CSoap User Guide

Rafik CHIBOUT, Mirco MUSOLESI.
Version 1.0
Feedback: rafik.chibout@inria.fr

Contents

1. Introduction

Before introducing CSoap, we describe the SOAP protocol and the WSDL language, because these technologies represent the basic knowledge required to understand CSoap and to use it.

1.1. Simple Object Access Protocol (SOAP)

It is important for application development to allow Internet communication between programs. Today's applications communicate using Remote Procedure Calls (RPC) between objects like DCOM and CORBA, but HTTP was not designed for this. RPC represents a compatibility and security problem; firewalls and proxy servers will normally block this kind of traffic. A better way to communicate between applications is over HTTP, because HTTP is supported by all Internet browsers and servers. SOAP was created to accomplish this. SOAP provides a way to communicate between applications running on different operating systems, with different technologies and programming languages.

SOAP is an XML-based communication protocol and encoding format. Originally conceived by Microsoft and Userland software, it has evolved through several generations and the current specification of W3C, SOAP 1.2, is fast growing in popularity and usage. The W3C's XML Protocol working group is in the process of turning SOAP into a true open standard. SOAP is widely viewed as the backbone to a new generation of cross-platform cross-language distributed computing applications, termed Web Services.

SOAP usually exchanges messages over HTTP: the client posts a SOAP request, and receives either an HTTP success code and a SOAP response or an HTTP error code.

A SOAP message is an ordinary XML document containing the following elements:

- A required Envelope element that identifies the XML document as a SOAP message. It contains:
  - An optional Header element that contains header information
  - A required Body element that contains call and response information and an optional Fault element that provides information about errors that occurred while processing the message

All the elements above are declared in the default namespace for the SOAP envelope `http://www.w3.org/2001/12/soap-envelope` and the default namespace for SOAP encoding and data types is `http://www.w3.org/2001/12/soap-encoding`.

Look at this SOAP Message skeleton:

```
<?xml version="1.0"?>
<soap:Envelope  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
                soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
     <soap:Header>
            ...
            ...
     </soap:Header>
     <soap:Body>
            ...
            ...
         <soap:Fault>
            ...
         </soap:Fault>
     </soap:Body>
</soap:Envelope>
```

1.2. Web Services Description Language (WSDL)

WSDL is an XML format used to describe network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete

endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate, however, the only bindings described in this document describe how to use WSDL in conjunction with SOAP 1.1, HTTP GET/POST, and MIME.

As communications protocols and message formats are standardized in the web community, it becomes increasingly possible and important to be able to describe the communications in some structured way. WSDL addresses this need by defining an XML grammar for describing network services as collections of communication endpoints capable of exchanging messages. WSDL service definitions provide documentation for distributed systems and serve as a recipe for automating the details involved in applications communication.

A WSDL document defines **services** as collections of network endpoints, or **ports**. In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings. This allows the reuse of abstract definitions: **messages**, which are abstract descriptions of the data being exchanged, and **port types** which are abstract collections of **operations**. The concrete protocol and data format specification for a particular port type constitutes a reusable **binding**. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service. Hence, a WSDL document uses the following elements in the definition of network services:

Types: a container for data type definitions using some type system (such as XSD).
Message: an abstract, typed definition of the data being communicated.
Operation: an abstract description of an action supported by the service.
Port Type: an abstract set of operations supported by one or more endpoints.
Binding: a concrete protocol and data format specification for a particular port type.
Port: a single endpoint defined as a combination of a binding and a network address.
Service: a collection of related endpoints.

These elements are described in detail at http://www.w3.org/TR/wsdl.

1.3. CSoap

CSoap is essentially a SOAP engine for resource-constrained devices such as PDA (*Personal Digital Assistant),* which, is able to deploy Web Services and to manage RPCs (Remote Procedure Call) from SOAP clients and dispatch them to services. CSoap is also a framework for developing Web Services and the clients of them. CSoap implementation follows the Sun's JAX-RPC Specification, referred as the JSR-101 specification, which gives a standard for SOAP-based RPC to support the development of SOAP-based interoperable and portable Web services (see `http://java.sun.com/xml/jaxrpc/index.jsp` for more detail). The current version of CSoap is an open source software written in Java language, and it may be running under Java VM or CVM (Virtual Machine for limited devices).

CSoap provides the following key features:

- **Speed.** CSoap uses SAX (event-based) XML parser to achieve a greater speed.
- **Flexibility.** The CSoap architecture gives the developer complete freedom to insert extensions into the engine for custom header processing, add some processing before/after service's running.
- **Interoperability/Portability/Stability.** CSoap follows Sun's JAXRPC (*Java API for XMLbased RPC*) specification that gives to CSoap interoperability, portability and stability.
- **Small footprint**. The memory footprint of CSoap implementation is of 90Kb that let CSoap run on resource-constrained devices, such as PDA.
- **Transport independent.** The core of the engine is completely transport-independent. The current version of CSoap is used only via HTTP transport but it's possible to use CSoap into other transport-protocols, such as SMTP, FTP….
- **WSDL support.** CSoap uses the WSDL documents to automatically generate the client stubs to access remote services and the necessary files to deploy services on CSoap Server.

To use CSoap you need the following knowledge:

- Core Java data types, classes and programming concepts.
- What threads are, race conditions, thread safety and synchronization.

- What a `classloader` is, what hierarchical `classloaders` are, and the common causes of a "`ClassNotFoundException`".
- How to diagnose trouble from exception traces, what a `NullPointerException` (NPE) and other common exceptions are, and how to fix them.
- What a web application is, what a `servlet` is, where classes, libraries and data go in a web application.
- How to start your application server and deploy a web application on it.
- What a network is, the core concepts of the IP protocol suite and the sockets API. Specifically, what is TCP/IP.
- What HTTP is. The core protocol and error codes and HTTP headers.
- What XML is. Not necessarily how to parse it or anything, just what constitutes well-formed and valid XML.

