JAVA

WEB SERVICES

- Satya Kaveti

Small Codes

Programming Simplified

A SmlCodes.Com Small presentation

In Association with Idleposts.com

JAVA WEB SERVICES

If you discover any errors on our website or in this tutorial, please notify us at support@smlcodes.com or smlcodes@gmail.com

First published on 10th JAN 2017, Published by **SmlCodes.com**

Author Credits

Name : **Satya Kaveti**

Email : satyakaveti@gmail.com

Website : smlcodes.com, satyajohnny.blogspot.com

Download Source Code: **https://codeload.github.com/smlcodes/WebServices-Examples/zip/master**

Digital Partners

1. Introduction

By using webservices we can communicate different applications on different platforms. For example java application in Windows platform can easily communicate with the application developed using .net/php in Linux operation system.

Web Services are mainly of 2 types,

1. **SOAP [Simple Object Access Protocol]**
2. **REST [Representational state transfer]**

1. SOAP [Simple Object Access Protocol]

SOAP stands for Simple Object Access Protocol. **SOAP is an XML based** industry standard protocol for designing and developing web services. **Since it's XML based, its platform and language independent**. So our server can be based on JAVA and client can be on .NET, PHP etc. and vice versa. **SOAP gives the output only in XML format**

We have following API's to implement SOAP Webservices in our java applications

- **JAX-WS**
- **Apache Axis2**

2. REST [Representational state transfer]

What ever the data/response we will get from the server is known as **Resource**.Each resource can be accessed by its URI's. We can get the resource from RESTful service in different formats like, **HTML,XML,JSON,TEXT,PDF** and in the Image formats as well, **but in real time we mainly we will prefer JSON.** REST guidelines always talks about stateless communication between client and the Server. Stateless means, every single request from client to server will be considered as a fresh request. Because of this reason REST always prefers to choose HTTP as it a stateless protocol

We have following API's to implement RESTful Webservices in our java applications

- **JAX-RS**

Apache CXF provides implementation for SOAP and RESTful services both.

REST is a style of software architecture.RESTful is typically used to refer to web services implementing such an architecture

No.	SOAP	REST
1)	SOAP is a **protocol**.	REST is an **architectural style**.
2)	SOAP stands for **Simple Object Access Protocol**.	REST stands for **REpresentational State Transfer**.
3)	SOAP **can't use REST** because it is a protocol.	REST **can use SOAP** web services because it is a concept and can use any protocol like HTTP, SOAP.
4)	SOAP **uses services interfaces to expose the business logic**.	REST **uses URI to expose business logic**.
5)	**JAX-WS** is the java API for SOAP web services.	**JAX-RS** is the java API for RESTful web services.
6)	SOAP **defines standards** to be strictly followed.	REST does not define too much standards like SOAP.
7)	SOAP **requires more bandwidth** and resource than REST.	REST **requires less bandwidth** and resource than SOAP.
8)	SOAP **defines its own security**.	RESTful web services **inherits security measures** from the underlying transport.
9)	SOAP **permits XML** data format only.	REST **permits different** data format such as Plain text, HTML, XML, JSON etc.
10)	SOAP is **less preferred** than REST.	REST **more preferred** than SOAP.

1.1 SOAP [Simple Object Access Protocol

Simple Object Access Protocol (SOAP) is a standard protocol specification for message exchange based on XML. Communication between the web service and client happens using XML messages.

A simple web service architecture have two components

1. **Client**
2. **Service provider**

To communicate clinet with service provider clinet must know about following things

- Location of webservices server
- Functions available, signature and return types of function.
- Communication protocol
- Input output formats

Service provider will create a standard XML file which will have all above information.So if this file is given to client then client will be able to access web service. **This XML file is called "WSDL"**.

WSDL (Web Services Description Language):

WSDL stands for **Web Service Description Language**. It is **an XML file that describes the technical details** of how to implement a web service, more specifically the **URI, port, method names, arguments, and data types**. Since WSDL is XML, it is both human-readable and machine-consumable, which aids in the ability to call and bind to services dynamically.using this WSDL file we can understand things like,

- Port / Endpoint – URL of the web service
- Input message format
- Output message format
- Security protocol that needs to be followed
- Which protocol the web service uses

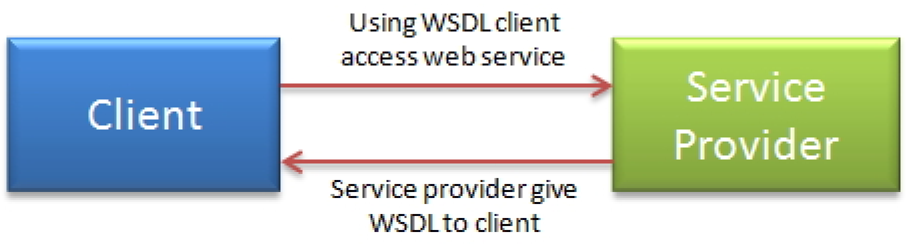
How to access web service:

There are two ways to access web service

1. **If Service provider knows client**
2. **If Service provider register its WSDL to UDDI and client can access it from UDDI**

1. If Service provider knows client

If Service provider knows client then it will provide its wsdl to client and client will be able to access web service.

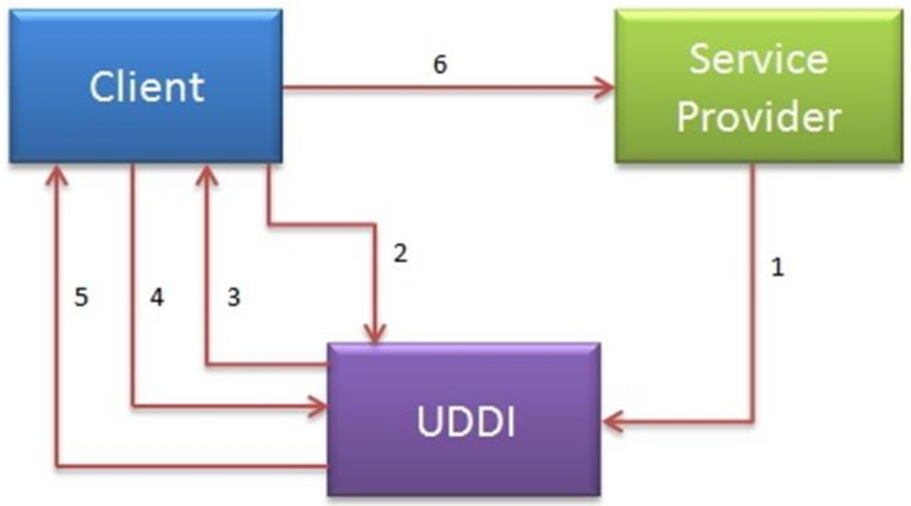

Using WSDL client access web service
Client
Service Provider
Service provider give WSDL to client

(If Service provider knows client)

2. If Service provider register its WSDL to UDDI and client can access it from UDDI

Service provider register its WSDL to UDDI and client can access it from UDDI:UDDI stands for Universal Description, Discovery and Integration.It is a directory service. Web services can register with a UDDI and make themselves available through it for discovery.So following steps are involved.

1. ***Service provider registers with UDDI.***
2. ***Client searches for service in UDDI.***
3. ***UDDI returns all service providers offering that service.***
4. ***Client chooses service provider***
5. ***UDDI returns WSDL of chosen service provider.***
6. ***Using WSDL of service provider,client accesses web service***



UDDI:

- UDDI is an XML-based standard for describing, publishing, and finding web services.
- UDDI is a specification for a distributed registry of web services

A business or a company can register three types of information into a UDDI registry. This information is contained in three elements of UDDI.

These three elements are:

1. **White Pages** : Basic information about the company and its business
2. **Yellow Pages**: contain more details about the company
3. **Green Pages**: contains technical information about a web service(url locations etc)

Now SOAP useswithout WSDL and UDDI. Instead of the discovery process described in the History of the Web Services Specification section below, SOAP messages are hard-coded or genereated without the use of a repository. The interaction is illustrated in the figure below. More on SOAP.

1.2 REST [Representation State Transfer]

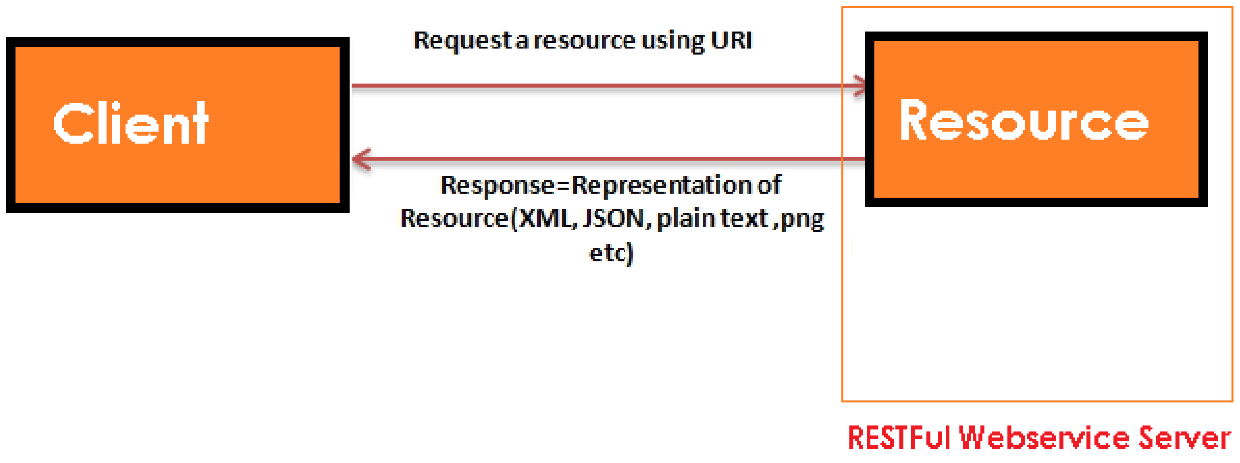
REpresentational State Transfer (REST) is a stateless client-server architecture in which the web services are viewed as **resources** and can **be identified by their URIs.**Web service clients that want to use these resources access via globally defined set of remote methods that describe the action to be performed on the resource.

It consists of two components

1. **REST server:** which provides access to the resources
2. **REST client** : which accesses and modify the REST resources.

In the REST architecture style, clients and servers exchange result representations of resources by using a standardized interface and protocol.**REST isn't protocol specific, but when people talk about REST they usually mean REST over HTTP.**

The response from server is considered as the result representation of the resources. This result representation can be generated from one resource or more number of resources. REST allows that resources have different result representations, **e.g.xml, json etc**. The rest client can ask for specific result representation via the HTTP protocol


Client
Request a resource using URI
Response=Representation of Resource(XML, JSON, plain text ,png etc)
Resource
RESTFul Webservice Server

HTTP methods:

RESTful web services use HTTP protocol methods for the operations they perform.

Methods are:

- **GET**:It defines a reading access of the resource without side-effects.This operation is idempotent i.e.they can be applied multiple times without changing the result
- **PUT**: It is generally used for updating resouce.It must also be idempotent.
- **DELETE:** It removes the resources. The operations are idempotent i.e. they can get repeated without leading to different results.
- **POST**: It is used for creating a new resource. It is not idempotent.

Idempotent

Idempotent means result of multiple successful request will not change state of resource after initial application

For example:
GET is idempotent. If Delete() is idempotent method because when you first time use delete, it will delete the resource (initial application) but after that, all other request will have no result because resource is already deleted.

Post is not idempotent method because when you use post to create resource, it will keep creating resource for each new request, so result of multiple successful request will not be same.

Some important features of Restful web services are:

1.Resource identification through URI:Resources are identified by their URIs (typically links on internet). So, a client can directly access a RESTful Web Services using the URIs of the resources (same as you put a website address in the browser's address bar and get some representation as response).

2.Uniform interface: Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE.

3.Client-Server: A clear separation concerns is the reason behind this constraint. Separating concerns between the Client and Server helps improve portability in the Client and Scalability of the server components.

4.Stateless: each request from client to server must contain all the information necessary to understand the request, and cannot take advantage of any stored context on the server.

5.Cache: to improve network efficiency responses must be capable of being labeled as cacheable or non-cacheable.

6.Named resources - the system is comprised of resources which are named using a URL.

7.Interconnected resource representations - the representations of the resources are interconnected using URLs, thereby enabling a client to progress from one state to another.

8.Layered components - intermediaries, such as proxy servers, cache servers, gateways, etc, can be inserted between clients and resources to support performance, security, etc.

9.Self-descriptive messages: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others.

1.3 Java Web Services API

Java web services tutorial provides concepts and examples of two main java web services api: **JAX-WS and JAX-RS**. The java web service application can **be accessed by other programming languages such as .Net and PHP.**

Java web service application perform communication through **WSDL (Web Services Description Language).** There are two ways to write java web service application code**: SOAP and RESTful**.

There are two main API's defined by Java for developing web service applications since JavaEE 6.

1. **JAX-WS:** for SOAP web services. The **are 2 ways to write JAX-WS** application code: by

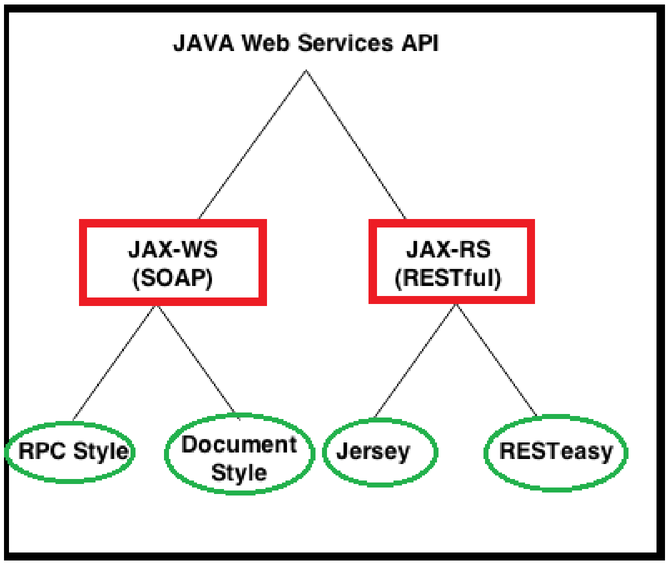
   i. ***RPC style***

   ii. ***Document style.***

2. **JAX-RS:** for RESTful web services. There are **mainly 2 implementation** currently in use for creating **JAX-RS** application:

   i. ***Jersey***

   ii. ***RESTeasy***.



We have some other RESTFul webservices providers like

- ***Jersey***
- ***RestEasy***
- ***Restlet***
- ***CFX***
- ***Spring Rest webservices***

2. JAX-WS (SOAP web services)

SOAP stands for Simple Object Access Protocol. It is a XML-based protocol for accessing web services.

SOAP is a W3C recommendation for communication between two applications.

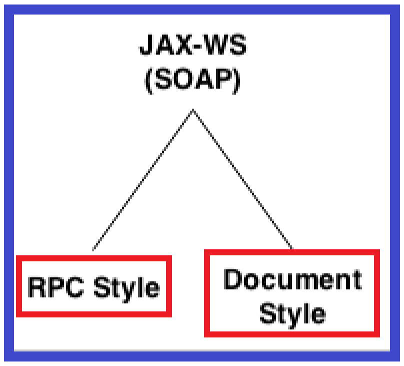
SOAP is XML based protocol. It is platform independent and language independent. By using SOAP, you will be able to interact with other programming language applications.

Advantages of Soap Web Services

- **WS Security**: SOAP defines its own security known as WS Security.
- **Language and Platform independent**: SOAP web services can be written in any programming language and executed in any platform

There are two ways to develop JAX-WS example.

1. **RPC style**
2. **Document style**



There are two encoding use models that are used to translate a WSDL binding to a SOAP message. They are: **literal, and encoded.**

The combination of the different style and use models give us four different ways to translate a WSDL binding to a SOAP message.

```
Document/literal
Document/encoded
RPC/literal
RPC/encoded
```

When using a literal use model, the body contents should conform to a user-defined **XML-schema (XSD) structure**. The advantage is two-fold. For one, you can validate the message body with the user-defined XML-schema, moreover, you can also transform the message using a transformation language like XSLT.

With a (SOAP) encoded use model, the message has to use XSD datatypes, but the structure of the message need not conform to any user-defined XML schema. This makes it difficult to validate the message body or use XSLT based transformations on the message body.

2.1 Diffrence between RPC-Style and Document Style

The way of generating SOAP message formate is main difffrence beteween them.

1. RPC Stlye:

SOAP Body must conform to a structure that indicates the **method name & Parameters name**

```
<soap:envelope>
    <soap:body>
        <myMethod>
            <x xsi:type="xsd:int">5</x>
            <y xsi:type="xsd:float">5.0</y>
        </myMethod>
    </soap:body>
</soap:envelope>
```

2. Document Style

SOAP Body can be structurted in any way you like.their is no TYPE attribute here

```
<soap:envelope>
    <soap:body>
        <xElement>5</xElement>
        <yElement>5.0</yElement>
    </soap:body>
</soap:envelope>
```

2.2 JAX-WS Annotations

We have following important annonotations in order to workwith JAX-WS webservices. They are

1. **@WebService**
2. **@SoapBinding**
3. **@WebMethod**
4. **@WebResult**
5. **@WebServiceClient**
6. **@RequestWrapper**
7. **@ResponseWrapper**
8. **@Oneway**
9. **@HandlerChain**

1.@WebService

This annotation can be used in 2 ways

1. To mark the class as the implementing the Web Service

```
Package webservice;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;

@WebService
@SOAPBinding(style=Style.RPC)
public interface HelloWorld {
        @WebMethod
        String getHelloworldMessage(String msg);
}
```

2. Defining a Web Service Interface (SEI), in other words Service Endpoint Interface

```
import javax.jws.WebService;

@WebService(endpointInterface="webservice.HelloWorld ")
public class HelloWorldImpl implements HelloWorld{
        @Override
        public String getHelloworldMessage (String name) {
                return "Hello World JAX-WS " + name;
        }
}
```

@Webservice with all attributes as below formate

```
@WebService(portName = "SoapPort", serviceName = " HelloWorld ",
 targetNamespace = "http://apache.org/hello_world_soap_http",
 endpointInterface="webservice.HelloWorld ")
```

2.@SoapBinding

This annotation is used to specify the SOAP messaging style which can either be **RPC** or **DOCUMENT**

```
/Service Endpoint Interface
@WebService
@SOAPBinding(style = Style.RPC)
//@SOAPBinding(style = Style. DOCUMENT)
public interface HelloWorld{
        @WebMethod
        String getHelloWorldAsString(String name);
}
```

@SoapBinding with all attributes as below formate

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

3.@WebMethod

@WebMethod JAX-WS annotation can be applied over a method only. This specified that the method represents a web service operation.it will be used in Interface(***Service Endpoint Interface***) level method only, not in implementation method level.()

```
/Service Endpoint Interface
@WebService
public interface HelloWorld{
        @WebMethod
        String getHelloWorldAsString(String name);
}
```

@WebMethod with all attributes as below formate

```
@WebMethod(operationName="echoComplexType", action=" SOAPAction")
```

4.@WebResult

@WebResult can be used to determine **what the generated WSDL shall look like**

```
@WebService
public interface HelloWorld{
@WebMethod
@WebResult(partName="Helloworld Method")
String getHelloWorldAsString(String name);
}
```

```
//Service Implementation
@WebService(endpointInterface = "com.mkyong.ws.HelloWorld")
public class HelloWorldImpl implements HelloWorld{
        @Override
        public String getHelloWorldAsString(String name) {
                return "Hello World JAX-WS " + name;
        }
}
```

```
public class WSPublisher {
        public static void main(String[] args) {
                Endpoint.publish("http://127.0.0.1:9999/ctf", new getHelloWorldAsString ());
        }
}
```

On publishing the generated WSDL (at URL: `http://127.0.0.1:9999/ctf?wsdl`) would be like:

```
<definitions
        xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
        xmlns:wsp="http://www.w3.org/ns/ws-policy"
xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
        xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:tns="http://webresult.jaxWsAnnotations.examples.smlcodes.com/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://schemas.xmlsoap.org/wsdl/"
        targetNamespace="http://webresult.jaxWsAnnotations.examples.smlcodes.com/"
        name="WSAnnotationsWebResultImplService">
        <types />
        <message name=" getHelloWorldAsString ">
                <part name="arg0" type="xsd: string " />
        </message>
        <message name=" Helloworld Method ">
                <part name=" getHelloWorldAsString " type="xsd:string" />
        </message>

</definitions>
```

5.@WebServiceClient

```
@WebServiceClient(name = "WsAnnotationsWebServiceImplService", targetNamespace =
"http://webservice.smlcodes.com/", wsdlLocation =
"file:/Users/satyakaveti/Downloads/ctf.wsdl")
```

The information specified in this annotation helps in identifying a wsdl:service element inside a WSDL document. This element represents the Web service for which the generated service interface provides a client view.

6.@RequestWrapper

`@RequestWrapper` JAX-WS annotation is used to annotate methods in the Service Endpoint Interface with the request wrapper bean to be used at runtime. It has 4 optional elements; `className` that represents the request wrapper bean name, `localName` that represents element's local name, `partName` that represent the part name of the wrapper part in the generated WSDL file, and `targetNamespace` that represents the element's namespace

```
@WebService
@SOAPBinding(style=Style.RPC)
public interface WSRequestWrapperInterface {
        @WebMethod
        @RequestWrapper(localName="CTF",
        targetNamespace="http://smlcodes.com/tempUtil",
        className="com.smlcodes.examples.jaxWsAnnotations.webservice.CTF")
        float celsiusToFarhenheit(float celsius);
}
```

7.@ResponseWrapper

`@ResponseWrapper` JAX-WS annotation is used to annotate methods in the Service Endpoint Interface with the response wrapper bean to be used at runtime. It has 4 optional elements; `className` that represents the response wrapper bean name, `localName` that represents element's local name, `partName` that represent the part name of the wrapper part in the generated WSDL file, and `targetNamespace` that represents the element's namespace.

```
public interface WSResponseWrapperInterfaceI {
        @WebMethod
        @ResponseWrapper(localName="CTFResponse",
        targetNamespace="http:// smlcodes.com/tempUtil",
        className="com. smlcodes.examples.jaxWsAnnotations.webservice.CTFResponse")
        float celsiusToFarhenheit(float celsius);
}
```

8.@Oneway

`@Oneway` JAX-WS annotation is applied to WebMethod which means that method will have only input and no output. When a `@Oneway` method is called, control is returned to calling method even before the actual operation is performed. It means that nothing will escape method neither response neither exception.

```
@WebService
@SOAPBinding(style = Style.RPC)
public interface WSAnnotationsOnewayI {
        @WebMethod
        @Oneway
        void sayHello();
}
```

9.@HandlerChain

Web Services and their clients may need to access the SOAP message for additional processing of the message request or response. A SOAP message handler provides a mechanism for intercepting the SOAP message during request and response.

