Java and SOAP

Simple Object Access Protocol

What?

- Simple and lightweight mechanism to exchange information between peers.
- specifies a modular packaging model to encode information to be exchanged
- Specifically, it uses XML documents, called *SOAP messages*, to represent information.
- Does not specify any implementation specific semantics
- As a result, it can be used in a wide variety of systems

Differences with XML-RPC

- XML-RPC was designed to be as simple as possible, while allowing complex data structures to be transmitted, processed and returned.
- Simplicity is also XML-RPC's greatest limitation
- Can't do
  - passing an object to a function,
  - Specifying which portion of a receiving application the message is intended for etc
- SOAP implements
  - 1) user defined data types,
  - 2) the ability to specify the recipient,
  - 3) message specific processing control, and
  - other features.

Soap Architecture

- Designed by Dave Winer et al. in 1998 and maintained by W3C.
- SOAP architecture consists of three parts:
  - The SOAP envelope construct that
    - defines an overall framework to express message content, the message exchangers.
  - The SOAP encoding rules that
    - define a mechanism that can be used successfully to exchange application-specific type values.
  - The SOAP RPC representation
    - defines a protocol that can be used for remote procedure calls and responses.

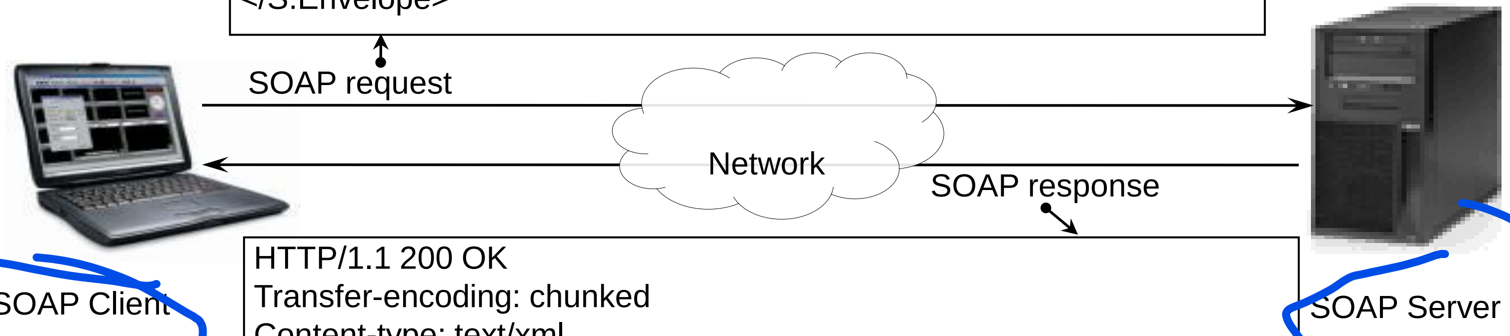
Soap Architecture

- The RPC part specification describes a standard, XML-based way to encode requests and responses, such as
  - Requests to invoke a method including parameters
  - Responses from a method including out parameters
  - Errors

RPC using SOAP

```
POST /ws/calc HTTP/1.1
Content-Type: text/xml
Content-Length: 215

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:add xmlns:ns2="http://ws.calc/">
          <arg0>7</arg0>
          <arg1>8</arg1>
       </ns2:add>
  </S:Body>
</S:Envelope>
```



```
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: text/xml

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:addResponse xmlns:ns2="http://ws.calc/">
      <return>15</return>
    </ns2:addResponse>
  </S:Body>
</S:Envelope>
```

SOAP Messages

- Primary part of the SOAP is the messaging framework
- Any SOAP message is an XML document having the following minimal specification.
  - The mandatory root element <Envelop>.
  - The <Envelop> element contains an optional <Header> element followed by a mandatory <Body> element.