1.4. What's in CSoap release?

This release includes the following features:

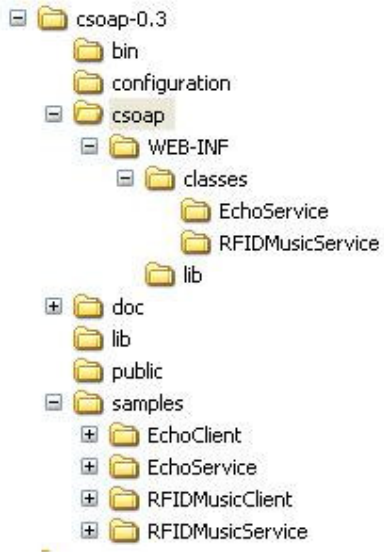
- Support of several Java types and their corresponding XML types: all Java primitive types, `Base64Binary` type, User-defined classes, Array of primitive Java type or of user-defined classes.
- Generator tool for building Java Stub and service's deployment files from WSDL documents.
- Support for session-oriented services, via HTTP cookies.
- HTTP servlet-based transport, which contains a CSoap Server instance and that can be plugged into servlet engines such as Jetty, Tomcat…
- Complete samples for getting start with CSoap.

2. Installing CSoap

This section describes how to install CSoap. CSoap have to be plugged in a Servlet Container. CSoap release is provided with Jetty Servlet Container, but it can be installed under others Servlet Containers that implement Servlet API version 2.2 or greater.

You find the latest CSoap release for Windows and Linux at http://www-rocq.inria.fr/arles/download/ozone/csoap-1.0.zip.

After having uncompressed the downloaded file, you will have this directories structure:



- `bin`: contains all CSoap scripts to start/stop/deploy… for Linux and Windows.
- `lib`: in this directory you will find csoap.jar and all JAR file libraries used by CSoap.
- `public`: is the public web directory, used by Jetty Server.

- `csoap`: this directory is CSoap context, that will contain the classes of the deployed services (see the next Section for more detail).
- `configuration/wsdd.conf`: this file contains the deployed services descriptions –WSDD (you will see more detail in WSDD Section).
- `csoap/samples`: this directory contains the java source and wsdd files of some services that you can deploy and see them in action to test CSoap installation.
- `doc`: contains this document and CSoap Architecture document.

- CSoap Context

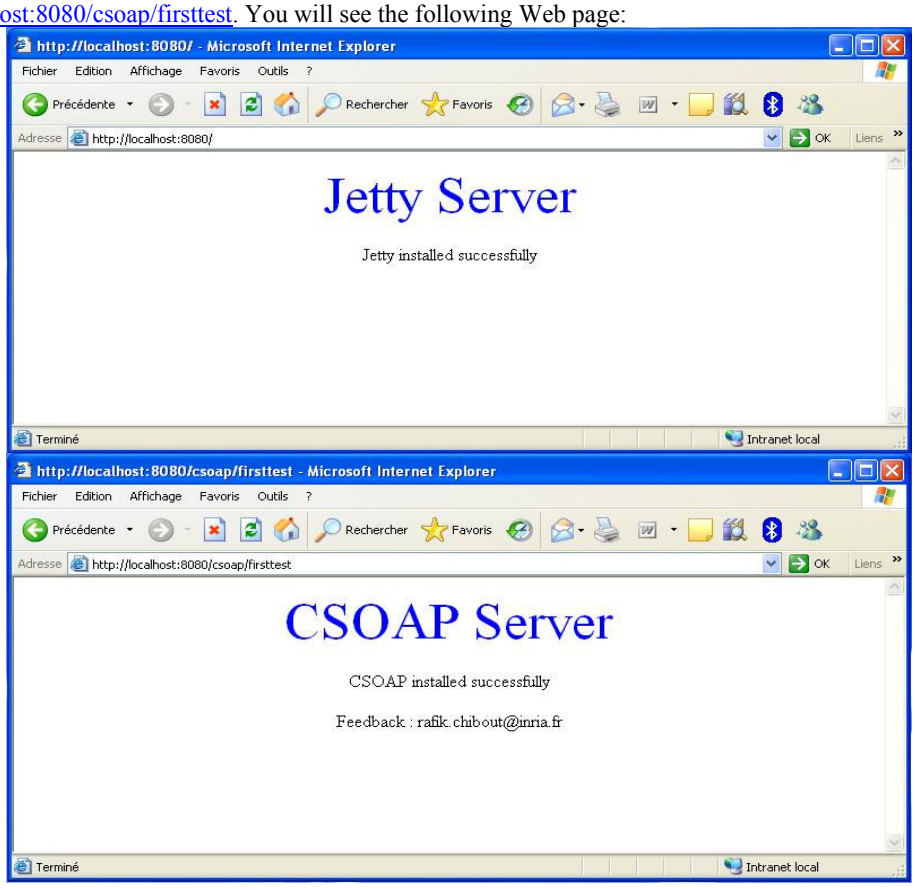
When you want to deploy a Web Service on CSoap, you have to copy the classes of the service to the `csoap/classes` directory. If your service classes are already packaged into JAR files, you can copy it into the `csoap/lib` directory and add it to the CLASSPATH variable. Also add any third party libraries that your service depends on into the same directories.

Before running CSoap you have to create an environment variable called **`CSOAP_HOME`**, which refers to the directory where CSoap is installed.

2.1. Starting and verifying the basic setup

To start CSoap Server use `bin/csoap.sh` (`bin/csoap.bat`) script. To stop CSoap use `bin/stopcsoap.sh` (`bin/stopcsoap.bat`) .

By default, Jetty server is configured to run on port 8080. You can change the port in `csoap.sh` (`csoap.bat`)
To verify that Jetty and CSoap are well-installed, look at the following addresses : http://localhost:8080/ http://localhost:8080/csoap/firsttest. You will see the following Web page:



3. Getting started

We consider as an example the `Echo` web service. This service provides an `echoString` method that receives an input string and returns the same string. The following is the Java code of the `Echo` service:

```
Package EchoService;

Public class EchoImpl
{
  String echoString(String str)
   {
     return str;
   }
}
```