A handler at server side can be a validator. Let's say we want to validate the temperature before the actual service method is called. To do this our validator class shall implement interface `SOAPHandler`

```
package handler;

public class TemperatureValidator implements SOAPHandler {

        @Override
        public boolean handleMessage(SOAPMessageContext context) {
                // TODO Auto-generated method stub
                return false;
        }

        @Override
        public boolean handleFault(SOAPMessageContext context) {
                // TODO Auto-generated method stub
                return false;
        }

        @Override
        public void close(MessageContext context) {
                // TODO Auto-generated method stub

        }

        @Override
        public Set getHeaders() {
                // TODO Auto-generated method stub
                return null;
        }

}
```

```
// soap-handler.xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<javaee:handler-chains xmlns:javaee="http://java.sun.com/xml/ns/javaee"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <javaee:handler-chain>
                <javaee:handler>
                        <javaee:handler-
class>com.smlcodes.examples.jaxWsAnnotations.handler.TemperatureValidator
                        </javaee:handler-class>
                </javaee:handler>
        </javaee:handler-chain>
</javaee:handler-chains>
```

```
package handler;

@WebService
@SOAPBinding(style = Style.RPC)
public interface WSAnnotationsHandlerChainI {
        @HandlerChain(file = "soap-handler.xml")
        @WebMethod
        float celsiusToFarhenheit(float celsius);
}
```

2.3 JAX-WS RPC Style

1. RPC style web services use **method name and parameters to generate XML structure**.
2. The generated **WSDL is difficult to be validated against schema**.
3. In RPC style, **SOAP message is sent as many elements**.
4. RPC **style message is tightly coupled.**
5. In RPC style, **SOAP message keeps the operation name**.
6. In RPC style, **parameters are sent as discrete values**.

Steps to create JAX-WS RPC Style Example

1. JAX-WS Web Service End Point files

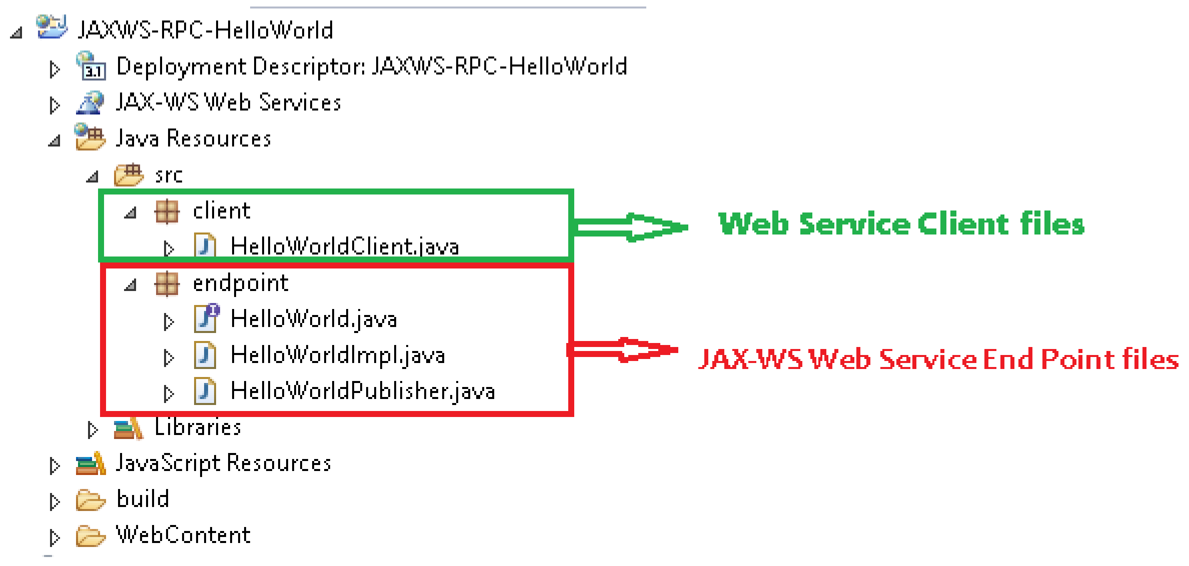
1. Create a Web Service Endpoint Interface with **@SOAPBinding(style = Style.RPC)**
2. Create a Web Service Endpoint Implementation
3. Create an Endpoint Publisher
4. Test generated WSDL. Ex: **http://localhost:8080/ws/hello?wsdl**

2. Web Service Client files

1. Java Web Service Client

In general words, "web service endpoint" is a service which published outside for user to access; where "web service client" is the party who access the published service.

Example : Hello World using JAX-WS RPC Style


```
JAXWS-RPC-HelloWorld
  Deployment Descriptor: JAXWS-RPC-HelloWorld
  JAX-WS Web Services
  Java Resources
    src
      client
        HelloWorldClient.java
      endpoint
        HelloWorld.java
        HelloWorldImpl.java
        HelloWorldPublisher.java
    Libraries
  JavaScript Resources
  build
  WebContent
```

1. JAX-WS Web Service End Point files

1. Create a Web Service Endpoint Interface

```
package endpoint;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;
//Service Endpoint Interface
@WebService
@SOAPBinding(style = Style.RPC)
public interface HelloWorld{
 @WebMethod
 String getHelloWorldMsg(String msg);
}
```

2. Create a Web Service Endpoint Implementation

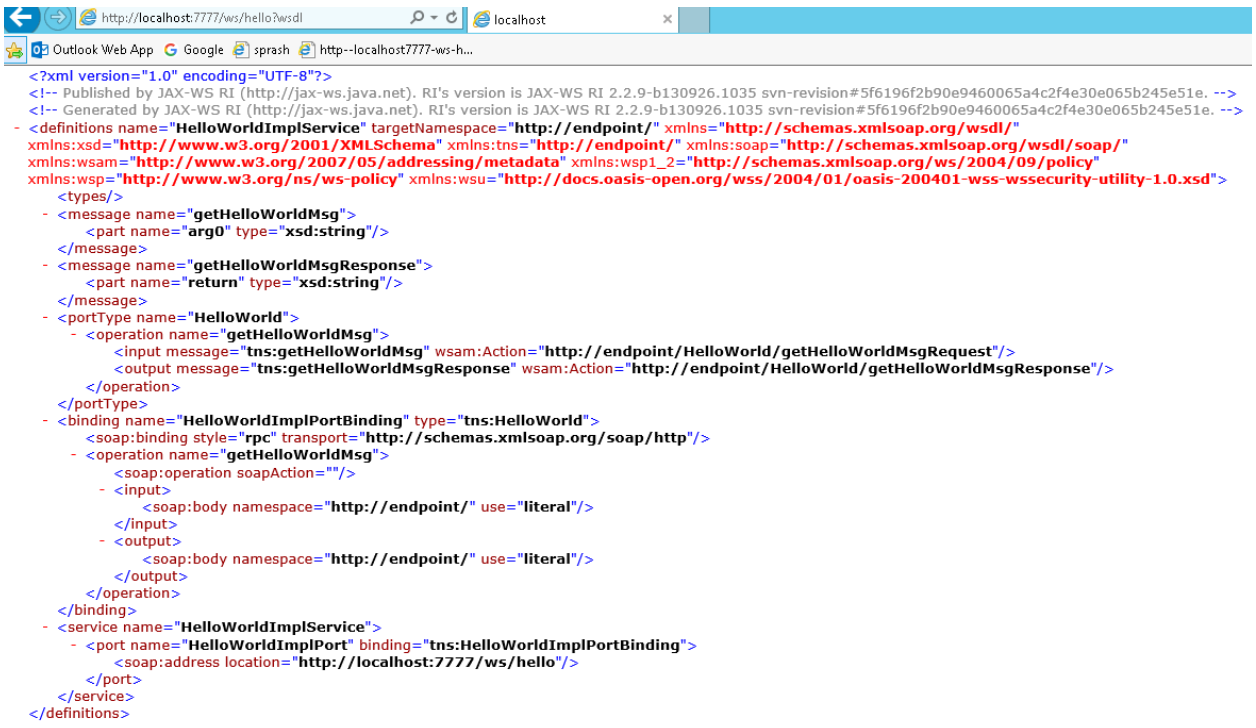
```
package endpoint;
import javax.jws.WebService;
//Service Implementation
@WebService(endpointInterface = "endpoint.HelloWorld")
public class HelloWorldImpl implements HelloWorld{
        @Override
        public String getHelloWorldMsg(String msg) {
                // TODO Auto-generated method stub
                return "Your Message from WebService is : "+msg;
        }
}
```

3. Create an Endpoint Publisher

```
package endpoint;
import javax.xml.ws.Endpoint;
//Endpoint publisher
public class HelloWorldPublisher{
    public static void main(String[] args) {
       Endpoint.publish("http://localhost:7777/ws/hello", new HelloWorldImpl());
       System.out.println("WSDL Published !!");
        }
}
```

4. Test generated WSDL

Run HelloWorldPublisher as Java Application & access url: ***http://localhost:7777/ws/hello?wsdl***



http://localhost:7777/ws/hello?wsdl

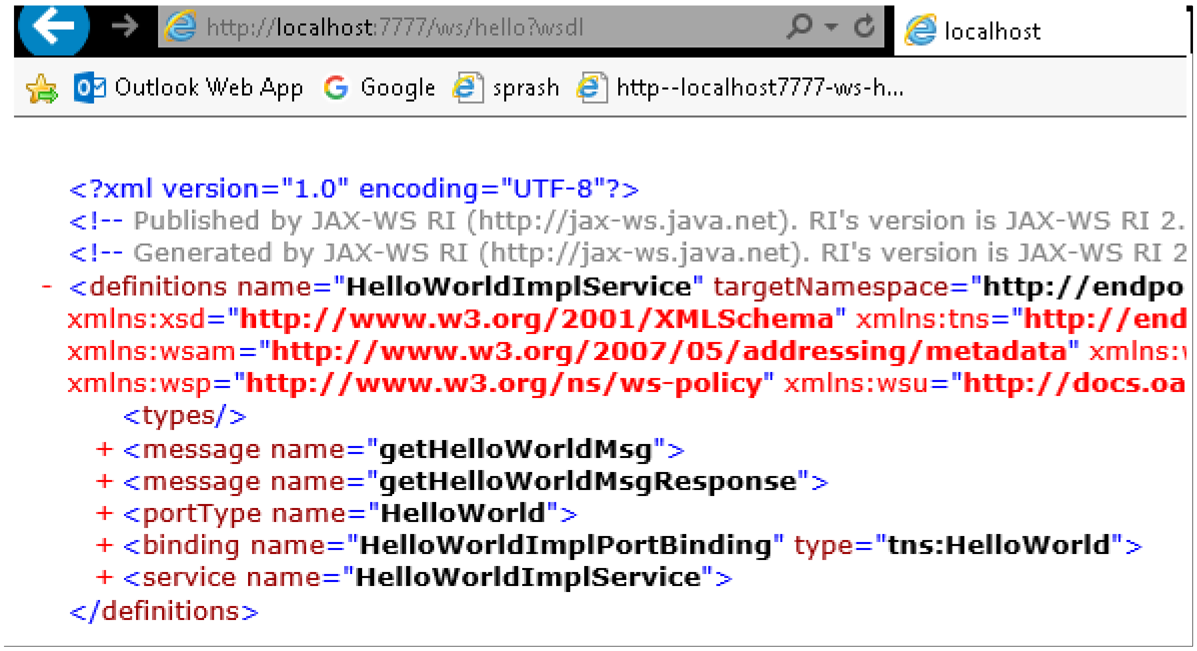
localhost

Outlook Web App Google sprash http--localhost7777-ws-h...

```
  <?xml version="1.0" encoding="UTF-8"?>
  <!-- Published by JAX-WS RI (http://jax-ws.java.net). RI's version is JAX-WS RI 2.2.9-b130926.1035 svn-revision#5f6196f2b90e9460065a4c2f4e30e065b245e51e. -->
  <!-- Generated by JAX-WS RI (http://jax-ws.java.net). RI's version is JAX-WS RI 2.2.9-b130926.1035 svn-revision#5f6196f2b90e9460065a4c2f4e30e065b245e51e. -->
- <definitions name="HelloWorldImplService" targetNamespace="http://endpoint/" xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://endpoint/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wsp="http://www.w3.org/ns/ws-policy" xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
     <types/>
   - <message name="getHelloWorldMsg">
        <part name="arg0" type="xsd:string"/>
     </message>
   - <message name="getHelloWorldMsgResponse">
        <part name="return" type="xsd:string"/>
     </message>
   - <portType name="HelloWorld">
      - <operation name="getHelloWorldMsg">
           <input message="tns:getHelloWorldMsg" wsam:Action="http://endpoint/HelloWorld/getHelloWorldMsgRequest"/>
           <output message="tns:getHelloWorldMsgResponse" wsam:Action="http://endpoint/HelloWorld/getHelloWorldMsgResponse"/>
        </operation>
     </portType>
   - <binding name="HelloWorldImplPortBinding" type="tns:HelloWorld">
        <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
      - <operation name="getHelloWorldMsg">
           <soap:operation soapAction=""/>
         - <input>
              <soap:body namespace="http://endpoint/" use="literal"/>
           </input>
         - <output>
              <soap:body namespace="http://endpoint/" use="literal"/>
           </output>
        </operation>
     </binding>
   - <service name="HelloWorldImplService">
      - <port name="HelloWorldImplPort" binding="tns:HelloWorldImplPortBinding">
           <soap:address location="http://localhost:7777/ws/hello"/>
        </port>
     </service>
  </definitions>
```

http://endpoint/" uses package name of Service endpoint publisher

the main components of WSDL documents are as below.



http://localhost:7777/ws/hello?wsdl

localhost

Outlook Web App Google sprash http--localhost7777-ws-h...

```
  <?xml version="1.0" encoding="UTF-8"?>
  <!-- Published by JAX-WS RI (http://jax-ws.java.net). RI's version is JAX-WS RI 2.
  <!-- Generated by JAX-WS RI (http://jax-ws.java.net). RI's version is JAX-WS RI 2
- <definitions name="HelloWorldImplService" targetNamespace="http://endpo
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://end
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" xmlns:
  xmlns:wsp="http://www.w3.org/ns/ws-policy" xmlns:wsu="http://docs.oa
     <types/>
   + <message name="getHelloWorldMsg">
   + <message name="getHelloWorldMsgResponse">
   + <portType name="HelloWorld">
   + <binding name="HelloWorldImplPortBinding" type="tns:HelloWorld">
   + <service name="HelloWorldImplService">
  </definitions>
```

hello.xml

WSDL Explanation

1. first Meaage part contains service method name & parameter list

```
<message name="getHelloWorldMsg">
        <part name="arg0" type="xsd:string"/>
</message>
```

2. Second Meaage part contains autogenerated Response method & return type

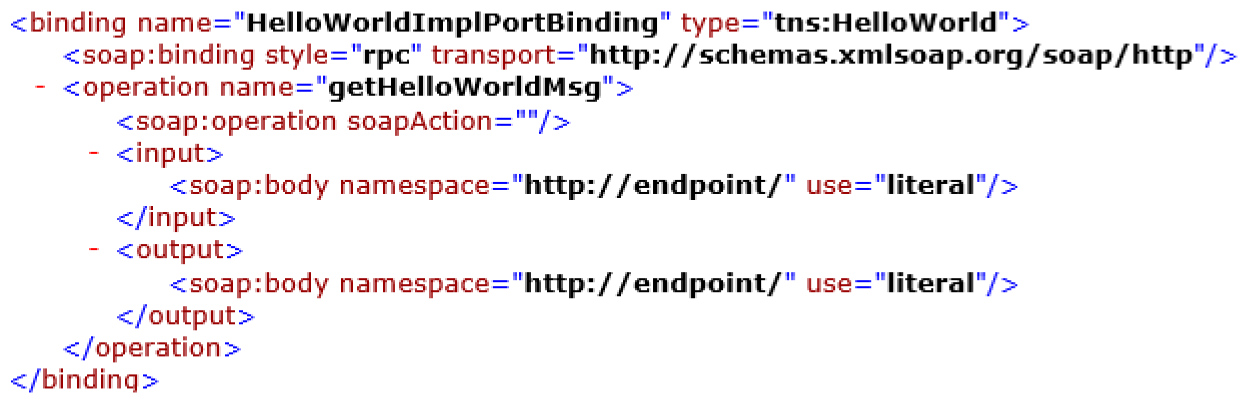
```
<message name="getHelloWorldMsgResponse">
        <part name="return" type="xsd:string"/>
</message>
```

3. PortType information is about ServiceEndpoint interface & input,output action urls

```
<portType name="HelloWorld">
<operation name="getHelloWorldMsg">
        <input message="tns:getHelloWorldMsg"
        wsam:Action="http://endpoint/HelloWorld/getHelloWorldMsgRequest"/>
        <output message="tns:getHelloWorldMsgResponse"
        wsam:Action="http://endpoint/HelloWorld/getHelloWorldMsgResponse"/>
</operation>
</portType>
```

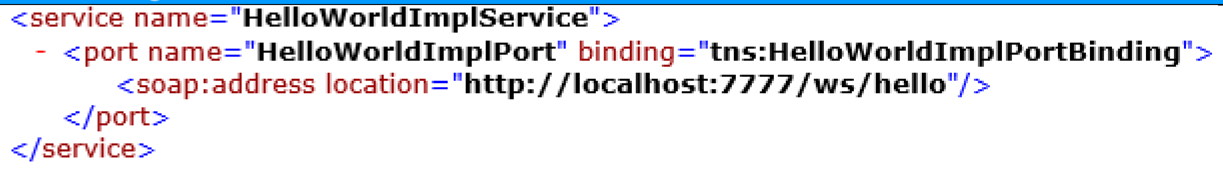
Here http://**endpoint** **it will take package name as automatically if we won't provide anything**

4. Binding will generate automatically by taking RPC Style/ Document Style



```
<binding name="HelloWorldImplPortBinding" type="tns:HelloWorld">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  - <operation name="getHelloWorldMsg">
        <soap:operation soapAction=""/>
      - <input>
            <soap:body namespace="http://endpoint/" use="literal"/>
        </input>
      - <output>
            <soap:body namespace="http://endpoint/" use="literal"/>
        </output>
    </operation>
</binding>
```

5. Service tag contains service details & WSDL document location



```
<service name="HelloWorldImplService">
  - <port name="HelloWorldImplPort" binding="tns:HelloWorldImplPortBinding">
        <soap:address location="http://localhost:7777/ws/hello"/>
    </port>
</service>
```

2. Web Service Client file

Follow below steps to write Webservice client

1. Create **URL** object by passing WSDL document location

   ```
   URL url = new URL("http://localhost:7777/ws/hello?wsdl");
   ```

2. Create **QName** by passing service URI, Service name as arguments

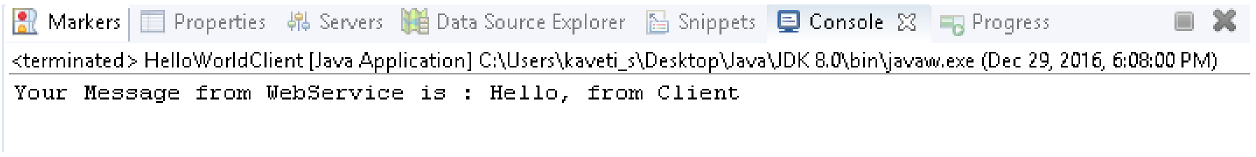
```
QName qname = new QName("http://endpoint/", "HelloWorldImplService");
```

3. Create Service Object by **calling create (-,-)** by passing URL,QName as arguments. Service objects provide the client view of a Web service. ports available on a service can be enumerated using the getPorts method

   ```
   Service service  = Service.create(url, qname);
   HelloWorld hello = service.getPort(HelloWorld.class);
   ```

```
package client;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;

import endpoint.HelloWorld;
public class HelloWorldClient{
    public static void main(String[] args) throws Exception {
    URL url = new URL("http://localhost:7777/ws/hello?wsdl");

        //1st argument service URI, refer to wsdl document above
        //2nd argument is service name, refer to wsdl document above
        QName qname = new QName("http://endpoint/", "HelloWorldImplService");
       Service service = Service.create(url, qname);
       HelloWorld hello = service.getPort(HelloWorld.class);
       System.out.println(hello.getHelloWorldMsg("Hello, from Client"));
    }
}
```

By running Clinet application we will get output as below



```
<terminated> HelloWorldClient [Java Application] C:\Users\kaveti_s\Desktop\Java\JDK 8.0\bin\javaw.exe (Dec 29, 2016, 6:08:00 PM)
Your Message from WebService is : Hello, from Client
```

2.4 JAX-WS Document Style

1. **SOAP Body can be structurted in any way you like**
2. Document style web services can be **validated against predefined schema**.
3. In document style, **SOAP message is sent as a single document**.
4. Document **style message is loosely coupled**.
5. In Document style, SOAP message loses the operation name.
6. In Document style, parameters are sent in XML format.

