JAVA WEB SERVICES

Study Material

India's No.1 Software Training Institute

DURGASOFT

www.durgasoft.com Ph: 9246212143 ,8096969696

JAVA WEB SERVICE

Cource Content:

1. Webservice introduction

2. Legacy techonologies before webservice and drawbacks with them

3. Webservice architecture and its components

4. Java webservice API's

-> JAX-RPC

-> JAX-M

-> JAX-WS

-> JAX-RS

5. Java webservice implmentation api's(SOAP webservices)

-> Axis1

-> JAX-RPC-SI

-> JAX-WS-RI

-> Metro

-> Axis2

-> Apache cxf

6. Webservice security

7. Restful webservice

-> Jersey

-> Resteasy

-> Restlet

-> Apache cxf

-> Apache wink

8. Restful webservice security.

Web service introduction:

-> If we want to share information between two interoperable applications then we will use webservices.

-> Before webservices also we have some other technologies to share information between 2 applications.

SOcket Programing

RMI(Remote method invocation)

EJB(Enterprice Java Beans)

CORBA(Common Object Request Broker Architecture)

DCOM(Distributed Component)

Socket Programing(Network programing):

-> It is given by sun micro system

-> If two applications are developed by using java language then only we can make communication between them using

socket programing, otherwise we can't use socket programing to make communication between them.

-> We need to write lot of code to integrate applications.

-> It has problem with location dependency.

RMI(Remote method invocation):

-> It is given by sun micro system.

-> RMI is a distributed technology and using this distributed technology we can develope distributed applications.

-> If two applications(client and server) are developed using java then only we can share information between two applications,

otherwise can't.

-> If we are using RMI to share information between 2 applications then we can't provide security and transaction

management to our applications.

EJB(Enterprice Java Beans):

-> It is also given sun micro system.

-> EJB is also distributed technology and using this we can develope distributed applications.

-> If both client and server applications are developed by using java then we use EJB to provide communication

between them.

-> To run EJB components we need application servers ie., weblogic, webspehere etc and using EJB we can provide

transaction management and security to our applications.

DCOM(Distributed Component):

-> It is given by microsoft.

-> It is also distributed technology and using this technology we can develope distributed applications.

-> If we are using DCOM technology to share information between two applications then that applications should be

developed by using dotnet technologies otherwise we can't make communication between them.

CORBA(Common Object Request Broker Architecture):

----- ----------

-> CORBA is a specification and it is given by Object Management group.

-> Sun microsystem, microsoft etc has given implmentation for corba specifications

-> Using CORBA we can make communication beween 2 interoperable applications.

-> But who are provided implmentations for corba, they are not provided proper implmentations due to that we some

cases we are unable to provide communication between 2 interoperable applications.

-> Even we can't provide transaction management and security for our applications using corba.

-> To resolve all these problems WS-I(Webservice - Interoperable) oranization has introduced one specification called

webservice.

Webservice:

-> Webservice is a specification and it has provided set of rules to make communication between 2 interoperable applications.

-> Its not an API, Its contains only set of rules.

-> It given WS-I oranization.

-> Using webservice specification we can make communication between two interoperable applications.

Note:

Interoperable Applications means, the client and service applications are independent applications ie. there is

dependency between 2 applications.

Webservice architecture:

-> Webservice specification has suggested mainly 6 components to provide communication between 2 interoperable

applications ie.,

1. WSDL(Webservice Description Language)
2. Skeleton.
3. UDDI registry.
4. STUB
5. SOAP

6. HTTP

-> All these 6 components are called as Webservice components.

WSDL:

-> WSDL is a xml file and it should be generated by service provider.

-> To generate wsdl file, service provider will use wsdl generation tool and tool is a predefined class.

-> WSDL file contains service class name, name of service class methods, parameter names, parameter datatypes and return types.

-> In addition to service class details, it contains endpoint url of the webservice.

-> After generating wsdl file by the service provider, service provider will share wsdl file to the client through an email

attachment or UDDI registry.

-> After receiving wsdl file by the client, he will generate stubs or proxies at client location.

-> Here stub generation tool or proxy generation is a predefined class.

UDDI Registry:

-> Service will use UDDI registry to share wsdl file to the client.

-> It is an optional component in webservice architecture because client can share wsdl to the client through an email

attachment.

STUB:

-> STUB is a generated class and it is generated by using stub generation tool.

-> Stub is class and it contains service class methods in the client native language.

-> Client will create stub object in his application and he will invoke required service methods on STUB object.

-> Stub will prepare soap request which contains client requested method details ie., name of the method, datatypes of the

parameters and parameter values.

-> Stub will receives soap response from skeleton and it will read soap response to handover service return value to

client.

SOAP:

-> SOAP is a protocol which contains set of predefined xml tags and these xml tags are called as soap tags.

-> STUB will uses these tags to prepare request xml document, so this request xml document is called as soap request.

-> Skeleton will uses these tags to prepare response xml document, so that the response xml document is called as

SOAP response.

HTTP:

-> As we know HTTP protocol is a transportion protocol and it will soap request from client location to service location

and it will move soap response service location to client location.

Skeleton:

-> Skeleton is a predefined class and will exist at service location.

-> Skeleton will receives soap request from client location and it will read soap request to get client request

method details.

-> It will requested method on actual service class and it will get return value from service class.

-> It will soap response by including service class retun value and finally it will send soap response tool

client using http protocol.

Java Webservice API'S:

-> Sun has provided total 4 API's to develop webservices and webservice clients ie.,

1. JAX-RPC(Java API for XML - Remote Procedure Call) - JDK1.4
2. JAX-M(Java API for XML - Mssaging) - JDK1.4
3. JAX-WS(Java API for XML - Webservice) - JDK1.5
4. JAX-RS(Java API for XML - Restful Services) - JDK1.6

-> We can have 2 types webservices ie.,

1. Synchronse Webservices
2. Asynchronse Webservices

Synchronse Webservices:

-> When ever client makes a request to service and if service provides immidiate response to client then that service is

called as Synchronse webservice.

-> To develop Synchronse webservices we will use below API'S ie.,

1. JAX-RPC
2. JAX-WS
3. JAX-RS

Asynchronse Webservices:

-> When ever client makes a request to service and if takes some time to process client then that service is called

as Asynchronse webservice.

-> In Asynchronse webservices client will never wait for service response.

-> To develop Asynchronse webservices we will use below API.

1. JAX-M

-> Again synchronse webservices has devided 2 types ie.,

1. SOAP based webservices
2. Restful webservices

-> To Develop SOAP based webservices we will use JAX-RPC and JAX-WS.

-> To develop Restful webservices we will use JAX-RS.

-> Unfortunatily all these API's are specifications but not implmentations.

-> So every specification need implmentation to develop webservice or webservice client.

-> So the above java webservice specifications need implementations.

-> Lot of third party vendors has provided implementations for above specifications.

JAX-RPC:

-> JAX-RPC-SI(Sun implmentation) given by sun.

-> Axis1 given Apache foundation

-> Weblogic implmentation -> Its came along with weblogic server(BEA)

-> Webspenere implmentation -> Its came along with webspehere server(IBM)

-> Jboss implmentation -> Its came along with Jboss server(Redhat)

Note:

-> Here JAX-RPC-SI and Axis1 are container independent implementations and remaining implementations are called as

container dependent implementations.

JAX-WS:

-> JAX-WS-RI(Reference Implementation) -> sun(JDK1.6)

-> Metro -> sun

-> Axis2 -> Apache foundation

-> ApacheCXF -> Apache foundation

-> Weblogic implmentation -> BEA

-> Jboss implmentation -> REDHAT

-> Webspenere implmentation -> IBM

-> Glass fish implmentation -> SUN

Note:

-> The first 4 implementations are called as container dependent implementations and last 4 implementations are called

as container independent implementations.

JAX-RS:

-> Jersey -> SUN

-> Resteasy -> Redhat

-> Restlet -> Jerome Louvel

-> Apache wink -> Apache foundation

-> Apache CXF -> Apache foudation

Note:

-> All these implementations are called as container independent implementations.

-> Axis2 doesn't support spring integration and ApacheCXF will have spring integration support.

-> Apache WINK doesn't support spring integration where as ApacheCXF supports spring integration.

Axis1 implmentation:

-> Axis1 implmentation follows JAX-RPC specification.

-> Using this implmentation we can develop soap based webervice and webservice client.

-> It is given Apache Foundation.

-> Before developing webservice or webservice client using axis1 implmentation first we need to downloas axis1 implmentation

software(jar files)

-> To download axis1 software we need to download axis-bin-1_4.zip file from below url.

http://apache.xmlcity.org/ws/axis/1_4/

-> After download above zip file, we need to extract some where in our local system.

D:\axis-1_4\

|_ lib -> *.jar(axis1 jars)

|_ samples -> sample webservices and webservices clients

|_ webapps -> axis web application.

-> Axis1 implmentation has given below classes which are acting as skeleton, wsdl generation tool and

stub generation tool.

1. org.apache.axis.wsdl.WSDL2Java -> Stub generation tool.
2. org.apache.axis.wsdl.Java2WSDL -> wsdl generation tool.
3. org.apache.axis.transport.http.AxisServlet -> sekeleton(servlet).

-> As we discussed axis1 implmentation has provided in built we server called Simple Axis Server, So if we develop webservice

using axis1 implmentation then we can deploy that webservice on simple axis server or any other webserver or simple

axis server. Because it is container independent implmentation.

Axis1 architecture

-> When we start Simple axis server bydefault one webapplication called axis will be run onto simple axis server.

-> axis webapplication is already contains webapplication structure with axis1 jars and web.xml which includes

org.apache.axis.transport.http.AxisServlet configutation with "/service/*" url-pattern.

-> If we want to deploy our service class onto axis webapplication we need follow below steps.

1. Create Service class called CalService.java and compiled it.
2. Generate wsdl file using wsdl generation tool.
3. Start simple axis server.
4. Prepare server-config.wsdd file and placed it into axis/WEB-INF folder.

Steps to create axis1 webservice and deploy it on to SimpleAxisServer:

Step1:

-> Create "CalService" folder in below location.

D:\backup\desktop\online class\webservices\class9\examples\axis1

-> Create CalService.java file and save it into above CalService folder.

-> Compile CalService.java

D:\backup\desktop\online class\webservices\class9\examples\axis1\CalService>javac -d . CalService.java <-|

Step2:

-> Download 2 more jar ie., java-mail-1.4.4.jar and activation.jar and copy them into our extracted axis s/w lib folder.

-> Set below axis1 jars into environment variable.

1. activation.jar
2. axis.jar

3. axis-ant.jar

4. commons-discovery-0.2.jar

5. commons-logging-1.0.4.jar

6. java-mail-1.4.4.jar

7. jaxrpc.jar

8. log4j-1.2.8.jar

9. saaj.jar

10. wsdl4j-1.5.1

-> Generate wsdl file by following below steps.

1. As we know wsdl file used by client,so that service provider need to generate wsdl file and share it client.

2. Axis1 implementation has given "org.apache.axis.wsdl.Java2WSDL" class to generate wsdl file.

3. To generate wsdl file, wsdl generation tool needs below inputs

a. wsdl file name(it can be any name, but normally we will use our service class name ie., CalService.wsdl)

b. endpoint url(http://localhost:65535/axis/services/cService)

c. target namespace(It can be any thing but it should be in url ei., http://www.venkat.com)

d. Our service class name.

4. Open command prompt and go to where our service class is available, and execute below command to

generate wsdl file.

D:\backup\desktop\online class\webservices\class9\examples\axis1\CalService>java org.apache.axis.wsdl.Java2WSDL -O CalService.wsdl -l http://localhost:65535/axis/services/cService -n http://www.venkat.com com.venkat.service.CalService <-|

Note: If we are getting any ClassNotFoundException then execute below command before executing above command.

D:\backup\desktop\online class\webservices\class9\examples\axis1\CalService>set classpath=.;%classpath% <-|

Step3:

-> In step3 we need to follow below steps to generate server-config.wsdd and to move CalService.class(axis/WEB-INF/classes)

and server-config.wsdd(axis/WEB-INF)

-> To do this we need use org.apache.axis.client.AdminClient and this class will perform above step.

-> To Perform above step by AdminClient, We need to prepare deploy.wsdd file and handover it to AdminClient.

-> To prepare deploy.wsdd file need to follow below steps

a. Copy existing deploy.wsdd file from D:\backup\mydata\webservicesandxml\axis-1_4\samples\stock into

our above "CalService" folder.

b. Modify deploy.wsdd file as per our requirement.

deploy.wsdd:

```
<deployment name="test"
xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

<service name="cService" provider="java:RPC">

<parameter name="className"
value="com.venkat.service.CalService"/>
<parameter name="allowedMethods" value="add"/>

</service>

</deployment>
```

-> handover deploy.wsdd file to AdminClient, So that AdminClient will generate server-config.wsdd file and it will move

Our service class and generated server-config.wsdd file into axis webapplication in the respective folders. But before

handover deploy.wsdd file to AdminClient we need to start SimpleAxisServer.

-> To start simpleaxisserver we need to execute below command.

1. Open new command prompt and go to our CalService folder location.

```
D:\backup\desktop\online
class\webservices\class9\examples\axis1\CalService>set
classpath=.;%classpath% <-|

D:\backup\desktop\online
class\webservices\class9\examples\axis1\CalService>java
org.apache.axis.transport.http.SimpleAxisServer -p 65535 <-|
```

-> Handover deploy.wsdd to AdminClient and to do this execute below step.

```
D:\backup\desktop\online
class\webservices\class9\examples\axis1\CalService>set
classpath=.;%classpath% <-|

D:\backup\desktop\online
class\webservices\class9\examples\axis1\CalService>java
org.apache.axis.client.AdminClient -p 65535 deploy.wsdd <-|
```

-> Use below url in browser to test wheather service is deployed or not.

http://localhost:65535/axis/services/cService?wsdl

Redeploying CalService:

-> If we want redeploy application then we need to follow steps.