Template

- Here is a sample template that shows the structure of a SOAP message:
  - <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  -   <soap:Header> <!-- optional -->
  -     <!—control information goes here... -->
  -   </soap:Header>
  -   <soap:Body><!-- mandatory -->
  -     <!-- payload or Fault element goes here... -->
  -   </soap:Body>
  - </soap:Envelope>

Template

- Elements must always belong to the namespace http://schemas.xmlsoap.org/soap/envelope/
- Root element <Envelop> and namespace together make an XML document a SOAP message.
- Optional <Header> element and one mandatory <Body> element that contains payload

Example

- <?xml version="1.0" ?>
- <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
- <soap:Body>
- <c:add xmlns:c="http://ws.calc/">
- <arg0>7</arg0>
- <arg1>8</arg1>
- </c:add>
- </soap:Body>
- </soap:Envelope>

Example explained

- This message represents a call of a method add() with two argument values 7 and 8.
- The <Body> element can have any number of attributes or children elements from any namespace.
- This is ultimately where the data, which we want to send, goes in.

Example

- The receiver of the above message may send the result using another message as follows:
  - <?xml version="1.0" ?>
  - <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  - <soap:Body>
  - <c:addResponse xmlns:c="http://ws.calc/">
  - <return>15</return>
  - </c:addResponse>
  - </soap:Body>
  - </soap:Envelope>

SOPA message validation

- Validated against the underlying schema which can be downloaded from http://schemas.xmlsoap.org/soap/envelope/
- We downloaded a free command line validator from https://dl.dropbox.com/u/10564628/xsd11-validator.jar.
- use the following command:
  - java -jar xsd11-validator.jar -sf SOAP.xsd -if soap.xml

<Header>

- SOAP itself does not specify any built-in headers.
- Example
  - <?xml version="1.0" ?>
  - <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  - <soap:Header>
  - <s:mode xmlns:s="sample">cheque</s:mode>
  - </soap:Header>
  - <soap:Body>
  - <pay>10000</pay>
  - <to>B. S. Roy</to>
  - </soap:Body>
  - </soap:Envelope>

(10 [illegible])

- The body contains an instruction to pay an amount 10000. The header tells that the payment should be made by cheque.

Fault message

- Special message used to indicate error
- The <Fault> element is used for this purpose.
- Example:
  - <?xml version="1.0" ?>
  - <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  - <soap:Body>
  - <soap:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
  - <faultcode>soap:Client</faultcode>
  - <faultstring>Cannot find dispatch method for {http://ws.calc/}add1</faultstring>
  - </soap:Fault>
  - </soap:Body>
  - </soap:Envelope>

Imp

Imp

RPC Using SOAP

- Although SOAP was originally intended for exchanging any type of messages, it is primarily used for Remote Procedure Call (RPC).
- Here is a sample HTTP SOAP request message

SOAP RPC request message

```
POST /ws/calc HTTP/1.1
Accept: text/xml, multipart/related
Content-Type: text/xml; charset=utf-8
SOAPAction: "http://ws.calc/Calculator/addRequest"
User-Agent: JAX-WS RI 2.2.4-b01
Host: 172.16.5.81:9999
Connection: keep-alive
Content-Length: 215

<?xml version="1.0" ?>
<S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:add xmlns:ns2="http://ws.calc/">
      <arg0>4</arg0>
      <arg1>3</arg1>
    </ns2:add>
  </S:Body>
</S:Envelope>
```

SOAP RPC request message

- A SOAP request message is an HTTP POST request message with two mandatory headers Content-Type and Content-Length.
- The body of the HTTP request message is a SOAP message which represents the call of a method add with two arguments 4 and 3.
- The corresponding response method may look like this:

SOAP RPC response message

```
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: text/xml; charset=utf-8
Date: Sun, 30 Mar 2014 07:53:06 GMT

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:addResponse xmlns:ns2="http://ws.calc/">
      <return>7</return>
    </ns2:addResponse>
  </S:Body>
</S:Envelope>
```

Web Service (WS)