In JAX-WS development, convert from "*RPC style*" to "*Document style*" is very easy, just change the @SOAPBinding style option

Example

1. JAX-WS Web Service End Point files

1. Create a Web Service Endpoint Interface with **@SOAPBinding(style = Style.Document)**

```
package endpoint;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;
//Service Endpoint Interface
@WebService
@SOAPBinding(style = Style.DOCUMENT)
public interface HelloWorld{
 @WebMethod
 String getHelloWorldMsg(String msg);
}
```

2. Create a Web Service Endpoint Implementation

```
package endpoint;

import javax.jws.WebService;
//Service Implementation
@WebService(endpointInterface = "endpoint.HelloWorld")
public class HelloWorldImpl implements HelloWorld{
        @Override
        public String getHelloWorldMsg(String msg) {
                // TODO Auto-generated method stub
                return "Your Message from WebService is : "+msg;
        }
}
```

3. Create an Endpoint Publisher & Run as Java Application

```
package endpoint;

import javax.xml.ws.Endpoint;
//Endpoint publisher
public class HelloWorldPublisher{
    public static void main(String[] args) {
       Endpoint.publish("http://localhost:7771/ws/hellodoc", new HelloWorldImpl());
       System.out.println("WSDL Published !!");
        }
}
```

4. Test generated WSDL. Ex: http://localhost:7771/ws/hellodoc**?wsdl**

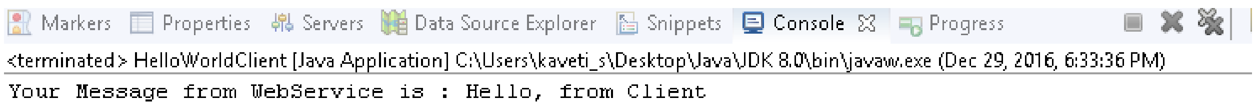
hellodoc.xml

```
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://endpoint/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wsp="http://www.w3.org/ns/ws-policy" xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
 - <types>
   - <xsd:schema>
       <xsd:import namespace="http://endpoint/" schemaLocation="http://localhost:7771/ws/hellodoc?xsd=1"/>
     </xsd:schema>
   </types>
 - <message name="getHelloWorldMsg">
     <part name="parameters" element="tns:getHelloWorldMsg"/>
   </message>
 - <message name="getHelloWorldMsgResponse">
     <part name="parameters" element="tns:getHelloWorldMsgResponse"/>
   </message>
 - <portType name="HelloWorld">
   - <operation name="getHelloWorldMsg">
       <input message="tns:getHelloWorldMsg" wsam:Action="http://endpoint/HelloWorld/getHelloWorldMsgRequest"/>
       <output message="tns:getHelloWorldMsgResponse" wsam:Action="http://endpoint/HelloWorld/getHelloWorldMsgResponse"/>
     </operation>
   </portType>
 - <binding name="HelloWorldImplPortBinding" type="tns:HelloWorld">
     <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
   - <operation name="getHelloWorldMsg">
       <soap:operation soapAction=""/>
     - <input>
         <soap:body use="literal"/>
       </input>
     - <output>
         <soap:body use="literal"/>
       </output>
     </operation>
   </binding>
 - <service name="HelloWorldImplService">
   - <port name="HelloWorldImplPort" binding="tns:HelloWorldImplPortBinding">
       <soap:address location="http://localhost:7771/ws/hellodoc"/>
     </port>
   </service>
</definitions>
```

2. Web Service Client files

Create Java Web Service Client & Run as Java Application

```java
package client;

import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;

import endpoint.HelloWorld;
public class HelloWorldClient{
    public static void main(String[] args) throws Exception {
    URL url = new URL("http://localhost:7771/ws/hellodoc?wsdl");

        //1st argument service URI, refer to wsdl document above
        //2nd argument is service name, refer to wsdl document above
        QName qname = new QName("http://endpoint/", "HelloWorldImplService");
       Service service = Service.create(url, qname);
       HelloWorld hello = service.getPort(HelloWorld.class);
       System.out.println(hello.getHelloWorldMsg("Hello, from Client"));
    }
}
```



```
<terminated> HelloWorldClient [Java Application] C:\Users\kaveti_s\Desktop\Java\JDK 8.0\bin\javaw.exe (Dec 29, 2016, 6:33:36 PM)
Your Message from WebService is : Hello, from Client
```

2.5 JAX-WS Tools

So far we created WebService applications manually. We have some tools to generate web service classes.lets start understanding them

2.5.1 wsimport tool

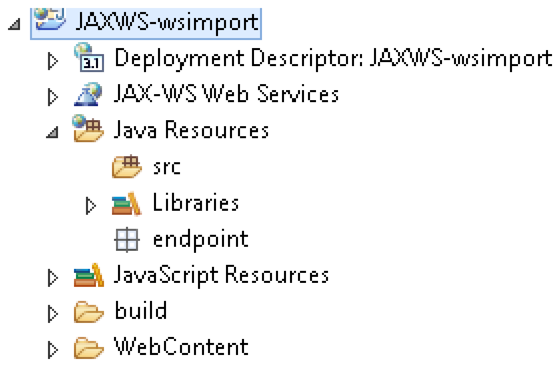
The **wsimport** tool is used to parse an existing Web Services Description Language (WSDL) file and generate required files (JAX-WS portable artifacts & JAX-WS Web Service End Point files).

We have to write web service client to access the published web services. This wsimport tool is available in the **$JDK/bin(**`C:\Users\kaveti_s\Desktop\Java\JDK 8.0\bin\wsimport.exe)` folder. We no need add these tools to PATH, because they are built in tools

In this example we are using JAXWS-Doc-HelloWorld published WSDL to generate JAX-WS portable artifacts. The WSDL URL is

To generate JAX-WS portable artifacts using wsimport tool follow below steps

1. Create an Empty Project & create endpoint package for saving generated artifacts

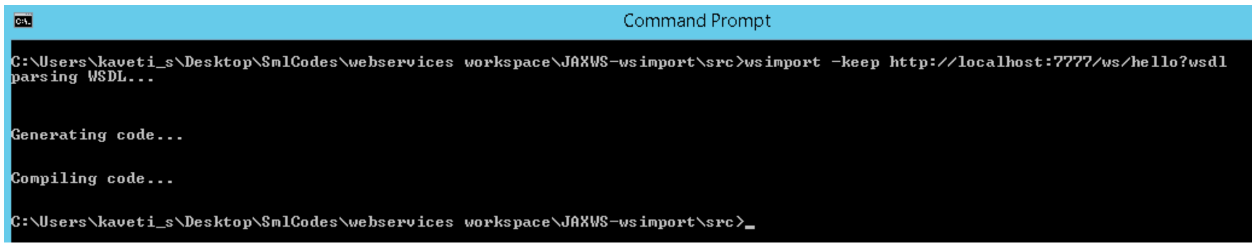


2. **Open command promt→ go to project location run wsimport with wsdl doc location as below**

```
wsimport  wsdl-location-path  -d -keep
```

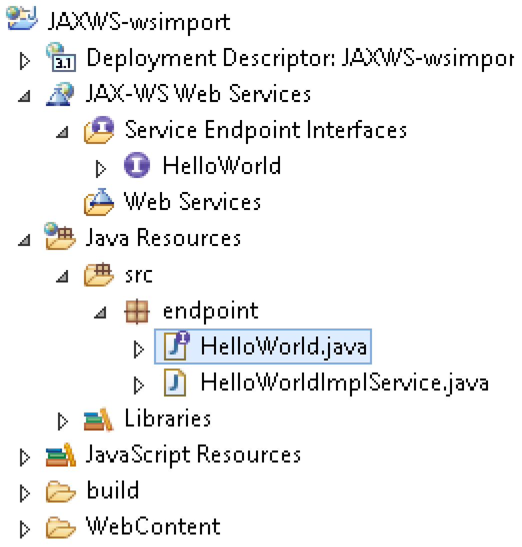
- **wsdl-location-path** : Is the location of wsdl file existence.
- **-d** : specify the directory where all the generated classes should be placed.
- **-keep** : It will keep the java source code of generated classes in the respective directory mentioned.

```
>wsimport -keep http://localhost:7777/ws/hello?wsdl
```



```
C:\Users\kaveti_s\Desktop\SmlCodes\webservices workspace\JAXWS-wsimport\src>wsimport -keep http://localhost:7777/ws/hello?wsdl
parsing WSDL...


Generating code...

Compiling code...

C:\Users\kaveti_s\Desktop\SmlCodes\webservices workspace\JAXWS-wsimport\src>_
```

By running this it is generated Service Endpoint files as below



```
JAXWS-wsimport
  ▷ Deployment Descriptor: JAXWS-wsimpor
  ◢ JAX-WS Web Services
      ◢ Service Endpoint Interfaces
          ▷ HelloWorld
        Web Services
  ◢ Java Resources
      ◢ src
          ◢ endpoint
              ▷ HelloWorld.java
              ▷ HelloWorldImplService.java
      ▷ Libraries
  ▷ JavaScript Resources
  ▷ build
  ▷ WebContent
```

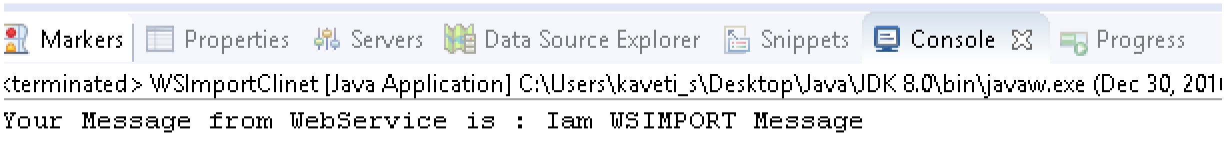
```java
// HelloWorld.java
package endpoint;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.xml.ws.Action;

/**
 * This class was generated by the JAX-WS RI. JAX-WS RI 2.2.9-b130926.1035 Generated source version: 2.2*/
@WebService(name = "HelloWorld", targetNamespace = "http://endpoint/")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public interface HelloWorld {

        /**
         * @param arg0
         * @return returns java.lang.String
         */
        @WebMethod
        @WebResult(partName = "return")
        @Action(input = "http://endpoint/HelloWorld/getHelloWorldMsgRequest", output =
"http://endpoint/HelloWorld/getHelloWorldMsgResponse")
        public String getHelloWorldMsg(@WebParam(name = "arg0", partName = "arg0") String arg0);

}
```

```java
package endpoint;
import java.net.MalformedURLException;
import java.net.URL;
import javax.xml.*;
@WebServiceClient(name = "HelloWorldImplService", targetNamespace = "http://endpoint/", wsdlLocation =
"http://localhost:7777/ws/hello?wsdl")
public class HelloWorldImplService extends Service {

        private final static URL HELLOWORLDIMPLSERVICE_WSDL_LOCATION;
        private final static WebServiceException HELLOWORLDIMPLSERVICE_EXCEPTION;
        private final static QName HELLOWORLDIMPLSERVICE_QNAME = new QName("http://endpoint/",
"HelloWorldImplService");

        static {
                URL url = null;
                WebServiceException e = null;
                try {
                        url = new URL("http://localhost:7777/ws/hello?wsdl");
                } catch (MalformedURLException ex) {
                        e = new WebServiceException(ex);
                }
                HELLOWORLDIMPLSERVICE_WSDL_LOCATION = url;
                HELLOWORLDIMPLSERVICE_EXCEPTION = e;
        }

        public HelloWorldImplService() {
                super(__getWsdlLocation(), HELLOWORLDIMPLSERVICE_QNAME);
        }

        public HelloWorldImplService(WebServiceFeature... features) {
                super(__getWsdlLocation(), HELLOWORLDIMPLSERVICE_QNAME, features);
        }

        public HelloWorldImplService(URL wsdlLocation) {
                super(wsdlLocation, HELLOWORLDIMPLSERVICE_QNAME);
        }

        public HelloWorldImplService(URL wsdlLocation, WebServiceFeature... features) {
                super(wsdlLocation, HELLOWORLDIMPLSERVICE_QNAME, features);
        }

        public HelloWorldImplService(URL wsdlLocation, QName serviceName) {
                super(wsdlLocation, serviceName);
        }

        public HelloWorldImplService(URL wsdlLocation, QName serviceName, WebServiceFeature... features) {
                super(wsdlLocation, serviceName, features);
        }

        /**
         * @return returns HelloWorld
         */
        @WebEndpoint(name = "HelloWorldImplPort")
        public HelloWorld getHelloWorldImplPort() {
           return super.getPort(new QName("http://endpoint/", "HelloWorldImplPort"), HelloWorld.class);
        }

        @WebEndpoint(name = "HelloWorldImplPort")
        public HelloWorld getHelloWorldImplPort(WebServiceFeature... features) {
           return super.getPort(new QName("http://endpoint/", "HelloWorldImplPort"), HelloWorld.class,
features);
        }

        private static URL __getWsdlLocation() {
                if (HELLOWORLDIMPLSERVICE_EXCEPTION != null) {
                        throw HELLOWORLDIMPLSERVICE_EXCEPTION;
                }
                return HELLOWORLDIMPLSERVICE_WSDL_LOCATION;
        }

}
```

Now, create a Java web service client which depends on the above generated files

```
package client;

import endpoint.HelloWorld;
import endpoint.HelloWorldImplService;

public class WSImportClinet {
public static void main(String[] args) {
        HelloWorldImplService service = new HelloWorldImplService();
        HelloWorld helloWorld = service.getHelloWorldImplPort();
        String output =helloWorld.getHelloWorldMsg("Iam WSIMPORT Message");
        System.out.println(output);      }
}
```

Run this application we will get following Output



```
<terminated> WSImportClinet [Java Application] C:\Users\kaveti_s\Desktop\Java\JDK 8.0\bin\javaw.exe (Dec 30, 201
Your Message from WebService is : Iam WSIMPORT Message
```

2.5.2 wsgen tool

The `wsgen` tool is used to parse an existing web service implementation class and generates required files (JAX-WS portable artifacts) for web service deployment. This `wsgen` tool is available in `$JDK/bin` folder.

2 common use cases for wsgen tool:

1. ***Generates JAX-WS portable artifacts (Java files) for web service deployment.***
2. ***Generates WSDL and xsd files***
3. ***Create web service client for testing***

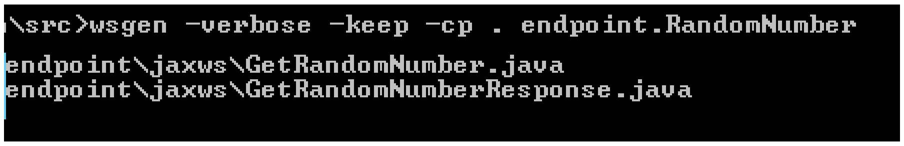
We need to create web service implementation class, remaing files will be generated by `wsgen` tool

```
package endpoint;
import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
public class RandomNumber {
        @WebMethod
        public String getRandomNumber() {
                return "Random Number Is : " + Math.random();
        }
}
```

1. Generates JAX-WS portable artifacts (Java files) for web service deployment.

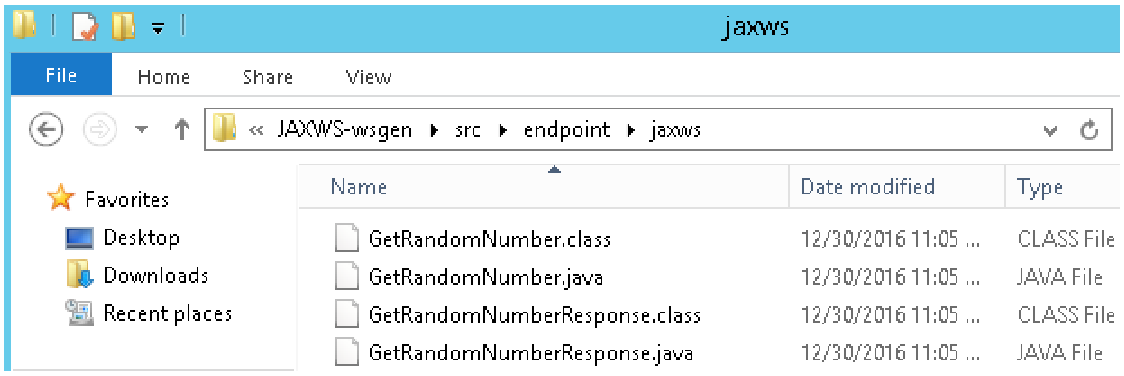
To generate all the JAX-WS portable artifacts for above web service implementation class (`RandomNumber.java`), use following command by going src folder from command prompt

```
>wsgen -verbose -keep -cp . endpoint.RandomNumber
```



```
\src>wsgen -verbose -keep -cp . endpoint.RandomNumber

endpoint\jaxws\GetRandomNumber.java
endpoint\jaxws\GetRandomNumberResponse.java
```

It will generate 2 .java files & 2 .class files



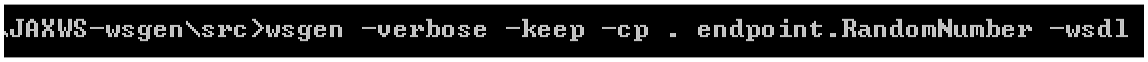
Name	Date modified	Type
GetRandomNumber.class	12/30/2016 11:05 ...	CLASS File
GetRandomNumber.java	12/30/2016 11:05 ...	JAVA File
GetRandomNumberResponse.class	12/30/2016 11:05 ...	CLASS File
GetRandomNumberResponse.java	12/30/2016 11:05 ...	JAVA File

```java
// GetRandomNumber.java
package endpoint.jaxws;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(name = "getRandomNumber", namespace = "http://endpoint/")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "getRandomNumber", namespace = "http://endpoint/")
public class GetRandomNumber {


}
```

```java
// GetRandomNumberResponse.java
package endpoint.jaxws;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(name = "getRandomNumberResponse", namespace = "http://endpoint/")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "getRandomNumberResponse", namespace = "http://endpoint/")
public class GetRandomNumberResponse {

        @XmlElement(name = "return", namespace = "")
        private String _return;

        /**
         *
         * @return returns String
         */
        public String getReturn() {
                return this._return;
        }

        /**
         *
         * @param _return
         *            the value for the _return property
         */
        public void setReturn(String _return) {
                this._return = _return;
        }

}
```

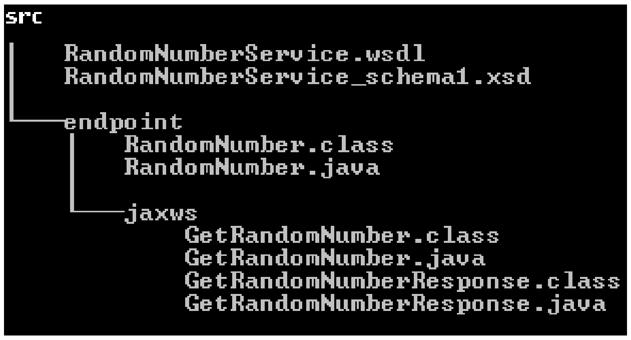
2. Genarates WSDL and xsd

To generate WSDL and xsd files for above web service implementation class (`RandomNumber.java`), add an extra **-wsdl** in the `wsgen` command



```
.JAXWS-wsgen\src>wsgen -verbose -keep -cp . endpoint.RandomNumber -wsdl
```

In this case it will generate 6 files (**2 java +2 class + 1 WSDL + 1 schema.xsd**). Files under src/ folder are



```
src
│   RandomNumberService.wsdl
│   RandomNumberService_schema1.xsd
│
└───endpoint
    │   RandomNumber.class
    │   RandomNumber.java
    │
    └───jaxws
            GetRandomNumber.class
            GetRandomNumber.java
            GetRandomNumberResponse.class
            GetRandomNumberResponse.java
```

RandomNumberService_schema1.xsd

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" targetNamespace="http://endpoint/" xmlns:tns="http://endpoint/"
xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="getRandomNumber" type="tns:getRandomNumber"/>
  <xs:element name="getRandomNumberResponse" type="tns:getRandomNumberResponse"/>
  <xs:complexType name="getRandomNumber">
    <xs:sequence/>
  </xs:complexType>
  <xs:complexType name="getRandomNumberResponse">
    <xs:sequence>
      <xs:element name="return" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

RandomNumberService.wsdl

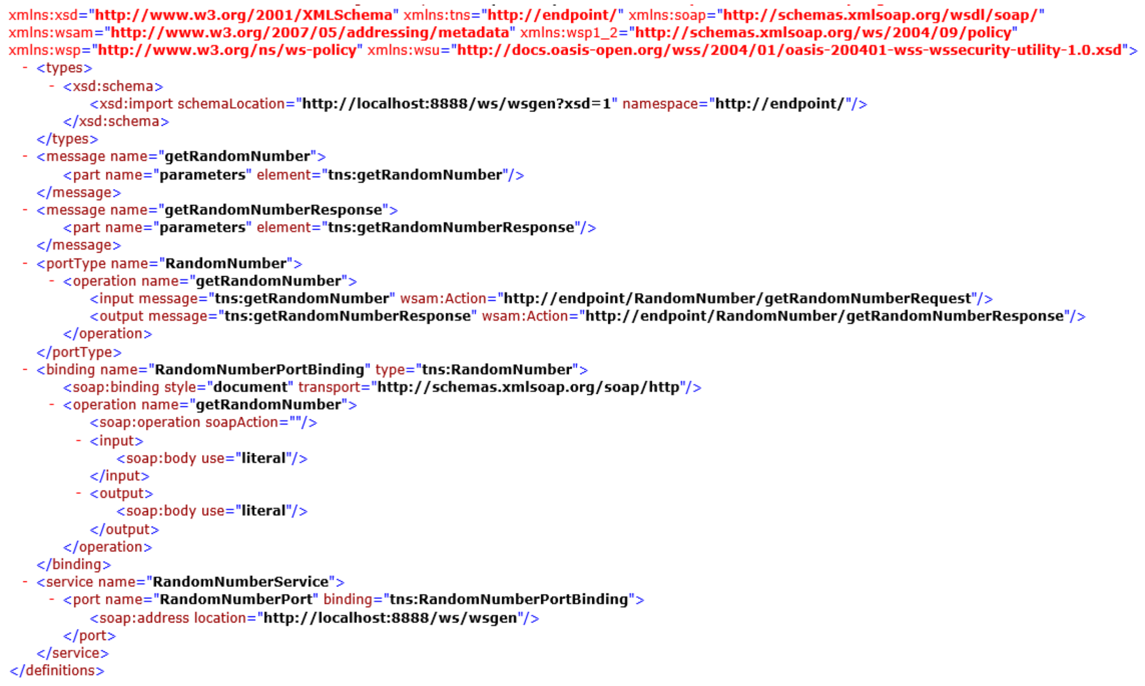
```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://endpoint/" name="RandomNumberService"
xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:wsp="http://www.w3.org/ns/ws-policy"
xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy" xmlns:tns="http://endpoint/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://endpoint/" schemaLocation="RandomNumberService_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="getRandomNumber">
    <part name="parameters" element="tns:getRandomNumber"/>
  </message>
  <message name="getRandomNumberResponse">
    <part name="parameters" element="tns:getRandomNumberResponse"/>
  </message>
  <portType name="RandomNumber">
    <operation name="getRandomNumber">
      <input wsam:Action="http://endpoint/RandomNumber/getRandomNumberRequest"
message="tns:getRandomNumber"/>
      <output wsam:Action="http://endpoint/RandomNumber/getRandomNumberResponse"
message="tns:getRandomNumberResponse"/>
    </operation>
  </portType>
  <binding name="RandomNumberPortBinding" type="tns:RandomNumber">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="getRandomNumber">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="RandomNumberService">
    <port name="RandomNumberPort" binding="tns:RandomNumberPortBinding">
      <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
    </port>
  </service>
</definitions>
```