1. Start simple axis server.

D:\backup\desktop\online class\webservices\class9\examples\axis1\CalService>java org.apache.axis.transport.http.SimpleAxisServer -p 65535 <-|

2. Remove generated server-config.xml file from "CalService" folder.

3. Run AminClient coomand to deploy service.

D:\backup\desktop\online class\webservices\class9\examples\axis1\CalService>java org.apache.axis.client.AdminClient -p 65535 deploy.wsdd <-|

Webservice client:

-> We have 2 types of clients are available in webservices ie.,

1. Proxy based client.
2. DII client.

1. Proxy based client:

-> In proxy based client, client will generate stubs and he will use generated stubs in his client application to

invoke webservice.

2. DII client:

-> In DII client, client will never generate stubs in his client application, instead webservice implementation provider(axis1, metero etc.,)

will provide some predefined classes which are acting as a stubs.

-> Client uses these predefined classes in his application to invo webservice.

Axis1 proxy based webservice client:

-> In proxy based client, we need to generate stubs at client location by using stub generation tool.

-> Axis1 implementation has provided below class which acting as stub generation tool.

org.apache.axis.wsdl.WSDL2Java

-> Using above wsdl generation we will generate stubs like below.

...>java org.apache.axis.wsdl.WSDL2Java CalService.wsdl <-|

Steps to create Axis1 proxy client:

-> Create "CalServiceClient" folder in below location.

D:\backup\desktop\online class\webservices\class9\examples\axis1

-> Copy cliet shared wsdl file into above folder.

-> Now execute below command from above location to generate stubs.

D:\backup\desktop\online class\webservices\class9\examples\axis1>java org.apache.axis.wsdl.WSDL2Java CalService.wsdl <-|

-> Compile generated stubs.

D:\backup\desktop\online class\webservices\class9\examples\axis1\com\venkat\www>javac *.java <-|

-> Now Create TestClient.java in above CalServiceClient folder with below code.

TestClient.java:

```
public class TestClient {

   public static void main(String[] args) throws Exception {

    java.net.URL url = new
java.net.URL("http://localhost:65535/axis/services/cService");


    org.apache.axis.client.Service service = new org.apache.axis.client.Service();


    com.venkat.www.Localhost65535SoapBindingStub stub = new
com.venkat.www.Localhost65535SoapBindingStub(url, service);

     int res = stub.add(10,10);

    System.out.println(res);

   }

}
```

-> Compile and run TestClient.java

-> But before compile and run TestClient.java, need to execute below command.

....>set classpath=.;%classpath%

DII client:

//http://localhost:65535/axis/services/cService

DIITestClient.java:

```
public class DIITestClient {

  public static void main(String[] args) throws Exception {

    org.apache.axis.client.Service ser =
        new org.apache.axis.client.Service();

    org.apache.axis.client.Call client =
     (org.apache.axis.client.Call)ser.createCall();
```

```
        client.setTargetEndpointAddress("http://localhost:65535/axis/services");

        javax.xml.namespace.QName q =
       new javax.xml.namespace.QName("cService","add");

    client.setOperationName(q);

        client.addParameter("in0",
        javax.xml.rpc.encoding.XMLType.XSD_INT,
        javax.xml.rpc.ParameterMode.IN);

        client.addParameter("in1",
        javax.xml.rpc.encoding.XMLType.XSD_INT,
        javax.xml.rpc.ParameterMode.IN);

        client.setReturnType(javax.xml.rpc.encoding.XMLType.XSD_INT);

        Object params[] = {new Integer(20), new Integer(20)};

        Integer res =(Integer)client.invoke(params);

   System.out.println(res);
}
}
```

-> Difference between proxy client and DII client:

-> If our method return type or parameter types are user defined type then we will use proxy based client.

-> If our method parameter or return types are predefined types then we will go for DII client.

Webservices in eclipse:

-> If we create webservice in eclipse then it will perform below steps.

1. It will generate wsdl file
2. It will copy webservice implementation jars into WEB-INF/lib folder.
3. It will create configuration(server-config.wsdd) file.
4. It will configure skeleton in web.xml with some url-pattern.
5. It will deploy application into server.

-> If we create webservice client in eclipse then it will perfome below steps.

1. It will generate stubs.
2. It will set webervice implementation jars into classpath.

-> Eclipse having only 3 webservice plugins ie., Axis1, Axis2 and Apache CXF.

-> So that if we create webservice or webservice client using any one of above implementation then eclipse will

perform necessary(above steps) steps.

-> If we create webservice or webservice client using other than above implementations then developer need to

perform above steps.

Steps to create Axis1 webservice in eclipse:

-> Create Dynamic web project with project name as Axis1WeatherService.

-> Add tomcat server to eclipse.

-> Create 2 classes ie., WeatherService.java and Weather.java

WeatherService.java:

```
package com.venkat.service;

import java.sql.DriverManager;
```

```
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class WeatherService {

  private Connection getConnection() {
    Connection con = null;

  try{
      Class.forName("oracle.jdbc.driver.OracleDriver");
      con =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","system"
,"xe");

   } catch(SQLException e) {e.printStackTrace();}
     catch(ClassNotFoundException e) {e.printStackTrace();}
     return con;
  }

  public Weather getWeatherByCityId(int cityId) {

        /* if(cityId < 1) {
              throw new ArithmeticException("In valid parameters");
```

```
        }
       */

    Weather w = null;

  try {
    Connection con = getConnection();
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("select * from weather where
cid="+cityId);

    System.out.println("SQL QUERY:$$$$$$$$$$$$$  "+ "select * from
weather where cid="+cityId);

    if(rs.next()) {
        int cid = rs.getInt(1);
        String cname = rs.getString(2);
        int temp = rs.getInt(3);
        w = new Weather();
        w.setCid(cid);
        w.setCname(cname);
        w.setTemp(temp);
    }
  }catch(Exception e) {e.printStackTrace();}
    return w;
```

```
  }


  public int getTempByWeather(Weather w) {
     String cname = w.getCname();
     int temp = -100;


     try {
      Connection con = getConnection();
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery("select * from weather where
cityname='"+cname+"'");
      System.out.println("SQL QUERY:$$$$$$$$$$$$  "+ "select * from
weather where cityname='"+cname+"'");


      if(rs.next()) {
         temp = rs.getInt(3);


      }
    }catch(Exception e) {e.printStackTrace();}
      return temp;
  }
}
```

Weather.java:

```
package com.venkat.service;

public class Weather {

  private int cid;
  private String cname;
  private int temp;


  public int getCid() {
      return cid;
  }

  public void setCid(int cid) {
      this.cid = cid;
  }

  public String getCname() {
      return cname;
  }

  public void setCname(String cname) {
      this.cname = cname;
```

```
 }

 public int getTemp() {
     return temp;
 }

 public void setTemp(int temp) {
    this.temp = temp;
 }
}
```

-> Copy ojdbc14.jar or classes12.jar file into WEB-INF/lib folder.

-> Follow below steps to convert WeatherService.java into webservice using Axis1 implementation.

right click on WeatherService.java -> new -> other.... -> expand WebServices folder -> select Webservice -> click on next -> next -> finish.

-> When we follow above step it will perfome below steps ie.,

-> It will generate wsdl file in WebContent/wsdl.

-> It will copy all axis1 jars into WEB-INF/lib folder.

-> It will configure axis1 skeleton ie., AxisServlet in web.xml file with /service/* url-pattern

-> It will generate server-config.wsdd file in WebContent/WEB-INF folder.

-> It will deploy application into server.

-> Deploy application into server.

-> Use below url in browser to check weather webservice has deployed or not.

http://localhost:8082/Axis1WeatherService/services/WeatherService?wsdl

Axis1 proxy based client in eclipse:

-> Create java project classed Axis1WeatherServiceClient.

-> copy wsdl file into our project root folder.

-> Generate stubs in src folder by following below step.

right click src folder -> new -> other... -> expand webservices folder -> select Webservice client -> click on next

-> click on Browse button to browse wsdl file -> click on finish.

-> Create TestClient.java in src folder to invoke webservice.

TestClient.java:

```
import com.venkat.service.Weather;
import com.venkat.service.WeatherServiceSoapBindingStub;



public class TestClient {

    public static void main(String[] args) throws Exception {
        java.net.URL endpointUrl =
                new
java.net.URL("http://localhost:8082/Axis1WeatherService/services/WeatherSer
vice");

            org.apache.axis.client.Service serviceName =
        new org.apache.axis.client.Service();

            WeatherServiceSoapBindingStub stub =
         new WeatherServiceSoapBindingStub(
          endpointUrl,
          serviceName);
```

```
                    Weather w = stub.getWeatherByCityId(100);


                    System.out.println(w.getCid());
                    System.out.println(w.getCname());
                    System.out.println(w.getTemp());
        }

}
```

-> Create WEATHER table with below data before compile and run above TestClient.java file.

WEATHER(CID, CITYNAME, TEMP)

Web client for above WeatherService:

-> Create Dynamic Webproject with project name as "Axis1WeatherServiceWebClient".

-> Copy wsdl file into project root directory.

-> Generate stubs in src folder.

right click on src folder -> new -> other... -> expand webservices folder -> select webservice client ->

click on next -> browse wsdl file -> click on finish.

-> Copy servlet-api.jar file into WEB-INF/lib folder.

-> Create weather.jsp in WebConten folder.

weather.jsp:

```
<html>
<body>

 <form action="getWeather">

  <% Integer temp = (Integer)request.getAttribute("TEMP");

    if(temp != null) {
  %>
     Temp: <%=temp%><br/>

   <%  }

  %>
```

```
 Enter Cityname: <input type="text" name="city"/><br/>

 <input type="submit" value="get weather"/>

</form>
</body>
</html>
```

-> Create WeatherServlet.java in src folder with com.venkat.servlet package.

WeatherServlet.java:

```
package com.venkat.servlet;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.venkat.service.Weather;
```

```
import com.venkat.service.WeatherServiceSoapBindingStub;


public class WeatherServlet extends HttpServlet {



        public void doGet(HttpServletRequest req,
                        HttpServletResponse res ) throws ServletException,
IOException {



                String city = req.getParameter("city");



                java.net.URL endpointUrl =
                                new
java.net.URL("http://localhost:8082/Axis1WeatherService/services/WeatherSer
vice");

                                org.apache.axis.client.Service serviceName =
                        new org.apache.axis.client.Service();

                                WeatherServiceSoapBindingStub stub =
                        new WeatherServiceSoapBindingStub(
                        endpointUrl,
```

```
                    serviceName);



                        Weather w = new Weather();

                        w.setCname(city);

                        int temp = stub.getTempByWeather(w);

                req.setAttribute("TEMP", temp);

                RequestDispatcher rd =
req.getRequestDispatcher("weather.jsp");
                rd.forward(req, res);
        }
}
```

-> Update web.xml file like below

web.xml:

```
<web-app>
  <welcome-file-list>
    <welcome-file>weather.jsp</welcome-file>
  </welcome-file-list>
```

```
  <servlet>
    <servlet-name>weatherServlet</servlet-name>
    <servlet-class>com.venkat.servlet.WeatherServlet</servlet-class>
  </servlet>

  <servlet-mapping>
     <servlet-name>weatherServlet</servlet-name>
     <url-pattern>/getWeather</url-pattern>
  </servlet-mapping>

</web-app>
```

Note:

-> Before run this application service application should be deploy on to server.

-> Deploy our client application into server and below url to access weather.jsp.

http://localhost:8082/Axis1WeatherServiceWebClient/

WSDL tags:

-> For each service class we should have one wsdl file.

-> Service will wsdl file using wsdl generation tool and he will shared that wsdl file to the client.

-> Client will generate stubs using stub generation from wsdl file and client uses that stubs to invoke webservices.

-> As we know wsdl file having service class details ie., name of the service class, names of service class methods,

parameter datatypes and return types.

-> In addition these details we should have endpoint url of the service in wsdl file.

-> WSDL file mainly contains 6 tags ie.,

1. <definitions>
2. <types>
3. <message>
4. <portType>
5. <binding>
6. <service>

<definitions>:

-> It is the root tag of the wsdl file and it contains remaining 5 tags.

<portType>:

-> It contains service class details and its method details.

-> <portType> tag contains one attribute called "name" and one or more <operation> tags.

-> "name" atrribute takes name of the service class and for each method in the service class we should have one

<operation> tag.

-> Each <operation> tag contians one attribute called "name" and one <input> tag and <output> tag.

-> Here "name" attribute contains name of the method and <input> tag contains method parameter details and

<output> tag contains method return type.

-> Each <input> tag and <output> tag internally refers one <message> tag ie., for ever <input> tag we should have one

<message> tag and <output> tag we should <message> tag.

-> Each <message> tag internally refers one <element> tag which is in <types> tag.

-> <element> tag contains actual return type or parameter types in the from of xml data type.

<message>

-> For each <input> tag we should have one <message> tag and it contains method parameter or return type details.

-> each <message> tag internally refers one <element> tag which is in <types> tag.

<types> :

-> Here <types> tag contains set of <element> tags and each <element> tag contians one parameter or return type

in the of xml data type.

<binding>:

-> Here <binding> tag will contains soap binding styles of the service methods.

-> Here we have 4 binding styles are available in webservices ie.,

1. RPC - literal
2. RPC - encoded
3. DOCUMENT - literal

4. DOCUMENT - encoded

-> These binding styles are used by stub to generate soap request format.

<service>:

-> Here <service> tag contains name of the service and endpoint-url of the webservice.

SOAP UI tool:

-> SOAP UI tool is used to test webservices.

-> SOAP UI tool needs physical wsdl file or wsdl url to test webservices.

Steps to follow test webservices using soap ui tool:

-> Open soap ui tool

-> Go to file -> new soap project -> enter name of the project and browse wsdl file or enter wsdl url -> click on ok

-> Expand any one of method "Request1" -> enter values in the place of "?" -> click on run button to test websservice.

Creating soap ui test cases:

-> Group of test methods are called as one test case.

-> Follow below steps to create soap ui testcase.

1. Right click on project -> New Test Suite -> enter test suite name -> click on OK.

2. Right click on testsuite -> New TestCase -> enter testcase name -> click on ok.

-> Now add test methods to testcase.

Right click on testcase -> Add Step -> SOAP Request -> enter test step name -> OK -> choose method -> OK.

-> To run testcase right click on testcase -> Show testcase editor -> click on run to execute all test steps.