- An application component that provides a service which is available over the web.
- This service is called service consumer/client
- Communication between a service provider and a consumer takes place through XML messages
- Web services are built on the top of HTTP and SOAP messages are used for communication. Imp
- The description (operations offered, message formats, bindings, location etc.) of a web service is provided as an XML document which is written in a language called **Web Service Description Language (WSDL)**. ← Imp

JAX-WS

Py Q

- Java API for XML Web Services (JAX-WS) is an Application Programming Interface (API) for developing web services and clients.
- JAX-WS web services and clients use SOAP messages for communication and HTTP for message transport

Imp

- JAX-WS is a part of the Java EE, it can also be used in Java SE version 6 and onwards
- Does not require any servlet of EJB container.

JAX-WS --basic Idea

Imp

- The web service developer specifies the operations by defining methods in a Java interface and also provides implementation of those methods.
- A client creates a proxy (a local object representing the service) and then simply calls methods on the proxy.
- JAX-WS runtime system converts the calls and responses to and from SOAP messages.

✓✓

JAX-WS packages

Package	Description
javax.jws	APIs for Java to WSDL mapping annotations
javax.jws.soap	APIs for mapping the Web Service to SOAP
javax.xml.ws	The Core JAX-WS APIs
javax.xml.ws.http	APIs for XML/HTTP Binding
javax.xml.ws.handler	APIs for message handlers
javax.xml.ws.soap	APIs for SOAP/HTTP Binding
javax.xml.ws.spi	SPIs for JAX-WS
javax.xml.ws.spi.http	HTTP SPI used for portable deployment of JAX-WS in containers
javax.xml.ws.wsaddressing	APIs for WS-Addressing

Advantage over JAX RPC

- It introduced some new features that were absent in JAX-RPC some of which are mentioned below:
  - Better platform independence for Java applications
  - Annotations
  - Invoking Web services asynchronously
  - Data binding with JAXB 2.2
  - Dynamic and static clients
  - Message Transmission Optimization Mechanism (MTOM)
  - Multiple payload structures
  - SOAP 1.2 support
  - Support for method parameters and return types

Developing web service

- We shall develop a web service for our calculator application
- We shall first concentrate on how to write and invoke a web service.
- Our web service will have a single operation add().
- We first write an interface Calculator as follows:

Developing web service

- package calc.ws;
- import javax.jws.WebMethod;
- import javax.jws.WebService;
- import javax.jws.soap.SOAPBinding;
- import javax.jws.soap.SOAPBinding.Style;
-
- //Service Endpoint Interface
- @WebService
- @SOAPBinding(style = Style.RPC)
- public interface Calculator {
-   @WebMethod public int add(int a, int b);
- }

Developing web service

- This is an ordinary Java interface except that it uses some annotations.
- JAX-WS uses annotations extensively, to simplify the development and deployment of web service.
- An interface, to be a web service interface, must be annotated by @WebService.
- The @SOAPBinding annotation maps this web service to SOAP.
- The element style instructs to use RPC encoding style for messages sent to and from the web service.
- Note that SOAP supports two kinds of encoding: *RPC style* and *document style*.
- The default is document.

Developing web service

- The @WebMethod tells that the method add() has to be exposed as a web service operation
- The exposed method must be public.
- interface is ready and acts as a contract between the web service and the client.
- The next step is to write an implementation of this interface.

Implementing web service

```
package calc.ws;
import javax.jws.WebService;
//Service Implementation
@WebService(endpointInterface = "calc.ws.Calculator")
public class SimpleCalculator implements Calculator {
  @Override
  public int add(int a, int b) {
    System.out.println("Received: " + a + " and " + b);
    int result = a + b;
    System.out.println("Sent: " + result);
    return result;
  }
}
```

Implementing web service

- This class defines the method add() declared in Calculator interface.
- When a class implements an endpoint interface, it is mandatory to use a @WebService annotation with an endpointInteface element specifying the fully qualified name of the interface.
- In the above implementation class, endpointInterface element tells that the name of the service endpoint interface defining the service's contract is calc.ws.Calculator.