All files are ready.we have to write Publisher class to publish the WSDL document

```
package endpoint;
import javax.xml.ws.Endpoint;
public class RandomNumberPublisher {
        public static void main(String[] args) {
                Endpoint.publish("http://localhost:8888/ws/wsgen", new RandomNumber());
                System.out.println("Service is published!");
        }
}
```

Run as Java Application. It will shows output as `Service is published!`

http://localhost:8888/ws/wsgen?wsdl it is as same as generated WSDL document



```
 xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://endpoint/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
 xmlns:wsp="http://www.w3.org/ns/ws-policy" xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
 - <types>
   - <xsd:schema>
       <xsd:import schemaLocation="http://localhost:8888/ws/wsgen?xsd=1" namespace="http://endpoint/"/>
     </xsd:schema>
   </types>
 - <message name="getRandomNumber">
     <part name="parameters" element="tns:getRandomNumber"/>
   </message>
 - <message name="getRandomNumberResponse">
     <part name="parameters" element="tns:getRandomNumberResponse"/>
   </message>
 - <portType name="RandomNumber">
   - <operation name="getRandomNumber">
       <input message="tns:getRandomNumber" wsam:Action="http://endpoint/RandomNumber/getRandomNumberRequest"/>
       <output message="tns:getRandomNumberResponse" wsam:Action="http://endpoint/RandomNumber/getRandomNumberResponse"/>
     </operation>
   </portType>
 - <binding name="RandomNumberPortBinding" type="tns:RandomNumber">
     <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
   - <operation name="getRandomNumber">
       <soap:operation soapAction=""/>
     - <input>
         <soap:body use="literal"/>
       </input>
     - <output>
         <soap:body use="literal"/>
       </output>
     </operation>
   </binding>
 - <service name="RandomNumberService">
   - <port name="RandomNumberPort" binding="tns:RandomNumberPortBinding">
       <soap:address location="http://localhost:8888/ws/wsgen"/>
     </port>
   </service>
 </definitions>
```

Finally Note these

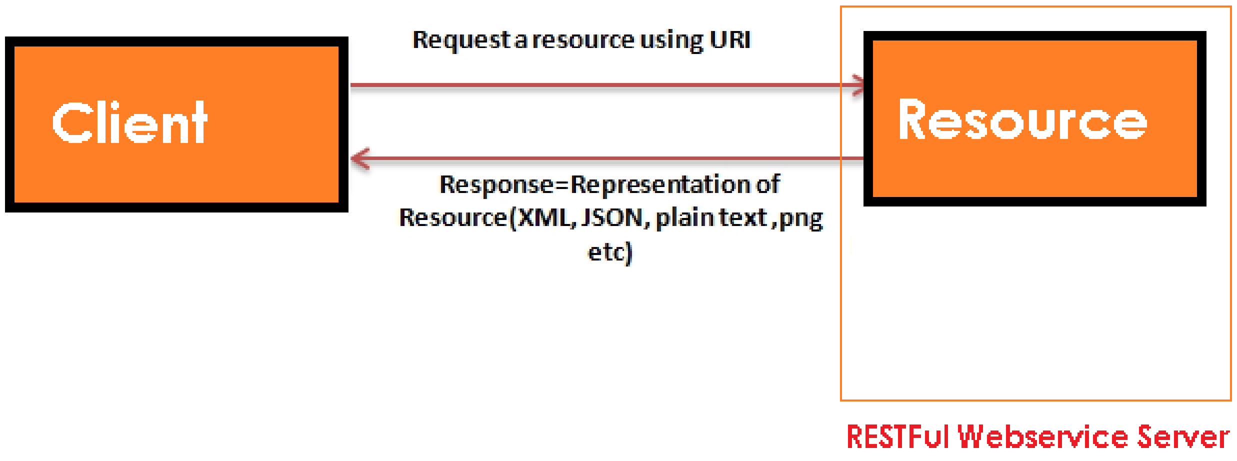
Wsimport → Uses the WSDL, generates java code for the service/client implementation.

Wsgen → Uses compiled code, generates WSDL (and artifacts)

3. JAX-RS (RESTFul web services)

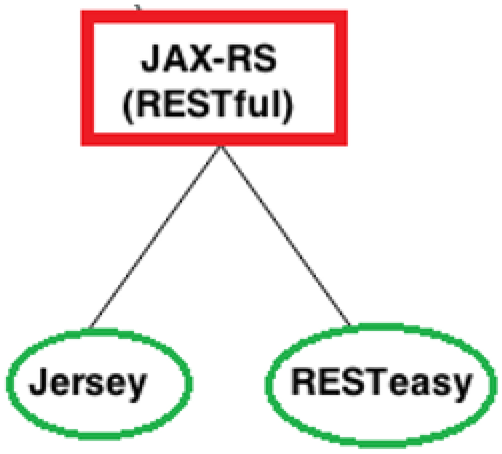
JAX-RS provides the implementation of **RESTful** web services, JAX-RS is a specification for RESTful Web Services with Java and it is given by Sun. Since it is a specification, other frameworks can be written to implement these specifications, and that includes ***Jersey*** from Oracle, ***Resteasy*** from Jboss, ***CXF*** from Apache, etc.

We can get the resource from RESTful service in different formats like, **HTML,XML,JSON,TEXT,PDF** and in the Image formats as well, but in real time we mainly we will prefer JSON. REST guidelines always talks about stateless communication between client and the Server. Stateless means, every single request from client to server will be considered as a fresh request. Because of this reason REST always prefers to choose HTTP as it a stateless protocol.


Request a resource using URI
Client
Resource
Response=Representation of Resource(XML, JSON, plain text ,png etc)
RESTFul Webservice Server

There are two main implementation of JAX-RS API.

1. **Jersey**
2. **RESTeasy**



3.1 JAX-RS Annotations

We have many annotations. But below are the majorly used annotations in RESTFul webservices

- **@Path('Path')**
- **@GET**
- **@POST**
- **@PUT**
- **@DELETE**
- **@Produces(MediaType.TEXT_PLAIN [, more-types])**
- **@Consumes(type[, more-types])**
- **@PathParam()**
- **@QueryParam()**
- **@MatrixParam()**
- **@FormParam()**

1.@Path()

- Its a Class & Method level of annotation
- This will check the path next to the base URL

Syntax: http**://localhost:(port)/<YourApplicationName>/<UrlPattern In Web.xml>/<path>**

Here `<path>` is the part of URI, and this will be identified by @path annotation at class/method level.

2.@GET

Its a method level of annotation, this annotation indicates that the following method should respond to the HTTP GET request only, if we annotate our method with @GET, the execution flow will enter that following method if we send GET request from the client

3.@POST

It's a method level of annotation, this annotation indicates that the following method should respond to the HTTP POST request only.

4.@PUT

It's a method level of annotation, this annotation indicates that the following method should respond to the HTTP PUT request only.

5.@DELETE

It's a method level of annotation, this annotation indicates that the following method should respond to the HTTP DELETE request only.

6.@Produces

It's a method or field level annotation, this tells which **MIME** type is delivered by the method annotated with **@GET**. Whenever we send a HTTP GET request to our RESTful service, it will invokes particular method and produces the output in different formats. There you can specifies in what are all formats (MIME) your method can produce the output, by using @produces annotation.

Remember: We will use @Produces annotation for GET requests only.

7.@Consumes

This is a class and method level annotation, this will define which MIME type is consumed by the particular method. It means in which format the **method can accept the input from the client**.

@PathParam, @QueryParam, @MatrixParam **annotations will come into picture in case if we are passing the input values to the restful service through the URL**

8.@PathParam

http://localhost:8001/<Rest Service Name>/rest/customers/100/Satya

Here the two parameters appear in the end of the above URL [100 & Satya], which are separated by forward slash **(/)** are called as path parameters

9.@QueryParam

http://localhost:8001/.../rest/customers?custNo=100&custName=Satya

If the client sends an input in the form of query string in the URL, then those parameters are called as Query Parameters. If you observe the above syntax, client passing **custNo=100&custName=Satya** started after question mark **(?)** symbol and each parameter is separated by **&** symbol, those parameters are called as query parameters.

10.@MatrixParam

http://localhost:8001/.../rest/customers;custNo=100;custName=Satya

Matrix parameters are another way defining the parameters to be added to URL. If you observe the above syntax, client is passing two parameters each are separated by **semicolon (;),** these parameters are called as matrix parameters. **Remember these parameters may appear any where in the path**.

11.@ FormParam

If we have a HTML form having two input fields and submit button. Lets client enter those details and submit to the RESTful web service. Then the rest service will extract those details by using this **@FormParam** annotation.

3.2 JAX-RS JERSEY

Jersey, reference implementation to develope RESTful web service based on the JAX-RS (JSR 311) specification.

If we want to implement Webservices using Jersey we need to download Jersey jar files from **Jersey website**

The major change between JERSEY & RESTEASY is just changing the configuration in web.xml

Download & install Maven, configure maven in Eclipse

Steps to Create Jersey Web Service Application

1. Create Dynamic web project in eclipse, convert that into Maven Project

2. Add Jersey jar files manually / through Maven by writing repo details in pom.xml

3. Create RESTFul webservice

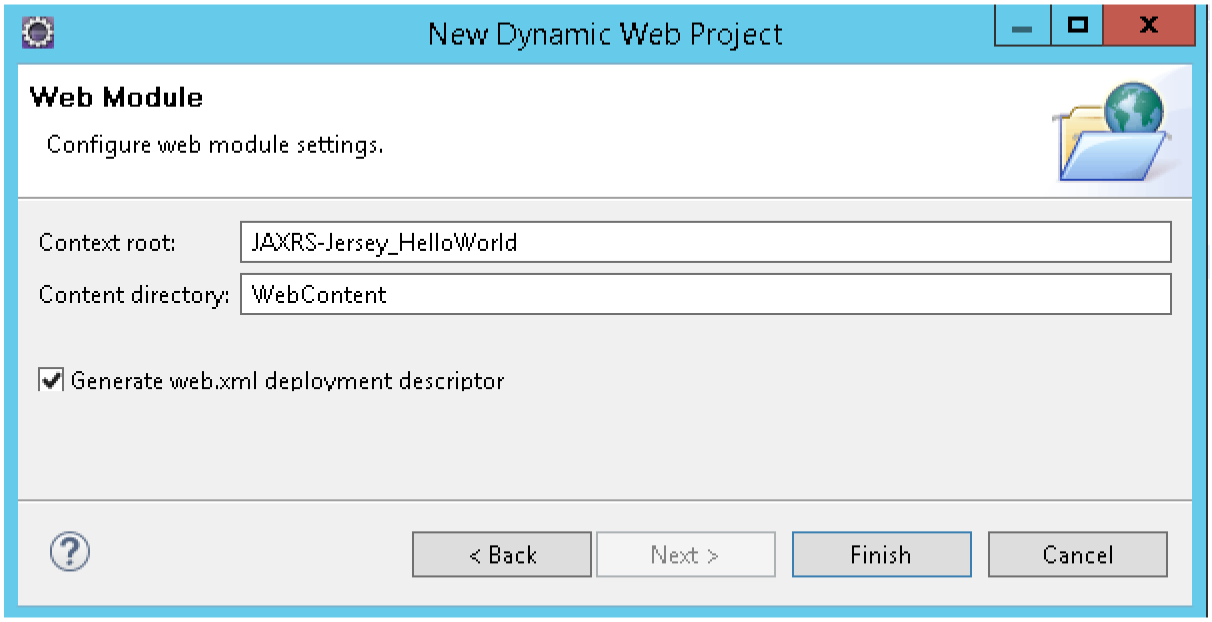
4. Configure `web.xml`

5. Test Webservice directly by using URL / writing webservice client

Example: JAXRS-Jersey-HelloWorld

1. Create Dynamic web project in eclipse, convert that into Maven Project

New → Dynamic web project → Provide project details → finish


New Dynamic Web Project

Web Module

Configure web module settings.

Context root: JAXRS-Jersey_HelloWorld

Content directory: WebContent

☑ Generate web.xml deployment descriptor

Right-click on Project →Configure → Convert to Maven Project .

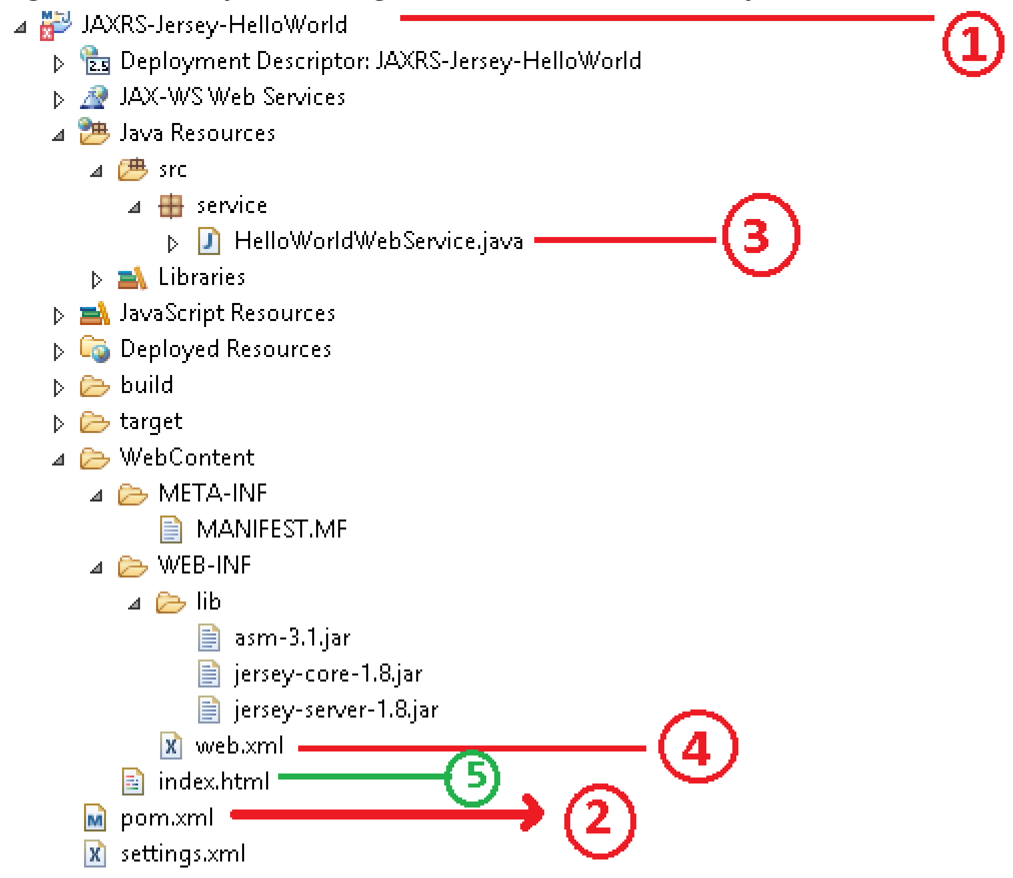


Figure 1 Directory Structure after adding all files

2. Add Jersey jar files manually / through Maven by writing repo details in pom.xml

Jersey is published in Java.net Maven repository. To develop Jersey REST application, just declares “jersey-server” in Maven **pom.xml**.

```
<project …>
        <repositories>
                <repository>
                        <id>maven2-repository.java.net</id>
                        <name>Java.net Repository for Maven</name>
                        <url>http://download.java.net/maven/2/</url>
                        <layout>default</layout>
                </repository>
        </repositories>

        <dependencies>
                <dependency>
                        <groupId>com.sun.jersey</groupId>
                        <artifactId>jersey-server</artifactId>
                        <version>1.8</version>
                </dependency>
                <dependency>
                        <groupId>com.sun.jersey</groupId>
                        <artifactId>jersey-client</artifactId>
                        <version>1.19.3</version>
                </dependency>
        </dependencies>
</project>
```

3. Create RESTFul webservice at Server End

```
package service;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
@Path("/hellojersey")
public class HelloWorldWebService {
  // This method is called if HTML and XML is not requested
  @GET
  @Produces(MediaType.TEXT_PLAIN)
  public String sayPlainTextHello() {
    return "Hello Jersey Plain";
  }

  // This method is called if HTML is requested
  @GET
  @Produces(MediaType.TEXT_HTML)
  public String sayHtmlHello() {
    return "<h1>" + "Hello Jersey HTML" + "</h1>";
  }
}
```

4.Configure `web.xml`

In `web.xml`, register "`com.sun.jersey.spi.container.servlet.ServletContainer`", and puts your Jersey service folder under "**init-param**", "`com.sun.jersey.config.property.packages`".

```
<?xml version="1.0" encoding="UTF-8"?><web-app id="WebApp_ID" version="2.4"
        xmlns="http://java.sun.com/xml/ns/j2ee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
        <display-name>Restful Web Application</display-name>

        <servlet>
                <servlet-name>jersey-serlvet</servlet-name>
                <servlet-class>
                     com.sun.jersey.spi.container.servlet.ServletContainer
                </servlet-class>
                <init-param>
                     <param-name>com.sun.jersey.config.property.packages</param-name>
                     <param-value>service</param-value>
                </init-param>
                <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>
```

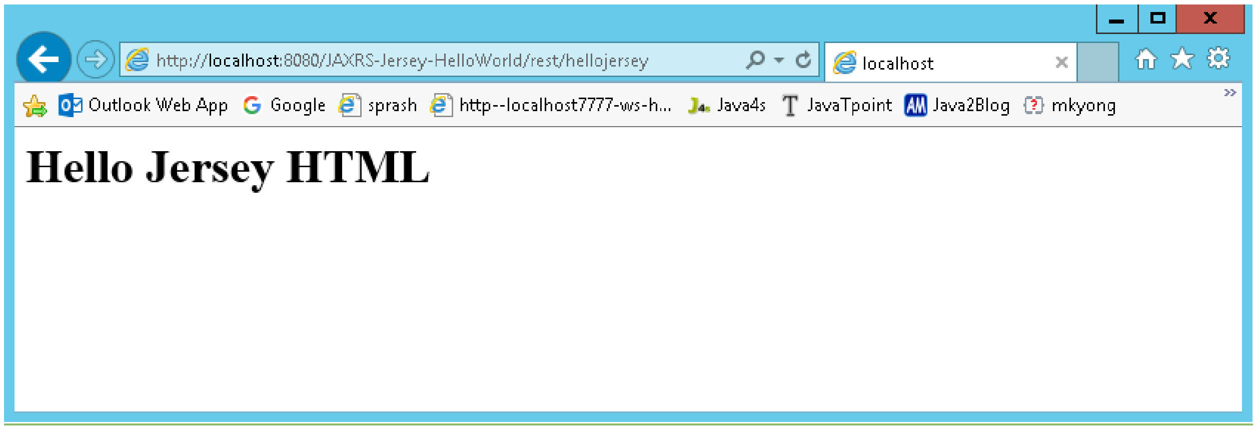
5. Test Webservice directly by using URL / writing webservice client

In this example, web request from "**projectURL/rest/hellojersy/**" will match to **"HelloWorldWebService",** via @Path("/hellojersey") So we are created a test index.html conatining following url for testing purpose

Index.html

```
<h1>Test JERSEY WEBSERVICE </h1>

<h3><a href="rest/hellojersey">Default</a></h3>
```

Direct Testing URL : **http://localhost:8080/JAXRS-Jersey-HelloWorld/rest/hellojersey**


http://localhost:8080/JAXRS-Jersey-HelloWorld/rest/hellojersey
localhost
Outlook Web App
Google
sprash
http--localhost7777-ws-h...
Java4s
JavaTpoint
Java2Blog
mkyong

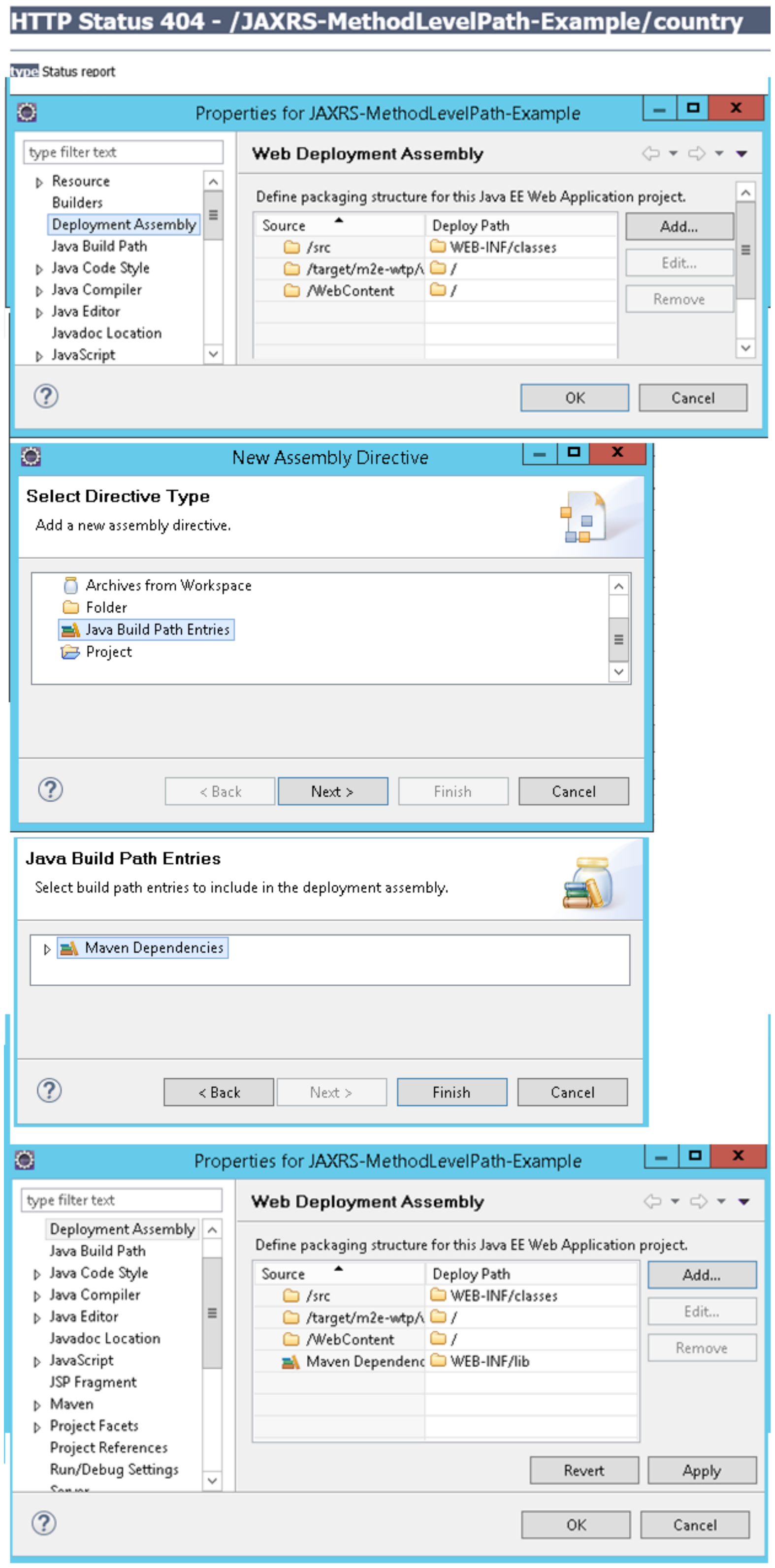
Hello Jersey HTML

We can write The HelloworldClientTest.java file is created inside the server application. But you can run client code by other application also by having service interface and jersey jar file.

```
package client;

import java.net.URI;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.UriBuilder;
import org.glassfish.jersey.client.ClientConfig;

public class HelloworldClientTest {
        public static void main(String[] args) {
                ClientConfig config = new ClientConfig();
                Client client = ClientBuilder.newClient(config);
                WebTarget target = client.target(getBaseURI());
                // Now printing the server code of different media type

        System.out.println(target.path("rest").path("hellojersey").request().accept(MediaType.TEXT_PLAIN).
get(String.class));

        System.out.println(target.path("rest").path("hellojersey").request().accept(MediaType.TEXT_HTML).g
et(String.class));
        }

        private static URI getBaseURI() {
                // here server is running on 4444 port number and project name is
                // restfuljersey
                return UriBuilder.fromUri("http://localhost:8080/JAXRS-Jersey-
HelloWorld/rest/hellojersey").build();
        }
}
```

If we got 404 error , follow below steps java.lang.ClassNotFoundException: com.sun.jersey.spi.container.servlet.ServletContainer

Right click on Project → Properties → Deployment Assembly : add : Java BuildPath Entities → Maven Dependencies → Finish



HTTP Status 404 - /JAXRS-MethodLevelPath-Example/country

type Status report

Properties for JAXRS-MethodLevelPath-Example

Web Deployment Assembly

Define packaging structure for this Java EE Web Application project.