Creating properties in SOAP UI:

-> To create properties in testcase, we need to follow below steps.

Right click on testcase -> Add Step -> Properties -> click OK. -> double click on properties to open properties window

-> click on + button to add property name and value.

-> After creating properties, we will use below syntax to use properties in soap request

${Properties#name of the prtoperty}

Ex:

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://service.venkat.com">
  <soapenv:Header/>
  <soapenv:Body>
    <ser:getWeatherByCityId>
      <ser:cityId>${Properties#cityId}</ser:cityId>
    </ser:getWeatherByCityId>
  </soapenv:Body>
</soapenv:Envelope>
```

SOAP tags:

-> We have lot predefined soap tag are avilable and using these soap tags stub will

prepare soap request and skeleton will prepare soap response.

SOAP request tags:

-> SOAP request contains mainly 3 tags ie.,

1. <Enevelop>
2. <Header>
3. <Body>

<Enevelop>:

-> It is the root tag of the soap request and it contains
remaining 2 tags ie., <body> tag and <header>

<Header>:

-> Some times webservices protected with some username and
password, So client we will use this send authentication
details of webservices.

<Body>:

-> It contains method details ie., name of the method,
parameter names, parameter datatypes and parameter values.

Ex:

```
<soapenv:Envelope>
  <soapenv:Header/>
  <soapenv:Body>
    <ser:getWeatherByCityId>
      <ser:cityId>100</ser:cityId>
    </ser:getWeatherByCityId>
  </soapenv:Body>
</soapenv:Envelope>
```

SOAP response tags:

-> SOAP response mainly contains 3 tags ie.,

1. <Enevelop>
2. <Body>
3. <Fault>

1. <Enevelop>:

-> It is the root tag of the soap response and it contains

<Body> tag as a child tag.

2. <Body>:

-> It contains service response or return value.

-> Some times it contains <Fault> tag.

3. <Fault>:

-> It is optional tag and it will appear in <Body> tag.

-> If webservices throws some exception then this tag

will be appear in soap in response.

-> This tag holds the webservice exception details.

-> <Fault> tag contains <faultCode> and <faultString> are

child tags.

Drawback with JAX-RPC:

-> JAX-RPC doesn't support any collection type or array type as method return type or parameter type.

JAX-WS

-> It is a soap based java webservice specification and using this implementation we can develop soap based webservices and clients in java.

-> It is introduced by sun along with jdk1.5.

-> It has lot of implementations which are given by some third party vendors ie.,

1. JAX-WS-RI(Java api for xml - reference implementation) -> sun
2. Metro -> sun
3. Axis2 -> apache foundation
4. Apache CXF implementation -> apache foudation
5. weblogic implementation -> BEA
6. webspehere implementation -> IBM
7. Jboss implementation -> Redhat
8. Glass Fish implementation(Metro) -> Sun

-> Here the first 4 implementations are called as container independent implementations and last 4 are container dependent implementations.

-> JAX-WS having some annotations ie.,

@WebService(javax.jws.WebService)

@WebMethod(javax.jws.WebMethod)

@WebParam(javax.jws.WebParam)

-> These annotations are commonly used in service class even if we develop webservices using any implementations from JAX-WS

specification.

@WebService:

-> It is a mandatory annotation and we will use this annotation in service interface and implementation class.

Ex:

```
@WebService
public interface CalServiceI {

    public int add(int i, int j);
    public int sub(int i, int j);
}
```

```
package com.venkat.service;

    @WebService(endpointInterface="com.venkat.service.CalServiceI",
            serviceName="CalServiceImpl",
                    targetNamespace="http://www.venkat.com")
public class CalServiceImpl implements CalServiceI {

      @WebMethod(name="addition")
      public int add(@WebParam(name="fno") int i,
@WebParam(name="sno") int j) {return i+j;}

      @WebMethod(name="subtraction")
      public int sub(int i, int j) {return i-j;}
    }
```

-> @WebService annotation having some optional properties ie.,

-> endpointInterface

-> serviceName

-> targetNamespace

-> We will use these properties inside service implementation classe but not in service interface.

1. endpointInterface:

-> It will take fully qualified name of the service interface and it is optional property.

-> The default value for this property is fully qualified name of service interface.

-> This value will be appear in wsdl file at <portType> tag name attribute.

2. serviceName:

-> It can be any name but it should be unique name.

-> It is an optional property.

-> The default value for this property is Service class name.

-> This value will be appear in wsdl file <service> tag name attribute value.

3. targetNamespace:

-> It can be any name but it should be in url format.

-> It is a default property.

-> The default values for this property is reverse order of the service class package name.

Ex: package com.venkat.service -> http://service.venkat.com

-> This value will appear in wsdl file at <definitions> tag targetNamespace attribute.

@WebMethod:

-> It is an optional annotation and it can be used in implementation class but not in interface.

-> If we want to generate different method name in th wsdl file then we will use this annotation.

```
@WebMethod(name="addition")
```

@WebParam:

-> It is an optional annotation and it can be used in implementation class but not in interface.

-> if we want to generate different parameter name in the wsdl file then we will use this annotation.

Note:

-> If we are using any implementation from JAX-WS to develop webservices then commonly we will use these annotations

in service interface and implementation class.

JAX-WS-RI implementation:

-> It is JAX-WS specification implementation and it is given by sun.

-> It is a reference implementation, So it is partial implementation for JAX-WS.

-> No body will use this implementation to develop realtime webservices.

-> It introduced by sun along with jdk1.6.

-> JAX-WS-RI implementation has given one in build called JAX-WS-RI server along with jdk1.6.

-> If we develop webservices using JAX-WS-RI implementation then we can deploy that webservices on any server

including JAX-WS-RI server.

-> If we develop webservice using any other implementations from JAX-WS then that webservices we cant deploy it onto

JAX-WS-RI server. Because it is specific to JAX-WS-RI implementation.

Developing webservices using JAX-WS-RI implementation and deploy it on to JAX-WS-RI server:

-> Create "CalService" folder in below location.

D:\backup\desktop\online class\webservices\class9\examples\jax-ws-ri

-> Create CalServiceI.java and CalServiceImpl.java in above location.

CalServiceI.java:

```
package com.venkat.service;

import javax.jws.WebService;

@WebService
public interface CalServiceI {

   public int add(int i,  int  j);
        public int sub(int i,  int  j);
}
```

CalServiceImpl.java:

```
package com.venkat.service;

import javax.jws.WebService;

@WebService(endpointInterface="com.venkat.service.CalServiceI",
                serviceName="CalServiceImpl",
                        targetNamespace="http://www.venkat.com")
public class CalServiceImpl implements CalServiceI {

   public int add(int i, int j) {return i+j;}

        public int sub(int i, int j) {return i-j;}

}
```

-> Compile above 2 files.

D:\backup\desktop\online class\webservices\class9\examples\jax-ws-ri\CalService>javac -d . CalServiceI.java

D:\backup\desktop\online class\webservices\class9\examples\jax-ws-ri\CalService>javac -d . CalServiceImpl.java

-> Create Publisher.java with below code in above CalService folder.

Publisher.java:

```
public class Publisher {

    public static void main(String[] arg) {

            javax.xml.ws.Endpoint.publish("http://localhost:9999/myapp", new
com.venkat.service.CalServiceImpl());

        }

}
```

-> Compile and run Publisher.java to deploy our webservice on jaxws-ri server.

```
  D:\backup\desktop\online class\webservices\class9\examples\jax-ws-
ri\CalService>set classpath=.;%classpath%

  D:\backup\desktop\online class\webservices\class9\examples\jax-ws-
ri\CalService>javac Publisher.java

  D:\backup\desktop\online class\webservices\class9\examples\jax-ws-
ri\CalService>java Publisher
```

-> After executing above commands, Open browser and use below url to access wsdl file.

```
    http://localhost:9999/myapp/?wsdl
```

JAX-WS-RI Architecture:

-> When we run above Publisher.java, It will follow below steps.

1. It will create one webapplication with names as "myapp".

2. It will move CalServiceI.class and CalServiceImpl.class file into myapp/WEB-INF/classes folder.

3. It will create web.xml in myapp/WEB-INF/ folder and it will configure JAX-WS-RI skeleton ie.,

WSServlet in web.xml file with url-pattern as "/*"

4. It will create sun-jaxws.xml file in myapp/WEB-INF/ folder and it will configure our service class ie.,

CalServiceImpl in sun-jaxws.xml file with unique name(url-pattern) as "/"

5. It will copy all jax-ws-ri jars into myapp/WEB-INF/lib folder

6. It will start jax-ws-ri server on 9999 port number.

7. It will deploy "myapp" application onto jax-ws-ri server.

JAX-WS-RI Client:

-> JAX-WS-RI implementation has provided 2 tools in jax1.6/bin directory ie.,

1. wsimort
2. wsgen

-> Here wsimport is acting as stub generation tool and wsgen is acting as wsdl generation tool.

Steps to create client app using JAX-WS-RI implementation:

-> Create CalServiceClient folder in below location.

D:\backup\desktop\online class\webservices\class9\examples\jax-ws-ri

-> Generate stubs by executing below command.

D:\backup\desktop\online class\webservices\class9\examples\jax-ws-ri>wsimport -keep -verbose http://localhost:9999/myapp/?wsdl <--|

Note:

-> -keep is optional option and If we this option then it won't delete stub source files.

-> -verbose is also optional option and if we use this option then it will print stacktrace.

-> Create TestClient.java in above CalServiceClient folder.

-> To invoke webservice from TestClient.java then we need 3 things from wsdl ie.,

1. targetNamespace which is in "targetNamespace" attribute at <definitions> tag.
2. service name which is at <service> tag.
3. endpoint-url which is inside <service> tag

TestClient.java:

```
public class TestClient {

    public static void main(String[] arg) throws Exception {

        java.net.URL url = new
java.net.URL("http://localhost:9999/myapp/?wsdl");
```

```
                javax.xml.namespace.QName q = new
javax.xml.namespace.QName("http://www.venkat.com","CalServiceImpl");


                javax.xml.ws.Service s = javax.xml.ws.Service.create(url, q);



              //here we need to pass generated service enpoint interface(SEI)

                com.venkat.CalServiceI ser =
s.getPort(com.venkat.CalServiceI.class);


            int res = ser.add(10,10);


                System.out.println(res);


        }
}
```

-> Compile and run TestClient.java

D:\backup\desktop\online class\webservices\class9\examples\jax-ws-ri\CalServiceClient>javac TestClient.java <--|

D:\backup\desktop\online class\webservices\class9\examples\jax-ws-ri\CalServiceClient>java TestClient <--|

Steps to Create webclient in eclipse:

-> Create DynamicWebProject with project names as CalServiceWebClient.

-> Generate stubs in src folder by following below steps.

1. Get physical location of src folder.

2. Execute below command

D:\backup\desktop\online class\webservices\class9\eclipse_examples\CalServiceWebClient\src>wsimport -keep -verbose http://localhost:9999/myapp/?wsdl <--|

3. Refresh eclipse project.

Note:

When we want generate stubs we need to deploy service on to jax-ws-ri server ie., need to run Publisher.java

-> Copy servlet-api.jar file in WEB-INF/lib folder.

-> Create Cal.jsp in WebContent folder.

Cal.jsp:

<html>

```
<body>
   <%if(request.getAttribute("RESULT") != null) {%>
            <%=request.getAttribute("RESULT")%>
        <%}%>

  <form action="calService">
     FNO: <input type="text" name="fno"/><br/>
          SNO: <input type="text" name="sno"/><br/>

          <input type="submit" value="ADD"/>

  </form>
</body>
</html>
```

-> Create CalServlet.java in src folder.

CalServlet.java:

```
package com.venkat.servlet;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
```

```
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;


public class CalServlet extends HttpServlet {


        protected void doGet(HttpServletRequest request,
                        HttpServletResponse response) throws ServletException,
IOException {



                String fno = request.getParameter("fno");
                String sno = request.getParameter("sno");


                java.net.URL url = new
java.net.URL("http://localhost:9999/myapp/?wsdl");


                javax.xml.namespace.QName q = new
javax.xml.namespace.QName(
                                "http://www.venkat.com", "CalServiceImpl");


                javax.xml.ws.Service s = javax.xml.ws.Service.create(url, q);


                // here we need to pass generated service enpoint interface(SEI)
```

```
            com.venkat.CalServiceI ser =
s.getPort(com.venkat.CalServiceI.class);

            int res = ser.add(Integer.parseInt(fno), Integer.parseInt(sno));

            request.setAttribute("RESULT", res);

            RequestDispatcher rd = request.getRequestDispatcher("cal.jsp");
            rd.forward(request, response);
        }
}
```

-> Create web.xml file in WEB-INF folder.

web.xml:

```
<web-app>
  <welcome-file-list>
   <welcome-file>cal.jsp.jsp</welcome-file>
  </welcome-file-list>
  <servlet>
    <servlet-name>calService</servlet-name>
    <servlet-class>com.venkat.servlet.CalServlet</servlet-class>
  </servlet>
  <servlet-mapping>
```

```
    <servlet-name>calService</servlet-name>
    <url-pattern>/calService</url-pattern>
  </servlet-mapping>
</web-app>
```

-> Deploy application server and use below url to access application.

http://localhost:8082/CalServiceWebClient

Creating webservices using JAX-WS-RI and deploy it onto tomcat server:

-> Download JAX-WS-RI JARS(jaxws-ri-2.2.10.zip) from below location.

https://jax-ws.java.net/2.2.10/

-> Extract some where in our local system and we will get below folder structure

D:\JAXWS-Ri

|_lib -> all jax-ws-ri jars

-> Create Dynamic web project with project names as "JAXWSRICalService".

-> Copy above jax-ws-ri jars into WebConten/WEB-INF/lib folder.

-> Create CalServiceI.java and CalServiceImpl.java in src folder with com.venkat.service package.

CalServiceI.java:

```
package com.venkat.service;

import javax.jws.WebService;

@WebService
public interface CalServiceI {

    public int add(int i,  int  j);

        public int sub(int i,  int  j);
}
```

CalServiceImpl.java:

```
package com.venkat.service;

import javax.jws.WebService;

