

Homework #2

Programming Web Services

ID2208

SOAP & WSDL

Hooman Peiro Sajjad (shps@kth.se)
KTH – ICT School
VT 2015

This presentation is based on a tutorial published by IBM and some Oracle documents pointed out in the reference.

Aims

To learn the followings:

- Design and Developing Web Services
- Developing Web Service Client
- SOAP processing

Homework #2- Part 1

Developing Web Service and Web
Service Client



In **THEORY** you could use any application server, any WS library with any IDE for developing Web Service technologies ,

But in **PRACTICE** you might face some strange problems (due to incompatibility of versions, non standard libraries,....)

So **BE READY** to face some headaches, just take your time and look for possible solutions. We have tested the recommended tools and libraries ,and hopefully you will get less headaches.

JAX-WS

- JAX-WS is designed to simplify building Web Services in Java (using Annotations, etc.)
- Part of Java SE 6-7 and Java EE 5-7 platforms
- It follows annotation based programming model
- It also includes :
SAAJ : provides a standard way to deal with SOAP messages with XML attachment

- JAX-WS download:

<https://jax-ws.java.net/>

Current Version is JAX-WS 2.2.10

Libraries & Tools Installation -1

1. Download and Install any of JWSDP 2.X or latest release of JAX-WS RI 2.2.10 (reference implementation) from: <http://jax-ws.java.net> which comes with more samples !

2. You might need to use Application Server.

- GlassFish: <http://glassfish.java.net/>
- Apache Tomcat : <http://tomcat.apache.org/>

Alternatively you can just use a light Web server, which doesn't require any installation (see sample code)

Libraries & Tools Installation -2

3. You MAY want to use IDEs as a development environment for web services and take advantage of their features for easy development:

NetBeans: <http://www.netbeans.org/downloads/index.html>

If you choose “All” version it also comes with GlassFish and Tomcat.

Eclipse : <http://www.eclipse.org/downloads/>

-Eclipse WPT plug-in: <http://www.eclipse.org/webtools/>

Eclipse has plug-ins for GlassFish <https://glassfish.java.net/ide.html>

and used to have a plug-in for Apache Tomcat .

(IntelliJ and Jdeveloper have plugins for Glassfish)

Or even better way, use no IDE, just “vi” or “Notepad” and follow the sample in the next slide!

Two Ways to Develop Web Service

1- *Top-Down approach*: Start with a WSDL contract, and generate the service Java classes. It requires good understanding of WSDL and XSD (*WSDL to JAVA*)

2- *Bottom-Up approach*: Start with developing Java classes, and use annotations to generate both a WSDL file and Java interfaces. Good starting point If you are new to Web Service (*JAVA to WSDL*).

For the time being, we follow **Bottom-Up** approach.

Homework #2- JAX-WS

APIs available to the developer in the Java SE 6-7 platform
(from Oracle JWS DP documents).

<i>API</i>	<i>Package</i>
JAX-WS	javax.xml.ws
SAAJ	javax.xml.soap
Web-service Meta-data	javax.xml.jws

Steps for Creating the Web Service and Client

(Easy way) (1/2)

1- Download Netbeans IDE for Java EE from here:

<https://netbeans.org/downloads/>

2- Follow the tutorial for creating a web service here:

https://netbeans.org/kb/docs/websvc/jax-ws.html#Exercise_1

3- For creating a client:

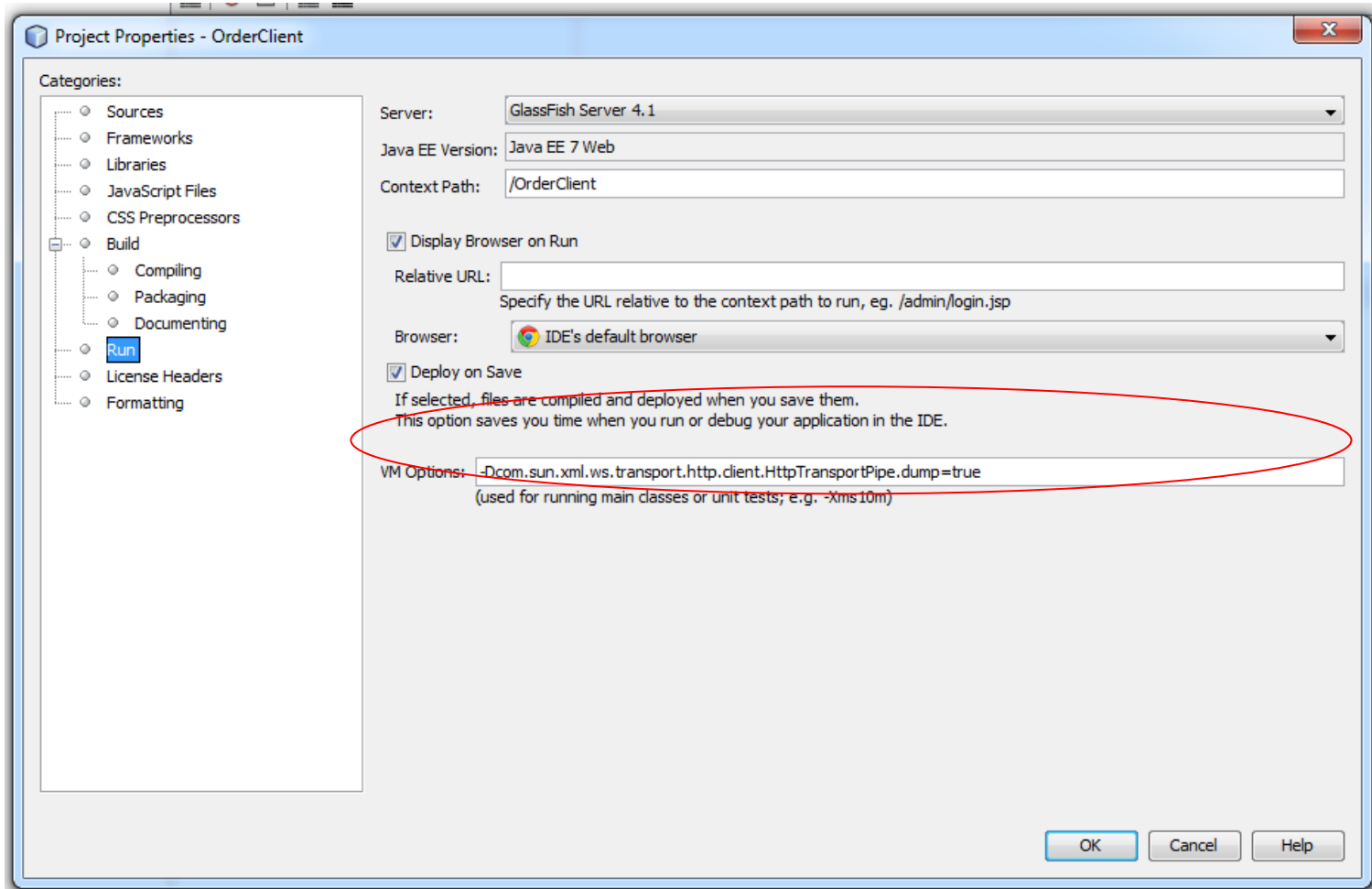
https://netbeans.org/kb/docs/websvc/jax-ws.html#Exercise_3_1

4- Compile the client class and Run the client.

Steps for Creating the Web Service and Client (Easy way) (2/2)

In order to make the Netbeans to display the SOAP messages, you need to set the following VM option, in the client project :

-Dcom.sun.xml.ws.transport.http.client.HttpTransportPipe.dump=true



Steps for Creating the Web Service and Client

- 1- Code the service implementation class.
- 2- Compile the class. Use *wsgen* to generate the artifacts required to deploy the service.
- 3- Package the files into a WAR file. Deploy the WAR file. The web service artifacts (which are used to communicate with clients) are generated by the Application Server during deployment.
- 4- Code the client class. Use *wsimport* to generate and compile the web service artifacts needed to connect to the service.
- 5- Compile the client class and Run the client.

Warming up Exercise – OrderProcessing Service

- Follow tutorial provided by IBM:
 - <https://www6.software.ibm.com/developerworks/education/ws-jax/ws-jax-a4.pdf>
- Download the sample code referenced at the end of the document
- Do the tutorial before starting the Homework

OrderProcessing Service

Scenario:

An Order-Processing Web service that

accepts :

order request, shipping information, ordered items,

generates:

a confirmation number as a response.