Deploying web service

- JAX-WS includes a class javax.xml.ws.Endpoint to easily publish and configure a web service
- We shall use a separate class for publishing web service. Here is the essential line of code in the bootstrap class:
  - Endpoint.publish("http://172.16.5.81:6789/ws/calc", new SimpleCalculator());
- This essentially publishes an endpoint for SimpleCalculator object with the URL http://172.16.5.81:6789/ws/calc.

Deploying web service

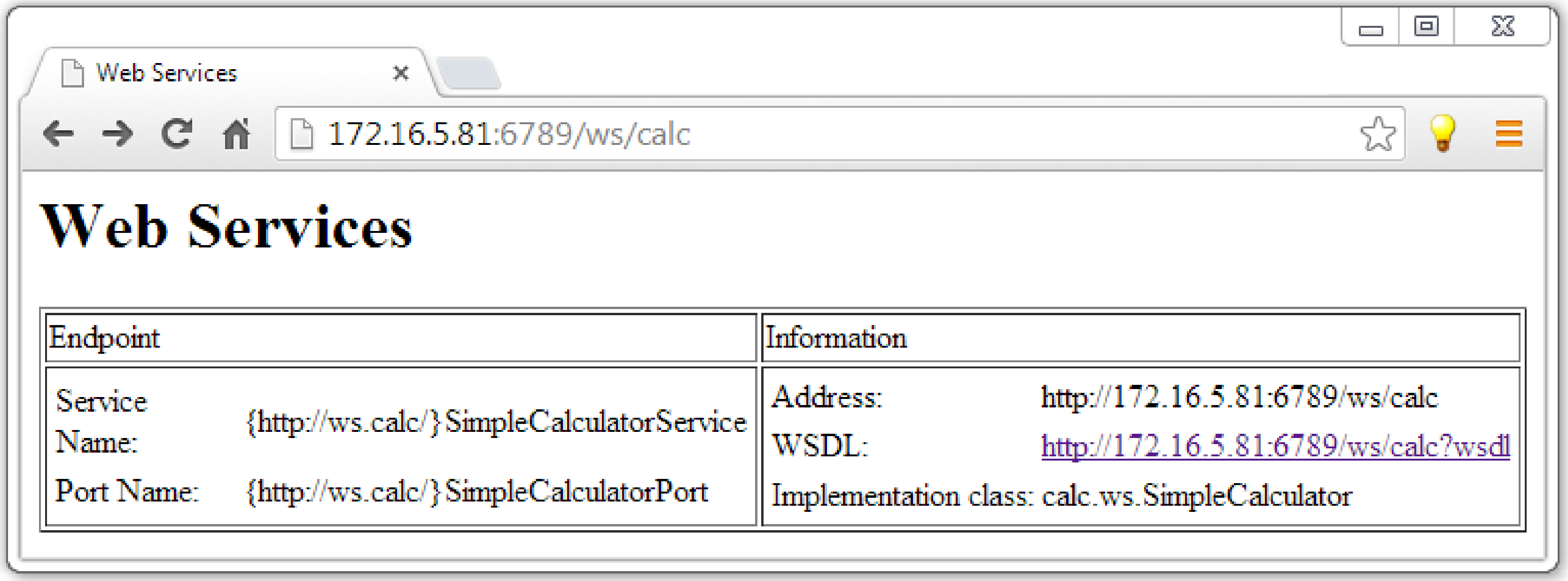
- The following is a complete source code (CalculatorPublisher.java) of the web service publisher:
  - import calc.ws.*;
  - import javax.xml.ws.Endpoint;
  - public class CalculatorPublisher {
  -   public static void main(String[] args) {
  - Endpoint.publish("http://172.16.5.81:6789/ws/calc", new SimpleCalculator());
  -   }
  - }

