

JAX-WS

Java API for XML Web Services (JAX-WS), is a standardized API for creating web services in XML format (SOAP). JAX-WS provides support for SOAP (Simple Object Access protocol) based web services.

Web services have been around a while now. First there was SOAP. But SOAP only described what the messages looked like. Then there was WSDL. But WSDL didn't tell you how to write web services in Java. Then along came JAX-RPC 1.0. After a few months of use, the Java Community Process (JCP) folks who wrote that specification realized that it needed a few tweaks, so released JAX-RPC 1.1. After a year or so of using that specification, the JCP folks wanted to build a better version: JAX-RPC 2.0. A primary goal was to align with industry direction, but the industry was not merely doing RPC web services, they were also doing message-oriented web services. So "RPC" was removed from the name and replaced with "WS" (which stands for web Services). Thus the successor to JAX-RPC 1.1 is JAX-WS 2.0 - the Java API for XML-based web services.

What remains the same in JAX-RPC and JAX-WS?

Before we itemize the differences between JAX-RPC 1.1 and JAX-WS 2.0, we should first discuss similarities between JAX-RPC and JAX-WS

- JAX-WS still supports SOAP 1.1 over HTTP 1.1, so interoperability will not be affected. The same messages can still flow across the wire.
- JAX-WS still supports WSDL 1.1, so what you've learned about that specification is still useful. A WSDL 2.0 specification is nearing completion, but it was still in the works at the time that JAX-WS 2.0 was finalized.

What is different between JAX-RPC and JAX-WS?

SOAP 1.2: JAX-RPC and JAX-WS support SOAP 1.1. JAX-WS also supports SOAP 1.2.

XML/HTTP: The WSDL 1.1 specification defined an HTTP binding, which is a means by which you can send XML messages over HTTP without SOAP. JAX-RPC ignored the HTTP binding. JAX-WS adds support for it.

WS-I's Basic Profiles: JAX-RPC supports WS-I's Basic Profile (BP) version 1.0. JAX-WS supports BP 1.1. (WS-I is the web services interoperability organization.)

New Java features: JAX-RPC maps to Java 1.4. JAX-WS maps to Java 5.0. JAX-WS relies on many of the features new in Java 5.0.

Java EE 5, the successor to J2EE 1.4, adds support for JAX-WS, but it also retains support for JAX-RPC, which could be confusing to today's web services novices.

The data mapping model:

- JAX-RPC has its own data mapping model, which covers about 90 percent of all schema types. Those that it does not cover are mapped to javax.xml.soap.SOAPElement.
- JAX-WS's data mapping model is JAXB. JAXB promises mappings for all XML schemas.

The interface mapping model:

- JAX-WS's basic interface mapping model is not extensively different from JAX-RPC's; however:
- JAX-WS's model makes use of new Java 5.0 features.
- JAX-WS's model introduces asynchronous functionality.

The dynamic programming model:

JAX-WS's dynamic client model is quite different from JAX-RPC's. Many of the changes acknowledge industry needs:

- It introduces message-oriented functionality.
- It introduces dynamic asynchronous functionality.
- JAX-WS also adds a dynamic server model, which JAX-RPC does not have.
- MTOM (Message Transmission Optimization Mechanism)
- JAX-WS, via JAXB, adds support for MTOM, the new attachment specification. Microsoft never bought into the SOAP with Attachments specification; but it appears that everyone supports MTOM, so attachment interoperability should become a reality.

The handler model:

The handler model has changed quite a bit from JAX-RPC to JAX-WS.

- JAX-RPC handlers rely on SAAJ 1.2. JAX-WS handlers rely on the new SAAJ 1.3 specification.

SOAP 1.2

There is really not a lot of difference, from a programming model point of view, between SOAP 1.1 And SOAP 1.2. As a Java programmer, the only place you will encounter these differences is When using the handlers.

XML/HTTP

Like the changes for SOAP 1.2, there is really not a lot of difference, from a programming model Point of view, between SOAP/HTTP and XML/HTTP messages. As a Java programmer, the only Place you will encounter these differences is when using the handlers. The HTTP binding has its own handler chain and its own set of message context Properties.

WS-I's basic profiles

JAX-RPC 1.1 supports WS-I's Basic Profile (BP) 1.0. Since that time, the WS-I folks have Developed BP 1.1 (and the associated AP 1.0 and SSBP 1.0). These new profiles clarify some Minor points, and more clearly define attachments. JAX-WS 2.0 supports these newer profiles. For the most part, the differences between them do not affect the Java programming model. The Exception is attachments. WS-I not only cleared up some questions about attachments, but they Also defined their own XML attachment type: wsi:swaRef. Many people are confused by all these profiles. You will need a little history to clear up the confusion.

WS-I's first basic profile (BP 1.0) did a good job of clarifying the various specs. But it wasn't perfect. And support for SOAP with Attachments (Sw/A) in particular was still rather fuzzy. In their second iteration, the WS-I folks pulled attachments out of the basic profile - BP 1.1 - and fixed some of the things they missed the first time around. At that point they also added two mutually exclusive supplements to the basic profile: AP 1.0 and SSBP 1.0. AP 1.0 is the Attachment Profile which describes how to use Sw/A. SSBP 1.0 is the Simple SOAP Binding Profile, which describes a web services engine that does not support Sw/A (such as Microsoft's .NET). The remaining profiles that WS-I has and is working on build on top of those basic profiles.

Summary

JAX-WS 2.0 is the successor to JAX-RPC 1.1. There are some things that haven't changed, but most of the programming model is different to a greater or lesser degree. The topics introduced in this tip will be expanded upon in a series of tips which we will publish over the coming months that will compare, in detail, JAX-WS and JAX-RPC. At a high level though, here are a few reasons why you would or would not want to move to JAX-WS from JAX-RPC.

Reasons to stay with JAX-RPC 1.1:

- If we want to stay with something that's been around a while, JAX-RPC will continue to be supported for some time to come.
- If we don't want to step up to Java 5.
- If we want to send SOAP encoded messages or create RPC/encoded style WSDL.