Developing Server Side Web Service - 1

First write the service code:

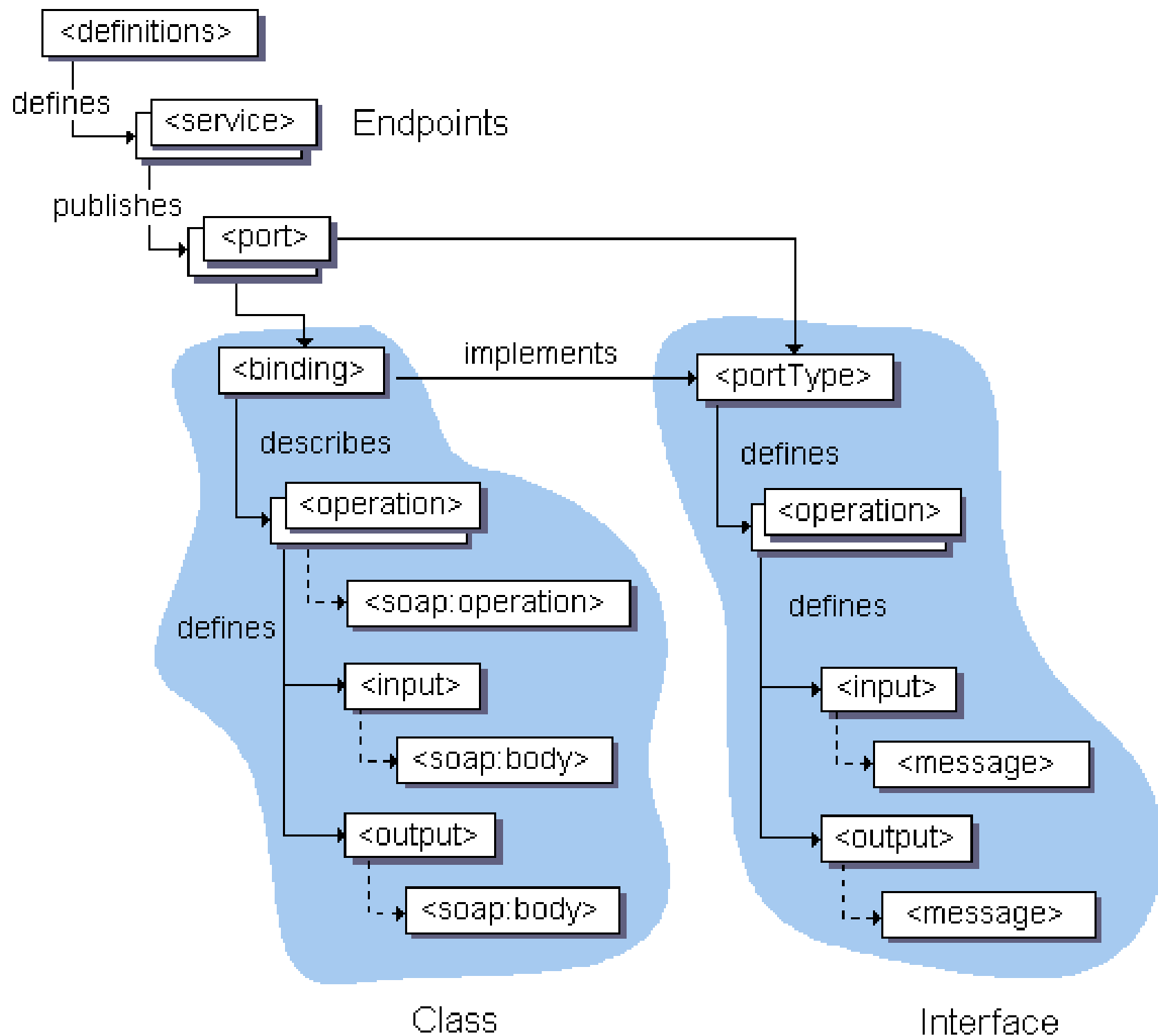
1- Write a service implementation in Plain Old Java Object.

2- Add JWS Annotation tags (JSR-181)

-Add One “@WebService” tag per class. It gives information like TargetNamespace, Port Name and Service Name

-**Optionally** add “@WebMethod” tag per each public methods to make them Web Service Operations or don't tag any of them.