3.1. Simple deployment of Web Services

To deploy the `Echo` service, the CSoap Server requires the definition of a deployment document referenced as **WSDD** (*Web Service Deployment Descriptor).* For more detail about WSDD see Section 4.
The following is the basic WSDD for `Echo` service:

```
<wsdd>
 <deployment xmlns="http://xml.csoap.org/csoap/wsdd/"
             xmlns:java="http://xml.csoap.org/csoap/wsdd/providers/java">
     <service name="Echo" scope="application" provider="JAVA:RPC">
       <parameter name="className" value="EchoService.EchoImpl"/>
           <operation name="echoString" qname="operNS:echoString"
                 xmlns:operNS="urn:EchoService" returnQName="echo"
                 returnType="xsd:string"
                 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
                <parameter name="str" type="xsd:string"
                 xmlns:xsd="http://www.w3.org/2001/XMLSchema" mode="IN"/>
           </operation>
     </service>
  </deployment>
</wsdd>
```

You'll find this file in `csoap/samples/EchoService/deployEcho.wsdd`

- `<service` **`name="Echo"`** `provider="java:RPC">`**:** define the name of the service. This name will be used from client to invoke the service.
- `<parameter name="className"` **`value="EchoService.EchoImpl"`**`/>`**:** define the service implementing class.
- `<operation` **`name="echoString"`** `qname="operNS:echoString"…>`**:** defines the service method `echoString`.
- **`<operation….returnType="xsd:string ….>:`** defines the return type of the `echoString` method.
- `<parameter` **`name="str" type="tns:string" … mode="IN"/>:`** the parameter **name="param"** defines the name of the `echoString`'s parameter, **`type="xsd:string"`** defines the xml type of this parameter and finally **`mode="IN"`** defines the mode of parameter (in this case is **`IN`**)**.**

To deploy the `Echo` service on the CSoap Server you can use the `bin/deploy.sh` (`bin/deploy.bat`) tool. This tool sends the service description file to CSoap Server. See this example:

```
$bin/deploy.sh samples/EchoService/deployEcho.wsdd
```

CSoap adds `Echo` service description in `configuration/wsdd.conf` file.
Now, you have only deployed the service description but not yet the service classes. To complete the deployment of this service, you have to copy the service classes (`EchoImpl.class and EchoPortType.class`) and all user-defined classes used by the `EchoImpl.class`, if any, into `csoap/WEB-INF/classes`.

Now the service Echo is available on CSoap and you can invoke its `echoString` method from client.
Use the `bin/EchoClient.sh` (`bin/EchoClient.bat`) to invoke the service Echo.

3.2. Web Services undeployment

To undeploy a service, CSoap requires an undeployment descriptor in which you must give the name of the service that you want to undeploy.

The following is an example of undeployment descriptor for `Echo` service:

```
<undeployment>
   <service name="Echo">
</undeployment>
```

To send this undeployment file to CSoap use also `deploy` tool, like in this example:

```
$bin/deploy.sh samples/EchoService/undeployEcho.wsdd
```

CSoap server removes automatically the service from the `configuration/wsdd.conf` file and the service becomes unavailable.
To undeploy a service you can also remove manually the description part of service from the `configuration/wsdd.conf`.

3.3. Simple CSoap Client

The following is a simple client of `EchoService` service:

```
1 package EchoClient;
2
3 import javax.xml.rpc.Call;
4 import javax.xml.namespace.QName;
5
6
7 public class SimpleEchoClient {
8     public static void main(String [] args) {
9       try {
10        String endpoint = "http://localhost:8080/csoap/Echo";
11
12        Call  call = (Call) new csoap.xml.rpc.CallImpl();
13
14        call.setTargetEndpointAddress(endpoint);
15        call.setOperationName(new QName("urn:EchoService","echoString"));
16        call.setReturnType(new QName("http://www.w3.org/2001/XMLSchema",
17                                     "string"));
18        call.addParameter("param",
19                          javax.xml.rpc.encoding.XMLType.XSD_STRING,
20                          javax.xml.rpc.ParameterMode.IN);
21
22        String ret = (String) call.invoke( new Object[] { "Hello!" } );
23        System.out.println("Sent 'Hello!', got '" + ret + "'");
24       }
25       catch (Exception e) {
26        System.err.println(e.toString());
27       }
28   }
}
```

You'll find this file in `csoap/samples/EchoClient/SimpleEchoClient.java`.

So what's happening here? In line 12 we create a new Call object. In line 14, we set up our endpoint URL - this is the destination for our SOAP message. In line 15 we define the operation (method) name of the Web Service. In line 16 we set return XML type of `echoString` method. In line 18 we assign the name `param` to the first (and alone) parameter on the `echoString` method and we also define the type of the parameter (`javax.xml.rpc.encoding.XMLType.XSD_STRING`) and whether it is an input, output or inout parameter (in this case it is an input parameter). And in line 22 we actually invoke the service, passing in an array of parameter values (in this case just one String value).
You can see what happens to the arguments by looking at the SOAP request that goes out on the wire (look at the coloured section below, and notice they match the values in the code above):

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xmlns:xsd=="http://www.w3.org/2001/XMLSchema">
     <SOAP-ENV:Body>
        <ns1:echoString xmlns:ns1="urn:EchoService">
            <param xsi:type="xsd:string">Hello!</testParam>
        </ns1:echoString>
     </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The client writing is simplified by the using of auto-generated Stub. See Section 6 for more details.

4. The CSoap Web Service Deployment Descriptor (WSDD)

4.1. Introducing WSDD

After having developed your services, you need to deploy them on CSoap Server to make them available for clients. Deploying a service involves copying the service classes into the deployed service directory and deploying service description on CSoap Server. The description of services is written in CSoap WSDD (*Web Service Deployment Description*) which gives all necessary XML-based targets to describe and configure a service.

4.2. WSDD general schema

Here you can see an example of the general schema of a WSDD file.

```
<deployment xmlns="http://arles.inria.fr/csoap/wsdd/">
   <service name="service Name" provider="java:RPC"
            scope="Request/Application/Session">
     <!-- handlers definition optional target -->
     </handler>
      ....
     </handler>
     <parameter name="className" value="java class name"/>
     <operation name="method name" qname="XML method name"
               returnQName="return QName" returnType="XML type">
       <parameter name="parameter name" type="XML parameter type"
                  mode="IN/OUT/INOUT">
     </operation>
     <!-- beanMapping defintion optional target -->
     <beanMapping>
      .....
     </beanMapping>
     <!-- typeMapping defintion optional target -->
     <typeMapping>
```

In this section we describe only mandatory targets, the optional targets are described in separate sections.