Compile and Run

- To compile and run the programs, create the following directory structure.
  - pub
  - │   CalculatorPublisher.java
  - ├───calc
  -      └───ws
  -              Calculator.java
  -                SimpleCalculator.java
- Go to the directory pub and use the following command to compile all classes:
  - javac calc\ws\*.java *.java
- To publish the web service, run the publisher using the following command
  - java CalculatorPublisher

Compile and Run

- You can check if the web service is published successfully or not by typing the following URL in a web browser:
  - http://172.16.5.81:6789/ws/calc
- If everything goes well, the browser's screen looks like this:



Web Services

Endpoint		Information	
Service Name:	{http://ws.calc/}SimpleCalculatorService	Address:	http://172.16.5.81:6789/ws/calc
Port Name:	{http://ws.calc/}SimpleCalculatorPort	WSDL:	http://172.16.5.81:6789/ws/calc?wsdl
		Implementation class:	calc.ws.SimpleCalculator

Invoking web service

- Client first creates a Service object that encapsulates a web service
- A Service object is usually created from WSDL contract which is available via WSDL URL.
- May be created
  - Manually
  - Using wsimport command

Creating a Service manually

- The create() method has many overloaded versions. The commonly used one takes a URL and a QName as follows:
  - URL url = new URL("http://172.16.5.81:6789/ws/calc");
  - QName qname = new QName("http://ws.calc/", "SimpleCalculatorService");
  -
  - **Service service = Service.create(url, qname);**

Invoking web service

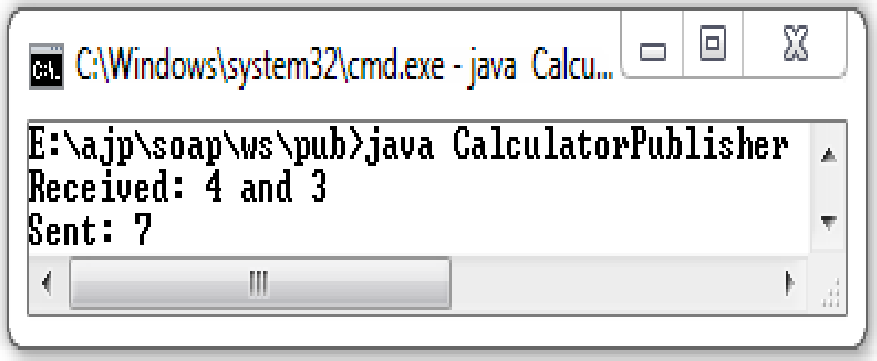
- This Service object is then used to create a local proxy to the web service:
  - Calculator cal = service.getPort(Calculator.class);
- Invoking an operation on the web service is now as simple as invoking a method on a local object:
  - int x = 4, y = 3;
  - int result = cal.add(x,y);

Compile and Run

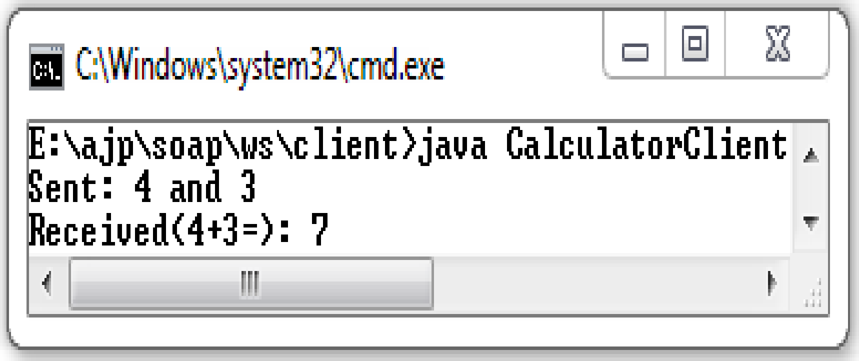
- Create the following directory structure in a machine for the web service client.
  - client
  - | CalculatorClient.java
  - ├───calc
  -     └───ws
  -             Calculator.java
- Note that the service endpoint interface Calculator.java is needed by this client.
- Had there been a provision, you may download it or may write one such interface manually.
- Open a terminal, go to the client directory and use thje following command to compile client files:
  - javac calc\ws\*.java *.java