-**Optionally**, add “@SOAPBinding” tag per class. It specifies the bindings to generate the WSDL file



@WebService annotation

Class	Name	Description
String	<u>endpointInterface</u>	The complete name of the service endpoint interface (SEI) defining the service's abstract web service contract.
String	name	The web service's name.
String	<u>portName</u>	The web service's port name.
String	<u>serviceName</u>	The web service's service name.
String	<u>targetNamespace</u>	If the <u>@WebService.targetNamespace</u> annotation is on an SEI, the <u>targetNamespace</u> is used for the <u>namespace</u> for the <u>wsdl:portType</u> and associated XML elements.
String	<u>wsdlLocation</u>	The location of a predefined WSDL file describing the service.

(from Oracle documents)

Useful hints on JAX-WS annotation:

http://docs.oracle.com/cd/E13222_01/wls/docs103/webserv_ref/annotations.html

Order Processing Web Service Implementation

Plain Java class

```
public class OrderProcessService
{
    public OrderBean processOrder(OrderBean orderBean) {
        // Do processing...
        System.out.println("processOrder called for customer"+
            orderBean.getCustomer().getCustomerId());
        // e.g print Items ordered are
        if (orderBean.getOrderItems() != null) {
            System.out.println("Number of items is " +
                orderBean.getOrderItems().length);
        }
        //Process Order.
        ...
        ...
        //Set the Order Id.
        orderBean.setOrderId("A2234");
        return orderBean;
    }
}
```

OrderBean class

```
public class OrderBean {  
  
    private Customer  customer;  
    private Address  shippingAddress;  
    private OrderItem[]  orderItems;  
    private String  orderId;  
  
    public Customer getCustomer() {  
        return customer;  
    }  
    public void setCustomer(Customer customer) {  
        this.customer = customer;  
    }  
    public String getOrderId() {  
        return orderId;  
    }  
    public void setOrderId(String orderId) {  
        this.orderId = orderId;  
    }  
    public Address getShippingAddress() {  
        return shippingAddress;  
    }  
}
```

Order Processing Service – Add Annotation

```
@WebService(serviceName = "OrderProcess",  
    portName = "OrderProcessPort",  
    targetNamespace = "http://jawxs.ibm.tutorial/jaxws/orderprocess")
```

```
@SOAPBinding(style = SOAPBinding.Style.DOCUMENT, use= SOAPBinding.Use.LITERAL,  
parameterStyle = SOAPBinding.ParameterStyle.WRAPPED)
```

```
public class OrderProcessService {
```

```
    @WebMethod
```

```
public OrderBean processOrder(OrderBean orderBean) {
```

```
    // Do processing...
```

```
        System.out.println("processOrder called for customer"+  
                        orderBean.getCustomer().getCustomerId());
```

```
    // Items ordered are
```

```
    if (orderBean.getOrderItems() != null) {
```

```
        System.out.println("Number of items is " + orderBean.getOrderItems().length);
```

```
    }
```

```
    //Process Order.
```

```
    //Set the Order Id.
```

```
    orderBean.setOrderId("A2234");
```

```
    return orderBean;}
```

Publishing the Web Service

Publish the Service:

1- Generate JAX-WS artifacts using “**wsgen**” tool.

```
wsgen -cp . com.ibm.jaxws.tutorial.service OrderProcessService -wsdl
```

Generated Artifacts: The WSDL and Schema files

2- Publish the service on a light weight web server

```
Endpoint.publish("http://localhost:8080/OrderProcessWeb/orderprocess",new OrderProcessService());
```

You should be able to see the WSDL at

<http://localhost:8080/OrderProcessWeb/orderprocess?wsdl>

Developing Web Service Client - 1

Create web service client (stubs) from WSDL using "wsimport".

`app_server_root/bin/wsimport.sh [options] -wsdlLocation URI`

- * Use the `-verbose` option to see a list of generated files when you run the command.
- * Use the `-keep` option to keep generated Java files.
- * Use the `-wsdlLocation` option to specify the location of the WSDL file.

```
wsimport -keep -p com.ibm.jaxws.tutorial.service.client  
http://localhost:8080/OrderProcessWeb/orderprocess?wsdl
```

The generated artifacts will be placed at:

`com.ibm.jaxws.tutorial.service.client`

You don't need to change anything in those generated artifacts (classes). Just use them!