@WebService(endpointInterface="com.venkat.service.CalServiceI",
```

```
                serviceName="CalServiceImpl",
                                targetNamespace="http://www.venkat.com")
public class CalServiceImpl implements CalServiceI {

   public int add(int i, int j) {return i+j;}

        public int sub(int i, int j) {return i-j;}

}
```

-> Configure JAX-WS-RI skeleton ie., WSServlet in web.xml file with "/calService" url-pattern.

web.xml:

```
<web-app>

  <listener>
    <listerner-class>
      com.sun.xml.ws.transport.http.servlet.WSServletContextListener
    </listerner-class>
  </listener>

  <servlet>
     <servlet-name>WSServlet</servlet-name>
```

```
    <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>WSServlet</servlet-name>
    <url-pattern>/calService</url-pattern>
  </servlet-mapping>

  <!-- <servlet-mapping>
    <servlet-name>WSServlet</servlet-name>
    <url-pattern>/fileService</url-pattern>
  </servlet-mapping>
 -->
 </web-app>
```

-> Create sun-jaxws.xml file in WEB-INF directory and configure our service implementation

in sun-jaxws.xml file

sun-jaxws.xml:

```
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">
```

```
  <endpoint name="cService"
implementation="com.venkat.service.CalServiceImpl" url-pattern="/calService"
/>


  <!-- <endpoint name="fileService"
implementation="com.venkat.service.FileServiceServiceImpl" url-
pattern="/fileService" /> -->


</endpoints>



<!--  http://localhost:8082/JAXWSRICalService/calService?wsdl  -->


<!--  http://localhost:8082/JAXWSRICalService/fileService?wsdl  -->
```

-> Now deploy application into tomcat server and use below url to access webservice.

http://localhost:8082/JAXWSRICalService/calService?wsdl

Creating client for above application:

-> Create Java project with project name as JAXWSRICalServiceClient.

-> Generate STUBS manually in src folder and to do this follow below steps.

1. Get client project physical location of src folder

2. Open command prompt and go to below url and execute below command to generate stubs.

D:\backup\desktop\online class\webservices\class9\eclipse_examples\JAXWSRICalServiceClient\src>wsimport -keep -verbose http://localhost:8082/JAXWSRICalService/calService?wsdl <--|

3. Refresh eclipse project to get generated stubs.

-> Create TestClient.java in src folder with below code to invoke webservices.

TestClient.java:

```
public class TestClient {

    public static void main(String[] arg) throws Exception {

        java.net.URL url = new java.net.URL("http://localhost:8082/JAXWSRICalService/calService?wsdl");

        javax.xml.namespace.QName q = new javax.xml.namespace.QName("http://www.venkat.com","CalServiceImpl");
```

```
                javax.xml.ws.Service s = javax.xml.ws.Service.create(url, q);



              //here we need to pass generated service enpoint interface(SEI)

                com.venkat.CalServiceI ser =
s.getPort(com.venkat.CalServiceI.class);


            int res = ser.add(10,10);


                System.out.println(res);


        }
}
```

Metro Implementation:

-> Metro implementation webservice development is same as JAX-WS-RI implementation,The only difference is jar files.

-> To work with metro implementation we need to dowload metro jars(metro-standalone-2.3.1.zip) by using below url.

https://metro.java.net/2.3.1/

-> Extract above downloaded zip file some where in our local system.

D:\Metro -> lib(metro jars)

Steps to create metro implementation webservice in eclipse:

-> Create Dynamic web project with project name as "MetroCalService"

-> Create CalServiceI.java and CalServiceImpl.java src folder with com.venkat.service package.

CalServiceI.java:

```
package com.venkat.service;

import javax.jws.WebService;

@WebService
public interface CalServiceI {

   public int add(int i,  int  j);
        public int sub(int i,  int  j);
}
```

CalServiceImpl.java:

```
package com.venkat.service;

import javax.jws.WebService;

@WebService(endpointInterface="com.venkat.service.CalServiceI",
                serviceName="CalServiceImpl",
                        targetNamespace="http://www.venkat.com")
public class CalServiceImpl implements CalServiceI {

   public int add(int i, int j) {return i+j;}

       public int sub(int i, int j) {return i-j;}

}
```

-> Create sun-jaxws.xml file in project WEB-INF directory.

sun-jaxws.xml:

```
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
version="2.0">
```

```
 <endpoint name="cService"
implementation="com.venkat.service.CalServiceImpl" url-pattern="/calService"
/>


 </endpoints>
```

-> Update web.xml file like below.

web.xml:

```
<web-app>


 <listener>
    <listerner-class>
       com.sun.xml.ws.transport.http.servlet.WSServletContextListener
    </listerner-class>
 </listener>


 <servlet>
     <servlet-name>WSServlet</servlet-name>
     <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-
class>
 </servlet>


 <servlet-mapping>
```

```
    <servlet-name>WSServlet</servlet-name>
    <url-pattern>/calService</url-pattern>
  </servlet-mapping>

 </web-app>
```

-> Copy all Metro jars which are in extracted metro/lib folder except "databinding" jars into our project WEB-INF/lib folder.

-> Deploy application into server and use url to test webservice wheather it is deployed or not.

http://localhost:8082/MetroCalService/calService?wsdl

Metro implementation client in eclipse:

-> Create Dynamic Web Project with project name as "MetroCalServiceClient".

-> Generate stubs in src folder by following below steps.

1. Get physical location of src folder

2. Open command prompt and go to physical location of src folder and execute below command.

D:\backup\desktop\online class\webservices\class9\eclipse_examples\MetroCalServiceClient\src>wsimport -impl -verbose http://localhost:8082/MetroCalService/calService?wsdl <-|

3. Refresh Eclipe project.

-> Create TestClient.java in src folder with below code.

TestClient.java:

```
public class TestClient {

    public static void main(String[] arg) throws Exception {

        java.net.URL url = new java.net.URL("http://localhost:8082/MetroCalService/calService?wsdl");

        javax.xml.namespace.QName q = new javax.xml.namespace.QName("http://www.venkat.com","CalServiceImpl");

        javax.xml.ws.Service s = javax.xml.ws.Service.create(url, q);

        //here we need to pass generated service enpoint interface(SEI)
```

```
                com.venkat.CalServiceI ser =
s.getPort(com.venkat.CalServiceI.class);


            int res = ser.add(10,10);


                System.out.println(res);


        }
}
```

File uploading and downloading webservice using metro implementation:

-> Create Dynamic Web Project with project name as "MetroFileService".

-> Create FileDetails.java, FileServiceI.java and FileServiceImpl.java in src folder with com.venkat.service package.

FileDetails.java:

```
package com.venkat.service;


public class FileDetails {


        private String fileName;

        private String fileType;
```

```
    private byte[] fileContent;


    public String getFileName() {
        return fileName;
    }
    public void setFileName(String fileName) {
        this.fileName = fileName;
    }
    public String getFileType() {
        return fileType;
    }
    public void setFileType(String fileType) {
        this.fileType = fileType;
    }
    public byte[] getFileContent() {
        return fileContent;
    }
    public void setFileContent(byte[] fileContent) {
        this.fileContent = fileContent;
    }
}
```

FileServiceI.java:

```
package com.venkat.service;

import javax.jws.WebService;

@WebService
public interface FileServiceI {

	public boolean uploadFile(FileDetails file);
	public FileDetails downloadFile();
}
```

FileServiceImpl.java:

```
package com.venkat.service;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

import javax.jws.WebService;

@WebService(endpointInterface = "com.venkat.service.FileServiceI",
serviceName = "FileServiceImpl", targetNamespace = "http://mvr.com")
```

```
public class FileServiceImpl implements FileServiceI {

    public boolean uploadFile(FileDetails fd) {

        String fileName = fd.getFileName();
        String fileType = fd.getFileType();
        byte[] fileData = fd.getFileContent();

        try {
            File f = new File("D:/clientUploads/" + fileName + "." +
fileType);
            FileOutputStream fos = new FileOutputStream(f);
            fos.write(fileData);
            fos.flush();
            fos.close();
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    public FileDetails downloadFile() {

        FileDetails fd = new FileDetails();
```

```
            File f = new File("E:/clientDownloads/abc.txt");
            int fileLength = (int) f.length();
            try {

                FileInputStream fis = new FileInputStream(f);

                byte[] fileContent = new byte[fileLength];

                fis.read(fileContent);
                fis.close();

                fd.setFileName("abc");
                fd.setFileType("txt");
                fd.setFileContent(fileContent);

            } catch (IOException e) {
                e.printStackTrace();
                return null;
            }

            return fd;
        }
}
```

-> Copy all metro jars into WEB-INF/lib folder except binding jars.

-> Configure skeleton in web.xml file.

web.xml:

```
<web-app>

  <listener>
     <listerner-class>
        com.sun.xml.ws.transport.http.servlet.WSServletContextListener
     </listerner-class>
  </listener>

  <servlet>
      <servlet-name>WSServlet</servlet-name>
      <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
  </servlet>

  <servlet-mapping>
     <servlet-name>WSServlet</servlet-name>
     <url-pattern>/fileService</url-pattern>
  </servlet-mapping>

 </web-app>
```

-> Create sun-jaxws.xml in WEB-INF folder.

sun-jaxws.xml:

```
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
version="2.0">

  <endpoint name="Stservice"

   implementation="com.venkat.service.FileServiceImpl"

    url-pattern="/fileService" />

</endpoints>
```

-> Deploy application into server and use below url to invoke webservice.

http://localhost:8082/MetroFileService/fileService?wsdl

File service client using metro implementation:

-> Creat Dynamic webproject with project MetroFileServiceClient.

-> Generate stubs in src folder by following below steps.

1. Get the physical location of src folder.

2. Open command prompt and go to physical location of src folder.

D:\backup\desktop\online class\webservices\class9\eclipse_examples\MetroFileService\src>wsimport -keep -verbose http://localhost:8082/MetroFileService/fileService?wsdl <-|

3. Refresh eclipse project.

-> Create TestClient.java in src folder with below code to invoke webservice.

-> Before run TestClient.java create necessary locations.

TestClient.java:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;

import com.venkat.service.FileDetails;



public class TestClient {
```

```
public static void main(String[] arg) throws Exception {



        java.net.URL url = new
java.net.URL("http://localhost:8082/MetroFileService/fileService?wsdl");


        javax.xml.namespace.QName q = new
javax.xml.namespace.QName("http://mvr.com","FileServiceImpl");


        javax.xml.ws.Service s = javax.xml.ws.Service.create(url, q);



       //here we need to pass generated service enpoint interface(SEI)

        com.mvr.FileServiceI ser =
s.getPort(com.mvr.FileServiceI.class);


        //file upload

        FileDetails fd = new FileDetails();

        File file = new File("E:\\uploadfile\\Jellyfish.jpg");

        FileInputStream fis = new FileInputStream(file);


        byte[] fileCont = new byte[(int)file.length()];

        fis.read(fileCont);

        fis.close();
```

```
fd.setFileName("Jellyfish");
fd.setFileType("jpg");
fd.setFileContent(fileCont);

boolean flag = ser.uploadFile(fd);

if(flag)
        System.out.println("File has uploaded successfully...");
else
        System.out.println("File has failed to upload...");

//file download
FileDetails fDetails = ser.downloadFile();
String fname = fDetails.getFileName();
String fType = fDetails.getFileType();
byte[] fContent = fDetails.getFileContent();


File f1 = new File("E:\\upload\\"+fname+"."+fType);
FileOutputStream fos1 = new FileOutputStream(f1);

fos1.write(fContent);
fos1.flush();
fos1.close();
```

```
                System.out.println("File has dowloaded successfully in
..."+f1.getPath()+"  location");
            }
}
```

Note:

-> We will MTOM(SOAP Message Transmission Optimization Mechanism) to minimize the file size in the network while transfer

the files in between client and server.

Axis2 implementation:

-> As we know eclipse having axis1 plugin, axis2 plugin and apache cxf plugin.

-> But eclipse having only axis1 jars but not axis2 and apache CXF jars.

-> So before working with eclipse to create webservice or webservice client using axis2 implementation or apache cxf

implementation then first we need to axis2 jars or apache cxf jars to eclipse. To do this we need to follow

below steps.

Adding axis2 jars to eclipse:

-> Download axis2 jars(axis2-1.6.3-bin.zip) using below url

https://axis.apache.org/axis2/java/core/download.cgi

-> Extract above dowloaded zip file some where in our local system

-> Add axis2 jars to eclipse by following below steps.

Windows -> preferences -> expand webservices folder and select Axis2 preferences -> click on browse button to

browse upto extracted axis2 root folder ie., D:\backup\mydata\webservicesandxml\axis2-1.6.2 -> click on apply and ok button.

Creating axis2 webservice in eclipse:

-> Create Dynamic webproject with project name as Axis2CalService.

-> Create CalService.java in src folder.

CalService.java:

```
package com.venkat.service;
```

```
public class CalService {


        public int add(int i, int j) {
                return i+j;
        }
}
```

-> Convert CalService.java to webservice using axis2 implementation by following below steps.

Right click on CalService.java -> new -> other... -> expand webservices folder -> select webservice ->click on next

button -> change webservice runtime to Apache axis2 -> click on finish.

-> Deploy webservice into server.

-> To test webservice use below url.

http://localhost:8082/Axis2CalService

-> To get wsdl file use below url.

http://localhost:8082/Axis2CalService/services/CalService?wsdl

Note:

-> In axis2 we will axis2.xml file as configuration file and it is same as like server-config.wsdd file in axis1 implementation

and sun-jaxws.xml in jax-ws-ri and metero implementation.

-> axis2 skeleton will use this file to identify service class when ever client makes a request and to generate wsdl file.

-> Axis2 webservice client in eclipse:

-> Create Dynamic Webproject with project name Axis2CalServiceClient.

-> Generate stubs in src folder by following below steps.

right click src folder -> new -> others... -> expand webservices folder and select webservice client -> click on next ->

enter wsdl url ie., http://localhost:8082/Axis2CalService/services/CalService?wsdl and

change webservice runtime to Apache axis2 -> click on finish.

-> Create TestClient.java in src folder with below code to invoke webservice.