- `<service name="service Name" provider="java:RPC">`
  In this part you define the name of the service which will be used by clients to invoke the service. The `provider` attribute defines the provider class that will be used to run the service. The default provider in CSoap is `csoap.server.RPCProvider` class and is specified by the `java:RPC` value of `provider` attribute. You can define your own provider class that will be used to run the services instead of the the default one. The provider class must contain a `runService()` method that has the role to implement the service call.

- `<parameter name="className" value="java class name"/>`
  Specifies the class that must be used by provider when a call to the service is received (e.g., `csoap.samples.echoService.echo` for the service `EchoService`).

- `<operation name="method name" qname="XML method name" xmlns:operNS="URI"`
  `returnQName="return QName" returnType="xml type">`
  `<parameter …..>`
  `</operation>`
  All the methods of the deployed service must be defined with the operation target. In this target you must define the java method name, the XML method name, XML return value name, method's XML return type. Each parameter of this method is specified by the following `<paramater>` tag:

  - `<parameter name="parameter name" type="parameter type"`
    `mode="IN/OUT/INOUT">`
    This tag contains the parameter name, type and mode (IN, OUT or INOUT).

To deploy more than one service you can write a WSDD file for each service or one WSDD file that contains all deployment descriptors of services into `<wsdd>` target. See the following schema:

```
<wsdd>
  <!- - deploy service 1 -->
  <deployment>
   <service name="MyService1" provider="java:RPC">
...
   </service>
  </deployment>

  <!-- deploy service 2 -->
  <deployment>
   <service name="MyService2" provider="java:RPC">
...
   </service>
  <deployment>

...

</wsdd>
```

4.3. Mapping between XML and Java Data in CSoap

Using WSDD requires knowing how XML type mapping is mapped to Java type. Interoperability is an ongoing challenge between SOAP implementations. For this reason the basic mapping between Java types and WSDL/XSD/SOAP in CSoap is determined by the JAX-RPC specification. See chapters 4 and 5 of the specification for details.

Standard mappings from XML to Java

The following table specifies the Java mapping for the built-in simple XML data types. These XML data types are as defined in the XML schema namespace [`http://www.w3.org/2001/XMLSchema`] referenced by **`xsd`** prefix and the SOAP encoding namespace [`http://schemas.xmlsoap.org/soap/encoding/`] referenced by **`soapenc`** prefix.

XML type	Java type
xsd:base64Binary	`byte[]`
xsd:Boolean	`Boolean`
xsd:byte	`Byte`
xsd:dateTime	`java.util.Calendar`
xsd:decimal	`java.math.BigDecimal`
xsd:double	`Double`
xsd:float	`Float`
xsd:int	`Int`
xsd:integer	`java.math.BigInteger`
xsd:long	`Long`
xsd:qname	`javax.xml.namespace.Qname`
xsd:short	`Short`
xsd:string	`java.lang.String`

An element declaration with nillable attribute set to true for a built-in simple XML data type is mapped to corresponding Java wrapper class for the Java primitive type.

For example the schema instance

```
<xsd:element name="code" type="xsd:int" nillable="true"/>
```

With the following element:

```
<code xsi:nil="true"/>
```

is mapped to `java.lang.Integer` class. The following table specifies the mapping of element declarations with nillable attribute set to true for the built-in simple XML types.

XML Type	Java Type
xsd:Boolean	`Java.lang.Boolean`
xsd:byte	`Java.lang.Byte`
xsd:double	`Java.lang.Double`
xsd:float	`Java.lang.Float`
xsd:int	`Java.lang.Int`
xsd:long	`Java.lang.Long`
xsd:short	`Java.lang.Short`

The SOAP 1.1 specification indicates that all SOAP encoded elements are nillable. So in the SOAP encoded case, SOAP encoded simple XML type is mapped to the corresponding Java wrapper class for the Java primitive type. An example is mapping of the `soapenc:int` to the `java.lang.Integer`. The following table shows the Java mapping of the SOAP encoded simple types.

SOAP Type	**`Java Type`**
soapenc:base64	`byte[]`
soapenc:boolean	`java.lang.Boolean`
soapenc:byte	`java.lang.Byte`
soapenc:decimal	`java.math.BigDecimal`
soapenc:double	`java.lang.Double`
soapenc:float	`java.lang.Float`
soapenc:int	`java.lang.Integer`
soapenc:long	`java.lang.Long`
soapenc:short	`java.lang.Short`
soapenc:string	`java.lang.String`

4.4. Advanced WSDD - Specifying more options

WSDD descriptors can also contain other advanced information about services that we cover in this section.

Scoped Services

CSoap supports three different ways of scoping service objects (the actual Java objects which implements your methods):

- "*Request*" scope, the default, will create a new object each time a *SOAP Request* comes in for your service.
- "*Application*" scope will create a singleton shared object to service all requests.
- "*Session*" scope will create a new object for each session-enabled client who accesses your service.

To specify the scope option, you add a scope attribute into the `<service>` target where the value is "Request", "Application" or "Session" like this example of service scoped with "Session":

```
<service  name="echoService"  Provider="java:RPC"
          scope="Session">
...
</service>
```

Handlers and Chains

Sometimes you need to do some processing over SOAP Messages before using them to invoke the service (ex security process, message tracking process) or before sending them (ex compression process, redirection process…). CSoap allows defining this process into a java class that called Handler. The handler is a java class which implements the `javax.xml.rpc.handler.Handler` interface and executes a SOAP message processing. The `handleRequest()` method of Handler perform any processing of the request message and `handleResponse()` method perform any processing of the response message. CSoap WSDD provides the `<handler>` target to define a handler, a `<requestFlow>` target, which, can contain a collection of defined-handlers that will be invoked when a SOAP request message is received, and a `<responseFlow>` target which contains a collection of defined-handlers that will be invoked when a SOAP response message is going to be sent.

The collection of handlers defined by `<requestFlow>` and `<responseFlow>` are either Global or Service-specific, the Global collection perform any processing of all messages which crosses the CSoap Engine and the Service-specific collection perform any processing of only the messages on its way to go or go back of the specified Service. The Global collection must be defined into the service named «*» and the Service-specific collection must be defined into the concerned service. We give here an example of Service-specific collection of handlers which contains one handler named `showMessages` implemented by class `csoap.util.handlers.ShowMessage`.