Reasons to step up to JAX-WS 2.0:

- If we want to use the new message-oriented APIs.
- If we want to use MTOM to send attachment data.
- If we want better support for XML schema through JAXB.
- If we want to use an asynchronous programming model in your web service clients.
- If we need to have clients or services that can handle SOAP 1.2 messages.
- If we want to eliminate the need for SOAP in your web services and just use the XML/HTTP binding.

SOAP

SOAP is an XML specification for sending messages over a network. SOAP messages are independent of any operating system and can use a variety of communication protocols including HTTP and SMTP.

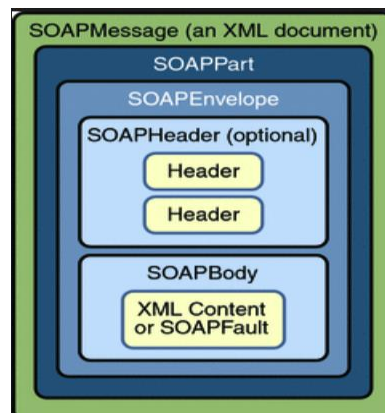
SOAP is XML heavy, hence best used with tools/frameworks. JAX-WS is a framework that simplifies using SOAP. It is part of standard Java.

SOAP message contains the following three parts:

Envelope: This Envelope element contains an optional Header element and a mandatory Body element.

Header: SOAP message contains the information which explains how the message is to be processed.

Body: Body part of the SOAP message contains the actual payload of the end to end message transmission.



Sample SOAP Request

```
01 <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" >
02   <soapenv:Header/>
03   <soapenv:Body>
04     <addPerson>
05       <person>
06         <id>0</id>
07         <name>This Person</name>
08       </person>
09     </addPerson>
10   </soapenv:Body>
11 </soapenv:Envelope>
```

Java-XML Data Binding

SOAP-based web services use XML to exchange request and response messages. This requires an architecture for converting Java objects to XML and the reverse. JAXB (Java Architecture for XML Binding) was developed for this purpose.

JAX-RPC uses its own data mapping model. This is because the JAXB specification had not been finalized when the first version of JAX-RPC was completed. The JAX-RPC data mapping model lacks support for some XML schemas.

JAX-WS uses JAXB for data binding. JAXB provides mapping for virtually all schemas.

We can use JAXB annotations on your Java bean and JAX-WS will convert it and its properties to XML elements at runtime when sending the SOAP message.

```
import javax.xml.bind.annotation.*;

@XmlRootElement(name = "person")
@XmlType(propOrder = {"id", "name"})
public class Person {

    @XmlElement(name = "id", required = true)
    int id;
    @XmlElement(name = "name", required = true)
    String name;
    // accessors and mutators
}
```

Web Services Definition Language (WSDL)

The Web Services Description Language is an XML-based interface definition language that is used for describing the functionality offered by a web service. The acronym is also used for any specific WSDL description of a web service (also referred to as a WSDL file), which provides a machine-readable description of how the service can be called, what parameters it expects, and what data structures it returns. Therefore, its purpose is roughly similar to that of a method signature in a programming language.

WSDL 1.0 (Sept. 2000) was developed by IBM, Microsoft, and Ariba to describe Web Services for their SOAP toolkit. It was built by combining two service description languages: NASSL (Network Application Service Specification Language) from IBM and SDL (Service Description Language) from Microsoft.

WSDL 1.1, published in March 2001, is the formalization of WSDL 1.0. No major changes were introduced between 1.0 and 1.1.

WSDL 1.2 (June 2003) was a working draft at W3C, but has become WSDL 2.0. According to W3C: WSDL 1.2 is easier and more flexible for developers than the previous version. WSDL 1.2 attempts

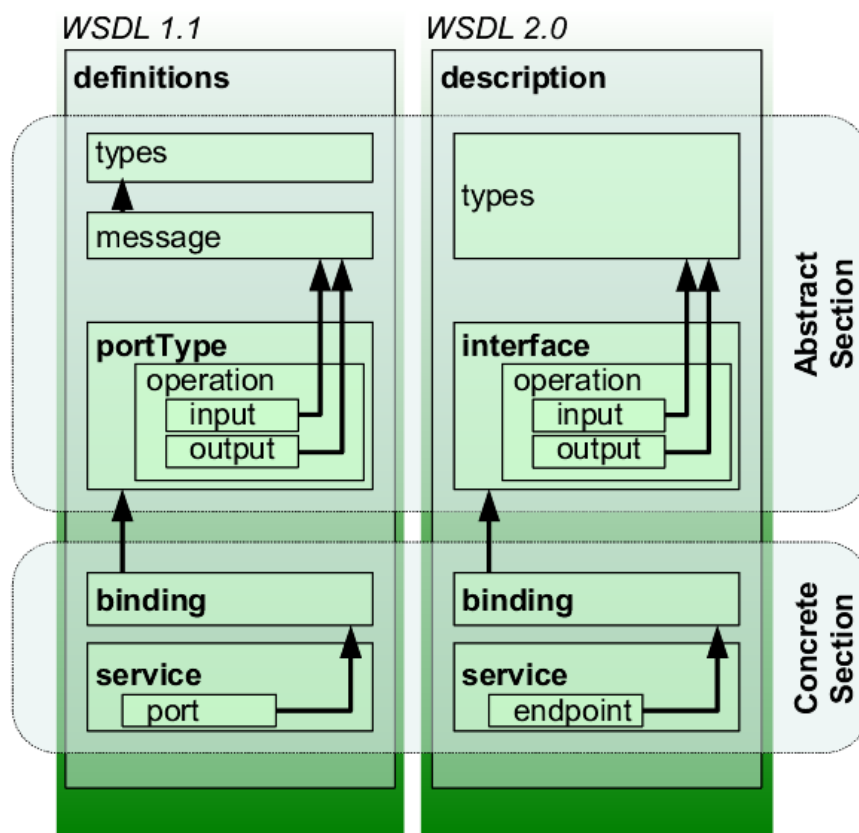
to remove non-interoperable features and also defines the HTTP 1.1 binding better. WSDL 1.2 was not supported by most SOAP servers/vendors.