TestClient.java:

```
import com.venkat.service.CalServiceStub;
```

```
public class TestClient {

    public static void main(
                    String[] args)
     throws Exception{

        CalServiceStub stub = new CalServiceStub();

        //Code to invoke add()
        CalServiceStub.Add params =
         new CalServiceStub.Add();

        params.setI(13);
        params.setJ(10);

        CalServiceStub.AddResponse
        res =  stub.add(params);

        int result = res.get_return();

        System.out.println(result);
    } }
```

-> Apache CXF implementation webservice:

-> Create Dynamic Webproject with project name as CXFCalService

-> Create CalServiceI.java and CalServiceImpl.java in src folder with jax-ws annotations.

CalServiceI.java:

```
package com.venkat.service;

import javax.jws.WebService;

@WebService
public interface CalServiceI {

   public int add(int i,  int  j);
        public int sub(int i,  int  j);
}
```

CalServiceImpl.java:

```
package com.venkat.service;
```

```
import javax.jws.WebService;

@WebService(endpointInterface="com.venkat.service.CalServiceI",
                    serviceName="CalServiceImpl",
                                targetNamespace="http://www.venkat.com")
public class CalServiceImpl implements CalServiceI {

   public int add(int i, int j) {return i+j;}

        public int sub(int i, int j) {return i-j;}

}
```

-> Download and copy apache cxf jars into WEB-INF/lib folder.

-> To download apache cxf s/w(apache-cxf-2.7.18.zip) need to use below link.

http://www.apache.org/dyn/closer.lua/cxf/2.7.18/apache-cxf-2.7.18.zip

-> Extract downloaded zip file some where in our local system.

D:\apache-cxf-2.7.18\

|

|__lib(apache cxf jars)

|__bin(java2ws.bat and wdl2java.bat)

|

|__samples

|

-> Configure cxf skeleton in web.xml file.

web.xml:

```
<web-app>
  <servlet>
    <servlet-name>cxf</servlet-name>
    <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>cxf</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>WEB-INF/cxf-servlet.xml</param-value>
  </context-param>
```

```
  <listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
</web-app>
```

-> Create configuration file called cxf-servlet.xml in WEB-INF folder.

-> Get existing cxf-servlet.xml file from dowloaded folder ie., from D:\apache-cxf-2.7.18\samples folder.

-> Better copy cxf-servlet.xml file which is having <jaxws:enpoint> tag.

cxf-servlet.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jaxws="http://cxf.apache.org/jaxws"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">
        <!-- <import resource="classpath:META-INF/cxf/cxf.xml" />
        <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
        <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />
```

-->

<jaxws:endpoint id="cService"
implementor="com.venkat.service.CalServiceImpl" address="/calService"/>

</beans>

-> Deploy application into service and use below url to test it.

http://localhost:8082/CXFCalService/services/calService?wsdl

Apache CXF client:

-> Create Dynamic web project with project name as CXFCalServiceClient.

-> Copy all cxf jars into WEB-INF/lib folder.

-> Generate stubs in src folder by using below command.

-> To generate stubs we need to use apache cxf stub generation tool which is in extracted apache cxf "bin" folder.

-> Set path variable in environment variables uptp extracted apache cxf/bin folder.

-> Get src physical location, open command prompt and go to physical location of src folder.

-> Execute below command to generate stubs.

D:\backup\desktop\online class\webservices\class9\eclipse_examples\CXFCalServiceClient\src>wsdl2java -impl http://localhost:8082/CXFCalService/services/calService?wsdl <-|

-> Refresh eclipse project to get generated stubs.

-> Create cxf.xml in src folder by configuring generated stubs.

cxf.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">


    <!--
      Here "id" attribute can have any name
      "serviceClass" attribute can take generated SEI
```

```
        "addredd" will take wsdl url
       -->

      <jaxws:client
        id="myclient"
            serviceClass="com.venkat.service.CalServiceI"
address="http://localhost:8082/CXFCalService/services/calService?wsdl">

      </jaxws:client>
</beans>
```

-> Create TestClient.java in src folder with below code to invoke webservice.

TestClient.java:

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.venkat.service.CalServiceI;


public class TestClient {

      public static void main(String[] args) {
```

```
        //It should be take client side configuration file name as argument.
        ApplicationContext ctx = new
ClassPathXmlApplicationContext("cxf.xml");



        //It should be taken as cxf.xml file <client> tag "id" attribute value
        CalServiceI ws = (CalServiceI)ctx.getBean("myclient");


        int res = ws.add(12, 12);


        System.out.println(res);



    }
}
```

Another way of creating CXF client:

-> Create Dynamic web project with project name as CXFCalServiceClient.

-> Copy all cxf jars into WEB-INF/lib folder.

-> Generate stubs in src folder by using below command.

-> To generate stubs we need to use apache cxf stub generation tool which is in extracted apache cxf "bin" folder.

-> Set path variable in environment variables uptp extracted apache cxf/bin folder.

-> Get src physical location, open command prompt and go to physical location of src folder.

-> Execute below command to generate stubs.

D:\backup\desktop\online class\webservices\class9\eclipse_examples\CXFCalServiceClient\src>wsdl2java -impl http://localhost:8082/CXFCalService/services/calService?wsdl <-|

-> Refresh eclipse project to get generated stubs.

-> Create TestClient.java in src folder with below Code.

TestClient.java:

```
import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;

import com.venkat.service.CalServiceI;
```

```
public class TestClient {

    public static void main(String[] s) {

        JaxWsProxyFactoryBean factory =
                new JaxWsProxyFactoryBean();

        factory.setAddress(
    "http://localhost:8082/CXFCalService/services/calService");

        factory.setServiceClass(CalServiceI.class);

        CalServiceI service = (CalServiceI) factory.create();

        System.out.println(service.add(12, 1));
    }
}
```

Apche CXF webservices with eclipse:

-> As we know eclipse having apache cxf plugin, but it doesn't have cxf jar files.

-> To add cxf jars to eclipse we need to follow below steps.

-> Download apache cxf jars.

-> follow below to add dowloaded jars.

windows -> preferences -> expand webservices folder and select cxf 2.x preferences -> click on ADD button to

browse upto extracted apache cxf root folder ie., D:\backup\mydata\webservicesandxml\apache-cxf-2.7.7 ->

enable apache cxf check box anf click on ok.

Steps to follow to create apache cxf webservices using eclipse:

-> Create DynamicWebProject web project with project name as CXFWeatherService.

-> Copy ojdbc14.jar file into WEB-INF/lib folder.

-> Create Weather.java, WeatherServiceI.java, WeatherServiceImpl.java, WeatherDAO.java in src folder.

Weather.java:

```
package com.venkat.service;

public class Weather {

  private int cid;
```

```
private String cname;
private int temp;



public int getCid() {
    return cid;
}

public void setCid(int cid) {
    this.cid = cid;
}

public String getCname() {
    return cname;
}

public void setCname(String cname) {
    this.cname = cname;
}

public int getTemp() {
    return temp;
}
```

```
  public void setTemp(int temp) {
      this.temp = temp;
  }
}
```

WeatherDAO.java:

```
package com.venkat.dao;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import com.venkat.service.Weather;

public class WeatherDAO {

        private Connection getConnection() {
                Connection con = null;

                try {
                        Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
                con = DriverManager.getConnection(
                                "jdbc:oracle:thin:@localhost:1521:XE",
"system", "xe");

            } catch (SQLException e) {
                e.printStackTrace();
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
            return con;
        }

        public Weather getWeatherByCityId(int cityId) {
            Weather w = null;

            try {
                Connection con = getConnection();
                Statement stmt = con.createStatement();
                ResultSet rs = stmt.executeQuery("select * from weather
where cid="
                                + cityId);

                if (rs.next()) {
                    int cid = rs.getInt(1);
                    String cname = rs.getString(2);
```

```
                int temp = rs.getInt(3);
                w = new Weather();
                w.setCid(cid);
                w.setCname(cname);
                w.setTemp(temp);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return w;

    }

    public int getTempByWeather(Weather w) {
        String cname = w.getCname();
        int temp = -100;

        try {
            Connection con = getConnection();
            Statement stmt = con.createStatement();
            ResultSet rs = stmt
                    .executeQuery("select * from weather where
cityname='"
                            + cname + "'");
```

```
                if (rs.next()) {
                        temp = rs.getInt(3);

                }
            } catch (Exception e) {
                e.printStackTrace();
            }
            return temp;
        }
}
```

WeatherServiceI.java:

```
package com.venkat.service;

public interface WeatherServiceI {

    public Weather getWeatherByCityId(int cityId);

    public int getTempByWeather(Weather w);
}
```

WeatherServiceImpl.java:

```
package com.venkat.service;

import com.venkat.dao.WeatherDAO;

public class WeatherServiceImpl implements WeatherServiceI {

	WeatherDAO weatherDAO = new WeatherDAO();


	public Weather getWeatherByCityId(int cityId) {

		return weatherDAO.getWeatherByCityId(cityId);
	}

	public int getTempByWeather(Weather w) {
		return weatherDAO.getTempByWeather(w);
	}
}
```

-> Convert weatherServiceImpl.java into apache cxf webservice by following below step.

Right click on WeatherServiceImpl.java -> new -> other... -> expand webservices and select webservice -> click on

Next -> change webservice runtime to apache cxf 2.x -> click on finish.

-> Use below url to test webservice.

http://localhost:8082/CXFWeatherService/services/WeatherServiceImplPort?wsdl

Note:

-> To run this service we need to create WEATHER table with CID, CITYNAME and TEMP columns.

ApacheCXF client using eclipse:

-> Create DynamicWebProject with project name as CXFWeatherServiceClient.

-> Generate stubs by following below steps.

ritht click on src folder -> new -> others... -> expand webservices and select webservice client -> click on next ->

change webservice runtime to apache cxf and enter wsdl url ie., http://localhost:8082/CXFWeatherService/services/WeatherServiceImplPort?wsdl -> click on finish.

-> Create cxf.xml file and configure generated SEI in cxf.xml file.

cxf.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

    <!--
      Here "id" attribute can have any name
      "serviceClass" attribute can take generated SEI
      "addredd" will take wsdl url
     -->

    <jaxws:client
       id="myclient"
            serviceClass="com.venkat.service.WeatherServiceImpl"

address="http://localhost:8082/CXFWeatherService/services/WeatherServiceIm
plPort?wsdl">

    </jaxws:client>
</beans>
```

-> Create TestClient.java in src folder.

TestClient.java:

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.venkat.service.Weather;
import com.venkat.service.WeatherServiceImpl;



public class TestClient {

	public static void main(String[] args) {

		//It should be take client side configuration file name as argument.
		ApplicationContext ctx = new
ClassPathXmlApplicationContext("cxf.xml");



		//It should be taken as cxf.xml file <client> tag "id" attribute value
		WeatherServiceImpl ws =
(WeatherServiceImpl)ctx.getBean("myclient");
```

```
            Weather res = ws.getWeatherByCityId(100);


            System.out.println(res.getTemp());
            System.out.println(res.getCname());
            System.out.println(res.getCid());
        }
}
```

-> Run above application to invoke service.

Spring integration in apache CXF:

-> In the above example WeatherDAO is dependent for WeatherServiceImpl.java, We will inject WeatherDAO object into

WeatherServiceImpl object by using below configutations in cxf-beans.xml file.

1. Change WeatherServiceImpl.java like below

WeatherServiceImpl.java:

```
public class WeatherServiceImpl implements WeatherServiceI {


        WeatherDAO weatherDAO;// = new WeatherDAO();
```

```
    public WeatherDAO getWeatherDAO() {
        return weatherDAO;
    }

    public void setWeatherDAO(WeatherDAO weatherDAO) {
        this.weatherDAO = weatherDAO;
    }

    public Weather getWeatherByCityId(int cityId) {

        return weatherDAO.getWeatherByCityId(cityId);
    }

    public int getTempByWeather(Weather w) {
        return weatherDAO.getTempByWeather(w);
    }
}
```

2. Change cxf-beans.xml file like below.

cxf-beans.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jaxws="http://cxf.apache.org/jaxws"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">
        <import resource="classpath:META-INF/cxf/cxf.xml" />
        <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
        <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />


        <jaxws:endpoint id="weatherservice" implementor="#weatherService"
                address="/WeatherServiceImplPort"/>


        <bean id="weatherService"
class="com.venkat.service.WeatherServiceImpl">
          <property name="weatherDAO" ref="weatherDAO"/>
        </bean>

        <bean id="weatherDAO" class="com.venkat.dao.WeatherDAO"/>

</beans>
```

Tracing SOAP request and response:

-> Some times werequirement to print soap request and responses in the console or log file at service and client side.

-> To do this apache cxf has given 2 classes ie.,

1. org.apache.cxf.interceptor.LoggingInInterceptor.
2. org.apache.cxf.interceptor.LoggingOutInterceptor.

-> We can these classes both at client side and service side.

-> Here LoggingInInterceptor will print incoming message in console at service and client side.

-> Here LoggingOutInterceptor will print out going message in the console at service and client side.

-> If we want print log messages at service side then we will configure these 2 classes in cxf-beans.xml file like below.

cxf-beans.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jaxws="http://cxf.apache.org/jaxws"
```

```
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

    <import resource="classpath:META-INF/cxf/cxf.xml" />

    <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />

    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />



    <jaxws:endpoint id="weatherservice" implementor="#weatherService"

        address="/WeatherServiceImplPort">


        <jaxws:inInterceptors>

          <ref bean="logInInterceptor"/>

        </jaxws:inInterceptors>


        <jaxws:outInterceptors>

          <ref bean="outInInterceptor"/>

        </jaxws:outInterceptors>


    </jaxws:endpoint>


    <bean id="weatherService"
class="com.venkat.service.WeatherServiceImpl">

      <property name="weatherDAO" ref="weatherDAO"/>

    </bean>
```

```
<bean id="weatherDAO" class="com.venkat.dao.WeatherDAO"/>

<bean id="logInInterceptor"
class="org.apache.cxf.interceptor.LoggingInInterceptor"/>
<bean id="outInInterceptor"
class="org.apache.cxf.interceptor.LoggingOutInterceptor"/>
</beans>
```

-> If we want to trace soap request and response at client side then we will LoggingInInterceptor anf LoggingOutInterceptor

in client cxf.xml file like below.

cxf.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jaxws="http://cxf.apache.org/jaxws"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">
<import resource="classpath:META-INF/cxf/cxf.xml" />
<import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
<import resource="classpath:META-INF/cxf/cxf-servlet.xml" />
```

```
<jaxws:endpoint id="weatherservice" implementor="#weatherService"
        address="/WeatherServiceImplPort">

        <jaxws:inInterceptors>
          <ref bean="logInInterceptor"/>
        </jaxws:inInterceptors>

        <jaxws:outInterceptors>
          <ref bean="outInInterceptor"/>
        </jaxws:outInterceptors>

</jaxws:endpoint>

<bean id="weatherService"
class="com.venkat.service.WeatherServiceImpl">
    <property name="weatherDAO" ref="weatherDAO"/>
</bean>

<bean id="weatherDAO" class="com.venkat.dao.WeatherDAO"/>

<bean id="logInInterceptor"
class="org.apache.cxf.interceptor.LoggingInInterceptor"/>
<bean id="outInInterceptor"
class="org.apache.cxf.interceptor.LoggingOutInterceptor"/>
</beans>
```

Top down approach webservice development:

Webservice security:

Difference between JAX-RPC and JAX-WS:

JAX-RPC:

-> It doesn't support any collection type or any array type as webservice method return type or parameter type.