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
   xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
   <!-- define the logging handler configuration -->
   <handler name="showmessages"
            type="csoap.util.handlers.HandlerRequestMessage">
   </handler>

   <!-- define the service, using the handlerRequest we just defined -->
   <service name="echoService" provider="java:RPC">
     <requestFlow>
        <handler type="showmessage"/>
     </requestFlow>
     <responseFlow>
        <handler type="showmessage"/>
     </responseFlow>

     <parameter name="className" value="csoap.examples.echoService.echo"/>
..
   </service>
</deployment>
```

The first section defines a Handler called "`showmessage`" that is implemented by the class `csoap.util.handlers.HandlerRequestMessage`. Then we define the `EchoService` service, already seen in the first example, and we add the <requestFlow> and `<responseFlow>` elements inside the `<service>` target. The `<requestFlow>` indicates a set of Handlers that should be invoked when the service is invoked, before the provider. By inserting a reference to "`showmessage`", we ensure that the Request message will be shown each time this service is invoked.
The `<responseFlow>` indicates a set of Handlers that should be invoked after service's execution and before sending the SOAP Response to the client.

The object that is passed to each Handler, when invoked, is a `SOAPMessageContext` object. To facilitate the access to the CSoap Message and to the main properties of CSoap Engine, CSoap puts a `SOAPMessage` object (implementing the `javax.xml.soap.SOAPMessage` interface and representing either a request Message or a response Message) and these properties into the `SOAPMessageContextImpl` class. The `SOAPMessageContextImpl` implements the `javax.xml.rpc.handler.soap.SOAPMessageContext` interface.
The following code is a part of the `showMessage` Handler which prints the SOAP Message on the screen.

```
package csoap.util.handlers;

import javax.xml.soap.*;

public class ShowMessage implements javax.xml.rpc.handler.Handler
{
    public void destroy() {... }

    public boolean handleFault(javax.xml.rpc.handler.MessageContext
                               messageContext) {
           return true;
           }

    public boolean handleRequest(javax.xml.rpc.handler.MessageContext
                                 messageContext) {
        try {
             javax.xml.soap.SOAPMessage msg = msgContext.getMessage();
             msg.writeTo(System.out);
            } catch (Exception e) {
              e.printStackTrace();
            }
        return true;
        }

    public boolean handleResponse(javax.xml.rpc.handler.MessageContext
                                  messageContext) {
        try {
             javax.xml.soap.SOAPMessage msg = msgContext.getMessage();
             msg.writeTo(System.out);
            } catch (Exception e) {
              e.printStackTrace();
            }
        return true;
          }

    public void init(javax.xml.rpc.handler.HandlerInfo handlerInfo) {
    }
}
```

The `handlerRequest` method is invoked by CSoap Engine after having received a SOAP request. The `handlerRequest` method gets the SOAP message from `SOAPMessageContext` which contains all information about the SOAP message and write the SOAP message on the screen.
The `handlerResponse` method is invoked by CSoap engine before sending a SOAP response message to the client.

Serialization and Deserialization

The serialization (also known as marshalling or encoding) is the process of transforming a Java data type into an XML representation. The deserialization (also known as unmarshalling or decoding) is the process of transforming an XML data type into the corresponding Java data type.
CSoap includes the ability to serialize/deserialize all java simple types and gives two classes (**`csoap.xml.rpc.encoding.SerializerComplexTypeFactory`**/ **`csoap.xml.rpc.encoding.DeserializerComplexTypeFactory`**) which are respectively able to serialize and to deserialize user defined classes that have only public attributes.
To deploy some services which use user-defined classes you must tell CSoap which Java classes map to which XML types. `To` do that, CSoap provides a `<beanMapping>` target allowing to specify a mapping between a Java type and an XML type.
As an example of serialization and deserialization we take a hotel service providing methods `listHotels()` and `reservation()`. The Hotel service uses a `csoap.samples.hotelService.ClientInfo` and `csoap.samples.hotelService.hotelDescription` user-defined complex-type. To deploy the Hotel service you need to specify to CSoap the type mapping between `ClientInfo` class and the XML Type `[urn:HotelService:Client]` and between the `hotelDescription` class and the XML Type `[urn:HotelService:HotelDescription].`

The follow code shows the h`otel` class.

```
Package csoap.samples.hotelService;

Public class hotel
{

   hotelDescription[] listHotels(String region)
   {
    ....
   }

   boolean reservation(int hotelId, Client client, Date date, int numdays)
   {
    ....
   }
}
```

The `listHotels(String region)` method returns an array of `hotelDescription` for the given region, and the `reservation(int hotelId, ClientInfo client, Date date, int numdays)` method makes a reservation in the hotel specified by it `hotelId`, for the given `client`, starting from `date` and lasting `numdays` days.

The following code is a part of Java code of `hotelDescription` class:

```
package csoap.samples.hotelService

public class hotelDescription
{
  public int hotelID
  public String hotelName;
  public string hotelAddress;
  public long hotelPrice;

  public void setHotelId(int id);
  public int getHotelId();
  public void setHotelName(String str);
  public String getHotelName();
…..
```

and the follow is a part of Java code of `ClientInfo` class:

```
package csoap.samples.hotelService

public class ClientInfo
{
  public String clientName;
  public string clientCard;

  public void setClientName(String str);
  public String getClientName();
…..
}
```

Look at the following WSDD description of hotel service which defines a type mapping for `ClientInfo` and `hotelDescription`:

```
<deployment xmlns="http://arles.inria.fr/csoap/wsdd/"
   xmlns:java="http://arles.inria.fr/csoap/wsdd/providers/java">
   <service name="hotelService" provider="java:RPC">
    <parameter name="className"
               value="csoap.examples.hotelService.hotelServer"/>
    <operation name="reservation" qname="operNS:reservation"
               xmlns:operNS="urn:HotelService" returnQName="reservationStat"
               returnType="xsd:boolean"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
          <parameter name="hotelId" type="xsd:int"
                     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
mode="IN"/>
          <parameter name="client" type="tNS:Client"
                     xmlns:tNS="urn:HotelService" mode="IN"/>
          <parameter name="reservationDate" type="xsd:datetime"
                     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
mode="IN"/>
          <parameter name="duration" type="xsd:int"
                     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
mode="IN"/>
    </operation>
    <beanMapping qname="tNS:Client" xmlns:tNS="urn:HotelService"
                 type="csoap.examples.hotelService.Client"/>
    <beanMapping qname="tNS:HotelDescription" xmlns:tNS="urn:HotelService"
                 type="csoap.examples.hotelService.hotelDescription"/>
   </service>
```

The `<beanMapping>` target allows defining a mapping between a `qname` that contains an XML type and `type` that contains a Java type. So in this case, we maps the `csoap.examples.hotelService.Client` class to

the XML QName `[urn:HotelService:Client]`. The `<beanMapping>` assigns the default serializer and deserializer factory to the specified java data type and xml type. These factories will be used to obtain a serializer/deserializer for the java data type. The default csoap serializer factory is `csoap.xml.rpc.encoding.SerializerComplexTypeFactory` and returns the `csoap.xml.rpc.encoding.SerializerComplexType` serializer. The default deserializer factory is `csoap.xml.rpc.encoding.DeserializerComplexTypeFactory` and returns the `csoap.xml.rpc.encoding.DeserializerComplexType` deserializer. This WSDD defines the mapping between `hotelDescription` class and `[HotelService:HotelDescription]`, but this is not enough to serialize an array of `hotelDescription` elements. It's necessary to define a specific (custom) serializer and deserializer of `hotelDescription[]`, by using `<typeMapping>` target which is described in the next Section.

Custom Serialization and Deserialization

To define custom serializer and deserializer factory which respectively contain a serializer and a deserializer able to serialize and deserialize our Java data type and it corresponding XML type, CSoap provides a `<typeMapping>` target. See the following schema of `<typeMapping>`:

```
<typeMapping qname="XML type"
           type="java class type"
           serializer="serializer Factory"
           deserializer="deserializer Factory"
           encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
```

Three extra attributes are added with respect to the `<beanMapping>` target:

- **`serializer`**, defines the Java class name of the Serializer factory which provides the serializer to be used to marshal an object of the specified Java class into XML.
- **`deserializer`**, define the Java class name of a Deserializer factory that provides the deserializer to be used to unmarshall XML into the correct Java class.
- **`encodingStyle`**, defines the used encoding.

(The `<beanMapping>` tag is really just shorthand for a `<typeMapping>` target with `serializer="csoap.xml.rpc.encoding.SerializerComplexTypeFactory"`, `deserializer="csoap.xml.rpc.encoding.DeserializerComplexTypeFactory"`, and `encodingStyle="http://schemas.xmlsoap.org/soap/encoding/")`.
CSoap provides a serializer and deserializer factory for array data types: `csoap.xml.rpc.encoding.SerializerArrayFactory` and `csoap.xml.rpc.encoding.DeserializerArrayFactory`.
Now look at the complete WSDD deployment of `hotelService`, with the mapping between `hotelDescription[]` and `[HotelService:ArrayOfHotel]`:

```
<deployment xmlns="http://arles.inria.fr/csoap/wsdd/"
   xmlns:java="http://arles.inria.fr/csoap/wsdd/providers/java">
   <service name="hotelService" provider="java:RPC">
    <parameter name="className"
               value="csoap.examples.hotelService.hotelServer"/>
    <operation name="listHotels" qname="operNS:listHotels"
               xmlns:operNS="urn:HotelService" returnQName="hotelEnumeration"
               returnType="rtNS:ArrayOfHotel" xmlns:rtNS="urn:HotelService">
          <parameter name="city" type="xsd:string"
                     xmlns:xsd="http://www.w3.org/2001/XMLSchema" mode="IN"/>
    </operation>
    <operation name="reservation" qname="operNS:reservation"
               xmlns:operNS="urn:HotelService" returnQName="reservationStat"
               returnType="xsd:boolean"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
          <parameter name="hotelId" type="xsd:int"
                     xmlns:xsd="http://www.w3.org/2001/XMLSchema" mode="IN"/>
          <parameter name="client" type="tNS:Client"
                     xmlns:tNS="urn:HotelService" mode="IN"/>
          <parameter name="reservationDate" type="xsd:datetime"
                     xmlns:xsd="http://www.w3.org/2001/XMLSchema" mode="IN"/>
          <parameter name="duration" type="xsd:int"
                     xmlns:xsd="http://www.w3.org/2001/XMLSchema" mode="IN"/>
    </operation>
    <beanMapping qname="tNS:Client" xmlns:tNS="urn:HotelService"
                 type="csoap.examples.hotelService.Client"/>
```

Sometimes you will need to write custom serializers and deserializers for application-specific data types, to do this you need to develop the four following classes:

- `SerializerFactory`: A factory class that returns a serializer based on the parsing mechanism, such as SAX and DOM. This class must implement `javax.xml.rpc.encoding.SerializerFactory` interface.
- `DeserializerFactory`: A factory class that returns a Deserializer based on the parsing mechanism. This class must implement `javax.xml.rpc.encoding.DeserializerFactory` interface.
- `Serializer`: This class is responsible for converting the java object into a corresponding XML structure. This class must implement `javax.xml.rpc.encoding.Serializer` interface
- `Deserializer`: This class is responsible for converting the XML structure into its corresponding java object. This class must implement `javax.xml.rpc.Deserializer` interface

As an example you can see the following CSoap classes provided to serialize and deserialize complex type objects which are instances of classes containing only public attributes:
`csoap.xml.rpc.encoding.SerializerComplexTypeFactory`
`csoap.xml.rpc.encoding.DeserializerComplexTypeFactory`
`csoap.xml.rpc.encoding.SerializerComplexType`
`csoap.xml.rpc.encoding.DeserializerComplexTypeFactory,`

After having developed your serializer and deserializer classes, you must tell CSoap which types they must be used for. You can do this with the `<typeMapping>` target in WSDD,

5. Advanced CSoap Client

Also on the client side it is possible to define handlers chains, bean mappings and type mappings options. To do so, you must create a WSDD client-side file (it is the same as the WSDD server-side) and use the `csoap.GlobalManagingClient` static class to pass the CSoap Client configuration contained in your WSDD client file. The following example is the CSoap client for hotel service. In this example we will make use of `clientDeploy.wsdd` file. The content of this file is the same of the `deploy.wsdd` file used above on the server side.