Developing Web Service Client - 2

- Artifacts ("wsimport" generated files) :
 - Service Endpoint Interface: *it contains the interface which your service implements* (OrderProcessService)
 - Service class (OrderProcess): the service stub class which the client uses to make requests to service.
 - JAXB - generated classes (Address, Customer, OrderBean, ...)
 - Exception classes mapped from WSDL (we don't have any)
 - Request and Response "Wrapper" classes (ProcessOrder, ProcessOrderResponse)

OrderProcessService (EndPoint Interface)

```
@WebService(name = "OrderProcessService", targetNamespace =  
"http://jawxs.ibm.tutorial/jaxws/orderprocess")  
@XmlSeeAlso({ ObjectFactory.class})
```

```
public interface OrderProcessService {
```

```
    /** @param arg0
```

```
    * @return returns com.ibm.jaxws.tutorial.service.client.OrderBean */
```

```
    @WebMethod
```

```
    @WebResult (targetNamespace = "")
```


```
    @RequestWrapper(localName = "processOrder", targetNamespace =  
"http://jawxs.ibm.tutorial/jaxws/orderprocess", className =  
"com.ibm.jaxws.tutorial.service.client.ProcessOrder") <----- wrapper
```

```
    @ResponseWrapper(localName = "processOrderResponse", targetNamespace =  
"http://jawxs.ibm.tutorial/jaxws/orderprocess", className =  
"com.ibm.jaxws.tutorial.service.client.ProcessOrderResponse") <----- wrapper
```

```
    public OrderBean processOrder( @WebParam(name = "arg0", targetNamespace = "")  
        OrderBean arg0);
```

```
}
```


OrderProcess (stub)

```
@WebServiceClient(name = "OrderProcess", targetNamespace =  
"http://jawxs.ibm.tutorial/jaxws/orderprocess", wsdlLocation =  
http://localhost:8080/OrderProcessWeb/orderprocess?wsdl)  
public class OrderProcess extends Service {  
    private final static URL ORDERPROCESS_WSDL_LOCATION;  
    static {  
        URL url = new URL("http://localhost:8080/OrderProcessWeb/orderprocess?wsdl");  
        ORDERPROCESS_WSDL_LOCATION = url;  
    }  
    public OrderProcess(URL wsdlLocation, QName serviceName) {  
        super(wsdlLocation, serviceName); }  
  
    public OrderProcess() {  
        super(ORDERPROCESS_WSDL_LOCATION, new  
QName("http://jawxs.ibm.tutorial/jaxws/orderprocess", "OrderProcess")); }  
    /* returns OrderProcessService*/  
  
    @WebEndpoint(name = "OrderProcessPort")  
    public OrderProcessService getOrderProcessPort() {   
        return (OrderProcessService)super.getPort(new  
QName("http://jawxs.ibm.tutorial/jaxws/orderprocess", "OrderProcessPort"),  
OrderProcessService.class);  
    }  
}
```

Service Client Code

```
.....  
// WSDL url of the OrderProcess web services  
URL url = getWSDLURL("  
http://localhost:8080/OrderProcessWeb/orderprocess?wsdl");  
    // Qualified name of the service  
QName qName = new QName(  
    "http://jawxs.ibm.tutorial/jaxws/orderprocess", "OrderProcess");  
// Call the stub class  
OrderProcess orderProcessingService = new OrderProcess(url, qName);  
  
    // initialize the request object  
OrderBean order = populateOrder();  
    // get the proxy to the service. The proxy implements the service interface defined  
    // by the service  
OrderProcessService port = orderProcessingService.getOrderProcessPort( );  
// invoke the proxy by passing the OrderBean instance  
// and get the response in form of OrderResponse object  
OrderBean orderResponse = port.processOrder( order );  
  
System.out.println("Order id is " + orderResponse.getOrderid());  
.....
```

Run Service Client

Run client class:

```
java com.ibm.jaxws.tutorial.service.client.OrderClient
```

```
http://localhost:8080/OrderProcessWeb/orderprocess?wsdl
```

Output:

Order id is A1234

References:

Design and Develop JAX-WS 2.0 web services, IBM corporation

<https://www6.software.ibm.com/developerworks/education/ws-jax/ws-jax-a4.pdf>

From WSDL to Java?