Source	Deploy Path
/src	WEB-INF/classes
/target/m2e-wtp/\	/
/WebContent	/

New Assembly Directive

Select Directive Type

Add a new assembly directive.

- Archives from Workspace
- Folder
- Java Build Path Entries
- Project

Java Build Path Entries

Select build path entries to include in the deployment assembly.

- Maven Dependencies

Properties for JAXRS-MethodLevelPath-Example

Web Deployment Assembly

Define packaging structure for this Java EE Web Application project.

Source	Deploy Path
/src	WEB-INF/classes
/target/m2e-wtp/\	/
/WebContent	/
Maven Dependenc	WEB-INF/lib

3.3 JAX-RS RESTEasy

RESTEasy, JBoss project, implementation of the **JAX-RS** specification. In this article, we show you how to use RESTEasy framework to create a simple REST style web application

Downlaod **RESTEasy** jars from here or add RESTEasy dependencies in POM.xml

Steps to Create RESTEasy Web Service Application

1. Create Dynamic web project in eclipse, convert that into Maven Project

2. Add RESTEasy jar files manually / through Maven by writing repo details in pom.xml

3. Create RESTFul webservice using RESTEasy

4. Configure `web.xml, Register` RESTEasy dependency class

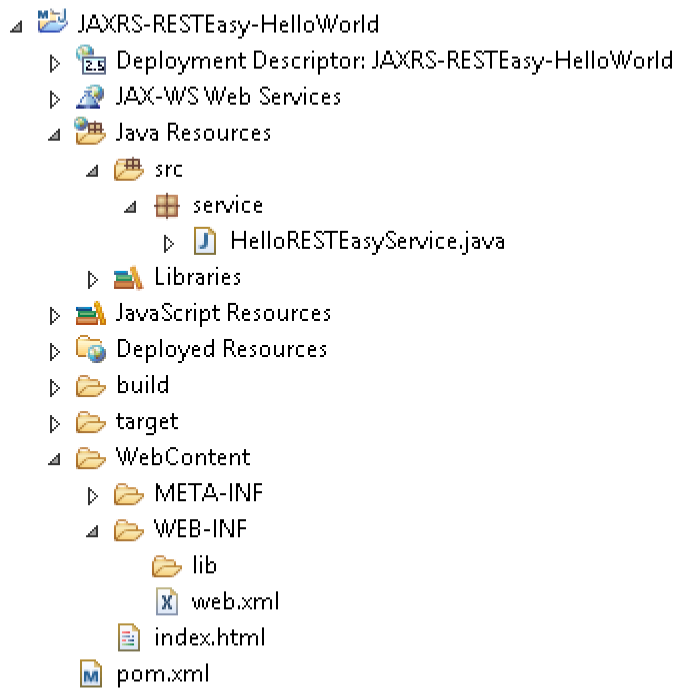
5. Test Webservice directly by using URL / writing webservice client

Example : JAXRS- RESTEasy –HelloWorld

1.Create Dynamic web project in eclipse, convert that into Maven Project

Create Dynamic Web Project : New → Dynamic web project → Provide project details → finish

Convert into Maven Project : Right-click on Project →Configure → Convert to Maven Project.



```
JAXRS-RESTEasy-HelloWorld
    Deployment Descriptor: JAXRS-RESTEasy-HelloWorld
    JAX-WS Web Services
    Java Resources
        src
            service
                HelloRESTEasyService.java
        Libraries
    JavaScript Resources
    Deployed Resources
    build
    target
    WebContent
        META-INF
        WEB-INF
            lib
            web.xml
        index.html
    pom.xml
```

2. Add RESTEasy jar files manually / through Maven by writing repo details in pom.xml

Declares JBoss public Maven repository and "**resteasy-jaxrs**" in your Maven pom.xml file. That's all you need to use **RESTEasy**.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>JAXRS-RESTEasy-HelloWorld</groupId>
  <artifactId>JAXRS-RESTEasy-HelloWorld</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
  <build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.5.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.0.0</version>
        <configuration>
          <warSourceDirectory>WebContent</warSourceDirectory>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <repositories>
           <repository>
                  <id>JBoss repository</id>
                  <url>https://repository.jboss.org/nexus/content/groups/public-jboss/</url>
           </repository>
        </repositories>

        <dependencies>
                <dependency>
                        <groupId>org.jboss.resteasy</groupId>
                        <artifactId>resteasy-jaxrs</artifactId>
                        <version>2.2.1.GA</version>
                </dependency>
        </dependencies>
</project>
```

3.Create RESTFul webservice using RESTEasy

```java
package service;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/helloresteasy")
public class HelloRESTEasyService {
        @GET
        @Produces(MediaType.TEXT_HTML)
        public String sayHtmlHello() {
                return "<h1>" + "Hello RESTEasy Service" + "</h1>";
        }
}
```

4. Configure `web.xml, Register` RESTEasy dependency class

Now, configure listener and servlet to support RESTEasy. Read this JBoss documentation for details

```
<web-app id="WebApp_ID" version="2.4"
        xmlns="http://java.sun.com/xml/ns/j2ee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
        <display-name>Restful Web Application</display-name>

        <!-- Auto scan REST service -->
        <context-param>
                <param-name>resteasy.scan</param-name>
                <param-value>true</param-value>
        </context-param>

        <!-- this need same with resteasy servlet url-pattern -->
        <context-param>
                <param-name>resteasy.servlet.mapping.prefix</param-name>
                <param-value>/rest</param-value>
        </context-param>

        <listener>
                <listener-class>
                        org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap
                </listener-class>
        </listener>

        <servlet>
                <servlet-name>resteasy-servlet</servlet-name>
                <servlet-class>
                        org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
                </servlet-class>
        </servlet>

        <servlet-mapping>
                <servlet-name>resteasy-servlet</servlet-name>
                <url-pattern>/rest/*</url-pattern>
        </servlet-mapping>

</web-app>
```

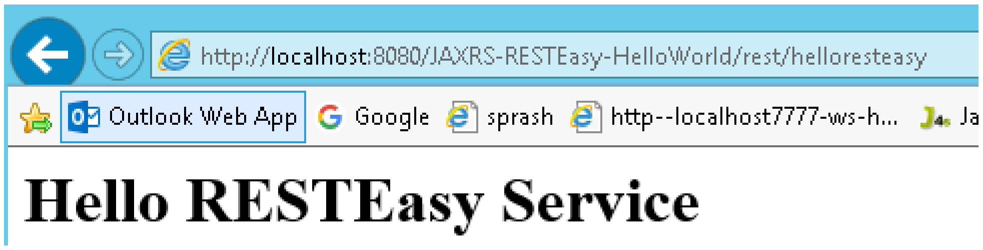
You need to set the "resteasy.servlet.mapping.prefix" if your servlet-mapping for the resteasy servlet has a url-pattern other than "/*".

In above example, the resteasy servlet url-pattern is "/rest/*", so you have to set the "resteasy.servlet.mapping.prefix" to "/rest" as well, otherwise, you will hit resource not found error message.

Remember to set "resteasy.scan" to true, so that RESTEasy will find and register your REST service automatically.

5.Test Webservice directly by using URL / writing webservice client

http://localhost:8080/JAXRS-RESTEasy-HelloWorld/rest/helloresteasy


http://localhost:8080/JAXRS-RESTEasy-HelloWorld/rest/helloresteasy
Outlook Web App
Google
sprash
http--localhost7777-ws-h...
Ja

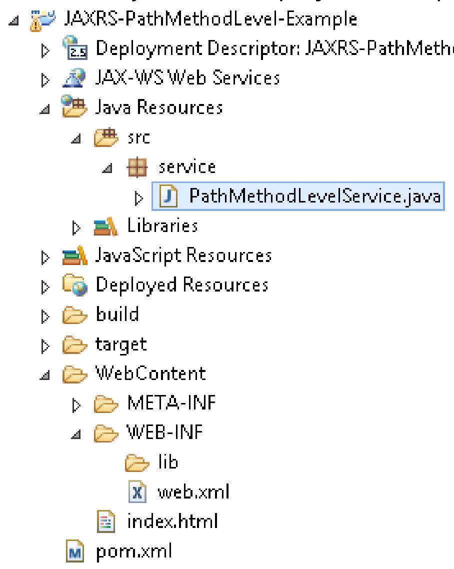
Hello RESTEasy Service

3.4 JAX-RS Examples

1: JAX-RS @Path annotation at Method Level Example

We can use `@Path` to bind URI pattern to a Java method

1. Create Dynamic web project in eclipse, convert that into Maven Project



```
JAXRS-PathMethodLevel-Example
  Deployment Descriptor: JAXRS-PathMethodLevel-Example
  JAX-WS Web Services
  Java Resources
    src
      service
        PathMethodLevelService.java
    Libraries
  JavaScript Resources
  Deployed Resources
  build
  target
  WebContent
    META-INF
    WEB-INF
      lib
      web.xml
    index.html
  pom.xml
```

2. Add Jersey jar files manually / through Maven by writing repo details in pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <repositories>
                <repository>
                        <id>maven2-repository.java.net</id>
                        <name>Java.net Repository for Maven</name>
                        <url>http://download.java.net/maven/2/</url>
                        <layout>default</layout>
                </repository>
        </repositories>

        <dependencies>
                <!-- https://mvnrepository.com/artifact/com.sun.jersey/jersey-server -->
                <dependency>
                        <groupId>com.sun.jersey</groupId>
                        <artifactId>jersey-server</artifactId>
                        <version>1.19.3</version>
                </dependency>


                <!-- https://mvnrepository.com/artifact/org.glassfish.jersey.core/jersey-client -->
                <dependency>
                        <groupId>org.glassfish.jersey.core</groupId>
                        <artifactId>jersey-client</artifactId>
                        <version>2.25</version>
                </dependency>

                <!-- https://mvnrepository.com/artifact/javax.ws.rs/javax.ws.rs-api -->
                <dependency>
                        <groupId>javax.ws.rs</groupId>
                        <artifactId>javax.ws.rs-api</artifactId>
                        <version>2.0</version>
                </dependency>
        </dependencies>
</project>
```

3. Create RESTFul webservice

```
package service;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Response;

@Path("/country")
public class PathMethodLevelService {

        @GET
        @Produces("text/html")
        public Response selectCountry() {
                String output = " Default Country : <h1>INDIA</h1>";
                return Response.status(200).entity(output).build();
        }

        @GET
        @Path("/usa")
        @Produces("text/html")
        public Response selectUSA() {
                String output = "Selected Country : <h1>United States of America(USA)</h1>";
                return Response.status(200).entity(output).build();
        }

        @GET
        @Path("/uk")
        @Produces("text/html")
        public Response selectUK() {
                String output = "Selected Country : <h1>UNITED KINGDOM(UK)</h1>";
                return Response.status(200).entity(output).build();
        }
}
```

4. Configure `web.xml`

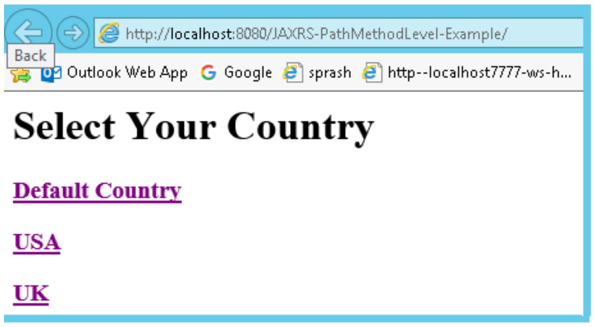
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:web="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" id="WebApp_ID" version="2.4">
  <display-name>JAXRS-PathMethodLevel-Example</display-name>
  <servlet>
    <servlet-name>jersey-serlvet</servlet-name>
    <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
    <init-param>
      <param-name>com.sun.jersey.config.property.packages</param-name>
      <param-value>service</param-value>
    </init-param>
    <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>
```

5. Test Webservice directly by using URL / writing webservice client

http://localhost:8080/JAXRS-PathMethodLevel-Example/ for Default country request

http://localhost:8080/JAXRS-PathMethodLevel-Example/rest/country/usa for usa

http://localhost:8080/JAXRS-PathMethodLevel-Example/rest/country/uk for UK

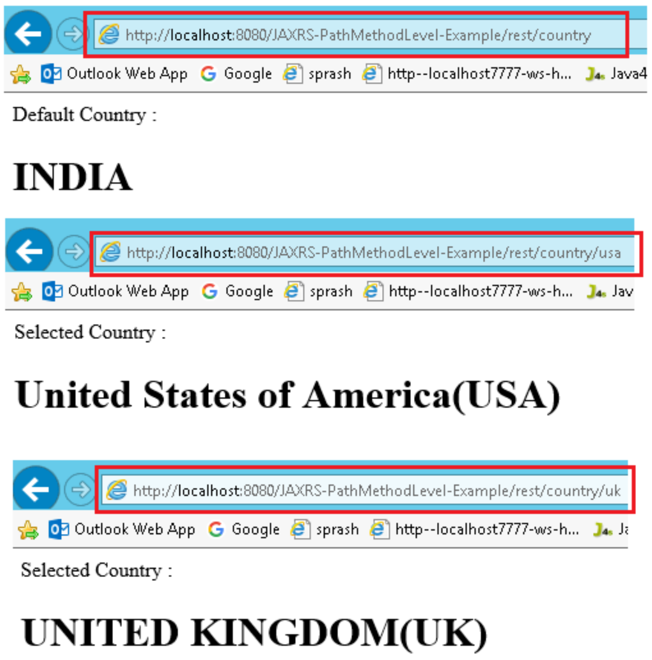

http://localhost:8080/JAXRS-PathMethodLevel-Example/

Select Your Country

Default Country

USA

UK


http://localhost:8080/JAXRS-PathMethodLevel-Example/rest/country

Default Country :

INDIA

http://localhost:8080/JAXRS-PathMethodLevel-Example/rest/country/usa

Selected Country :

United States of America(USA)

http://localhost:8080/JAXRS-PathMethodLevel-Example/rest/country/uk

Selected Country :

UNITED KINGDOM(UK)

Response Class in JAX-RS

javax.ws.rs.core.Response class is reserved for an extension by a JAX-RS implementation providers. An application should use one of the static methods to create a Response instance using a **ResponseBuilder**. An application class should not extend this class directly

We have following methods in Responnse classs which are used majorly

1. public abstract int **getStatus**()
2. public abstract MultivaluedMap<String,Object> **getMetadata**()
3. public static ResponseBuilder **status**(Response.StatusType status)
4. public static Response.ResponseBuilder **ok**()

2: JAX-RS @PathParam annotation Example

In RESTful (JAX-RS) web services **@PathParam** annotation will be used to bind RESTful URL parameter values with the method arguments

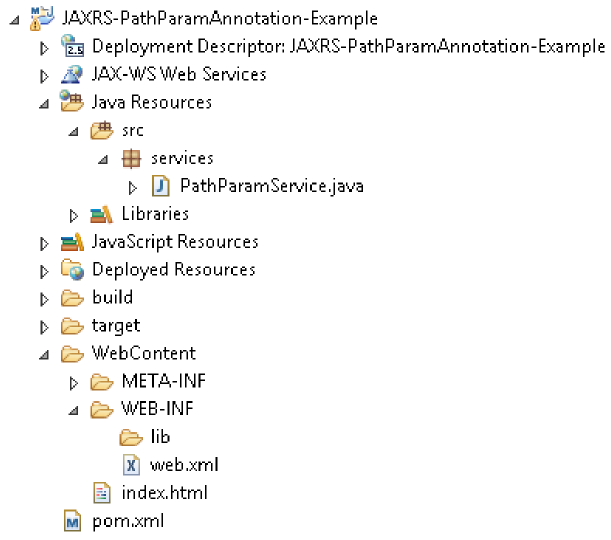
```
http://localhost:8001/<Rest Service Name>/rest/customers/100/Satya
```

Here the two parameters appear in the end of the above URL [100 & Satya], which are separated by forward slash **(/)** are called as path parameters

We will read those URL paramters in our webservice method using

```
@PathParam("paramname") String variablename
```

1. Create Dynamic web project in eclipse, convert that into Maven Project


```
JAXRS-PathParamAnnotation-Example
  Deployment Descriptor: JAXRS-PathParamAnnotation-Example
  JAX-WS Web Services
  Java Resources
    src
      services
        PathParamService.java
    Libraries
  JavaScript Resources
  Deployed Resources
  build
  target
  WebContent
    META-INF
    WEB-INF
      lib
      web.xml
    index.html
  pom.xml
```

2. **Add RESTEasy jar files manually / through Maven by writing repo details in pom.xml**

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>JAXRS-PathParamAnnotation-Example</groupId>
  <artifactId>JAXRS-PathParamAnnotation-Example</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>

  <repositories>
              <repository>
                      <id>maven2-repository.java.net</id>
                      <name>Java.net Repository for Maven</name>
                      <url>http://download.java.net/maven/2/</url>
                      <layout>default</layout>
              </repository>
      </repositories>

      <dependencies>
                      <dependency>
                      <groupId>com.sun.jersey</groupId>
                      <artifactId>jersey-server</artifactId>
                      <version>1.8</version>
              </dependency>
      </dependencies>

      <build>
              <finalName>JAXRS-PathParamAnnotation-Example</finalName>
              <plugins>
                      <plugin>
                              <artifactId>maven-compiler-plugin</artifactId>
                              <configuration>
                                      <compilerVersion>1.5</compilerVersion>
                                      <source>1.5</source>
                                      <target>1.5</target>
                              </configuration>
                      </plugin>
              </plugins>
      </build>

</project>
```

3. Create RESTFul webservice using RESTEasy

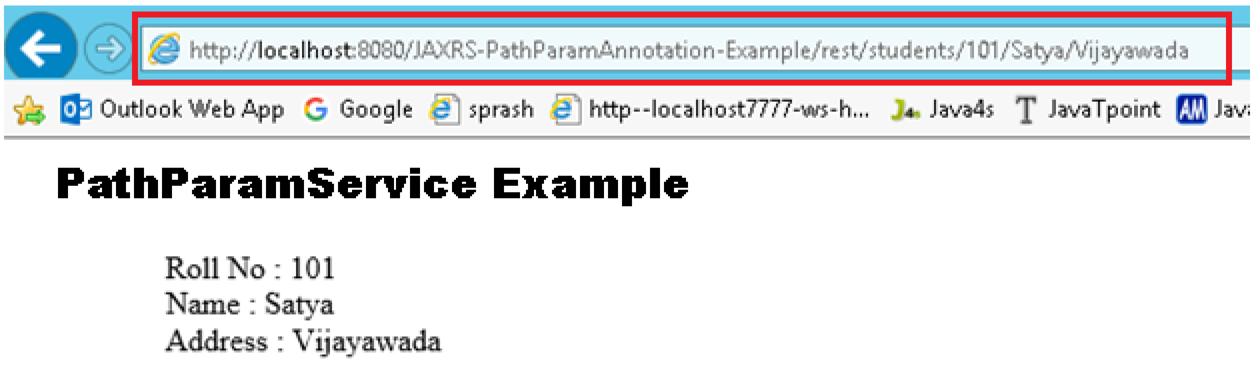
```
package services;


@Path("/students")
public class PathParamService {

        @GET
        @Path("{rollno}/{name}/{address}")
        @Produces("text/html")
        public Response getResultByPassingValue(
                        @PathParam("rollno") String rollno,
                                        @PathParam("name") String name,
                                        @PathParam("address") String address) {
                String output = "<h1>PathParamService Example</h1>";
                output = output+"<br>Roll No : "+rollno;
                output = output+"<br>Name : "+name;
                output = output+"<br>Address : "+address;
                return Response.status(200).entity(output).build();
        }
}
```