-> It uses Http1.1 version protocol

-> It uses SOAP1.1 version protocol

-> It uses WSDL1.1 version

-> It is introduced with JDK1.4

-> It doesn't support annotations

JAX-WS:

-> It sopports any collection type or any array type as webservice method return type or parameter type.

-> It Uses HTTP1.1 version protocol

-> It uses both SOAP1.1 and SOAP1.2 version protocol

-> It uses both WSDL1.1 and 2.0

-> It is introduced with JDK1.5

-> It supports annotations

Restful webservices(JAX-RS):

-> JAX-RS is a specification and using we can develope restful webservices and webservices clients.

-> It is introduced in JDK1.6 version.

-> It has lot of 3rd party implementations are available ie.,

1. Jersey implementation from sun microsystem
2. Restlet implementation from redhat
3. Resteasy implementation from jerome louvel
4. Apache wink implementation from apache foudation
5. Apache CXF implementation from apache foundation.

-> Here ApacheCXF supports spring integration where as Apache wink doesn't support spring integration.

Differences between SOAP and RESTFUL webservices:

-> SOAP webservices supports only one format ie., SOAP request(XML) and SOAP response(xml) to share information between

client and service.

-> Restful webservices supports many formats to share information between client and service ie., xml, json, plain text,

and html etc.,

-> SOAP based webservices follows heavy weight architecture becuase some validation will be happned at client side and

server side. So that it may performance issue.

-> RESTFUL webservices follows leight weight architecture becuase any validations will never happned at client side and

server side. So that it increases the performance of the application.

-> Here soap webservice supports 3 types of securities ie., Username token security, Message level security and

server level security.

-> RESTFUL webservices only 2 types of securities ie., Basic Authentication security(OAuth Security) and server level security,

It doesn't supports message level security.

-> Restful webservices are mainly used in mobile applications becuase it is leight weight components.

-> SOAP based webservices are mainly used in desktop applications and web applications.

JSON(Java script object notations):

-> JSON is universal notation because it can be understable by all the programing languages like xml.

-> To interact with JSON from java application, we need Java JSON api's.

-> Here we have 3 java JSON API'S are available ie.,

-> Jackson api

-> GSON(Google son) api

-> Simple JSON api

-> Using any of these api's we can convert java object to json and json to java object.

Convert java object to JSON and JSON to java obect:

-> Create java project in eclipse with Project is JSONJava

-> Download jackson api jars and put them into build path.

-> Create Student.java, Address.java and JavaToJson.java in src folder.

Student.java:

```
public class Student {
	private int sid;
	private String name;
	private int marks;
	public int getSid() {
		return sid;
	}
```

```
	public void setSid(int sid) {
		this.sid = sid;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getMarks() {
		return marks;
	}
	public void setMarks(int marks) {
		this.marks = marks;
	}
}
```

Address.java:

```
public class Address {

	private int flatno;
	private String buildingName;
```

```
    private String area;
    public int getFlatno() {
        return flatno;
    }
    public void setFlatno(int flatno) {
        this.flatno = flatno;
    }
    public String getBuildingName() {
        return buildingName;
    }
    public void setBuildingName(String buildingName) {
        this.buildingName = buildingName;
    }
    public String getArea() {
        return area;
    }
    public void setArea(String area) {
        this.area = area;
    }
}
```

JavaToJson.java:

```
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;


import org.codehaus.jackson.JsonGenerationException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;



public class JavaToJson {


	public static void main(String[] args) throws JsonGenerationException,
JsonMappingException, IOException {


		Student student = new Student();


		student.setSid(100);
		student.setName("venkat");
		student.setMarks(400);


		Address address = new Address();
		address.setFlatno(101);
		address.setArea("hyd");
		address.setBuildingName("nitin residency");
```

```
        student.setAddress(address);

        ObjectMapper mapper = new ObjectMapper();

        mapper.writeValue(new File("Student.json"), student);
        System.out.println("Json generated...");

    }}
```

-> Create another java class called JsonToJava.java in src folder.

JsonToJava.java:

```
import java.io.File;
import java.io.IOException;

import org.codehaus.jackson.JsonGenerationException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;


public class JsonToJava {
```

```
    public static void main(String[] args) throws JsonGenerationException,
JsonMappingException, IOException {


            ObjectMapper mapper = new ObjectMapper();

            Student std = mapper.readValue(new File("Student.json"),
Student.class);


            System.out.println(std.getSid());

            System.out.println(std.getName());

            System.out.println(std.getMarks());


            Address ad = std.getAddress();


            System.out.println("\nAddress: ");

            System.out.println(ad.getFlatno());

            System.out.println(ad.getBuildingName());

            System.out.println(ad.getArea());

    }

}
```

REST(Representational State Transform):

-> Converting object state into different formats ie., xml, json etc is called as REST.

-> In restful webservices, service class look like a servlet becuase servlet can be accessed by url-pattern and even

restful webservice class is also accessed by url.

JAX-RS annotation:

-> JAX-RS having lot of annotations and these annotations we will use in restful webservice service class

irrespective of the implementation.

1. @Path(javax.ws.rs.Path)
2. @GET(javax.ws.rs.GET)
3. @POST(javax.ws.rs.POST)
4. @PUT(javax.ws.rs.PUT)
5. @DELETE(javax.ws.rs.DELETE)
6. @QueryParam(javax.ws.rs.QueryParam)
7. @PathParam(javax.ws.rs.PathParam)
8. @MatrixParam(javax.ws.rs.MatrixParam)
9. @FormParam(javax.ws.rs.FormParam)
10. @Consumes(javax.ws.rs.Consumes)
11. @Produces(javax.ws.rs.Produces)

JAX-RS architecture:

@Path:

-> To define class level path and method level path we will use this annotation.

```
   @Path("/helloService")
 public class HelloService {

    @Path("/sayHello")
    public String sayHello(String name) {
            return "Hello... "+name;
          }
 }
```

@QueryParam, @PathParam, @MatrixParam and @FormParam:

-> If our rest method parameters are the predefined types then we will use any one of these annotation to define the parameter.

-> These annotations are called parameter level annotations.

@QueryParam:

Ex1:

```
@Path("/helloService")
public class HelloService {

   @Path("/sayHello")
   public String sayHello(@QueryParam("name1") String name) {
          return "Hello... "+name;
        }
}
```

-> If the rest method parameter is the @QuertParam then client will use query string to send the parameters like below.

http://localhost:8080/webappname/rest/helloService/sayHello?name1=venkat

Ex2:

```
@Path("/helloService")
public class HelloService {

   @Path("/sayHello")
```

```
    public String sayHello(@QueryParam("name1") String name,
@QueryParam("age") int age) {
            return "Hello... "+name+" You age is "+age;
        }
 }
```

http://localhost:8080/webappname/rest/helloService/sayHello?name1=venkat&age=32

@MatrixParam:

Ex1:

```
 @Path("/helloService")
 public class HelloService {

    @Path("/sayHello")
    public String sayHello(@MatrixParam("name1") String name,
@MatrixParam("age") int age) {
            return "Hello... "+name+" You age is "+age;
        }
 }
```

http://localhost:8080/webappname/rest/helloService/sayHello;name1=venkat;age=32

@PathParam:

Ex1:

```
@Path("/helloService")
public class HelloService {

    @Path("/sayHello/{name1}/{age}")
    public String sayHello(@PathParam("name1") String name,
@PathParam("age") int age) {
            return "Hello... "+name+" You age is "+age;
        }
}
```

-> If rest service methods having Path parameters then client will send those parameters along with path itself like below.

http://localhost:8080/webappname/rest/helloService/sayHello/venkat/32

Ex2:

```
@Path("/helloService")
public class HelloService {
```

```
    @Path("{name1}/{age}")
    public String sayHello(@PathParam("name1") String name,
@PathParam("age") int age) {
            return "Hello... "+name+" You age is "+age;
        }
 }
```

http://localhost:8080/webappname/rest/helloService/venkat/32

Ex3:

-> Some times we can also mix these parameters.

```
 @Path("/helloService")
 public class HelloService {

    @Path("/sayHello/{name1}")
    public String sayHello(@PathParam("name1") String name,
@QueryParam("age") int age) {
            return "Hello... "+name+" You age is "+age;
        }
 }
```

http://localhost:8080/webappname/rest/helloService/venkat?age=32

@FormParam:

-> If we want to invoke rest method directly from form ie., html or jsp then we will these parameters.

```
@Path("/helloService")
public class HelloService {

    @POST
    @Path("/sayHello")
    public String sayHello(@FormParam("name1") String name,
@FormParam("age") int age) {
            return "Hello... "+name+" You age is "+age;
        }
}
```

-> If client wants to invoke above service method then client will design form like below.

```
<html>
   <body>
        <form action="rest/helloService/sayHello" method="POST">
```

```
            Enter Name: <input type="text" name="name1"/><br/>
                Enter Age: <input type="text" name="age"/><br/>
                <input type="submit" value="send"/>
        </form>
    </body>
</html>
```

Note:

-> There is no rule where we need to use @QueryParam, @PathParam, @MatrixParam and @FormParam, We can use any

annotation at any situation, But if rest method having @POST annotation then we can't use @QueryParam becuase

query string will never accept if method type is @POST.

@Consumes:

-> If rest method having user defined type parameter then we will @Consumes annotation to specify that type ie.,

JSON, XML, PLAIN TEXT or HTML etc.

Ex1:

```
@Path("/helloService")
```

```
public class HelloService {


   @POST
   @Path("/sayHello")
        @Consumes("application/xml")
   public String saveStudent(Student student) {
           return "Hello...";
        }
}
```

Ex2:

```
@Path("/helloService")
public class HelloService {


   @POST
   @Path("/sayHello")
        @Consumes("application/json")
   public String saveStudent(Student student) {
           return "Hello... ";
        }
}
```

Ex3:

```
@Path("/helloService")
public class HelloService {


   @POST
   @Path("/sayHello")
        @Consumes("{application/json,application/xml}")
   public String saveStudent(Student student) {
          return "Hello... ";
        }
}
```

Note:

-> If rest method parameter type user defined type then we can't send that parameters along with url instead we will

send through request body.

-> If rest method parameter is user defined type then that method type should be of type @POST or @PUT or @DELETE, but

we can't use @GET.

-> If rest method parameter cosuming in xml format, then client needs to pass that parameter in only xml format, he can't

use any other formats like json, plain text, html etc.

@Produces:

-> The rest method can accept only below types as a return types ie.,.

-> String.

-> StringBuffer.

-> StringBuilder.

-> User defined types.

-> Response

-> It will never any primitive type or wrapper types as rest method return type.

-> If rest method return type is User defined type then we need to use @Produces annotation to specify the producing format

ie., JSON, XML, HTML etc.

Ex1:

```
@Path("/helloService")
public class HelloService {

   @POST
   @Path("/sayHello")
       @Consumes({"application/json", "application/xml")
       @Produces("application/xml")
   public Employee saveStudent(Student student) {
```

```
        Employee e = new Employee();
              ---
              ---
              ---

        return e;
    }
}
```

Ex2:

```
@Path("/helloService")
public class HelloService {

  @POST
  @Path("/sayHello")
      @Consumes("application/json")
      @Produces({"application/xml","application/json"})
      public Employee saveStudent(Student student) {

        Employee e = new Employee();
              ---
              ---
```

```
                ---


          return e;
        }
}
```

-> If we want to send some status code to the client in addition to return value(predefined typed or user defined type),

then we will use method return type as Response.

Ex3:

```
----
@Path("/helloService")
public class HelloService {

  @POST
  @Path("/sayHello")
      @Consumes("application/json")
      @Produces({"application/xml","application/json"})
      public Response saveStudent(Student student) {

        ---
            ---
            ---
```

```
                ResponseBuilder builder = Response.status(201);
                builder.entity("Filed to save data");

        return builder.build();
          }
  }
```

-> Normally we will 200 status code for successful processing the request.

@GET(javax.ws.rs.GET), @POST(javax.ws.rs.POST), @PUT(javax.ws.rs.PUT) and @DELETE(javax.ws.rs.DELETE):

@GET:

-> If rest method returning some resources or if rest method parameters are @QuertParam parameters then we will

use this annotation before defining method.

```
@Path("/helloService")
public class HelloService {
```

```
@Path("/sayHello/{name}")
    @Consumes("application/json")
    @Produces({"application/xml","application/json"})
    @GET
    public Employee saveStudent(@PathParam("name") String name) {

       Employee e = new Employee();
            ---
            ---
            ---

      return e;
    }
}
```

-> If rest method parameter is user defined type then we can't use @GET, instead we can use @POST, @PUT or @DELETE irrespective

of the return type.