WSDL 2.0 became a W3C recommendation on June 2007. WSDL 1.2 was renamed to WSDL 2.0 because it has substantial differences from WSDL 1.1. The changes are the following:

Added further semantics to the description language

- Removed message constructs
- Operator overloading not supported
- PortTypes renamed to interfaces
- Ports renamed to endpoints

The current version of WSDL is WSDL 2.0.



Definition: The definition element must be the root element of the WSDL specification.

This element defines the name of the web service, information about the multiple namespaces used throughout the WSDL document.

Types: This element is very important while sharing the data across multiple platforms using web services. The data type of the variables we are using in the web services should be compatible with all platforms. The types element is used to define all the data types used between the end-to-end transmissions. XSD or XML Schema Definition is used to define the types used in the WSDL. The detailed overview of XSD will explain in the next section.

Message: Each message describes either the input or output of the web service transmission and it has the name of the message and zero or more part message elements. Input message contains the request information and the corresponding part message details. The output message element contains the details about the response from the server.

portType: PortType elements are an entity like a Java class containing group of operations.

This element describes a complete operation definition by combining input and output messages. One portType element can describe multiple operations.

Binding: The binding element describes how the service will be implemented in the transmission.

Service: Service element provides the address, information about the service. This element defines the URL of the service we are invoking.

Top-Down vs. Bottom-Up

There are two ways of building SOAP web services. We can go with a top-down approach or a bottom-up approach.

In a top-down (contract-first) approach, a WSDL document is created, and the necessary Java classes are generated from the WSDL. In a bottom-up (contract-last) approach, the Java classes are written, and the WSDL is generated from the WSDL.

Writing a WSDL file can be quite difficult depending on how complex your web service is. This makes the bottom-up approach an easier option. On the other hand, since your WSDL is generated from the Java classes, any change in code might cause a change in the WSDL. This is not the case for the top-down approach.

Features of JAX-WS

- Requests and responses are transmitted as SOAP messages (XML files) over HTTP.
- Even though SOAP structure is complex, developer need not bother about the complexity in messaging. The JAX-WS run time system manages all the parsing related stuffs.
- Since JAX-WS uses the W3C defined technologies, a SOAP web service client can access a SOAP web service that is running in a non-Java platform.
- A machine-readable WSDL (Web Service Description Language) file which describes all the operations about a service can be present in case of SOAP web services.
- Annotations can be used with JAX-WS to simplify the development.

Annotations used in JAX WS

Annotations can be used in Java classes in creating web services. The most commonly used annotations are listed below:

- **@WebService**
- **@WebMethod**
- **@WebParam**
- **@WebResult**
- **@SOAPBinding**
- **@OneWay**

As JAX-WS is an API we can't start the development directly with API. We need an implementation for this API.

JAX-WS implementations

There are several implementations for JAX-WS. Some of them are:

- Reference Implementation
- Apache Axis
- Apache cxf
- JBossWS
- Metro Project in Glassfish

Developing WebService using JAX-WS API with RI implementation

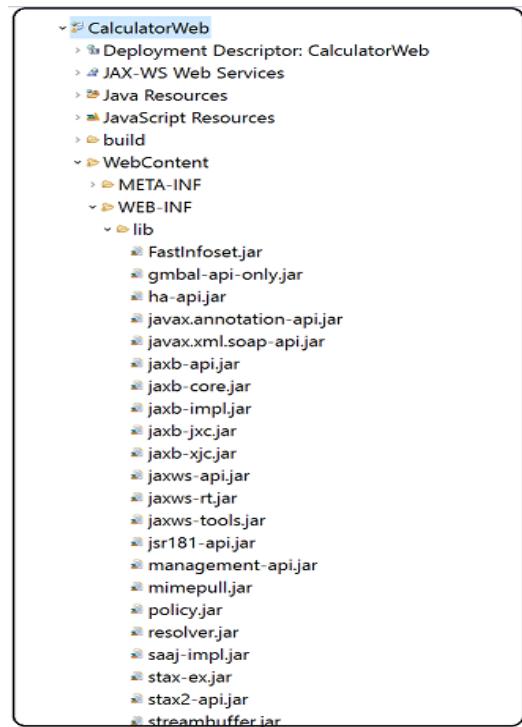
Deciding Technology Stack

Specification	: B.P 1.1
API	: JAX-WS
Implementation	: Reference Implementation (RI)
Role	: Provider
Approach	: Contract Last Approach
Endpoint	: Servlet Endpoint
MEP	: Synchronous request - reply
MEF	: Document Literal (By default)

Requirement: Develop an application to perform Air thematic Operations through WebService

Procedure of developing Provider

Step 1:- Create Dynamic web project in IDE and add jax-ws RI related jars in project lib folder



Step 2:- Create Service Endpoint Interface (This is optional in jax-ws api)

```
Calculator.java
1 package com.cal.service;
2
3 import javax.jws.WebMethod;
4 import javax.jws.WebService;
5
6 @WebService(name = "ICalculator")
7 public interface ICalculator {
8
9     @WebMethod(operationName = "Add")
10    public Integer add(Integer a, Integer b);
11
12    @WebMethod(operationName = "Subtract")
13    public Integer subtract(Integer a, Integer b);
14
15 }
```

Step3:- Create Service Implementation class

```

1 package com.cal.service;
2
3 import javax.xml.ws.WebService;
4
5 @WebService(endpointInterface = "com.cal.service.ICalculator")
6 public class CalculatorImpl implements ICalculator {
7
8     @Override
9     public Integer add(Integer a, Integer b) {
10         return a + b;
11     }
12
13     @Override
14     public Integer subtract(Integer a, Integer b) {
15         return a - b;
16     }
17
18 }
19