1. Generate a service endpoint interface (using [wsimport](#))
2. Implement the service endpoint interface.
3. Create a WAR file (deployment descriptor web.xml , sun-jaxws.xml , WSDL file and Service Impl. Class) to deploy.
4. Publish the End-point
5. Develop Client side as before

Look at “[fromwsdl](#)” example in folder [.../jaxws/samples/](#)

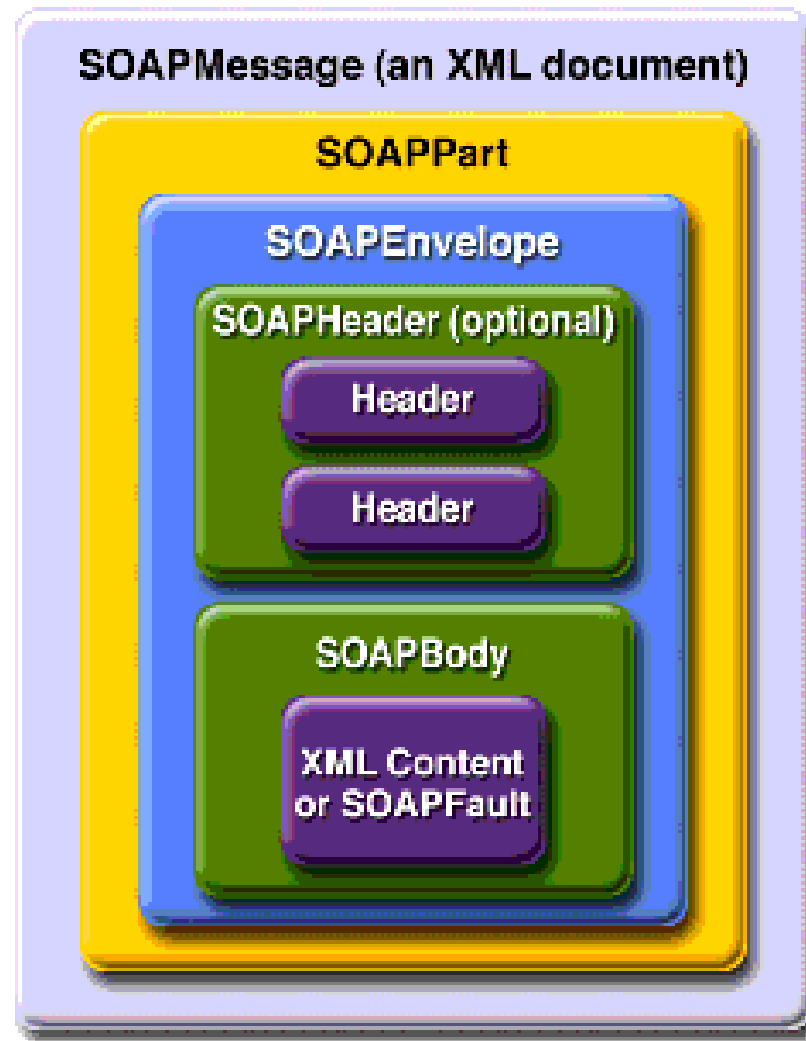
Homework #2 – Part 2

SOAP

SAAJ (SOAP Attachment API for Java)

- SAAJ API is used to create, populate and access **SOAP** messages applications directly , read and write SOAP-based XML messages.
- SAAJ is used in SOAP Message handlers for reading and modifying SOAP headers
- This tutorial and most of the related materials are based on SAAJ section in Oracle WSDP tutorial
- http://docs.oracle.com/cd/E17802_01/webservices/webservices/docs/2.0/tutorial/doc/JavaWSTutorial.pdf (For JWSDP version 2.0)

SOAP Message with no Attachment



Building SOAP Message

- 1- Create (Get) SOAP message Part, Body and Header
- 2- Add element to Header (optional)
- 3- Add elements (e.g. request, response) to Body
- 4- Add content (e.g. an XML document) to Body
- 5- Add Attachments (optional) (not covered in this homework)
- 6- Add Attributes (optional)

Steps to Create a SOAP Message -1

//get insance of SOAP factory

```
MessageFactory factory = MessageFactory.newInstance();
```

// get empty soap message

```
SOAPMessage message = factory.createMessage();
```

// get Part piece of the message

```
SOAPPart soapPart = message.getSOAPPart();
```

// get Envelop piece of the message

```
SOAPEnvelope envelope = soapPart.getEnvelope();
```

// access the empty header of the message

```
SOAPHeader header = envelope.getHeader();
```

// access the empty body of the message

```
SOAPBody body = envelope.getBody();
```