@POST:

-> If rest method having user defined types as parameters and if we writing some logic inside the method to insert those resources into

DB or any external system then we will use @POST annotation.

```
@Path("/helloService")
public class HelloService {

   @POST
   @Path("/sayHello")
        @Consumes("application/json")
        @Produces({"application/xml","application/json"})
        public Employee saveStudent(Student student) {

            Employee e = new Employee();
                  ---
                  ---
                  ---

          return e;
        }
}
```

@PUT:

-> If rest method having some modification logic inside the method then we will use @PUT annotation.

```
@Path("/helloService")
```

```
public class HelloService {


    @PUT
    @Path("/sayHello/{salary}")
        @Produces({"application/xml","application/json"})
        public Employee saveStudent(@QuertParam("eid") int empId,
@PathParam("salary") int salary) {


            Employee e = new Employee();
                ---
                ---
                ---


           return e;
        }
 }
```

-> Irrspective of the return type or parameter type we can use @PUT annotation.

@DELETE:

-> If rest method deleting some resources based on client sending parameters then we will use @DELETE annotation.

-> Irrspective of the return type or parameter type we can use @DELETE annotation.

```
@Path("/helloService")
public class HelloService {

   @DELETE
   @Path("/sayHello")
        @Produces({"application/xml","application/json"})
        public Employee saveStudent(@QuertParam("eid") int empId) {

            Employee e = new Employee();
                  ---
                  ---
                  ---

           return e;
        }
}
```

Restful webservices using jersey implementation:

-> Before work with jersey implementation(jersey-archive-1.19.zip) we need to download jersey jars using below url.

https://jersey.java.net/download.html

-> Once dowloaded above extracted file then we will get below folder structure.

D:\jersey-archive-1.19

|

|_ lib(jersey jars)

|

|_ apidocs

Steps to create restful webservice using jersey implementation:

-> Create Dynamic web project with Project name is "JerseyHelloService"

-> Copy jersey jars into our project WEB-INF/lib folder.

-> Configure jersey implementation skeleton in web.xml file with some url pattern ie., /rest/*

web.xml:

```
<web-app>
    <servlet>
```

```
        <servlet-name>jersey-serlvet</servlet-name>
        <servlet-
class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>

        <load-on-startup>1</load-on-startup>

    </servlet>

    <servlet-mapping>

        <servlet-name>jersey-serlvet</servlet-name>
        <url-pattern>/rest/*</url-pattern>

    </servlet-mapping>

</web-app>
```

-> Create HelloService.java in src folder with package name is com.venkat.service.

HelloService.java:

```
package com.venkat.service;

import javax.ws.rs.GET;
```

```
import javax.ws.rs.Path;

import javax.ws.rs.PathParam;

import javax.ws.rs.QueryParam;


@Path("/helloService")

public class HelloService {


	@Path("/sayHello/{age}")

	@GET

	public String sayHello(@QueryParam("name") String name,
@PathParam("age") int age) {

		return "Hello....  "+name+" your age is "+age;

	}

}
```

-> Deploy application into server and use below url to acess sayHello method.

http://localhost:8082/JerseyHelloService/rest/helloService/sayHello/32?name=venkat

Restful webservice client using Jersey implementation:

-> Create Dynamic Web Project with name as JerseyHelloServiceClient.

-> Copy all jersey jars into WEB-INF/lib folder.

-> Create TestClient.java in src folder with below code to invoke webservice.

TestClient.java:

```
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;

public class TestClient {

    public static void main(String[] args) {


        Client c = Client.create();

        WebResource resources1 =
c.resource("http://localhost:8082/JerseyHelloService/rest/helloService/sayHello/
32?name=venkat");

            String res = resources1.get(String.class);

            System.out.println(res);
```

```
        }
}
```

Create Restful webservice using Jersey implementation:

-> Create Dynamic Web Project with Project name as JerseyEmployeeService.

-> Copy all jersey jars into WEB-INF/lib folder.

-> Configure Jersey skeleton in web.xml file with "/rest/*" url-pattern.

-> Create Employee.java, ConnectionFactory.java and EmployeeService.java in src folder with com.venkat.service package.

Employee.java:

```
package com.venkat.service;

public class Employee {

        private int eid;
        private String name;
        private int salary;

        public int getEid() {
```

```
		return eid;
	}
	public void setEid(int eid) {
		this.eid = eid;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getSalary() {
		return salary;
	}
	public void setSalary(int salary) {
		this.salary = salary;
	}
}
```

ConnectionFactory.java:

```
package com.venkat.service;
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionFactory {

	private static Connection getConnection() {
		Connection con = null;

		try {
			Class.forName("oracle.jdbc.driver.OracleDriver");
			con = DriverManager.getConnection(
					"jdbc:oracle:thin:@localhost:1521:XE",
"system", "xe");

		} catch (SQLException e) {
			e.printStackTrace();
		} catch (ClassNotFoundException e) {
			e.printStackTrace();
		}
		return con;
	}
}
```

EmployeeService.java:

```
package com.venkat.service;


import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;


import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;


@Path("/employeeService")
public class EmployeeService {
```

```
    @GET
    @Path("/getEmployeeById")
    @Produces("application/json")
    public Employee getEmployeeById(@QueryParam("eid") int eid) {
        Connection con = ConnectionFactory.getConnection();
        Employee emp = null;
        try {
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM
EMPLOYEE WHERE EID='"+eid+"'");
            if(rs.next()) {
              emp = new Employee();
              emp.setEid(rs.getInt(1));
              emp.setName(rs.getString(2));
              emp.setSalary(rs.getInt(3));
            }
    } catch (SQLException e) {
            e.printStackTrace();
        }
        return emp;
    }

    @POST
    @Path("/createEmployee")
```

```
@Consumes(MediaType.APPLICATION_JSON)
public Response createEmployee(Employee emp) {
        int eid = emp.getEid();
        String name = emp.getName();
        int salary = emp.getSalary();
        Response res = null;

        Connection con = ConnectionFactory.getConnection();
        try {
                PreparedStatement stmt = con.prepareStatement("insert into
Employee values(?,?,?)");
                stmt.setInt(1, eid);
                stmt.setString(2, name);
                stmt.setInt(3, salary);
                int noOfRecordsUpdated = stmt.executeUpdate();
                if(noOfRecordsUpdated > 0) {
                        res = Response.status(200).entity("Employee inserted
successfully...").build();
                } else {
                        res = Response.status(201).entity("Employee inserted
failed...").build();
                }

    } catch (SQLException e) {
      res = Response.status(202).entity("Employee inserted failed...").build();
```

```
                e.printStackTrace();
            }
            return res;
        }


        @GET
        @Path("/deleteEmployeeById")
        public Response deleteEmployeeById(@QueryParam("eid") int eid) {

      Response res = null;

            Connection con = ConnectionFactory.getConnection();
            try {
                PreparedStatement stmt = con.prepareStatement("delete
Employee where eid=?");
                stmt.setInt(1, eid);
                int noOfRecordsUpdated = stmt.executeUpdate();
                if(noOfRecordsUpdated > 0) {
                    res = Response.status(200).entity("Employee deleted
successfully...").build();
                } else {
                    res = Response.status(201).entity("Employee deleted
failed...").build();
                }
```

```
        } catch (SQLException e) {
          res = Response.status(202).entity("Employee deleted failed...").build();
                        e.printStackTrace();
                }


                return null;
        }



        @DELETE
        @Path("/deleteEmployee")
        @Consumes(MediaType.APPLICATION_JSON)
        public Response deleteEmployee(Employee emp) {


              Response res = null;
              int eid = emp.getEid();


                        Connection con = ConnectionFactory.getConnection();
                        try {
                                PreparedStatement stmt =
con.prepareStatement("delete Employee where eid=?");
                                stmt.setInt(1, eid);
                                int noOfRecordsUpdated = stmt.executeUpdate();
                                if(noOfRecordsUpdated > 0) {
```

```
                    res = Response.status(200).entity("Employee
deleted successfully...").build();
                } else {
                    res = Response.status(201).entity("Employee
deleted failed...").build();
                }

            } catch (SQLException e) {
              res = Response.status(202).entity("Employee deleted
failed...").build();
                    e.printStackTrace();
                }

              return res;
        }

        @GET
        @Path("updateEmployeeSalaryById")
        public Response updateEmployeeSalaryById(int eid,  int salary) {

            Response res = null;

                Connection con = ConnectionFactory.getConnection();
                try {
```

```
                    PreparedStatement stmt =
con.prepareStatement("update Employee set salary=? where eid=?");
                    stmt.setInt(1, salary);
                    stmt.setInt(2, eid);

                    int noOfRecordsUpdated = stmt.executeUpdate();
                    if(noOfRecordsUpdated > 0) {
                        res = Response.status(200).entity("Employee
updated successfully...").build();
                    } else {
                        res = Response.status(201).entity("Employee
updated failed...").build();
                    }

            } catch (SQLException e) {
              res = Response.status(202).entity("Employee updated
failed...").build();
                    e.printStackTrace();
                }

            return res;
        }

        @PUT
        @Path("updateEmployeeSalary")
        @Consumes("application/json")
```

```
public Response updateEmployeeSalary(Employee emp) {


Response res = null;
int eid = emp.getEid();
        int salary = emp.getSalary();


            Connection con = ConnectionFactory.getConnection();
            try {
                PreparedStatement stmt =
con.prepareStatement("update Employee set salary=? where eid=?");
                stmt.setInt(1, salary);
                stmt.setInt(2, eid);


                int noOfRecordsUpdated = stmt.executeUpdate();
                if(noOfRecordsUpdated > 0) {
                    res = Response.status(200).entity("Employee
updated successfully...").build();
                } else {
                    res = Response.status(201).entity("Employee
updated failed...").build();
                }


        } catch (SQLException e) {
          res = Response.status(202).entity("Employee updated
failed...").build();
                e.printStackTrace();
```

```
                }

            return res;
        }

}
```

-> Copy ojdbc14.jar file into WEB-INF/lib folder.

-> deploy application into server and use below url to test webservice.

http://localhost:8082/JerseyEmployeeService/rest/employeeService/getEmployeeById?eid=100

Note:

CREATE TABLE EMPLOYEE(EID number(6),NAME varchar2(25),SALARY number(6));

insert into Employee values(100,'venkat',6000);

-> Webservice client for above service using jersey implementation:

-> Create Dynamic Web Project with project name as JerseyEmployeeServiceClient.

-> Copy all jersey jars into /WEB-INF/lib folder.

-> Create Employee.java and TestClient1.java in src folder.

Employee.java:

```
package com.venkat.service;

public class Employee {

	private int eid;
	private String name;
	private int salary;

	public int getEid() {
		return eid;
	}
	public void setEid(int eid) {
		this.eid = eid;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
```

```
        this.name = name;
    }
    public int getSalary() {
        return salary;
    }
    public void setSalary(int salary) {
        this.salary = salary;
    }
}
```

TestClient1.java:

```
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.config.ClientConfig;
import com.sun.jersey.api.client.config.DefaultClientConfig;
import com.sun.jersey.api.json.JSONConfiguration;


public class TestClient1 {

    public static void main(String[] args) {
```

```
                ClientConfig clientConfig = new DefaultClientConfig();

clientConfig.getFeatures().put(JSONConfiguration.FEATURE_POJO_MAPPI
NG, Boolean.TRUE);

                Client c = Client.create(clientConfig);

            WebResource resources1 =
c.resource("http://localhost:8082/JerseyEmployeeService/rest/employeeService/
getEmployeeById?eid=100");

                Employee res =
resources1.accept("application/json").get(Employee.class);

                System.out.println(res.getEid());
                System.out.println(res.getName());
                System.out.println(res.getSalary());
        }

}
```

-> Create TestClient2.java in src folder.

TestClient2.java:

```
import javax.ws.rs.core.MediaType;
```

```
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.config.ClientConfig;
import com.sun.jersey.api.client.config.DefaultClientConfig;
import com.sun.jersey.api.json.JSONConfiguration;


public class TestClient2 {

        public static void main(String[] args) {

                ClientConfig clientConfig = new DefaultClientConfig();

clientConfig.getFeatures().put(JSONConfiguration.FEATURE_POJO_MAPPI
NG, Boolean.TRUE);

                Client c = Client.create(clientConfig);

            WebResource resources1 =
c.resource("http://localhost:8082/JerseyEmployeeService/rest/employeeService/
createEmployee");

            Employee e = new Employee();
            e.setEid(300);
```

```
            e.setName("naresh1");
            e.setSalary(10000);

            ClientResponse res =
resources1.type("application/json").post(ClientResponse.class, e);

            int statusCode = res.getStatus();

            String resp = res.getEntity(String.class);

            System.out.println("STATUS CODE: "+statusCode);
            System.out.println("OUTPUT:"+resp);
 }
}
```

TestClient3.java:

```
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.WebResource;


public class TestClient3 {

      public static void main(String[] args) {
```

```
            Client c = Client.create();


        WebResource resources1 =
c.resource("http://localhost:8082/JerseyEmployeeService/rest/employeeService/
deleteEmployeeById?eid="+200);
        ClientResponse resp = resources1.get(ClientResponse.class);


        int statusCode = resp.getStatus();
        String res = resp.getEntity(String.class);




        System.out.println(statusCode);
        System.out.println(res);
    }
}
```

TestClient4.java

```
import javax.ws.rs.core.MediaType;


import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.config.ClientConfig;
```

```
import com.sun.jersey.api.client.config.DefaultClientConfig;

import com.sun.jersey.api.json.JSONConfiguration;



public class TestClient4 {

        public static void main(String[] args) {


                ClientConfig clientConfig = new DefaultClientConfig();

clientConfig.getFeatures().put(JSONConfiguration.FEATURE_POJO_MAPPI
NG, Boolean.TRUE);



                 Client c = Client.create(clientConfig);


            WebResource resources1 =
c.resource("http://localhost:8082/JerseyEmployeeService/rest/employeeService/
deledeleteEmployee");


            Employee e = new Employee();

            e.setEid(200);


            String resp  =  resources1.type("application/json").delete(String.class,
e);


            System.out.println(resp);
```

```
        }

}
```

TestClient5.java:

```
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.WebResource;



public class TestClient5 {

        public static void main(String[] args) {

                Client c = Client.create();

            WebResource resources1 =
c.resource("http://localhost:8082/JerseyEmployeeService/rest/employeeService/
updateEmployeeSalaryById/100/12000");

            ClientResponse res = resources1.get(ClientResponse.class);

            System.out.println(res.getStatus());
```

```
            System.out.println(res.getEntity(String.class));


        }


}
```

TestClient6.java:

Rest service with xml:

-> All restful webservice implementation skeleton's uses JAXB(Java Api for xml binding) api to convert java object into

xml and xml into java objects.