```

Step 4:- Create Web service deployment descriptor file in project WEB-INF folder (sun-jaxws.xml) with endpoint and url-pattern details

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">
3     <endpoint name="ICalculator"
4         implementation="com.cal.service.CalculatorImpl"
5         url-pattern="/calculator" />
6 </endpoints>

```

Step 5: Configure WSServletContextListener & WSServlet in web.xml

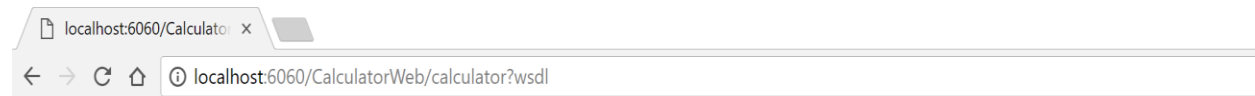
```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee"
3     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
4     id="WebApp_ID" version="3.0">
5 <listener>
6     <listener-class>
7         com.sun.xml.ws.transport.http.servlet.WSServletContextListener
8     </listener-class>
9 </listener>
10 <servlet>
11     <servlet-name>calculator</servlet-name>
12     <servlet-class> com.sun.xml.ws.transport.http.servlet.WSServlet </servlet-class>
13     <load-on-startup>1</load-on-startup>
14 </servlet>
15 <servlet-mapping>
16     <servlet-name>calculator</servlet-name>
17     <url-pattern>/calculator</url-pattern>
18 </servlet-mapping>
19 </web-app>

```

Step 6: Deploy the application and access using Endpoint URL**Web Services**

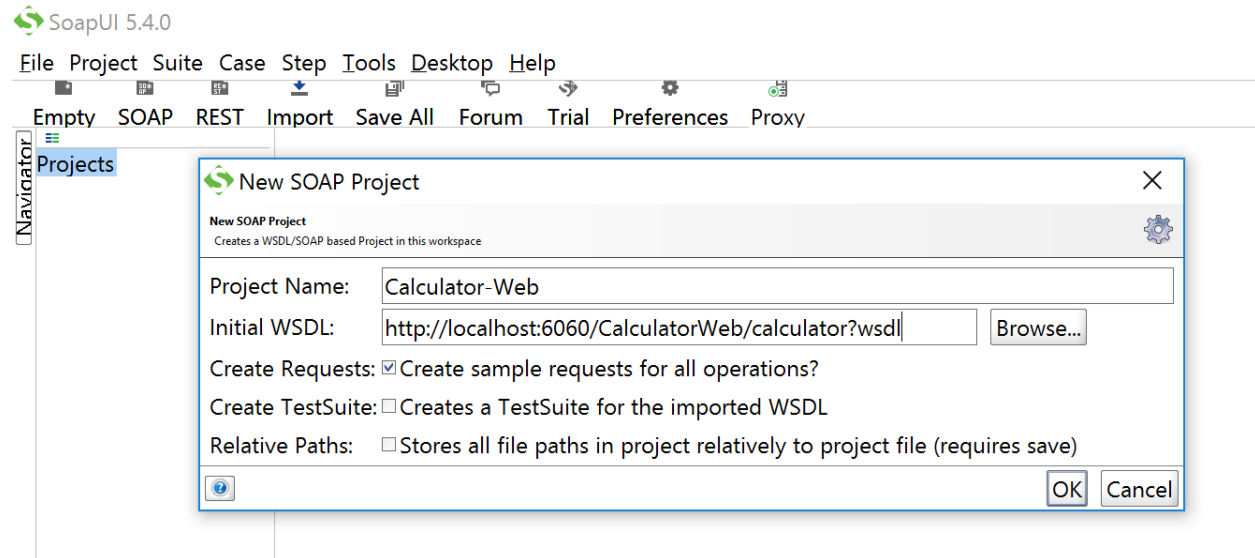
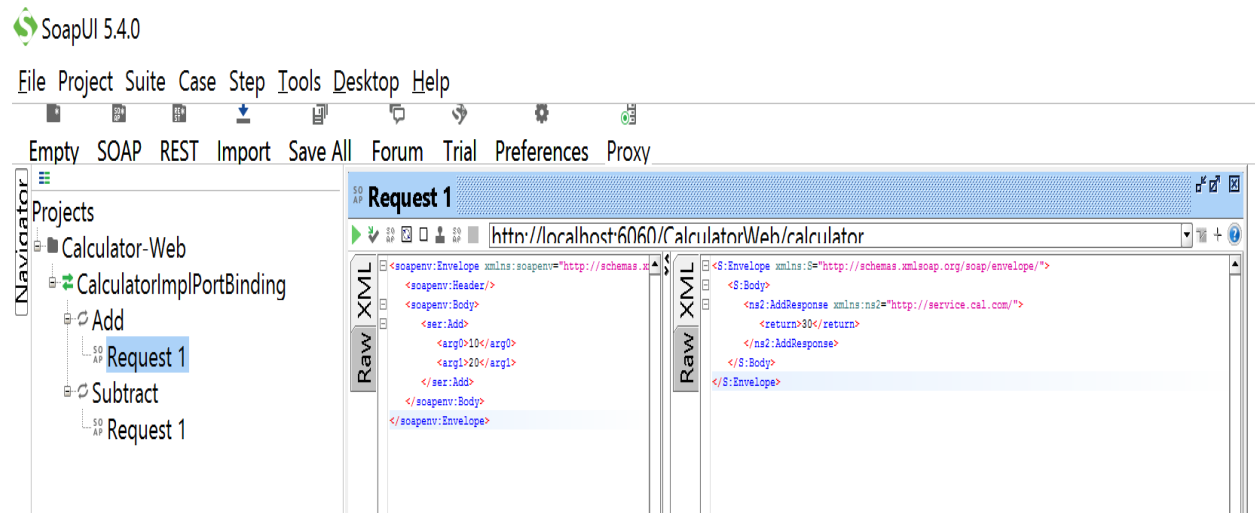
Endpoint	Information
Service Name: {http://service.cal.com/}CalculatorImplService Port Name: {http://service.cal.com/}CalculatorImplPort	Address: http://localhost:6060/CalculatorWeb/calculator WSDL: http://localhost:6060/CalculatorWeb/calculator?wsdl Implementation class: com.cal.service.CalculatorImpl

Access the WSDL using below URL

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<!--
  Published by JAX-WS RI (http://jax-ws.java.net). RI's version is JAX-WS RI 2.2.10 svn-revision#919b322c92f13ad085a933e8dd6dd35d4947364
-->
<!--
  Generated by JAX-WS RI (http://jax-ws.java.net). RI's version is JAX-WS RI 2.2.10 svn-revision#919b322c92f13ad085a933e8dd6dd35d4947364
-->
<definitions xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" xmlns:wsp="http://www.w3.org/
xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy" xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" xmlns:soap="http://
xmlns:tns="http://service.cal.com/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace
name="CalculatorImplService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://service.cal.com/" schemaLocation="http://localhost:6060/CalculatorWeb/calculator?xsd=1"/>
    </xsd:schema>
  </types>
```