4. Configure `web.xml, Register` Jersey dependency class

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:web="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" id="WebApp_ID" version="2.4">
  <display-name>JAXRS-PathParamAnnotation-Example</display-name>
  <servlet>
    <servlet-name>jersey-serlvet</servlet-name>
    <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
    <init-param>
      <param-name>com.sun.jersey.config.property.packages</param-name>
      <param-value>services</param-value>
    </init-param>
    <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>
```

5. Test Webservice directly by using URL / writing webservice client

http://localhost:8080/JAXRS-PathParamAnnotation-Example/rest/students/101/Satya/Vijayawada


http://localhost:8080/JAXRS-PathParamAnnotation-Example/rest/students/101/Satya/Vijayawada
Outlook Web App
Google
sprash
http--localhost7777-ws-h...
Java4s
JavaTpoint

PathParamService Example

Roll No : 101
Name : Satya
Address : Vijayawada

Note : In Upcomming Examples POM.XML , Web.xml are same for all applications. So iam skipping those. If any changes in those files I will mention don't worry ☺

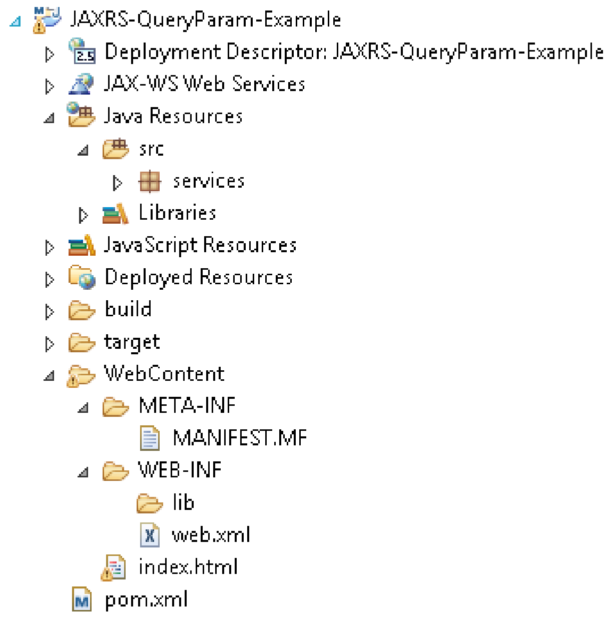
3: JAX-RS @QueryParam annotation Example

```
http://localhost:8001/…/rest/customers?custNo=100&custName=Satya
```

If the client sends an input in the form of query string in the URL, then those parameters are called as Query Parameters. If you observe the above syntax, client passing **custNo=100&custName=Satya** started after question mark **(?)** symbol and each parameter is separated by **&** symbol, those parameters are called as query parameters.

Steps to Implement this Web Service Application

1. Create Dynamic web project in eclipse, convert that into Maven Project


JAXRS-QueryParam-Example
Deployment Descriptor: JAXRS-QueryParam-Example
JAX-WS Web Services
Java Resources
src
services
Libraries
JavaScript Resources
Deployed Resources
build
target
WebContent
META-INF
MANIFEST.MF
WEB-INF
lib
web.xml
index.html
pom.xml

2. Add RESTEasy jar files manually / through Maven by writing repo details in **pom.xml(skip)**

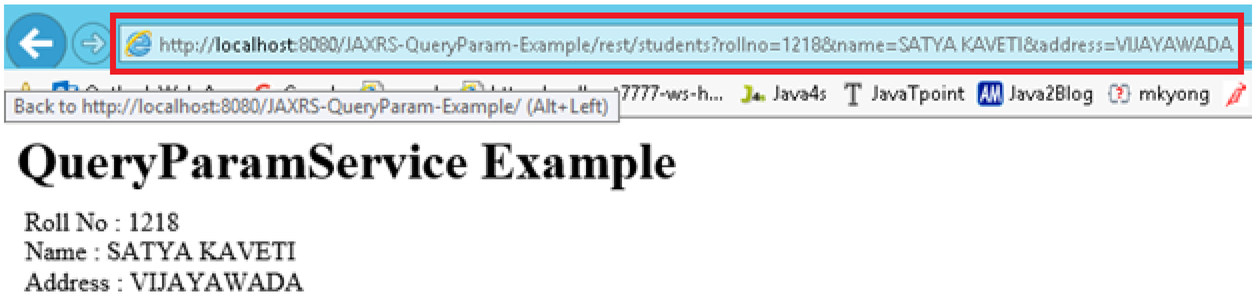
3. Create RESTFul webservice using Jersy

```
package services;

@Path("/students")
public class QueryParamService {
        @GET
        @Produces("text/html")
        public Response getResultByPassingValue(
                                        @QueryParam("rollno") String rollno,
                                        @QueryParam("name") String name,
                                        @QueryParam("address") String address) {
                String output = "<h1>QueryParamService Example</h1>";
                output = output+"<br>Roll No : "+rollno;
                output = output+"<br>Name : "+name;
                output = output+"<br>Address : "+address;
                return Response.status(200).entity(output).build();
        }
}
```

4. Configure `web.xml` **(SKIPING)**

5. Test Webservice directly by using URL / writing webservice client

http://localhost:8080/JAXRS-QueryParam-Example/rest/students?rollno=1218&name=SATYA KAVETI&address=VIJAYAWADA


http://localhost:8080/JAXRS-QueryParam-Example/rest/students?rollno=1218&name=SATYA KAVETI&address=VIJAYAWADA
Back to http://localhost:8080/JAXRS-QueryParam-Example/ (Alt+Left)
7777-ws-h... Java4s JavaTpoint Java2Blog mkyong

QueryParamService Example

Roll No : 1218
Name : SATYA KAVETI
Address : VIJAYAWADA

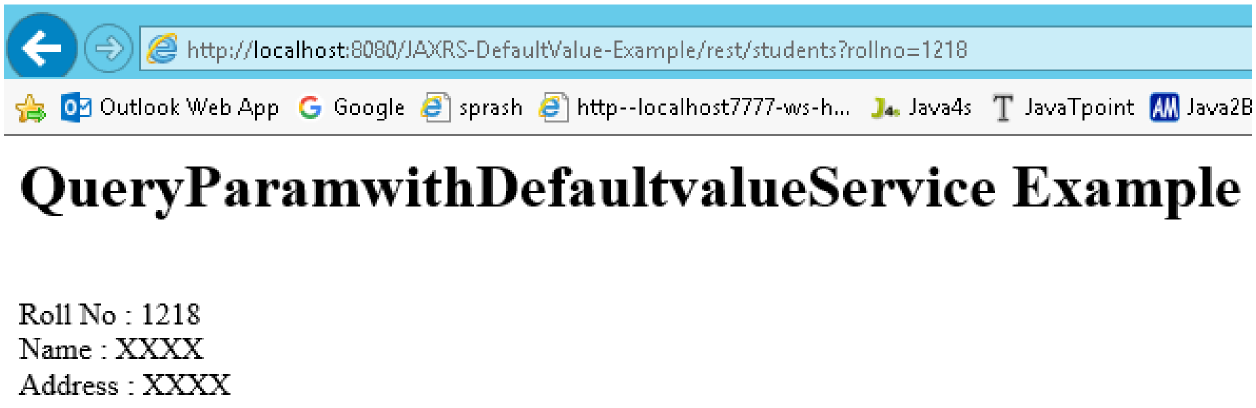
4. JAX-RS @DefaultValue Annotation

Sometimes URL doesn't contain the values which are expected the methods. In that situation we can use `@DefaultValue` for passing default values to method parameters. `@DefaultValue` is good for optional parameter.

```
package services;

import javax.ws.rs.DefaultValue;
import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("/students")
public class QueryParamwithDefaultvalueService {
        @GET
        @Produces("text/html")
        public Response getResultByPassingValue(@DefaultValue("1000") @QueryParam("rollno") String rollno,
                                @DefaultValue("XXXX") @QueryParam("name") String name,
                                @DefaultValue("XXXX") @QueryParam("address") String address) {
                String output = "<h1>QueryParamwithDefaultvalueService Example</h1>";
                output = output + "<br>Roll No : " + rollno;
                output = output + "<br>Name : " + name;
                output = output + "<br>Address : " + address;
                return Response.status(200).entity(output).build();
        }
}
```

http://localhost:8080/JAXRS-DefaultValue-Example/rest/students?rollno=1218

in Above URL we are not passing Name, Address paramaeter values. So it will take Default values passed in @DefaultValue("XXXX") annotation


http://localhost:8080/JAXRS-DefaultValue-Example/rest/students?rollno=1218
Outlook Web App Google sprash http--localhost7777-ws-h... Java4s JavaTpoint Java2B

QueryParamwithDefaultvalueService Example

Roll No : 1218
Name : XXXX
Address : XXXX

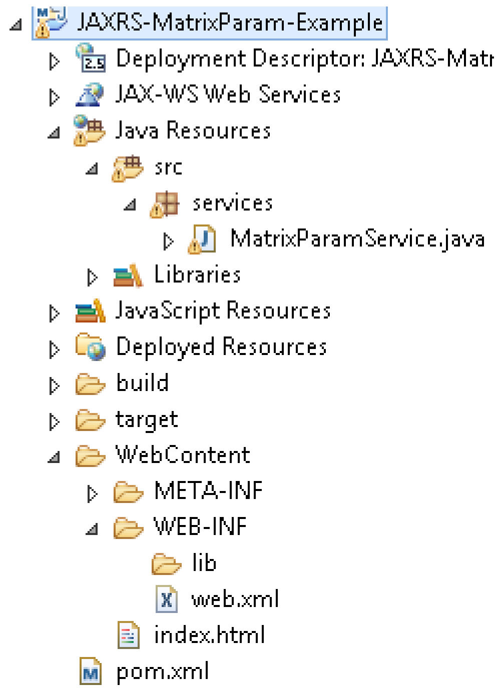
5: JAX-RS @MatrixParam annotation Example

http://localhost:8001/…/rest/customers;custNo=100;custName=Satya

Matrix parameters are another way defining the parameters to be added to URL. If you observe the above syntax, client is passing two parameters each are separated by **semicolon(;),** these parameters are called as matrix parameters. **Remember these parameters may appear any where in the path**

Steps to Implement this Web Service Application

1. Create Dynamic web project in eclipse, convert that into Maven Project



```
JAXRS-MatrixParam-Example
    Deployment Descriptor: JAXRS-Matr
    JAX-WS Web Services
    Java Resources
        src
            services
                MatrixParamService.java
        Libraries
    JavaScript Resources
    Deployed Resources
    build
    target
    WebContent
        META-INF
        WEB-INF
            lib
            web.xml
        index.html
    pom.xml
```

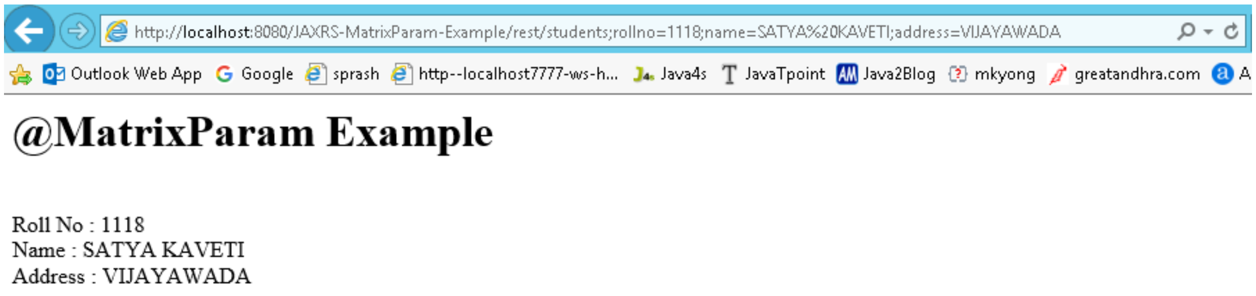
2. Add RESTEasy jar files manually / through Maven by writing repo details in **pom.xml(Skipping)**

3. Create RESTFul webservice using Jersey

```
package services;
import javax.ws.rs.*;
import javax.ws.rs.core.Response;

@Path("/students")
public class MatrixParamService{
        @GET
        @Produces("text/html")
        public Response getResultByPassingValue(
                        @MatrixParam("rollno") String rollno,
                        @MatrixParam("name") String name,
                        @MatrixParam("address") String address) {

                String output = "<h1>@MatrixParam Example</h1>";
                output = output+"<br>Roll No : "+rollno;
                output = output+"<br>Name : "+name;
                output = output+"<br>Address : "+address;
                return Response.status(200).entity(output).build();
        }
}
```

4. Configure `web.xml, Register` RESTEasy dependency class **(Skipping)**

5. Test Webservice directly by using URL / writing webservice client
http://localhost:8080/JAXRS-MatrixParam-Example/rest/students;rollno=1118;name=SATYA%20KAVETI;address=VIJAYAWADA


http://localhost:8080/JAXRS-MatrixParam-Example/rest/students;rollno=1118;name=SATYA%20KAVETI;address=VIJAYAWADA
Outlook Web App
Google
sprash
http--localhost7777-ws-h...
Java4s
JavaTpoint
Java2Blog
mkyong
greatandhra.com
A

@MatrixParam Example

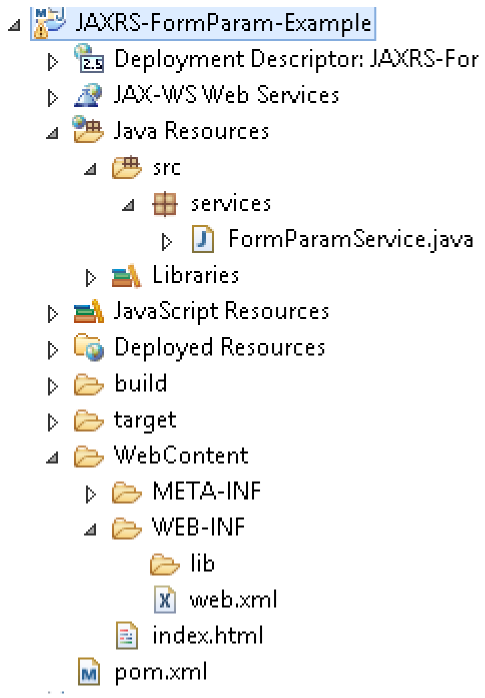
```
Roll No : 1118
Name : SATYA KAVETI
Address : VIJAYAWADA
```

6: JAX-RS @FormParam annotation Example

If we have a HTML form having two input fields and submit button. Lets client enter those details and submit to the RESTful web service. Then the rest service will extract those details by using this **@FormParam** annotation.we can use `@FormParam` annotation to bind HTML form parameters value to a Java method

Steps to Implement this Web Service Application

1. Create Dynamic web project in eclipse, convert that into Maven Project



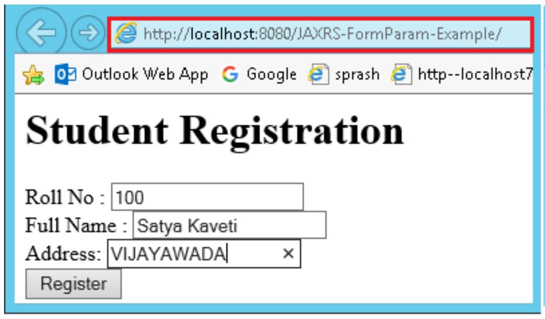
2. Configure **pom.xml, web.xml (Skipping)**

3. Create RESTFul webservice Jersey

```
package services;
import javax.ws.rs.FormParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Response;

@Path("/students")
public class FormParamService {

        @POST
        @Path("/registerStudent")
        @Produces("text/html")
        public Response getResultByPassingValue(
                        @FormParam("rollno") String rollno,
                        @FormParam("name") String name,
                        @FormParam("address") String address) {

                String output = "<h1>@FormParam Example - REGISTRATION COMPLETED!!!</h1>";
                output = output+"<br>Roll No : "+rollno;
                output = output+"<br>Name : "+name;
                output = output+"<br>Address : "+address;
                return Response.status(200).entity(output).build();
        }
}
```

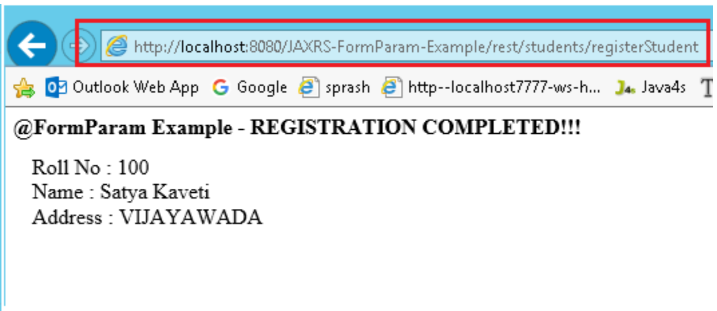
5. Test Webservice directly by using URL / writing webservice client

i. http://localhost:8080/JAXRS-FormParam-Example/
ii. http://localhost:8080/JAXRS-FormParam-Example/rest/students/registerStudent


http://localhost:8080/JAXRS-FormParam-Example/

Student Registration

Roll No : 100
Full Name : Satya Kaveti
Address: VIJAYAWADA

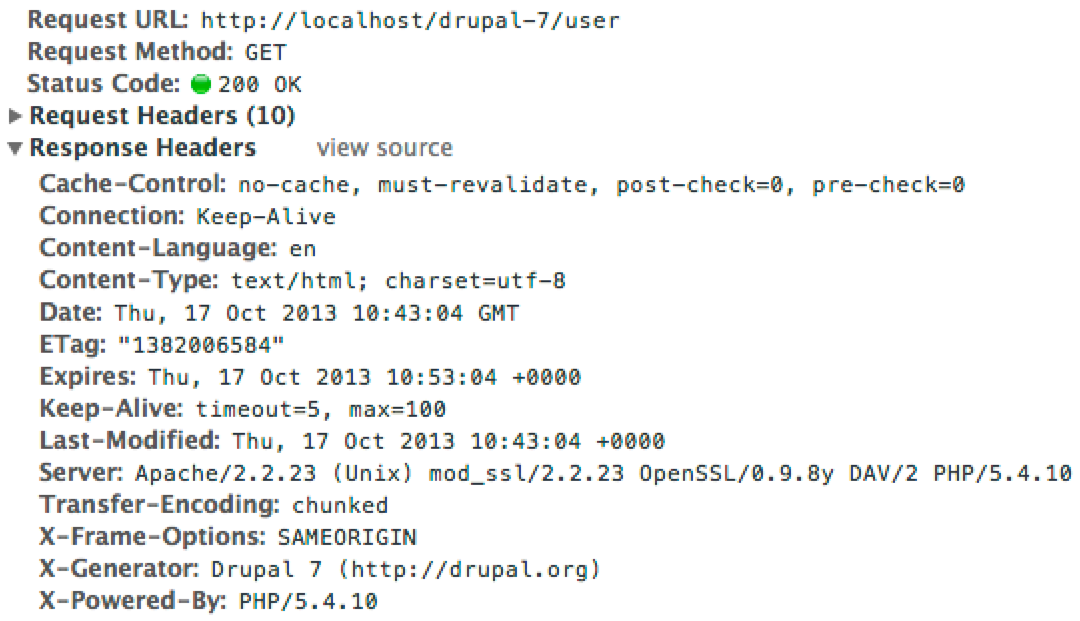

http://localhost:8080/JAXRS-FormParam-Example/rest/students/registerStudent

@FormParam Example - REGISTRATION COMPLETED!!!

Roll No : 100
Name : Satya Kaveti
Address : VIJAYAWADA

7: JAX-RS @HeaderParam Example

We can get the HTTP header details using JAXRS. HTTP headers like below formate



```
Request URL: http://localhost/drupal-7/user
Request Method: GET
Status Code: ● 200 OK
▶ Request Headers (10)
▼ Response Headers        view source
  Cache-Control: no-cache, must-revalidate, post-check=0, pre-check=0
  Connection: Keep-Alive
  Content-Language: en
  Content-Type: text/html; charset=utf-8
  Date: Thu, 17 Oct 2013 10:43:04 GMT
  ETag: "1382006584"
  Expires: Thu, 17 Oct 2013 10:53:04 +0000
  Keep-Alive: timeout=5, max=100
  Last-Modified: Thu, 17 Oct 2013 10:43:04 +0000
  Server: Apache/2.2.23 (Unix) mod_ssl/2.2.23 OpenSSL/0.9.8y DAV/2 PHP/5.4.10
  Transfer-Encoding: chunked
  X-Frame-Options: SAMEORIGIN
  X-Generator: Drupal 7 (http://drupal.org)
  X-Powered-By: PHP/5.4.10
```

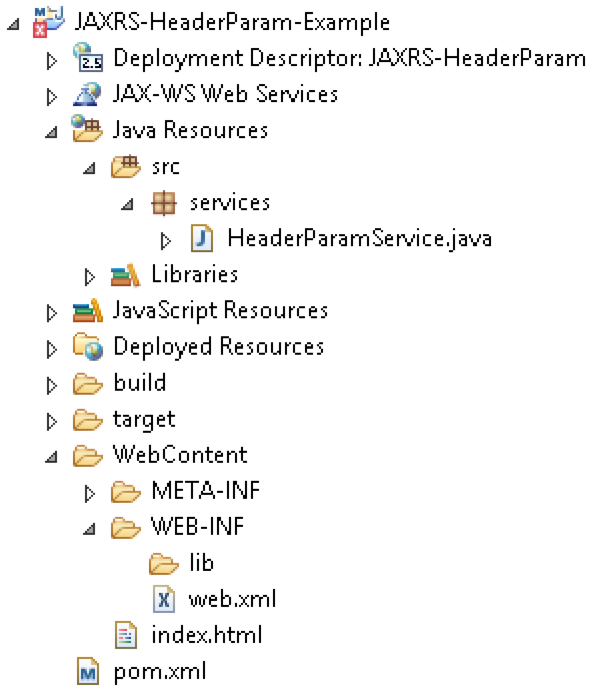
We have two ways to get HTTP request header in JAX-RS :

1. Inject directly with `@HeaderParam`

2. Pragmatically via `@Context`

JAX-RS @HeaderParam Example

1. Create Dynamic web project in eclipse, convert that into Maven Project