-> So JAXB api will expect JAXB annotations from pojo class to convert java object to xml and xml to java object.

-> Mainly we use 3 JAXB annotations in pojo classes to convert xml to java and java to xml.

@XmlRootElement

@XmlElement

@XmlAtrribute

Employee.java:

```
package com.venkat.service;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name="emp")
public class Employee {

	private int eid;
	private String name;
	private int salary;

	@XmlElement(name="id")
	public int getEid() {
		return eid;
	}
	public void setEid(int eid) {
		this.eid = eid;
	}
```

```
    @XmlElement
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    @XmlElement
    public int getSalary() {
        return salary;
    }
    public void setSalary(int salary) {
        this.salary = salary;
    }
}
```

Restful webservice security:

-> Restful webservice supports only username token security but not message level security.

-> In restfull webservices we can achive username token security by using basic authentication security and

OAuth security.

-> In basic authentication security client will send static username and password as http header in http request and

the service provider will get username and password from http header, And he will validate wheather username and password

are valid or not, if valid then it will allow user to process the request otherwise it will throw the authentication failed exception message

to client.

-> In OAuth security client always will send dynamic value(token) to the service provider and the service provider

will verify that token is valid or not, if it is valid, then it will allow client to process the request.

-> Here the token having some expire time(age of token), if the token is used with in age of token, then only it will

be valid otherwise it autometically invalidated.

Restful webservice security using basic authentication:

-> Apply basic authentication security to existing EmployService project to do this need to follow below steps.

1. Create AuthenticationService.java and AuthFilter.java in src folder.

AuthenticationService.java:

```
package com.venkat.auth.service;

import java.util.StringTokenizer;

import com.sun.org.apache.xml.internal.security.utils.Base64;

public class AuthenticationService {

	//Basic encryptedusername:encryptedpassword

	public boolean isAuthenticate(String authenticateString) {

		if(authenticateString == null) {
		    return false;
		}
		String encodedUsernamePassword =
authenticateString.replaceFirst("Basic", "");

		try {
			byte[] decodedUserPass =
Base64.decode(encodedUsernamePassword);//abc:xyz

			String decodeUserPass = new
String(decodedUserPass,"UTF-8");//un:pass
```

```
                StringTokenizer tokenizer = new
StringTokenizer(decodeUserPass, ":");


                String userName = tokenizer.nextToken();
                String password = tokenizer.nextToken();


                if(userName.equals("mvr") &&
password.equals("@durga")) {
                        return true;
                } else {
                        return false;
                }


            } catch(Exception e) {
                        e.printStackTrace();
                return false;
                }
        }


}
```

AuthFilter.java:

```
package com.venkat.auth.filter;
```

```
import java.io.IOException;


import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;


import com.venkat.auth.service.AuthenticationService;


public class AuthFilter implements Filter {



        public void destroy() {}


        public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {


                HttpServletRequest req = (HttpServletRequest)request;


                String authUsernamePass = req.getHeader("Authorization");
```

```
            AuthenticationService as = new AuthenticationService();

            boolean isAuthenticated = as.isAuthenticate(authUsernamePass);

            if(isAuthenticated)
                chain.doFilter(request, response);
            else {
                HttpServletResponse res = (HttpServletResponse) response;

                res.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
            }
        }
        public void init(FilterConfig fConfig) throws ServletException {}
}
```

-> Copy servlet-api.jar into WEB-INF/lib folder.

-> Configure AuthFilter.java in web.xml file like below.

web.xml:

```
<web-app>
        <servlet>
```

```
        <servlet-name>jersey-serlvet</servlet-name>
        <servlet-
class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>

        <load-on-startup>1</load-on-startup>

    </servlet>

    <servlet-mapping>

        <servlet-name>jersey-serlvet</servlet-name>
        <url-pattern>/rest/*</url-pattern>

    </servlet-mapping>

    <filter>
        <filter-name>AuthFilter</filter-name>
        <filter-class>com.venkat.auth.filter.AuthFilter</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>AuthFilter</filter-name>
    <url-pattern>/rest/*</url-pattern>
    </filter-mapping>
```

```
</web-app>
```

Creating Basic authentication client using jersey implementation:

-> To invoke basic authentication service we need to use below client.

TestClient1.java:

```
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.config.ClientConfig;
import com.sun.jersey.api.client.config.DefaultClientConfig;
import com.sun.jersey.api.client.filter.HTTPBasicAuthFilter;
import com.sun.jersey.api.json.JSONConfiguration;


public class TestClient1 {

        public static void main(String[] args) {

                ClientConfig clientConfig = new DefaultClientConfig();

clientConfig.getFeatures().put(JSONConfiguration.FEATURE_POJO_MAPPI
NG, Boolean.TRUE);
```

```
                Client c = Client.create(clientConfig);
                c.addFilter(new HTTPBasicAuthFilter("mvr","@durga"));

            WebResource resources1 =
c.resource("http://localhost:8082/JerseyEmployeeService/rest/employeeService/
getEmployeeById?eid=100");

                Employee res =
resources1.accept("application/xml").get(Employee.class);

                System.out.println(res.getEid());
                System.out.println(res.getName());
                System.out.println(res.getSalary());
        }

}
```

Restfull webservice with apache cxf:

-> Create Dynamic web project with project names as CXFRestService

-> Copy CXF jars into project WEB-INF/lib folder.

-> Copy ojdbc14.jar file into WEB-INF/lib folder.

-> Create ConnectionFactory.java, Employee.java and EmployeeService.java in src folder.

Employee.java:

```
package com.venkat.service;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Employee {

	private int eid;
	private String name;
	private int salary;

	@XmlElement
	public int getEid() {
		return eid;
	}
	public void setEid(int eid) {
		this.eid = eid;
	}
```

```
	@XmlElement
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}

	@XmlElement
	public int getSalary() {
		return salary;
	}
	public void setSalary(int salary) {
		this.salary = salary;
	}
}
```

ConnectionFactory.java:

```
package com.venkat.service;

import java.sql.Connection;
```

```
import java.sql.DriverManager;

import java.sql.SQLException;


public class ConnectionFactory {


	public static Connection getConnection() throws SQLException,
ClassNotFoundException{

		Connection con = null;


			Class.forName("oracle.jdbc.driver.OracleDriver");

			con = DriverManager.getConnection(

					"jdbc:oracle:thin:@localhost:1521:XE",
"system", "xe");


		return con;

	}


}
```

EmployeeService.java:

```
package com.venkat.service;


import java.sql.Connection;

import java.sql.PreparedStatement;
```

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

@Path("/employeeService")
public class EmployeeService {

    @GET
    @Path("/getEmployeeById")
    @Produces("application/xml")
    public Employee getEmployeeById(@QueryParam("eid") int eid) {
        Connection con = null;
```

```
            Employee emp = null;
            try {
                  con = ConnectionFactory.getConnection();
                  Statement stmt = con.createStatement();
                  ResultSet rs = stmt.executeQuery("SELECT * FROM
EMPLOYEE WHERE EID='"+eid+"'");
                  if(rs.next()) {
                    emp = new Employee();
                    emp.setEid(rs.getInt(1));
                    emp.setName(rs.getString(2));
                    emp.setSalary(rs.getInt(3));
                  }
      } catch (SQLException e) {
       e.printStackTrace();
            } catch (ClassNotFoundException e) {
                  e.printStackTrace();
            }
            return emp;
      }

      @POST
      @Path("/createEmployee")
      @Consumes(MediaType.APPLICATION_JSON)
      public Response createEmployee(Employee emp) {
            int eid = emp.getEid();
```

```
            String name = emp.getName();
            int salary = emp.getSalary();
            Response res = null;

            Connection con = null;
            try {
                  con = ConnectionFactory.getConnection();
                  PreparedStatement stmt = con.prepareStatement("insert into
Employee values(?,?,?)");
                  stmt.setInt(1, eid);
                  stmt.setString(2, name);
                  stmt.setInt(3, salary);
                  int noOfRecordsUpdated = stmt.executeUpdate();
                  if(noOfRecordsUpdated > 0) {
                        res = Response.status(200).entity("Employee inserted
successfully...").build();
                  } else {
                        res = Response.status(201).entity("Employee inserted
failed...").build();
                  }

      } catch (SQLException e) {
        res = Response.status(202).entity("Employee inserted failed...").build();
                  e.printStackTrace();
            } catch (ClassNotFoundException e) {
```

```
                res = Response.status(203).entity("Employee inserted
failed...").build();
                e.printStackTrace();
            }
            return res;
        }


        @GET
        @Path("/deleteEmployeeById")
        public Response deleteEmployeeById(@QueryParam("eid") int eid) {

      Response res = null;


            Connection con = null;
            try {
                con = ConnectionFactory.getConnection();
                PreparedStatement stmt = con.prepareStatement("delete
Employee where eid=?");
                stmt.setInt(1, eid);
                int noOfRecordsUpdated = stmt.executeUpdate();
                if(noOfRecordsUpdated > 0) {
                    res = Response.status(200).entity("Employee deleted
successfully...").build();
                } else {
```

```
                    res = Response.status(201).entity("Employee deleted
failed...").build();
                }
        } catch (SQLException e) {
          res = Response.status(202).entity("Employee deleted failed...").build();
                e.printStackTrace();
            } catch (ClassNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

            return null;
        }


        @DELETE
        @Path("/deleteEmployee")
        @Consumes(MediaType.APPLICATION_JSON)
        //@Produces(MediaType.APPLICATION_JSON)
        public String deleteEmployee(Employee emp) {

            String res = null;
            int eid = emp.getEid();

                Connection con = null;
```

```
                try {
                        con = ConnectionFactory.getConnection();
                        PreparedStatement stmt =
con.prepareStatement("delete Employee where eid=?");
                        stmt.setInt(1, eid);
                        int noOfRecordsUpdated = stmt.executeUpdate();
                        if(noOfRecordsUpdated > 0) {
                                res = "Employee successfully deleted...";
                        } else {
                                res = "Employee failed to delete";
                        }

            } catch (SQLException e) {
              res = "Employee failed to delete";
                        e.printStackTrace();
                } catch (ClassNotFoundException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

               return res;
        }

        @GET
```

```
@Path("updateEmployeeSalaryById/{eid}/{salary}")
public Response updateEmployeeSalaryById(@PathParam("eid") int eid,
@PathParam("salary") int salary) {

    Response res = null;

        Connection con = null;
        try {
            con = ConnectionFactory.getConnection();
            PreparedStatement stmt =
con.prepareStatement("update Employee set salary=? where eid=?");
            stmt.setInt(1, salary);
            stmt.setInt(2, eid);

            int noOfRecordsUpdated = stmt.executeUpdate();
            if(noOfRecordsUpdated > 0) {
                res = Response.status(200).entity("Employee
updated successfully...").build();
            } else {
                res = Response.status(201).entity("Employee
updated failed...").build();
            }

    } catch (SQLException e) {
      res = Response.status(202).entity("Employee updated
failed...").build();
```

```
                    e.printStackTrace();
                } catch (ClassNotFoundException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }

        return res;
    }

    @PUT
    @Path("updateEmployeeSalary")
    @Consumes("application/json")
    public Response updateEmployeeSalary(Employee emp) {

     Response res = null;
     int eid = emp.getEid();
            int salary = emp.getSalary();

                Connection con = null;
                try {
                    con = ConnectionFactory.getConnection();
                    PreparedStatement stmt =
con.prepareStatement("update Employee set salary=? where eid=?");
                    stmt.setInt(1, salary);
                    stmt.setInt(2, eid);
```

```
                    int noOfRecordsUpdated = stmt.executeUpdate();
                    if(noOfRecordsUpdated > 0) {
                        res = Response.status(200).entity("Employee
updated successfully...").build();
                    } else {
                        res = Response.status(201).entity("Employee
updated failed...").build();
                    }
            } catch (SQLException e) {
              res = Response.status(202).entity("Employee updated
failed...").build();
                    e.printStackTrace();
                } catch (ClassNotFoundException e) {
                    e.printStackTrace();
                }

            return res;
        }
}
```

-> Create cxf-servlet.xml file in WEB-INF folder.

cxf-servlet.xml:

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jaxws="http://cxf.apache.org/jaxws"

        xmlns:jaxrs="http://cxf.apache.org/jaxrs"

        xsi:schemaLocation="http://www.springframework.org/schema/beans

         http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

          http://cxf.apache.org/jaxws

           http://cxf.apache.org/schemas/jaxws.xsd

            http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd">


        <import resource="classpath:META-INF/cxf/cxf.xml" />

        <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />

        <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />



   <bean id="jsonProvider"
class="org.codehaus.jackson.jaxrs.JacksonJsonProvider"/>



   <!--
http://localhost:8082/CXFRestService/services/empServiceRestPort/employeeS
ervice/getEmployeeById?eid=100 -->


   <jaxrs:server id="empServiceRest" address="/empServiceRestPort">
```

```
        <jaxrs:serviceBeans>
            <ref bean="empService"/>
        </jaxrs:serviceBeans>

       <jaxrs:providers>
           <ref bean="jsonProvider"/>
       </jaxrs:providers>

       <jaxrs:extensionMappings>
         <entry key="xml" value="application/xml"/>
         <entry key="json" value="application/json"/>
       </jaxrs:extensionMappings>

   </jaxrs:server>

        <bean id="empService" class="com.venkat.service.EmployeeService">
        </bean>
</beans>
```

-> Configure cxf skeleton in web.xml file

web.xml:

```
<web-app>
```

```
<servlet>
  <servlet-name>cxf</servlet-name>
  <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>cxf</servlet-name>
  <url-pattern>/services/*</url-pattern>
</servlet-mapping>
<session-config>
  <session-timeout>60</session-timeout>
</session-config>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>WEB-INF/cxf-servlet.xml</param-value>
</context-param>
<listener>
  <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
</web-app>
```

-> Deploy and run application.

Note:

-> Download below 2 jars and copy them into project WEB-INF/lib folder.

jackson-all-1.9.0.jar

jackson-jaxrs.jar