Note: If we are able access the WSDL like above, that means Provider is up and running

Step 7: Test the Provider using SOAP UI tool**Open SOAP UI - > Click on File - > New SOAP Project -> Enter Project Name and WSDL URL****Step 8: Send SOAP request like below - > we can SOAP Response**

Developing Consumer

After developing the provider, in order to access provider, we need to build the Consumer.

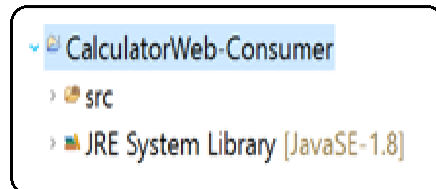
JAX-WS API supports both the developments of Provider as well as Consumer. JAX-WS API supports two types of consumer, they are

- Stub based consumer
- Dispatch API

From above two approaches, most of the projects use stub based consumer and following section describes how to build a stub based consumer for JAX-WS Provider.

Procedure for Developing STUB Based Consumer for JAX-WS Provider

Step 1: Open IDE and create a Java Project



Step 2: Generate classes required for consumer using wsimport tool by giving wsdl URL as input

Syntax: `wsimport -d src -keep -verbose WSDL_URL`

Note: wsimport tool will be shipped as part of JDK itself

```
Command Prompt
C:\Ashok\CalculatorWeb-Consumer>wsimport -d src -keep -verbose http://localhost:6060/CalculatorWeb/calculator?wsdl
parsing WSDL...

Generating code...

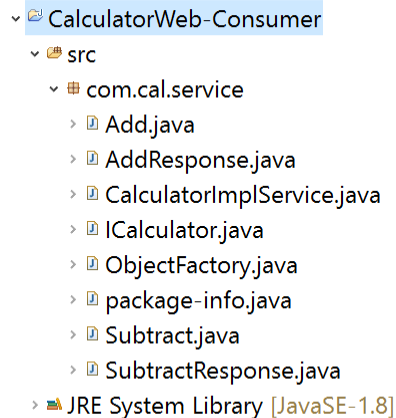
com\cal\service\Add.java
com\cal\service\AddResponse.java
com\cal\service\CalculatorImplService.java
com\cal\service\ICalculator.java
com\cal\service\ObjectFactory.java
com\cal\service\Subtract.java
com\cal\service\SubtractResponse.java
com\cal\service\package-info.java

Compiling code...

javac -d C:\Ashok\CalculatorWeb-Consumer\src -classpath C:\Program Files\Java\jdk1.8.0_161\lib\tools.jar;C:\Program Files\Java\jdk1.8.0_161\classes -Xbootclasspath/p:C:\Program Files\Java\jdk1.8.0_161\jre\lib\rt.jar;C:\Program Files\Java\jdk1.8.0_161\jre\lib\rt.jar C:\Ashok\CalculatorWeb-Consumer\src\com\cal\service\Add.java C:\Ashok\CalculatorWeb-Consumer\src\com\cal\service\AddResponse.java C:\Ashok\CalculatorWeb-Consumer\src\com\cal\service\CalculatorImplService.java C:\Ashok\CalculatorWeb-Consumer\src\com\cal\service\ICalculator.java C:\Ashok\CalculatorWeb-Consumer\src\com\cal\service\ObjectFactory.java C:\Ashok\CalculatorWeb-Consumer\src\com\cal\service\Subtract.java C:\Ashok\CalculatorWeb-Consumer\src\com\cal\service\SubtractResponse.java C:\Ashok\CalculatorWeb-Consumer\src\com\cal\service\package-info.java

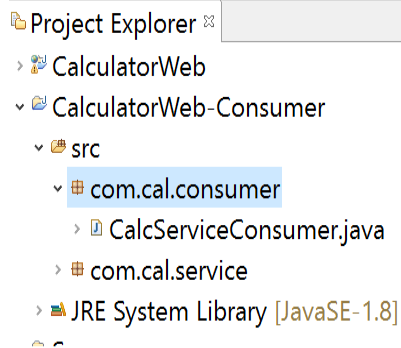
C:\Ashok\CalculatorWeb-Consumer>
```

After classes are generated -> Refresh the project - > we can see classes like below