Compile and Run

- Now, execute the client to invoke the web service as follows:
  - java CalculatorClient
- A sample result is shown



```
E:\ajp\soap\ws\pub>java CalculatorPublisher
Received: 4 and 3
Sent: 7
```



```
E:\ajp\soap\ws\client>java CalculatorClient
Sent: 4 and 3
Received(4+3=): 7
```

Tracking SOAP messages

- If you want to see the underlying SOAP messages at the publisher side, insert the following line of code in CalculatorPublisher.java:
  - `System.setProperty("com.sun.xml.internal.ws.transport.http.HttpAdapter.dump", "true");`
- Similarly, insert the following line of code to see SOAP messages at the client side:
  - `System.setProperty("com.sun.xml.internal.ws.transport.http.client.HttpTransportPipe.dump", "true");`

Tracking SOAP messages

- You will see that the client sends a SOAP message which looks like this:
  - <?xml version="1.0" ?>
  - <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  -   <S:Body>
  -     <ns2:add xmlns:ns2="http://ws.calc/">
  -       <arg0>4</arg0>
  -       <arg1>3</arg1><
  -     /ns2:add>
  -   </S:Body>
  - </S:Envelope>
  -

Tracking SOAP messages

- The web service responds with the following SOAP message:
  - <?xml version="1.0" ?>
  - <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  - <S:Body>
  - <ns2:addResponse xmlns:ns2="http://ws.calc/">
  - <return>7</return>
  - </ns2:addResponse>
  - </S:Body>
  - </S:Envelope>

Using WSDL

- WSDL is used to develop XML files that specify rules for communication between different systems such as:
  - How one system can talk to another system
  - Which specific data are needed in the request
  - What would be the structure of the XML file containing data
  - What error messages to display when a certain rule for communication is not observed, to make troubleshooting easier

Using WSDL

- When we published our web service using publish() method of Endpoint class, a WSDL document was also generated which can be accessed using the URL
  - http://172.16.5.81:6789/ws/calc?wsdl.

- This WSDL file can be used to generate helper classes such as Service Endpoint Interface (SEI), Service and Exception classes etc. using the application wsimport

Using WSDL

- The wsimport takes URL of WSDL file as a parameter, and generates a set of files, structured in a directory tree.
- In order to create these artifacts, create a directory (say client1), open a terminal, go to this directory and use the following command to generate JAX-WS artifacts:
  - wsimport -keep http://172.16.5.81:6789/ws/calc?wsdl

Using WSDL

- This will generate the Java artifacts and compile them by importing the http://172.16.5.81:6789/ws/calc?wsdl.
- A sample output of this command is shown below:
  - parsing WSDL...
  - Generating code.
  - Compiling code..

```
client1
 └──calc
      └──ws
            Calculator.class
            Calculator.java
            SimpleCalculatorService.class
            SimpleCalculatorService.java
```

  - A sample directory after creating artifacts is shown here:

Developing client

- Developing the client using these artifacts is very easy. We first create a service as follows:
  - SimpleCalculatorService calcService = new SimpleCalculatorService();
- The service class also has methods each of which returns a local proxy, called *dynamic proxy*, of service implementation.
- We get a reference to this proxy as follows:
  - Calculator cal = calcService.getSimpleCalculatorPort();

Client

- The complete source code of the client is shown below:
  - import calc.ws.*;
  - public class CalculatorClient {
  -   public static void main(String[] args) {
  -     **SimpleCalculatorService calcService = new SimpleCalculatorService();**
  -     **Calculator cal = calcService.getSimpleCalculatorPort();**
  -     int x = 4, y = 3;
  -     int result = cal.add(x,y);
  -     System.out.println("Sent: " + x +" and "+y);
  -     System.out.print("Received("+x+"+"+y+"=): " + result);
  -   }
  - }