Steps to Create a SOAP message -2

What we have built so far:

```
<SOAP-ENV:Envelope xmlns:SOAP-  
ENV="http://schemas.xmlsoap.org/soap/envelope/">
```

```
  <SOAP-ENV:Header/>
```

```
  <SOAP-ENV:Body/>
```

```
</SOAP-ENV:Envelope>
```

Steps to Create a SOAP message - 3

Adding content to Body :

```
SOAPBody body = message.getSOAPBody();  
QName bodyName = new QName  
("http://wombat.ztrade.com", "GetLastTradePrice", "m");  
  
SOAPBodyElement bodyElement=  
body.addBodyElement(bodyName);  
  
QName name = new QName("symbol");  
  
SOAPElement symbol= bodyElement.addChildElement(name);  
  
symbol.addTextNode("SUNW");
```

Steps to Create a SOAP message - 3

We will have :

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/
  envelope/">
<SOAP-ENV:Body>
  <m:GetLastTradePrice
    xmlns:m="http://wombat.ztrade.com">
    <symbol>SUNW</symbol>
  </m:GetLastTradePrice>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Send a SOAP Message

Get a SOAPConnection Object :

```
SOAPConnectionFactory soapConnectionFactory =  
SOAPConnectionFactory.newInstance();
```

```
SOAPConnection connection =  
soapConnectionFactory.createConnection();
```

Send SOAP Message :

```
// Create an endpoint which is a URL to the published web service  
java.net.URL endpoint = new URL("http://wombat.ztrade.com/quotes");
```

```
// Send the SOAP Message request and then wait for SOAP Message  
response
```

```
SOAPMessage response = connection.call(message, endpoint);
```

Read SOAP Message

```
SOAPBody soapBody = response.getSOAPBody();
```

```
Iterator iterator = soapBody.getChildElements(bodyName);
```

```
SOAPBodyElement bodyElement =  
    (SOAPBodyElement)iterator.next();
```

```
String lastPrice =  
    bodyElement.getValue();
```

```
System.out.print("The last price for SUNW is ");  
System.out.println(lastPrice);
```

```
<SOAP-ENV:Body>  
    <elementName> 11.11 </elementName>  
    . . . .  
</SOAP-ENV:Body>
```

Adding Content to Empty SOAP Body-2

Target XML fragment of body content :

```
<PO:PurchaseLineItems
  xmlns:PO="http://sonata.fruitsgalore.com">
  <Order>
    <Product> Apple </Product>
    <Price>1.56</Price>
  </Order>
  <Order>
    <Product>Peach</Product>
    <Price>1.48</Price>
  </Order>
</PO:PurchaseLineItems>
```

Adding Content to Empty SOAP Body-1

```
SOAPBody body = message.getSOAPBody();
QName bodyName = new QName("http://sonata.fruitsgalore.com",
    "PurchaseLineItems", "PO");

SOAPBodyElement purchaseLineItems = body.addBodyElement(bodyName);
QName childName = new QName("Order");
SOAPElement order = purchaseLineItems.addChildElement(childName);
childName = new QName("Product");

SOAPElement product = order.addChildElement(childName);
product.addTextNode("Apple");
childName = new QName("Price");
SOAPElement price = order.addChildElement(childName);
price.addTextNode("1.56");
childName = new QName("Order");
SOAPElement order2 = purchaseLineItems.addChildElement(childName);
childName = new QName("Product");
SOAPElement product2 = order2.addChildElement(childName);

product2.addTextNode("Peach");
childName = soapFactory.new QName("Price");
SOAPElement price2 = order2.addChildElement(childName);
price2.addTextNode("1.48");
```


Adding Content to Header

```
SOAPHeader header = message.getSOAPHeader();
```

```
QName headerName = new QName("http://ws-  
org/schemas/conformanceClaim/", "Claim", "wsi");
```

```
SOAPHeaderElement headerElement =  
header.addHeaderElement(headerName);
```

```
headerElement.addAttribute(new  
QName("conformsTo"), "http://ws-i.org/profiles/basic/1.1/");
```

We will have:

```
<SOAP-ENV:Header>  
  <wsi:Claim xmlns:wsi="http://ws-i.org/schemas/conformanceClaim/"  
    conformsTo="http://ws-i.org/profiles/basic/1.1/">  
</SOAP-ENV:Header>
```