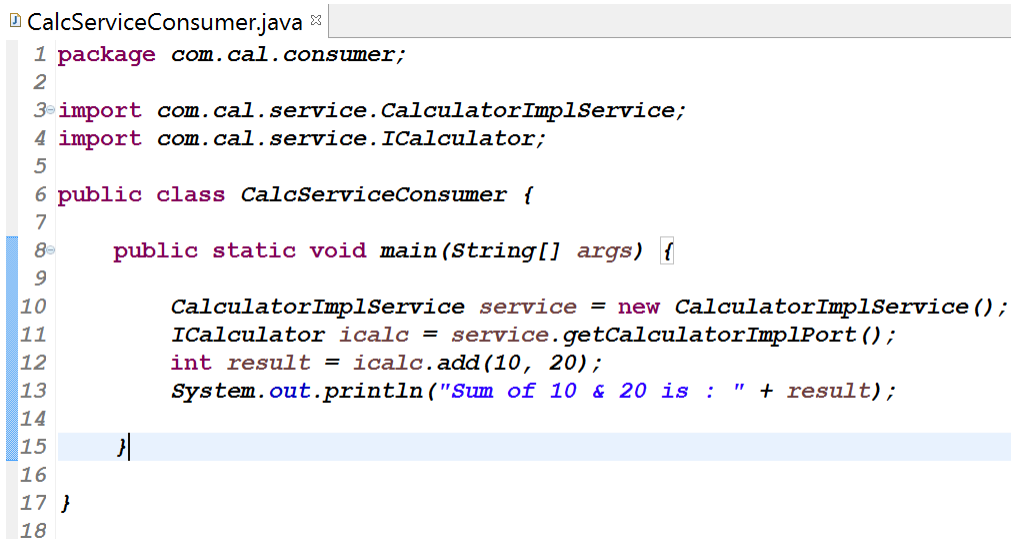


```
CalculatorWeb-Consumer
├── src
│   ├── com.cal.service
│   │   ├── Add.java
│   │   ├── AddResponse.java
│   │   ├── CalculatorImplService.java
│   │   ├── ICalculator.java
│   │   ├── ObjectFactory.java
│   │   ├── package-info.java
│   │   ├── Subtract.java
│   │   └── SubtractResponse.java
│   └── JRE System Library [JavaSE-1.8]
```

Step 3: Develop Consumer class to access the provider

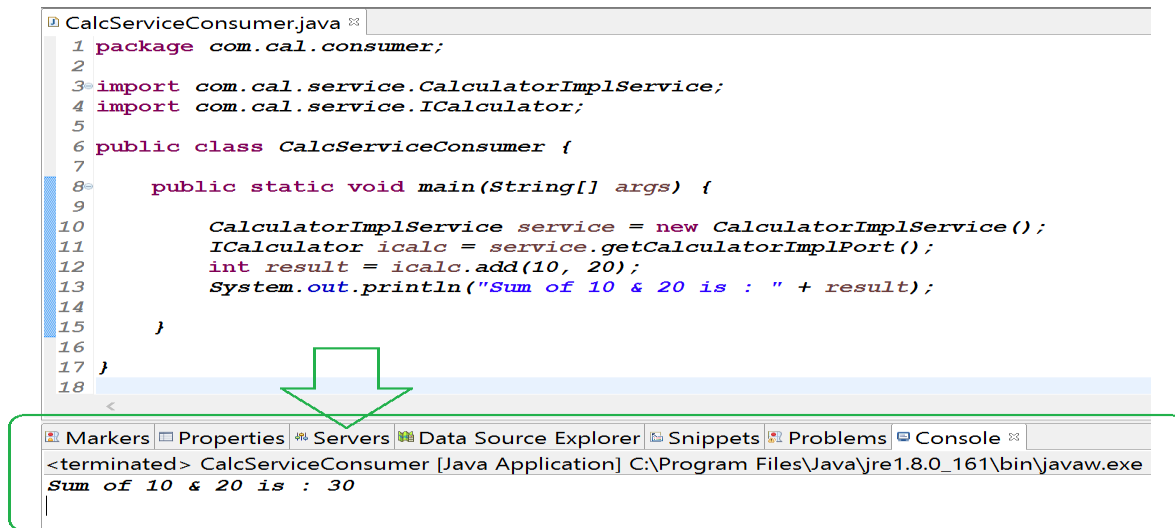


```
Project Explorer
├── CalculatorWeb
├── CalculatorWeb-Consumer
│   ├── src
│   │   ├── com.cal.consumer
│   │   │   └── CalcServiceConsumer.java
│   │   ├── com.cal.service
│   │   └── JRE System Library [JavaSE-1.8]
```



```
CalcServiceConsumer.java
1 package com.cal.consumer;
2
3 import com.cal.service.CalculatorImplService;
4 import com.cal.service.ICalculator;
5
6 public class CalcServiceConsumer {
7
8     public static void main(String[] args) {
9
10         CalculatorImplService service = new CalculatorImplService();
11         ICalculator icalc = service.getCalculatorImplPort();
12         int result = icalc.add(10, 20);
13         System.out.println("Sum of 10 & 20 is : " + result);
14     }
15 }
16
17 }
18
```

Step 4: Run CalcServiceConsumer.java class



```

1 package com.cal.consumer;
2
3 import com.cal.service.CalculatorImplService;
4 import com.cal.service.ICalculator;
5
6 public class CalcServiceConsumer {
7
8     public static void main(String[] args) {
9
10         CalculatorImplService service = new CalculatorImplService();
11         ICalculator icalc = service.getCalculatorImplPort();
12         int result = icalc.add(10, 20);
13         System.out.println("Sum of 10 & 20 is : " + result);
14
15     }
16
17 }
18

```

Markers Properties Servers Data Source Explorer Snippets Problems Console

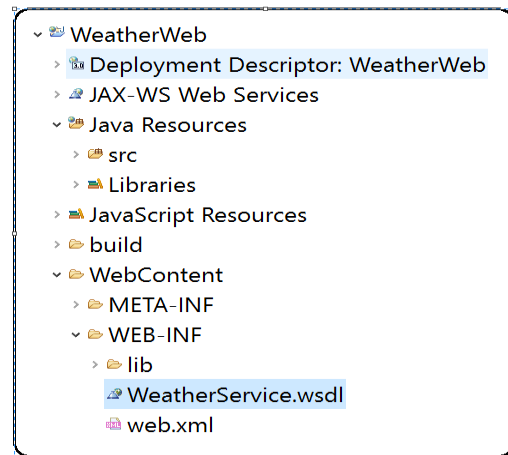
<terminated> CalcServiceConsumer [Java Application] C:\Program Files\Java\jre1.8.0_161\bin\javaw.exe
Sum of 10 & 20 is : 30