Web service

- Since, in the above example, a client generates artifacts from WSDL document, the web service need not implement an interface and may be coded as :

```
package calc.ws;
import javax.jws.*;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;
@WebService
@SOAPBinding(style = Style.RPC)
public class SimpleCalculator {
  @WebMethod public int add(int a, int b) {
    System.out.println("Received: " + a + " and " + b);
    int result = a + b;
    System.out.println("Sent: " + result);
    return result;
  }
}
```

Deploy Web service

- Deploy it using Endpoint class as before.
- If you now generate the client artifacts in a directory client2, using wsimport, the following files are generated:
  - client2
  - └──calc
  -     └──ws
  -         SimpleCalculator.class
  -         SimpleCalculator.java
  -         SimpleCalculatorService.class
  -         SimpleCalculatorService.java

Client

- With these artifacts, the client code in the client2 directory will look like this:

```
import calc.ws.*;
public class CalculatorClient {
  public static void main(String[] args) {
    SimpleCalculatorService calcService = new SimpleCalculatorService();
    SimpleCalculator cal = calcService.getSimpleCalculatorPort();
    int x = 4, y = 3;
    int result = cal.add(x,y);
    System.out.println("Sent: " + x +" and "+y);
    System.out.print("Received("+x+"+"+y+"=): " + result);
  }
}
```

Document Style

- there are two different ways to encode and construct SOAP messages: *RPC style* and *Document style*.
- The WSDL document for RPC style merely tells us how to construct a SOAP message.
- Consider the WSDL document fragment or our previous web service:
  - …
  - **<message name="add">**
  - **<part name="arg0" type="xsd:int"/>**
  - **<part name="arg1" type="xsd:int"/>**
  - **</message>**
  - **<message name="addResponse">**
  - **<part name="return" type="xsd:int"/>**
  - **</message>**
  - …
  - <binding name="SimpleCalculatorPortBinding" type="tns:Calculator">
  - <soap:binding transport="http://schemas.xmlsoap.org/soap/http" **style="rpc"**/>

Document Style

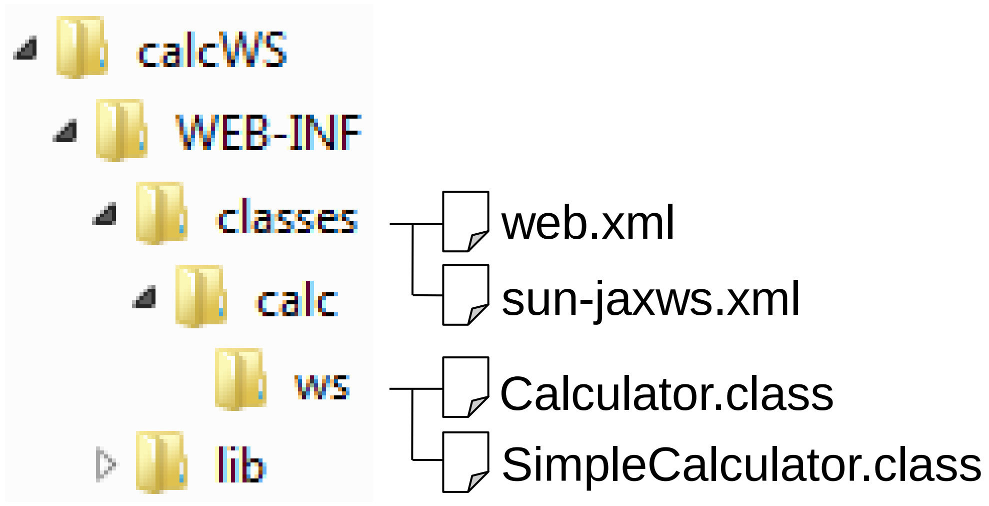
- It tells us that to call a method add, SOAP body should contain an XML document fragment as follows:
  - <add>
  - <arg0>4</arg0>
  - <arg1>3</arg1>
  - </add>
- However, there is no provision to verify this XML representation.
- Using document style web service is very easy.
- Simply include @SOAPBinding annotation as follows:
  - @SOAPBinding(style = Style.DOCUMENT)