```
1  import javax.xml.rpc.Call;
2  import javax.xml.namespace.QName;
3  import csoap.examples.hotelService.hotelDescription;
4
5  public class TestClient {
6    public static void main(String [] args) {
7    try {
8      FileInputStream wsddClient = new
9      FileInputStream("csoap/examples/hotelService/clientDeploy.wsdd");
10     csoap.GlobalManagingClient.receiveWsdd(wsddClient)
```

In lines 8, 9 and 10 we configure CSoap client with `clientDeploy.wsdd` file. In line 20 we set return XML type of `listHotels()` method `[urn:HotelService:ArrayOfHotelDescription]. In line` 24 we invoke `listHotels()` with region value "*Paris*".
If you want only to specify type mappings for current service Client you can do this without a WSDD file configuration, in fact CSoap provides the method `registerTypeMapping()` method of `csoap.xml.rpc.CallImpl` that can be used to configure the method call. See the following example:

```
1  import javax.xml.rpc.Call;
2  import javax.xml.rpc.encoding.SerializerFactory;
3  import javax.xml.rpc.encoding.DeserializerFactory;
4  import javax.xml.namespace.QName;
5
6  import csoap.xml.rpc.encoding.SerializerComplexTypeFactory;
7  import csoap.xml.rpc.encoding.DeserializerComplexTypeFactory;
6  import csoap.xml.rpc.encoding.SerializerArrayFactory;
8  import csoap.xml.rpc.encoding.DeserializerArrayFactory;
9
10
11  import csoap.examples.hotelService.hotelDescription;
12
13  public class TestClient {
14   public static void main(String [] args) {
15    try {
16
17    String endpoint ="http://localhost:8080/csoap/services/HotelService";
18      SerializerFactory sf = null;
19      DeserializerFactory df = null;
20
21      csoap.xml.rpc.CallImpl csoapCall = new csoap.xml.rpc.CallImpl();
22      Class javaClass = hotelDescription.class ;
23      QName xmlType = new Qname("urn:HotelService","HotelDescription");
24      sf = (SerializerFactory)(new SerializerComplexTypeFactory());
25      df = (DeserializerFactory)(new DeserializerComplexTypeFactory());
26      csoapCall.registerTypeMapping(javaClass,xmlType,sf,df);
27
28      javaClass = HotelDescription[].class;
29      xmlType=new Qname("urn:HotelService","ArrayOfHotelDescription");
30      sf = (SerializerFactory)(new SerializerArrayFactory());
31      df = (DeserializerFactory)(new DeserializerArrayFactory());
```

In line 21 we create a csoap Call and in line 26 we register the type mapping between `csoap.example.hotelService.hotelDescription` class and `[urn:HotelService:HotelDescription]` XML type and we define serializer and deserializer factories `csoap.xml.rpc.encoding.SerializerComplexTypeFactory` and `csoap.xml.rpc.encoding.DeserializerComplexTypeFactory`

In line 32 we register a type mapping between `csoap.example.hotelService.hotelDescription[]` java type and `[urn:HotelService:ArrayOfHotelDescription]` XML Type and we define serializer and deserializer factories `csoap.xml.rpc.encoding.SerializerArrayFactory` and `csoap.xml.rpc.encoding.DeserializerArrayFactory`

6. CSoap Generator -Using WSDL with CSoap

6.1. Introducing CSoap Generator

The development of a Web service is a process that requires writing large pieces of code that follow in many cases the same patterns. This is more evident in the case of the development of the software components that are necessary for Web Service remote invocation. You have to write the stub and all the classes related to the data serialization process. For these reasons, we have developed a code generator that automatically generates a large part of client-side Java files that are involved in these mechanisms and the WSDD deployment file given its well-formed WSDL file. These files are generated according to CSoap.

More specifically, with respect to Web services invocation, our component creates all the files that are necessary to invoke a service that is described by a standard WSDL document; in other words, it generates a stub class, an interface class, the related user-defined types (including holders for data serialization) and exceptions classes.

6.2. Using CSoap Generator

The following is the syntax of the generator provided by CSoap:

```
%java csoap.wsdl.Generator –n fileName –s side –d dirName –c ImplClass –v
```

- **`fileName`** : is the name of the WSDL file.
- **`side`** : indicates if the code has to be generated for the server or the client side (**s** for server side and **c** for client side).
- **`dirName`** : is the name of the directory where the files will be generated.
- **`implClass`** : is the name of a class that implements the service.
- **`v`** : display debug information.

6.3. Generated files

The following table shows the files generated by CSoap Generator:

WSDL clause	**Java class(es) generated**
For each entry in the type section	A java class A holder if this type is used as an inout/out parameter
For each portType	A java interface
For each binding	A stub class
For each service	A service interface A service implementation One deploy.WSDD file with operation meta data One undeploy.WSDD file

6.4. WSDL elements and java generated classes

In this section we give some examples of the classes generated for each WSDL elements.

Types

The names of Java class generated from a WSDL file will depend on the names of the types defined in the WSDL file. For example, given the WSDL:

```
<xsd:complexType name="hotelDescription">
  <xsd:sequence>
   <xsd:element name="hotelId" type="xsd:int"/>
   <xsd:element name="hotelName" type="xsd:string"/>
   <xsd:element name="hotelAddess" type="xsd:string"/>
   <xsd:element name="hotelPrice" type="xsd:long"/>
  </xsd:sequence>
</xsd:complexType>
```

CSoap Generator will generate

```
public class Phone implements java.io.Serializable {
  public hotelDescription() {...}
  public int getHotelId() {...}
  public void setHotelId(int id) {...}
  public java.lang.String getHotelName() {...}
  public void setHotelName(java.lang.String name) {...}
  public java.lang.String getHotelAddess() {...}
  public void setHotelPrice(long price) {...}
  public long getHotelPrice() {...}
  public void setHotelAddress(java.lang.String address) {...}
  public boolean equals(Object obj) {...}
}
```

Complex types

Arrays declaration

According to WSDL specification, arrays must be declared by restriction. In other words, if you have to declare an array of Java String, you define it as an element belonging to a subset (a restriction) of the set of standard SOAP arrays. More formally, an array is derived from a `soapenc:array` by restriction using the `wsdl:arrayType` attribute, with the soapenc prefix associated to the URI `http://schemas.xmlsoap.org/soap/encoding` namespace and the wsdl prefix `http://schemas.xmlsoap.org/wsdl`.

The following example shows the definition of an array of String derived from the `soapenc:array` by restriction:

```
<xsd:complexType name="ArrayOfHotelDescription">
    <xsd:complexContent>
      <xsd:restriction base="soapenc:Array">
        <xsd:sequence>
          <xsd:element name="hotelDescription" minOccurs="0"
          maxOccurs="unbounded" type="tns:HotelDescription"/>
        </xsd:sequence>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
```

It is worth noting that the name of the complex type must start with `ArrayOf`; this limitation derives from the implementation of the WSIF library that we have used to manipulate the XML Schema declaration inside the WSDL file.

Holders

This type may be used as an *inout* or *out* parameter. Java does not have the concept of *inout/out* parameters. In order to achieve this behaviour, JAX-RPC specifies the use of holder classes. A holder class is simply a class that contains an instance of its type. For example, the holder for the `hotelDescription` class would be

```
package csoap.examples.hotelService.holders;
public final class HotelDescriptionHolder implements javax.xml.rpc.holders.Holder {
 public csoap.examples.hotelService.HotelDescription value;

 public HotelDescription Holder()
 {
 }

 public HotelDescription Holder(csoap.examples.hotelService.HotelDescription value) {
  this.value = value;
 }
}
```

A holder class is only generated for a type if that type is used as an inout or out parameter. Note that the holder class has the suffix "Holder" appended to the class name, and it is generated in a sub-package with the "holders".
The holder classes for the primitive types can be found in `javax.xml.rpc.holders`.

PortTypes

The Service Definition Interface (SDI) is the interface that's derived from a WSDL's portType. This is the interface you use to access the operations on the service. For example, given the WSDL

```
<message name="listHotelsRequest">
  <part name="region" type="xsd:string"/>
</message>

<message name="listHotelsResponse">
<part name="hotels" type="tns:ArrayOfHotelDescription"/>
</message>

<portType name="HotelInterface">
  <operation name="listHotles">
    <input message="tns:listHotelsRequest"/>
    <output message="tns:listHotelResponse"/>
  </operation>
</portType>
```

CSoap Generator will generate

```
public interface HotelInterface extends java.rmi.Remote {
   public csoap.examples.hotelService.HotelDescription[] listHotels(String region)
      throws  java.rmi.RemoteException;
}
```

A note about the name of the SDI: the name of the SDI is typically the name of the `portType`. However, to construct the SDI, CSoap Generator needs information from both the `portType` and the `binding`. (This is unfortunate and is a topic of discussion for WSDL version 2.)
JAX-RPC says (section 4.3.3): "The name of the Java interface is mapped from the name attribute of the `wsdl:portType` element. ... If the mapping to a service definition interface uses elements of the `wsdl:binding` ..., then the name of the service definition interface is mapped from the name of the `wsdl:binding` element."
Note the name of the spec. It contains the string "RPC". So this spec, and CSoap Generator, assumes that the interface generated from the `portType` is an RPC interface.

Bindings

A Stub class implements the SDI. Its name is the binding name with the suffix "Stub". It contains the code which turns the method invocations into SOAP calls using the CSoap Service and Call objects. It stands in as a proxy (another term for the same idea) for the remote service, letting you call it exactly as if it were a local object. In other words, you don't need to deal with the endpoint URL, namespace, or parameter arrays which are involved in dynamic invocation via the Service and Call objects. The stub hides all that work for you.
Given the following WSDL:

```
<binding name="HotelSOAPBinding" type="tns:HotelInterface">
  ...
</binding>
```

CSoap Generator will generate:

```
public class HotelSOAPBindingStub implements HotelService.HotelInterface {
{
 public HotelSOAPBindingStub(URL endpointURL,
           javax.xml.rpc.Service service)
 {...}

public HotelService.Hotel[] listHotels(java.lang.String city)
      throws java.rmi.RemoteException,HotelService.HotelException {
// implementation of Call listHotels
// return the result of invokation
}
}
```

Exceptions

You should pay particular attention in declaring exceptions; in fact, you must follow the standard defined in JAX-RPC specification. Now we focussed in particular on service specific exceptions. To sum up, a fault element, which is an optional element inside an operation block, specifies the format of any error message related to the remote invocation of a particular service method. It is worth noting that according to the WSDL specification, a fault message must have a single part.
The name of the Java exception is mapped from the name attribute of the message. For reference, you can look at the following example:

```
<message name="HotelException">
  <part name="errorMessage" type="xsd:string"/>
</message>
...
<portType name="HotelInterface">
  ...
  <operation name="listHotels">
    <input>...</input>
    <output>...</output>
    <fault name="HotelException" message="HotelException"/>
  </operation>
  ...
</portType>
```

The following is a fragment of the Java service endpoint interface derived from the above WSDL port type definition:

```
public interface HotelInterface extend java.rmi.Remote {
   public csoap.examples.hotelService.HotelDescription(String region)
      throws java.rmi.RemoteException,
           HotelService.HotelException;
}
```

In this example you can note that the fault element is mapped to the `com.HotelService.HotelException` exception that extends the `java.lang.Exception` class.

In this case the generator produces the following code for the definition of the user-defined exception class:

```
public class HotelException extends java.lang.Exception {
  public HotelException(java.lang.String errorMessage) {...}
  public getErrorMessage(){...}
}
```

It is worth observing that the single message part in the fault maps to a field with a getter method in the mapped exception. Moreover, it is declared as input parameter in the exception constructor.

6.5. CSoap Client With generated Stub

A typical usage of the stub classes would be as follows

```
public class client
{
 public static void main(String [] args) throws Exception {
  // Make a service
   String endpointURL="http://localhost:8080/csoap/services/HotelService"
   hotelInterface service = new HotelSOAPBindingStub(endpointURL,null);


  // Make the actual call
   HotelDescription[] list = service.listHotles("Paris");
}
}
```