Contract First Approach

Step1: Create Dynamic web project and add jar files in project lib folder



Step 2: Create WSDL file in project WEB-INF folder and generate Binding classes using wsimport tool



-----WeatherService.wsdl Start-----

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://service.weather.com"
  xmlns:apache="http://xml.apache.org/xml-soap"
  xmlns:impl="http://service.weather.com"
  xmlns:intf="http://service.weather.com"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!--WSDL created by Apache Axis version: 1.4 Built on Apr 22, 2006 (06:55:48
  PDT) -->
  <wsdl:types>
    <schema elementFormDefault="qualified"
      targetNamespace="http://service.weather.com"
      xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="getTemperature">
        <complexType>
          <sequence>
            <element name="zip" type="xsd:int" />
          </sequence>
        </complexType>
      </element>
      <element name="getTemperatureResponse">
        <complexType>
          <sequence>
            <element name="getTemperatureReturn"
              type="xsd:double" />
          </sequence>
        </complexType>
      </element>
    </schema>
  </wsdl:types>

  <wsdl:message name="getTemperatureRequest">
    <wsdl:part element="impl:getTemperature" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="getTemperatureResponse">
    <wsdl:part element="impl:getTemperatureResponse" name="parameters">
    </wsdl:part>
  </wsdl:message>
```



```

</wsdl:message>
<wsdl:portType name="WeatherService">
  <wsdl:operation name="getTemperature">
    <wsdl:input message="impl:getTemperatureRequest"
name="getTemperatureRequest">
    </wsdl:input>
    <wsdl:output message="impl:getTemperatureResponse"
name="getTemperatureResponse">
    </wsdl:output>
  </wsdl:operation>

</wsdl:portType>
<wsdl:binding name="WeatherServiceSoapBinding" type="impl:WeatherService">
  <wsdlsoap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="getTemperature">
    <wsdlsoap:operation soapAction="" />
    <wsdl:input name="getTemperatureRequest">
      <wsdlsoap:body use="literal" />
    </wsdl:input>
    <wsdl:output name="getTemperatureResponse">
      <wsdlsoap:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

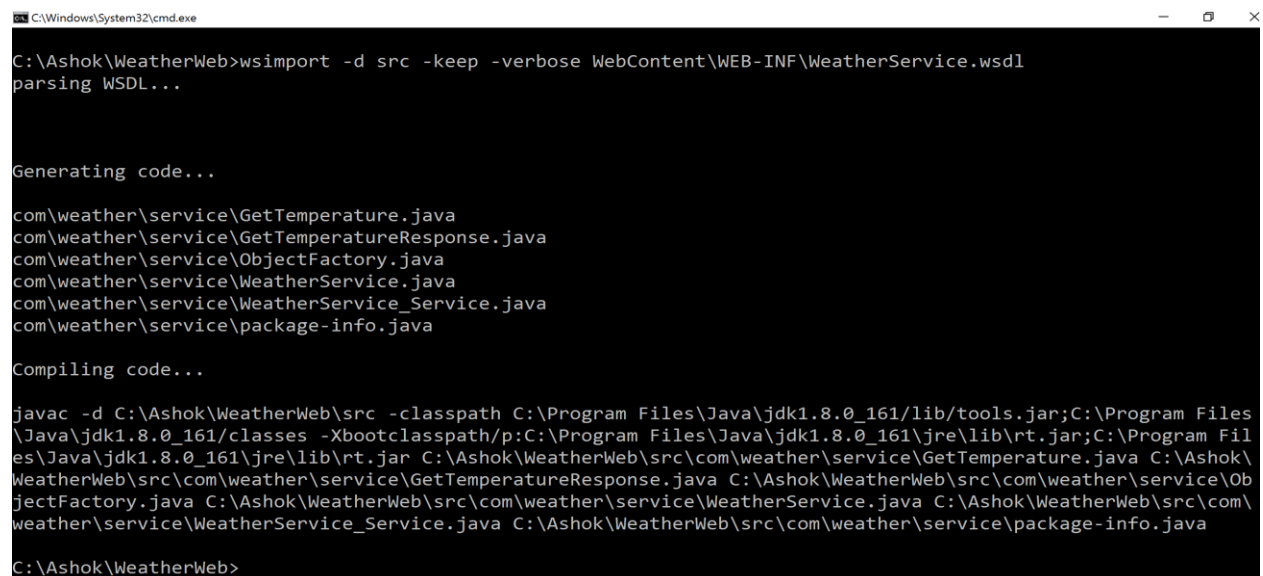
<wsdl:service name="WeatherService">
  <wsdl:port binding="impl:WeatherServiceSoapBinding"
name="WeatherService">
    <wsdlsoap:address

      location="http://localhost:4040/WeatherWeb/services/WeatherServiceImpl" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>

```

-----WeatherService WSDL End-----

Syntax: wsimport -d src -keep -verbose PATH_OF_WSDL



```

C:\Windows\System32\cmd.exe

C:\Ashok\WeatherWeb>wsimport -d src -keep -verbose WebContent\WEB-INF\WeatherService.wsdl
parsing WSDL...

Generating code...