```
◢ JAXRS-HeaderParam-Example
  ▷ Deployment Descriptor: JAXRS-HeaderParam
  ▷ JAX-WS Web Services
  ◢ Java Resources
    ◢ src
      ◢ services
        ▷ HeaderParamService.java
    ▷ Libraries
  ▷ JavaScript Resources
  ▷ Deployed Resources
  ▷ build
  ▷ target
  ◢ WebContent
    ▷ META-INF
    ◢ WEB-INF
        lib
        web.xml
      index.html
    pom.xml
```

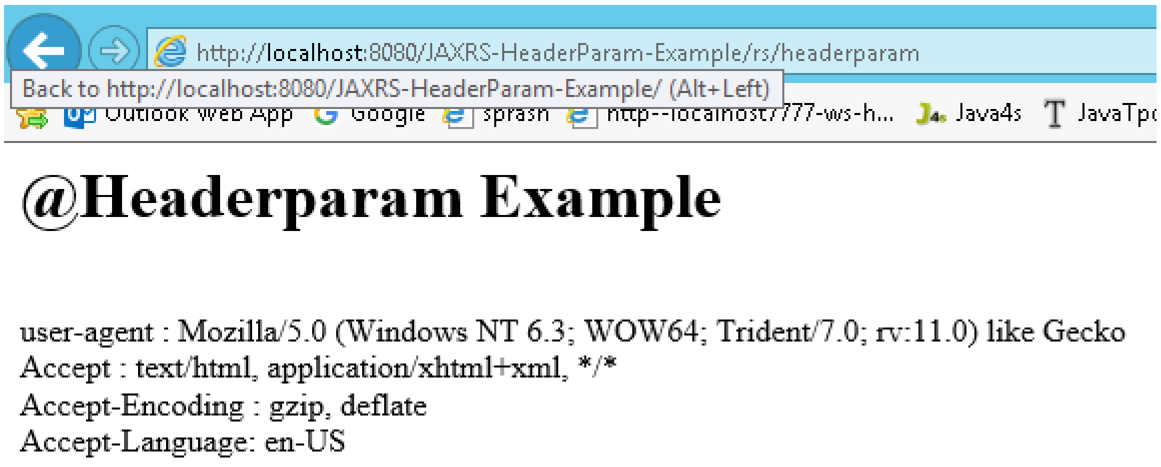
2. Configure **pom.xml, web.xml (Skipping)**

3. Create RESTFul webservice

```
package services;

import javax.ws.rs.GET;
import javax.ws.rs.HeaderParam;
import javax.ws.rs.Path;
import javax.ws.rs.core.Response;

@Path("/rs")
public class HeaderParamService {

        @GET
        @Path("/headerparam")
        public Response getHeader(
                        @HeaderParam("user-agent") String userAgent,
                        @HeaderParam("Accept") String accept,
            @HeaderParam("Accept-Encoding") String encoding,
            @HeaderParam("Accept-Language") String lang) {

                String output = "<h1>@Headerparam Example</h1>";
                output = output+"<br>user-agent : "+userAgent;
                output = output+"<br>Accept : "+accept;
                output = output+"<br>Accept-Encoding : "+encoding;
                output = output+"<br>Accept-Language: "+lang;

                return Response.status(200)
                        .entity(output)
                        .build();

        }
}
```

5. Test Webservice directly by using URL / writing webservice client

http://localhost:8080/JAXRS-HeaderParam-Example/rs/headerparam


http://localhost:8080/JAXRS-HeaderParam-Example/rs/headerparam
Back to http://localhost:8080/JAXRS-HeaderParam-Example/ (Alt+Left)
Outlook Web App
Google
sprash
http--localhost:7777-ws-h...
Java4s
JavaTp

@Headerparam Example

user-agent : Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko
Accept : text/html, application/xhtml+xml, */*
Accept-Encoding : gzip, deflate
Accept-Language: en-US

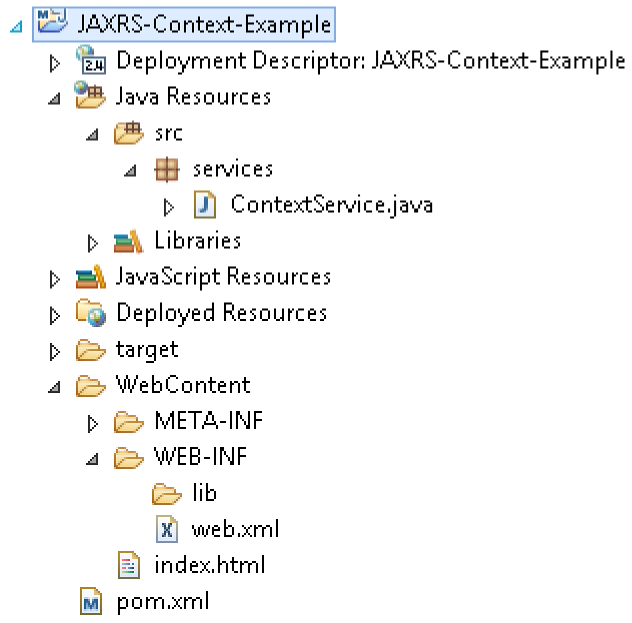
8: JAX-RS @Context Example

JAX-RS provides the **@Context** annotation to inject a variety of resources in your RESTful services. Some of the most commonly injected components are HTTP headers, HTTP URI related information

- ***HTTP headers***
- ***HTTP URI details***
- ***Security Context***
- ***Resource Context***
- ***Request***
- ***Configuration***
- ***Application***
- ***Providers***

JAX-RS @Context Example

1. Create Dynamic web project in eclipse, convert that into Maven Project



```
JAXRS-Context-Example
  Deployment Descriptor: JAXRS-Context-Example
  Java Resources
    src
      services
        ContextService.java
    Libraries
  JavaScript Resources
  Deployed Resources
  target
  WebContent
    META-INF
    WEB-INF
      lib
      web.xml
    index.html
  pom.xml
```

2. Configure **pom.xml, web.xml (Skipping)**

3. Create RESTFul webservice

```
package services;

import javax.ws.rs.GET;

import javax.ws.rs.Path;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.HttpHeaders;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.SecurityContext;
import javax.ws.rs.core.UriInfo;
```

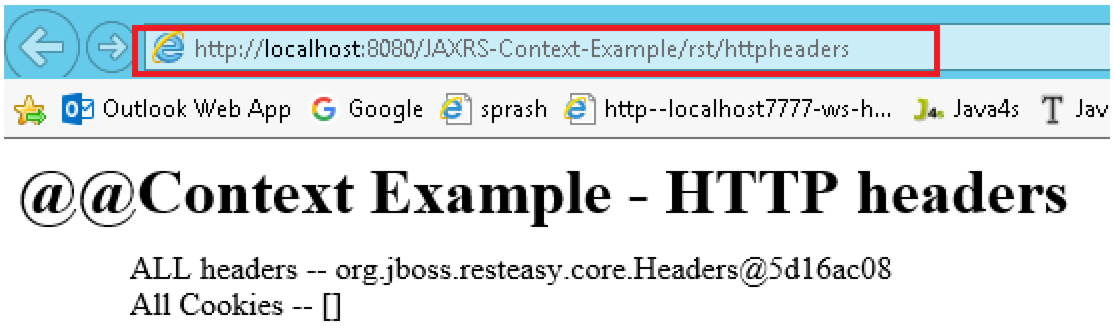
```
@Path("/rst")
public class ContextService {

        @GET
    @Path("httpheaders")
        public Response getHttpheaders(@Context HttpHeaders headers){
                String output = "<h1>@@Context Example - HTTP headers</h1>";
                 output =  output+"<br>ALL headers -- "+ headers.getRequestHeaders().toString();
                 output =  output+"<br>All Cookies -- "+ headers.getCookies().values();
                 return Response.status(200)
                                        .entity(output)
                                        .build();
    }


        @GET
          @Path("uriinfo")
          public Response test(@Context UriInfo uriDetails){
                String output = "<h1>@@Context Example - HTTP URI details</h1>";
        output =  output+"<br>ALL query parameters -- "+ uriDetails.getQueryParameters().toString();
        output =  output+"<br>'id' query parameter -- "+ uriDetails.getQueryParameters().get("id");
        output =  output+"<br>Complete URI -- "+ uriDetails.getRequestUri();
                 return Response.status(200)
                                        .entity(output)
                                        .build();
          }

        @GET
          @Path("securitycontext")
          public Response test(@Context SecurityContext secContext){
                String output = "<h1>@@Context Example - Security Context</h1>";
                //output =  output+"<br>Caller -- "+ secContext.getUserPrincipal().getName();
                output =  output+"<br>Authentication Scheme -- "+ secContext.getAuthenticationScheme();
                output =  output+"<br>Over HTTPS ? -- "+ secContext.isSecure();
                output =  output+"<br>Belongs to 'admin' role? -- "+ secContext.isUserInRole("admin");
              return Response.status(200)
                                        .entity(output)
                                        .build();
          }


}
```

5. Test Webservice directly by using URL / writing webservice client

http://localhost:8080/JAXRS-Context-Example/rst/httpheaders

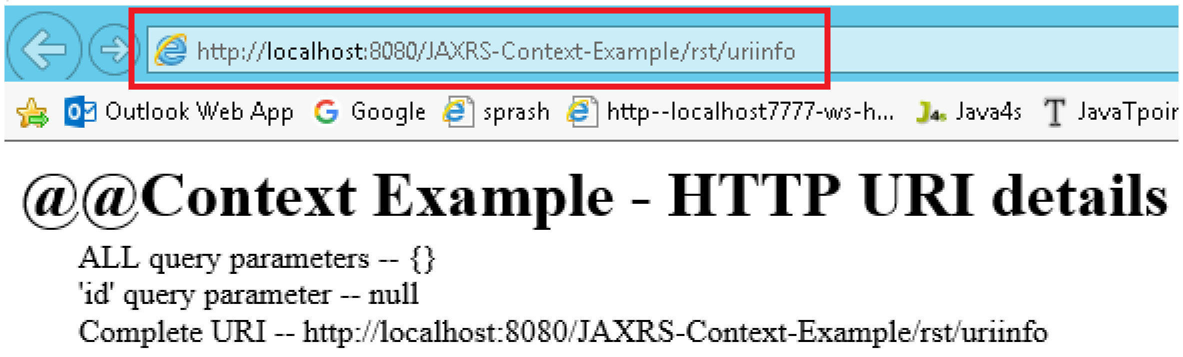
http://localhost:8080/JAXRS-Context-Example/rst/uriinfo

http://localhost:8080/JAXRS-Context-Example/rst/securitycontext


http://localhost:8080/JAXRS-Context-Example/rst/httpheaders
Outlook Web App
Google
sprash
http--localhost7777-ws-h...
Java4s
Jav

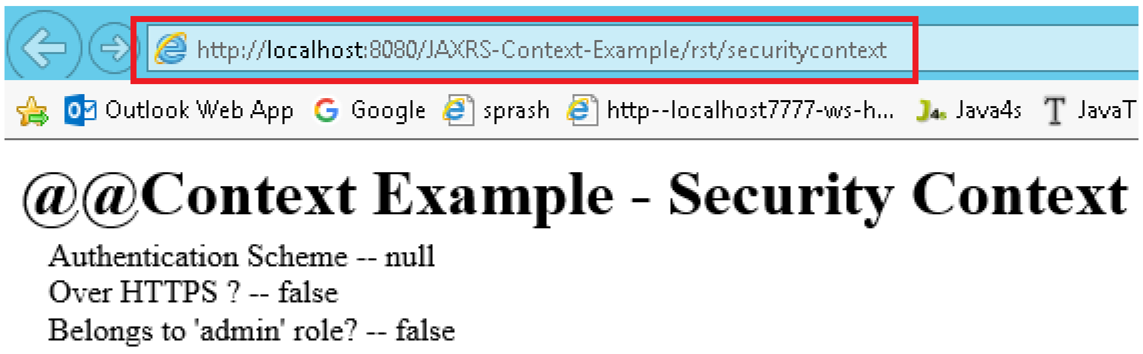
@@Context Example - HTTP headers

ALL headers -- org.jboss.resteasy.core.Headers@5d16ac08
All Cookies -- []


http://localhost:8080/JAXRS-Context-Example/rst/uriinfo
Outlook Web App
Google
sprash
http--localhost7777-ws-h...
Java4s
JavaTpoir

@@Context Example - HTTP URI details

ALL query parameters -- {}
'id' query parameter -- null
Complete URI -- http://localhost:8080/JAXRS-Context-Example/rst/uriinfo


http://localhost:8080/JAXRS-Context-Example/rst/securitycontext
Outlook Web App
Google
sprash
http--localhost7777-ws-h...
Java4s
JavaT

@@Context Example - Security Context

Authentication Scheme -- null
Over HTTPS ? -- false
Belongs to 'admin' role? -- false

9: JAX-RS Download files (text/image/pdf/execel) Example

We can download any type of files from the RESTful web services, **@produces** annotation

We should annotate our method with

- **@Produces("text/plain")** If you are expecting Text file as response
- **@Produces("image/your image type[.jpg/.png/.gif]")** for downloading any Image files
- **@Produces("application/pdf")** for downloading PDF files

Steps to Implement this Web Service Application

1. Create **Dynamic web project in eclipse**, **pom.xml, web.xml (Skipping)**

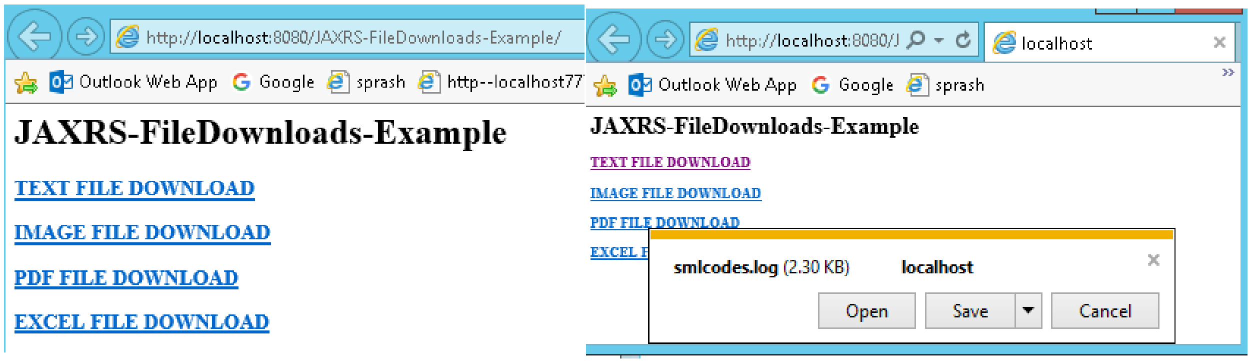
2. Create RESTFul webservice Jersey

```java
package service;
import java.io.File;
import javax.ws.rs*;

@Path("/download")
public class FileDownloadService {
        private static final String TEXT_FILE_PATH = "C:\\Users\\kaveti_s\\textfile.txt";
        private static final String IMG_FILE_PATH = "C:\\Users\\kaveti_s\\img.jpg";
        private static final String PDF_FILE_PATH = "C:\\Users\\kaveti_s\\pdffile.pdf";
        private static final String XLS_FILE_PATH = "C:\\Users\\kaveti_s\\excel.xlsx";

        //TEXTFILE DOWNLOAD
        @GET
        @Path("/textfile")
        @Produces("text/plain")
        public Response downloadTextFile() {
                File file = new File(TEXT_FILE_PATH);
                ResponseBuilder response = Response.ok((Object) file);
                response.header("Content-Disposition",
                        "attachment; filename=\"smlcodes.log\"");
                return response.build();
        }

        //IMAGE DOWNLOAD
        @GET
        @Path("/image")
        @Produces("image/jpg")
        public Response downloadImage() {
                File file = new File(IMG_FILE_PATH);
                ResponseBuilder response = Response.ok((Object) file);
                response.header("Content-Disposition",
                        "attachment; filename=smlcodes.jpg");
                return response.build();
        }

        //PDF DOWNLOAD
        @GET
        @Path("/pdf")
        @Produces("application/pdf")
        public Response downloadPDF() {
                File file = new File(PDF_FILE_PATH);
                ResponseBuilder response = Response.ok((Object) file);
                response.header("Content-Disposition",
                                "attachment; filename=smlcodes.pdf");
                return response.build();
        }


        //XLS DOWNLOAD
        @GET
        @Path("/xls")
        @Produces("application/vnd.ms-excel")
        public Response downloadXLS() {
                File file = new File(XLS_FILE_PATH);
                ResponseBuilder response = Response.ok((Object) file);
                response.header("Content-Disposition",
                        "attachment; filename=new-smlcodes.xls");
                return response.build();

        }
}
```

```html
//index.html
<h1>JAXRS-FileDownloads-Example</h1>

<h3><a href="download/textfile">TEXT FILE DOWNLOAD</a></h3>
 <h3><a href="download/image">IMAGE FILE DOWNLOAD</a></h3>
 <h3><a href="download/pdf">PDF FILE DOWNLOAD</a></h3>
 <h3><a href="download/xls">EXCEL FILE DOWNLOAD</a></h3>
```

5. Test Webservice directly by using URL / writing webservice client

http://localhost:8080/JAXRS-FileDownloads-Example/



JAXRS-FileDownloads-Example

TEXT FILE DOWNLOAD

IMAGE FILE DOWNLOAD

PDF FILE DOWNLOAD

EXCEL FILE DOWNLOAD

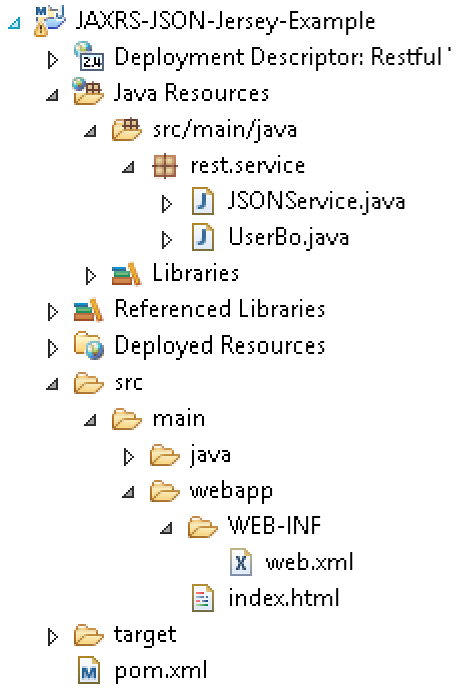
smlcodes.log (2.30 KB) localhost

10. JAX-RS JSON Example Using Jersey

Jersey uses Jackson to convert **object to / form JSON**. In this example, we show you how to convert a "user" object into JSON format, and return it back to user

Steps to Implement this Web Service Application

1. Create Dynamic web project in eclipse, convert that into Maven Project



```
JAXRS-JSON-Jersey-Example
  Deployment Descriptor: Restful '
  Java Resources
    src/main/java
      rest.service
        JSONService.java
        UserBo.java
    Libraries
  Referenced Libraries
  Deployed Resources
  src
    main
      java
      webapp
        WEB-INF
          web.xml
        index.html
  target
  pom.xml
```

2. Configure pom.xml

To make Jersey support JSON mapping, declares “jersey-json.jar” in Maven `pom.xml` file.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>JAX-RS-JSON_Example-Jersey</groupId>
  <artifactId>JAX-RS-JSON_Example-Jersey</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
  <build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.5.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.0.0</version>
        <configuration>
          <warSourceDirectory>WebContent</warSourceDirectory>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <repositories>
                <repository>
                        <id>maven2-repository.java.net</id>
                        <name>Java.net Repository for Maven</name>
                        <url>http://download.java.net/maven/2/</url>
                        <layout>default</layout>
                </repository>
        </repositories>

        <dependencies>


                <dependency>
                        <groupId>com.sun.jersey</groupId>
                        <artifactId>jersey-server</artifactId>
                        <version>1.8</version>
                </dependency>

                <dependency>
                        <groupId>com.sun.jersey</groupId>
                        <artifactId>jersey-json</artifactId>
                        <version>1.8</version>
                </dependency>

        </dependencies>
</project>
```

3. Configure **web.xml**

In `web.xml`, declares “`com.sun.jersey.api.json.POJOMappingFeature`” as “`init-param`” in Jersey mapped servlet. It will make Jersey support JSON/object mapping.

```
<web-app id="WebApp_ID" version="2.4"
        xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
        <display-name>Restful Web Application</display-name>

        <servlet>
                <servlet-name>jersey-serlvet</servlet-name>
                <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
                <init-param>
                        <param-name>com.sun.jersey.config.property.packages</param-name>
                        <param-value>rest.service</param-value>
                </init-param>
                <init-param>
                        <param-name>com.sun.jersey.api.json.POJOMappingFeature</param-name>
                        <param-value>true</param-value>
                </init-param>
                <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>
```

4. Write "UserBo" class

Write "UserBo" class object, Jersey will convert this object into JSON format.

```
package services;

public class UserBo {
        String username;
        String password;

        public String getUsername() {
                return username;
        }

        public void setUsername(String username) {
                this.username = username;
        }

        public String getPassword() {
                return password;
        }

        public void setPassword(String password) {
                this.password = password;
        }

        @Override
        public String toString() {
                // TODO Auto-generated method stub
                return "User [username=" + username + ", password=" + password + "]";
        };
}
```

5. Create RESTFul webservice Jersey

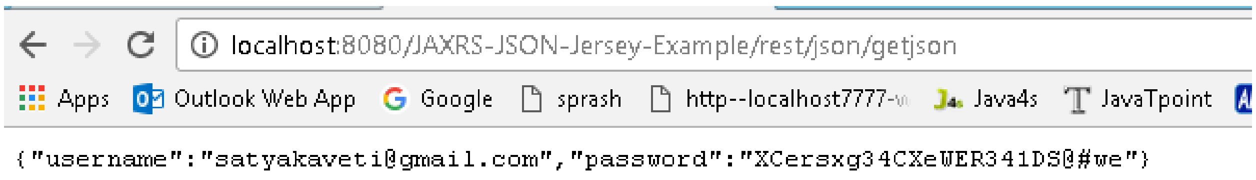
Annotate the method with `@Produces(MediaType.APPLICATION_JSON)`. Jersey will use Jackson to handle the JSON conversion automatically.

```
package rest.service;
```

```
@Path("/json")
public class JSONService {
        @GET
        @Path("/getjson")
        @Produces(MediaType.APPLICATION_JSON)
        public UserBo getboInJSON() {

                UserBo bo = new UserBo();
                bo.setUsername("satyakaveti@gmail.com");
                bo.setPassword("XCersxg34CXeWER341DS@#we");
                return bo;
        }

}
```