Using tomcat to deploy web service

- Usually, a web service runs under the supervision of a servlet container such as Tomcat, JBoss etc.
- We need to perform the following steps, at a minimum:
  - Create the web application deployment descriptor (web.xml) and a proprietary web service deployment descriptor (for example, sun-jaxws.xml).
  - Package generated artifacts, service implementation class and those descriptors into a web archive (.war file).
  - Deploy the .war archive into the servlet container.

Using tomcat to deploy web service

- To deploy a JAX-WS Web Service using tomcat, we need JAX-WS API that can be downloaded from https://jax-ws.java.net/.
- We downloaded jaxws-ri-2.2.8.zip and when unzipped, the following directory structure was created.



Using tomcat to deploy web service

- Create a web application directory (say calcWS) in tomcat’s webapps directory.
- The API jars may be copied in the application’s lib (i.e. calcWS/WEB-INF/lib) directory.
- Now we create a standard web deployment descriptor web.xml for the deployment. It specifies WSServletContextListener as listener class and WSServlet as servlet class.
- Relevant portion is shown

Web.xml

```
<listener>
    <listener-class>
com.sun.xml.ws.transport.http.servlet.WSServletContextListener
    </listener-class>
  </listener>
  <servlet>
    <servlet-name>ws</servlet-name>
    <servlet-class>
        com.sun.xml.ws.transport.http.servlet.WSServlet
    </servlet-class>
  </servlet>
    <servlet-mapping>
      <servlet-name>ws</servlet-name>
      <url-pattern>/ws</url-pattern>
    </servlet-mapping>
```

Deployment descriptor

- We then create a web service deployment descriptor. The name of this file for JAX-WS is sun-jaxws.xml. A sample file is shown below:
  - <?xml version="1.0" encoding="UTF-8"?>
  - <endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">
  -   <endpoint name="Calculator" implementation="calc.ws.SimpleCalculator"
  -     url-pattern="/ws/calc"/>
  - </endpoints>

Using Ant to build war file

- If you want to build a web archive (WAR) file, create the following directory structure.
- Place all JAR files from jaxws-ri/lib in calcWS/WebContent/WEB-INF/lib directory.

```
calcWS
 │    build.xml
 ├──src
 │    └──calc
 │         └──ws
 │               Calculator.java
 │               SimpleCalculator.java
 └──WebContent
      └──WEB-INF
          │    sun-jaxws.xml
          │    web.xml
          └──lib
```

Sample ant build.xml file:

```
<project name="calcWS" default="war" basedir=".">
  <description>Web Services build file</description>
  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist"  location="dist"/>
  <property name="webcontent"  location="WebContent"/>

  <target name="compile" description="compile the source " >
   <mkdir dir="${build}"/>
   <!-- Compile the java code from ${src} into ${build} -->
   <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="war" depends="compile" description="generate war" >
    <!-- Create the war distribution directory -->
    <mkdir dir="${dist}/war"/>
    <!-- Follow standard WAR structure -->
    <copy todir="${dist}/war/build/"><fileset dir="${webcontent}"/></copy>
    <copy todir="${dist}/war/build/WEB-INF/classes/"><fileset dir="${build}"/></copy>

    <jar jarfile="${dist}/war/calcWS.war" basedir="${dist}/war/build/"/>
  </target>

</project>
```

Using Ant to build war file

- Open a terminal, go the directory calcWS and use the following command to build the WAR file:
  - ant
- Make sure that the ant application is in our PATH environment variable. If everything goes fine, the following message appears:

```
Buildfile: E:\ajp\soap\calcWS\build.xml

compile:
    [mkdir] Created dir: E:\ajp\soap\calcWS\build
    [javac] Compiling 2 source files to
E:\ajp\soap\calcWS\build
...
```