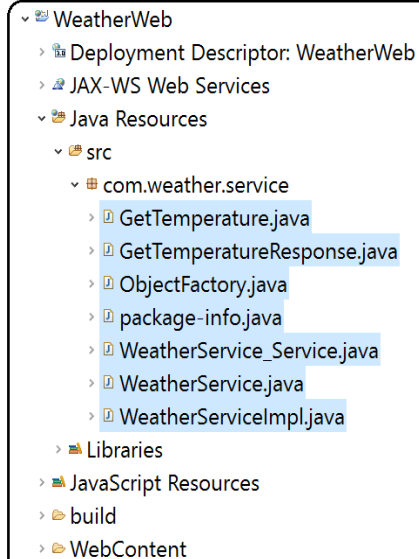
com\weather\service\GetTemperature.java
com\weather\service\GetTemperatureResponse.java
com\weather\service\ObjectFactory.java
com\weather\service\WeatherService.java
com\weather\service\WeatherService_Service.java
com\weather\service\package-info.java

Compiling code...

javac -d C:\Ashok\WeatherWeb\src -classpath C:\Program Files\Java\jdk1.8.0_161\lib\tools.jar;C:\Program Files\Java\jdk1.8.0_161\classes -Xbootclasspath/p:C:\Program Files\Java\jdk1.8.0_161\jre\lib\rt.jar;C:\Program Files\Java\jdk1.8.0_161\jre\lib\rt.jar C:\Ashok\WeatherWeb\src\com\weather\service\GetTemperature.java C:\Ashok\WeatherWeb\src\com\weather\service\GetTemperatureResponse.java C:\Ashok\WeatherWeb\src\com\weather\service\ObjectFactory.java C:\Ashok\WeatherWeb\src\com\weather\service\WeatherService.java C:\Ashok\WeatherWeb\src\com\weather\service\WeatherService_Service.java C:\Ashok\WeatherWeb\src\com\weather\service\package-info.java

C:\Ashok\WeatherWeb>

```



Step 3: Create WeatherServiceImpl.java with business logic

```
WeatherServiceImpl.java
1 package com.weather.service;
2
3 import javax.xml.ws.WebService;
4
5 @WebService(endpointInterface = "com.weather.service.WeatherService")
6 public class WeatherServiceImpl implements WeatherService {
7
8     @Override
9     public double getTemperature(int zip) {
10
11         if (zip == 500081) {
12             return 38.45;
13         } else if (zip == 500082) {
14             return 40.56;
15         }
16         return 0;
17     }
18
19 }
20
```

Step4: Create WebService deployment descriptor file in project WEB-INF folder (sun-jaxws.xml)

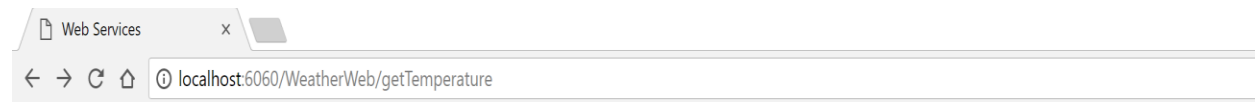
```
sun-jaxws.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">
3     <endpoint name="WeatherService"
4         implementation="com.weather.service.WeatherServiceImpl"
5         url-pattern="/getTemperature" />
6 </endpoints>
```

Step 5: Configure WSServlet in web.xml file

```

web.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee"
3   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
4   id="WebApp_ID" version="3.0">
5 <listener>
6   <listener-class>
7     com.sun.xml.ws.transport.http.servlet.WSServletContextListener
8   </listener-class>
9 </listener>
10 <servlet>
11   <servlet-name>weather</servlet-name>
12   <servlet-class> com.sun.xml.ws.transport.http.servlet.WSServlet </servlet-class>
13   <load-on-startup>1</load-on-startup>
14 </servlet>
15 <servlet-mapping>
16   <servlet-name>weather</servlet-name>
17   <url-pattern>/getTemperature</url-pattern>
18 </servlet-mapping>
19 </web-app>

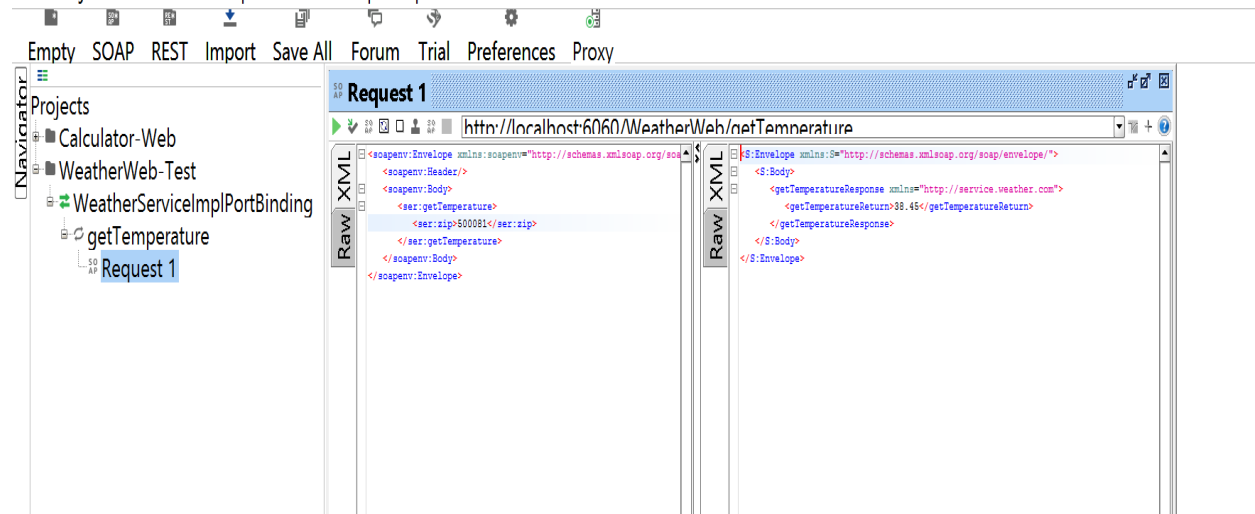
```

Step 6: Deploy the application and access the Endpoint**Web Services**

Endpoint	Information
Service Name: {http://service.weather.com/} WeatherServiceImplService	Address: http://localhost:6060/WeatherWeb/getTemperature
Port Name: {http://service.weather.com/} WeatherServiceImplPort	WSDL: http://localhost:6060/WeatherWeb/getTemperature?wsdl
	Implementation class: com.weather.service.WeatherServiceImpl

Step 7: Test the Provider using SOAP UI tool

File Project Suite Case Step Tools Desktop Help



====0000====