Adding XML document to SOAP Body as a DOM object

```
DocumentBuilderFactory dfactory = DocumentBuilderFactory.newInstance();  
  
dfactory.setNamespaceAware(true);  
  
dfactory.setValidating(true);  
  
DocumentBuilder builder = dfactory.newDocumentBuilder();  
  
Document document = builder.parse("CV.xml");  
  
SOAPBodyElement docElement = body.addDocument(document);
```

soapUI

- soapUI is a free and open source application for monitoring, invoking, developing, simulating and functional testing of web services over HTTP.
- More on soapUI on <http://www.soapui.org> .
- It provides plug-ins for both NetBean and Eclipse
- SoapUI in NetBeans:
 - <http://www.soapui.org/IDE-Plugins/installation.html>
- SoapUI in Eclipse :
 - <http://www.mastertheboss.com/jboss-web-services/soapui-tutorial-for-eclipse>

soapUI (in netbeans)

The screenshot displays the NetBeans IDE 6.0 interface with the soapUI plugin. The main workspace is divided into several panes:

- Projects:** Shows a project named "MyWebServiceTestingProject" with a tree view containing various test cases like "BrowseNodeLookup", "CartAdd", "Request 1", "CartClear", "CartCreate", "CartGet", "CartModify", "CustomerContentLookup", "CustomerContentSearch", "Help", "ItemLookup", "ItemSearch", and "Request 1".
- Request 1 - Properties:** A table showing properties for the selected request:

Request 1	
Name	Request 1
Description	
Message Size	6887
Encoding	UTF-8
Endpoint	http://soap.amazon.com/o...
Bind Address	
- Request 1:** The main pane showing the raw XML of the SOAP request and response. The request is a SOAP envelope with a header and a body containing an `ItemSearch` operation. The response is a SOAP envelope with a header and a body containing an `ItemSearchResponse` operation, which includes a `RequestId` and an `Errors` section with a `Code` of "AWS.InvalidParameterValue" and a `Message` indicating a value is not valid.
- soapUI Log:** A log window at the bottom showing debug messages from the SOAP service, including timestamps and XML snippets.

The status bar at the bottom indicates the response time: 645ms (698 bytes).

You want more samples on JAX-WS?

Look at the “samples” directory in the “JAX-WS” installation directory.

You just need to install “ant” <http://ant.apache.org/manual/install.html>
and set “**AS_HOME**” to point to the Application Server installation directory.

Then run the Application server , next build the Server and Client side components.

Homework #2 – Part 3

Tasks and Deliverables

Tasks (1)

Design and implement flight ticket reservation services with the following services:

1) Authorization of customers. Only the user with the correct user-name and password can have access to other services. This service should accept customer's Id and Password, check them with the list of registered customers and if found then give a reply that allows system access.

Tasks (2)

2) Checking possible itinerary for flights given a departure city and a destination city. In case there is no direct flight from the departure to the destination, the service should find a route that combine several flights.

3) Checking availability of tickets and finding their price for a given itinerary and given date.

Tasks (3)

(4) Output the price of available itineraries .

5) Booking tickets for requested itinerary. Credit card number is required to book the tickets.

6) Issue tickets. The ticket has to be booked before issuing.

Requirements:

a - You have to implement half of the services above, in top-down fashion (writing WSDL file first and then generating source codes) and the rest in the bottom-up fashion (starting from Java code and then generating WSDL files).

Tasks (3)

b- **You SHOULD** develop a test client for each Web Service in order to show that each service works independently.

c- You have to extend generated SOAP messages with **headers** related to the above task. **You do not have to implement it.** You can just present them in your documentation and explain how you wrote the headers for your chosen scenario.

Deliverables

- 1- Textual report explaining what you did
- 2- The Source code, WSDLs and Schema of the implemented Web services.
- 3- The XML of constructed, sent and received SOAP messages communicated among services.
- 4- A short and precise description showing your architecture and deployment of your web services.
- 5- Executable version of your system
- 6- Show your fully functional system in a 10-15 minutes presentation.

HW #2 - Delivery

- Send your deliverables by e-mail to **BOTH**:

misha@kth.se and shps@kth.se

*e-mail subject: **PWS15-HW2***

Please add your names in the body of the email

Attach: source code + instructions how to run your code,

- Deadline : 09 **Feb. 2015, 11:59 PM CET**

Presentation: **To be decided**

GOOD LUCK!

Question?