6. Test Webservice directly by using URL / writing webservice client

http://localhost:8080/JAX-RS-JSON_Example-Jersey/json/getjson

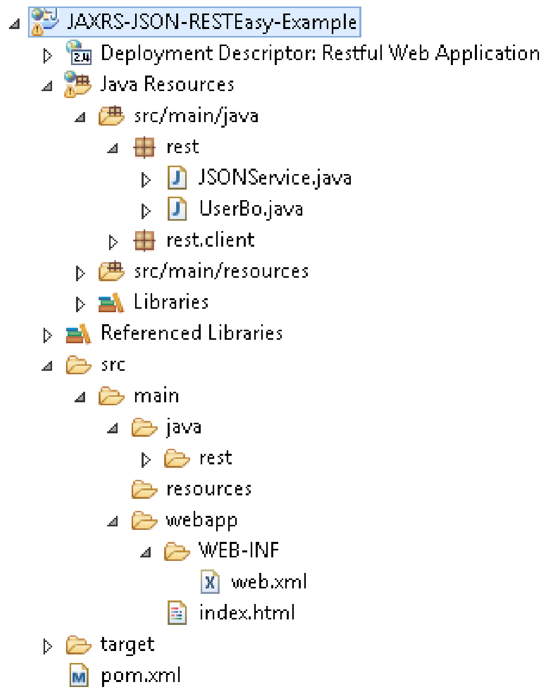


```
{"username":"satyakaveti@gmail.com","password":"XCersxg34CXeWER341DS@#we"}
```

11. JAX-RS JSON Example Using RESTEasy

To integrate Jackson with RESTEasy, you just need to include **"resteasy-jackson-provider.jar".**

1. Create Dynamic web project in eclipse, convert that into Maven Project



```
JAXRS-JSON-RESTEasy-Example
  Deployment Descriptor: Restful Web Application
  Java Resources
    src/main/java
      rest
        JSONService.java
        UserBo.java
      rest.client
    src/main/resources
    Libraries
  Referenced Libraries
  src
    main
      java
        rest
      resources
      webapp
        WEB-INF
          web.xml
        index.html
  target
  pom.xml
```

2. Configure pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>JAXRS-JSON-RESTEasy-Example</groupId>
        <artifactId>JAXRS-JSON-RESTEasy-Example</artifactId>
        <packaging>war</packaging>
        <version>1.0-SNAPSHOT</version>
        <name>JAXRS-JSON-RESTEasy-Example</name>
        <url>http://maven.apache.org</url>

        <repositories>
                <repository>
                        <id>JBoss repository</id>
                        <url>https://repository.jboss.org/nexus/content/groups/public-jboss/</url>
                </repository>
        </repositories>

        <dependencies>
                <dependency>
                        <groupId>junit</groupId>
                        <artifactId>junit</artifactId>
                        <version>4.8.2</version>
                        <scope>test</scope>
                </dependency>

                <dependency>
                        <groupId>org.jboss.resteasy</groupId>
                        <artifactId>resteasy-jaxrs</artifactId>
                        <version>2.2.1.GA</version>
                </dependency>

                <dependency>
                        <groupId>org.jboss.resteasy</groupId>
                        <artifactId>resteasy-jackson-provider</artifactId>
                        <version>2.2.1.GA</version>
                </dependency>

        </dependencies>

        <build>
                <finalName>JAXRS-JSON-RESTEasy-Example</finalName>
                <plugins>
                        <plugin>
                                <artifactId>maven-compiler-plugin</artifactId>
                                <configuration>
                                        <source>1.6</source>
                                        <target>1.6</target>
                                </configuration>
                        </plugin>
                </plugins>
        </build>

</project>
```

3. Configure web.xml

In web.xml Disable RESTEasy auto scanning and register your REST service manually, otherwise, you will get ***Illegal to inject a message body into a singleton into public.JacksonJsonProvider Error***

```
<web-app>
        <display-name>JAXRS-JSON-RESTEasy-Example</display-name>
        <context-param>
                <param-name>resteasy.resources</param-name>
                <param-value>rest.JSONService</param-value>
        </context-param>

        <listener>
```

```
                <listener-class>
                        org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap
                </listener-class>
        </listener>
        <servlet>
                <servlet-name>resteasy-servlet</servlet-name>
                <servlet-class>
                        org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
                </servlet-class>
        </servlet>

        <servlet-mapping>
                <servlet-name>resteasy-servlet</servlet-name>
                <url-pattern>/rest/*</url-pattern>
        </servlet-mapping>

</web-app>
```

4. Write "UserBo" class

Write "UserBo" class object, Jersey will convert this object into JSON format.

```
package rest;

public class UserBo {

        String username;
        String password;

        public String getUsername() {
                return username;
        }

        public void setUsername(String username) {
                this.username = username;
        }

        public String getPassword() {
                return password;
        }

        public void setPassword(String password) {
                this.password = password;
        }

        @Override
        public String toString() {
                return "USER [username=" + username + ", password=" + password + "]";
        }

}
```

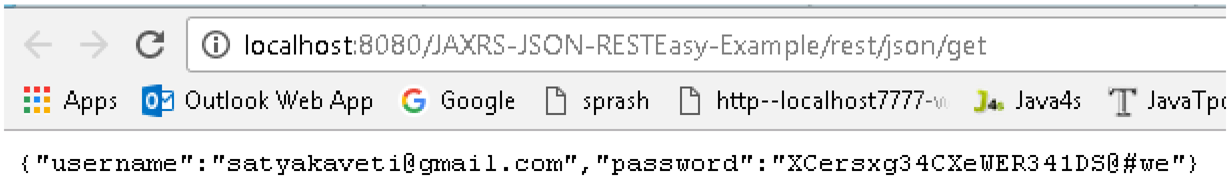
5. Create RESTFul webservice

```
package rest;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Response;


@Path("/rest/json")
```

```
public class JSONService {

        @GET
        @Path("/get")
        @Produces("application/json")
        public UserBo getUserBoInJSON() {

                UserBo bo = new UserBo();
                bo.setUsername("satyakaveti@gmail.com");
                bo.setPassword("XCersxg34CXeWER341DS@#we");

                return bo;

        }

        @POST
        @Path("/post")
        @Consumes("application/json")
        public Response createUserBoInJSON(UserBo UserBo) {

                String result = "UserBo created : " + UserBo;
                return Response.status(201).entity(result).build();

        }

}
```

6. Test Webservice directly by using URL / writing webservice client

http://localhost:8080/JAXRS-JSON-RESTEasy-Example/rest/json/get



```
{"username":"satyakaveti@gmail.com","password":"XCersxg34CXeWER341DS@#we"}
```

12. JAX-RS XML Example Using Jersey

JAX-RS supports conversion of java objects into XML with the help of JAXB. As Jersey it self contains JAXB libraries we no need to worry about JAXB-Jersey integration. Just include **"jersey-server.jar"**

Steps to Implement this Web Service Application

1. Create Dynamic web project in eclipse, convert that into Maven Project

2. Configure **pom.xml**

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>JAXRS-XML-Jersey-Example</groupId>
  <artifactId>JAXRS-XML-Jersey-Example</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>

  <repositories>
                <repository>
                        <id>maven2-repository.java.net</id>
                        <name>Java.net Repository for Maven</name>
                        <url>http://download.java.net/maven/2/</url>
                        <layout>default</layout>
                </repository>
        </repositories>

        <dependencies>
                <dependency>
                        <groupId>junit</groupId>
                        <artifactId>junit</artifactId>
                        <version>4.8.2</version>
                        <scope>test</scope>
                </dependency>

                <dependency>
                        <groupId>com.sun.jersey</groupId>
                        <artifactId>jersey-server</artifactId>
                        <version>1.8</version>
                </dependency>

        </dependencies>

        <build>
                <finalName>JAXRS-XML-Jersey-Example</finalName>
                <plugins>
                        <plugin>
                                <artifactId>maven-compiler-plugin</artifactId>
                                <configuration>
                                        <compilerVersion>1.5</compilerVersion>
                                        <source>1.5</source>
                                        <target>1.5</target>
                                </configuration>
                        </plugin>
                </plugins>
        </build>

</project>
```

3. Configure **web.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  <display-name>RestPathAnnotationExample</display-name>
  <servlet>
    <servlet-name>jersey-serlvet</servlet-name>
    <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
    <init-param>
      <param-name>com.sun.jersey.config.property.packages</param-name>
      <param-value>rest.service</param-value>
    </init-param>
    <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>
```

4. write Customer POJO class

Write **Customer POJO class &** Annotate object with JAXB annotation, for conversion later.

```
package rest.service;

import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
```

```
@XmlRootElement(name = "customer")
public class Customer {

        String custName;
        String custCountry;
        int custId;

        @XmlElement
        public String getCustName() {
                return custName;
        }
        public void setCustName(String custName) {
                this.custName = custName;
        }

        @XmlElement
        public String getCustCountry() {
                return custCountry;
        }
        public void setCustCountry(String custCountry) {
                this.custCountry = custCountry;
        }

        @XmlAttribute
        public int getCustId() {
                return custId;
        }
        public void setCustId(int custId) {
                this.custId = custId;
        }
}
```

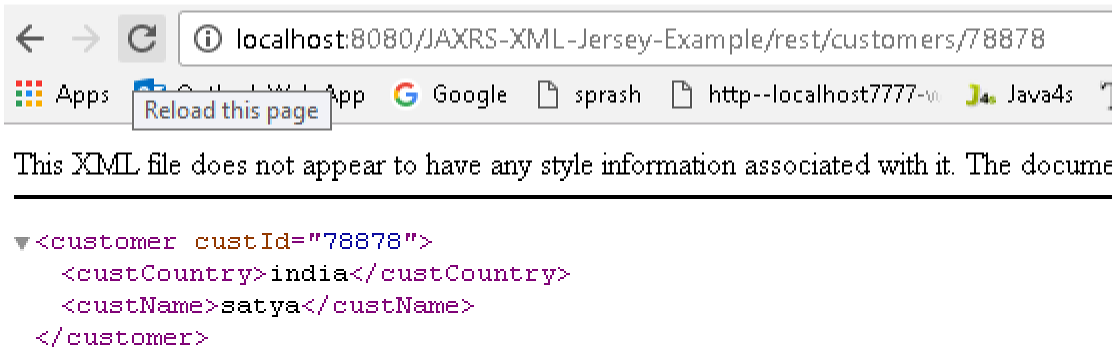
5. Create RESTFul webservice Jersey

To return a XML file, annotate the method with `@Produces(MediaType.APPLICATION_XML)`. Jersey will convert the JAXB annotated object into XML file automatically.

```
package rest.service;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/customers")
public class RestfulXMLExample {

        @GET
        @Path("/{id}")
        @Produces(MediaType.APPLICATION_XML)
        public Customer getCustomerDetails(@PathParam("id") int custId) {

                // WRITE DATABASE LOGIC TO RETRIEVE THE CUSTOMER RECORD WITH 'custID'

                Customer cust = new Customer();
                cust.setCustName("satya");
                cust.setCustCountry("india");
                cust.setCustId(custId);
                return cust;
        }
}
```

6. Test Webservice directly by using URL / writing webservice client

http://localhost:8080/JAXRS-XML-Jersey-Example/rest/customers/78878


localhost:8080/JAXRS-XML-Jersey-Example/rest/customers/78878
Apps
Reload this page
Google
sprash
http--localhost7777-
Java4s

This XML file does not appear to have any style information associated with it. The docume

```
▼<customer custId="78878">
   <custCountry>india</custCountry>
   <custName>satya</custName>
 </customer>
```

13. JAX-RS RESTFul Java Clinets

So far we used URL directly to Test our RESTful service. But in the real time we will call the services by writing some client application logic. We have different ways to write a RESTful client

They are

- **java.net.URL**
- **Apache HttpClient**
- **RESTEasy client**
- **Jersey client**

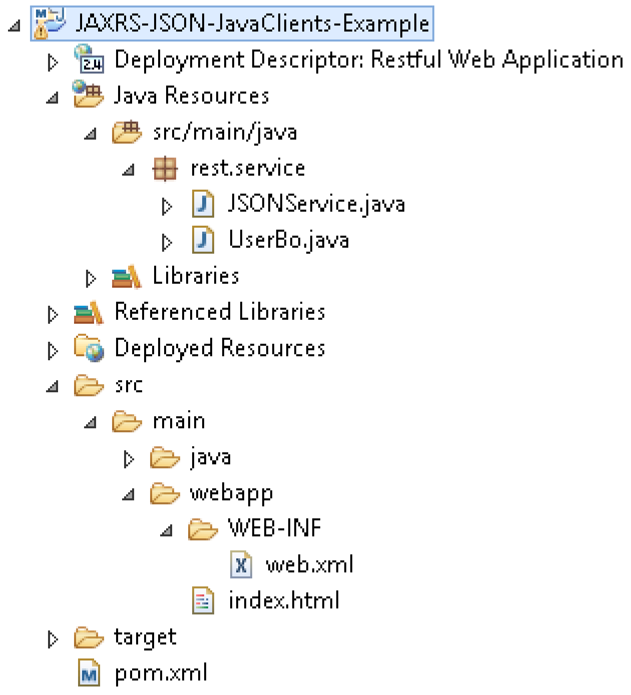
Every Java Clinet can send two types of requests

1. **GET**
2. **POST**

We will see one by one by Example. Here we are Taking JAXRS-JSON-Jersey-Example for writing clinets. For all webservices are same. Only difference in Java Clinets

JAXRS-JSON-Jersey-Example

1. Create Dynamic web project in eclipse, convert that into Maven Project



```
JAXRS-JSON-JavaClients-Example
  Deployment Descriptor: Restful Web Application
  Java Resources
    src/main/java
      rest.service
        JSONService.java
        UserBo.java
    Libraries
  Referenced Libraries
  Deployed Resources
  src
    main
      java
      webapp
        WEB-INF
          web.xml
        index.html
  target
  pom.xml
```

2. Configure **pom.xml, web.xml (May change for Every Java Clinet, please Observe)**

3. Craete UserBo for Converting Java Object to JSON data

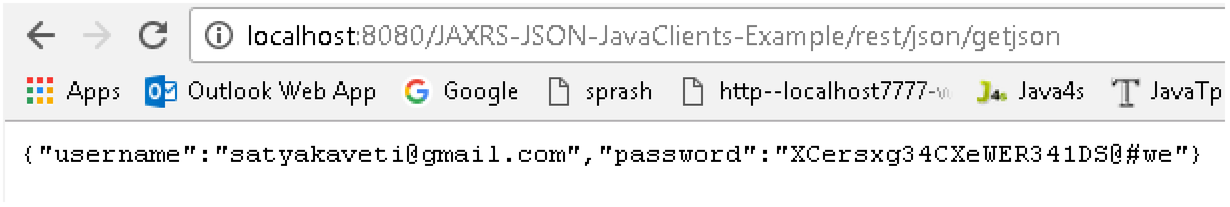
```java
package rest.service;

public class UserBo {

	String username;
	String password;

	public String getUsername() {
		return username;
	}

	public void setUsername(String username) {
		this.username = username;
	}

	public String getPassword() {
		return password;
	}

	public void setPassword(String password) {
		this.password = password;
	}

	@Override
	public String toString() {
		return "USER [username=" + username + ", password=" + password + "]";
	}

}
```

4. Create Web Service having both @GET @POST for testing with Java Clinets

```java
package rest.service;

import javax.ws.rs.core.Response;

@Path("/json")
public class JSONService {
	@GET
	@Path("/getjson")
	@Produces(MediaType.APPLICATION_JSON)
	public UserBo getboInJSON() {

		UserBo bo = new UserBo();
		bo.setUsername("satyakaveti@gmail.com");
		bo.setPassword("XCersxg34CXeWER341DS@#we");
		return bo;
	}

	@POST
	@Path("/postjson")
	@Consumes(MediaType.APPLICATION_JSON)
	public Response createTrackInJSON(UserBo bo) {

		String result = "USER DATA SAVED!! " + bo;
		return Response.status(201).entity(result).build();

	}

}
```

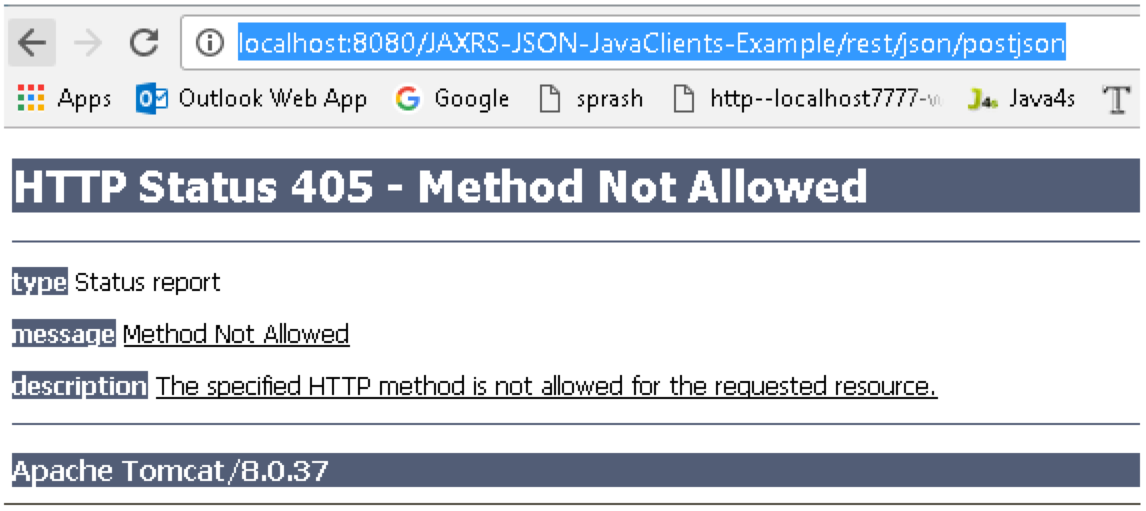
5. Test Webservice directly by using URL / writing webservice client

GET: **http://localhost:8080/JAXRS-JSON-JavaClients-Example/rest/json/getjson**



```
{"username":"satyakaveti@gmail.com","password":"XCersxg34CXeWER341DS@#we"}
```

POST: **http://localhost:8080/JAXRS-JSON-JavaClients-Example/rest/json/postjson**


localhost:8080/JAXRS-JSON-JavaClients-Example/rest/json/postjson
Apps
Outlook Web App
Google
sprash
http--localhost7777-w
Java4s

HTTP Status 405 - Method Not Allowed

type Status report

message Method Not Allowed

description The specified HTTP method is not allowed for the requested resource.

Apache Tomcat/8.0.37

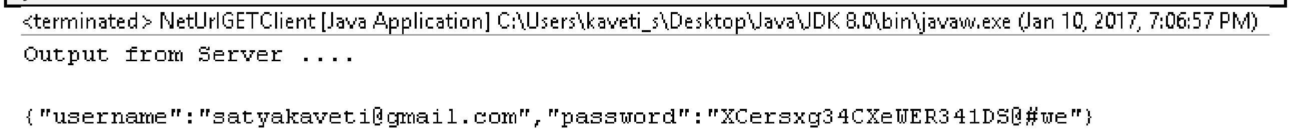
So far we are used above process to Test the Web Services. Now lets see how to test webservices with Java clinets.

1. java.net.URL

Here we will use "`java.net.URL`" and "`java.net.HttpURLConnection`" to create a simple Java client to send **"GET" and "POST"** request.

GET Request Example

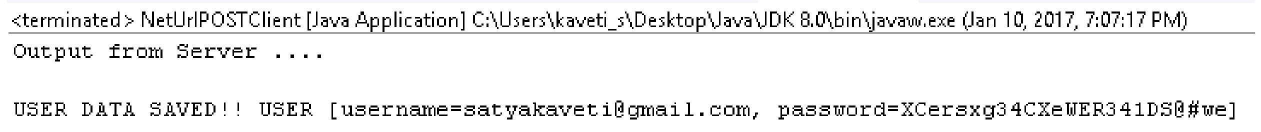
```
package rest.clients;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;

public class NetUrlGETClient {

        public static void main(String[] args) {

                try {

        URL url = new URL("http://localhost:8080/JAXRS-JSON-JavaClients-Example/rest/json/getjson");
                        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
                        conn.setRequestMethod("GET");
                        conn.setRequestProperty("Accept", "application/json");

                        if (conn.getResponseCode() != 200) {
                throw new RuntimeException("Failed : HTTP error code : " + conn.getResponseCode());
                        }

                        BufferedReader br = new BufferedReader(new
InputStreamReader((conn.getInputStream())));

                        String output;
                        System.out.println("Output from Server .... \n");
                        while ((output = br.readLine()) != null) {
                                System.out.println(output);
                        }

                        conn.disconnect();

                } catch (MalformedURLException e) {

                        e.printStackTrace();

                } catch (IOException e) {

                        e.printStackTrace();

                }

        }
}
```



```
<terminated> NetUrlGETClient [Java Application] C:\Users\kaveti_s\Desktop\Java\JDK 8.0\bin\javaw.exe (Jan 10, 2017, 7:06:57 PM)
Output from Server ....

{"username":"satyakaveti@gmail.com","password":"XCersxg34CXeWER341DS@#we"}
```

POST Request Example

```java
package rest.clients;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;

public class NetUrlPOSTClient {
        public static void main(String[] args) {

                try {

        URL url = new URL("http://localhost:8080/JAXRS-JSON-JavaClients-Example/rest/json/postjson");
                        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
                        conn.setDoOutput(true);
                        conn.setRequestMethod("POST");
                        conn.setRequestProperty("Content-Type", "application/json");

String input = "{\"username\":\"satyakaveti@gmail.com\",\"password\":\"XCersxg34CXeWER341DS@#we\"}";

                        OutputStream os = conn.getOutputStream();
                        os.write(input.getBytes());
                        os.flush();

                        if (conn.getResponseCode() != HttpURLConnection.HTTP_CREATED) {
                throw new RuntimeException("Failed : HTTP error code : " + conn.getResponseCode());
                        }

                        BufferedReader br = new BufferedReader(new
InputStreamReader((conn.getInputStream())));

                        String output;
                        System.out.println("Output from Server .... \n");
                        while ((output = br.readLine()) != null) {
                                System.out.println(output);
                        }

                        conn.disconnect();

                } catch (MalformedURLException e) {

                        e.printStackTrace();

                } catch (IOException e) {

                        e.printStackTrace();

                }

        }
}
```



```
<terminated> NetUrlPOSTClient [Java Application] C:\Users\kaveti_s\Desktop\Java\JDK 8.0\bin\javaw.exe (Jan 10, 2017, 7:07:17 PM)
Output from Server ....

USER DATA SAVED!! USER [username=satyakaveti@gmail.com, password=XCersxg34CXeWER341DS@#we]
```

2. Apache HttpClient

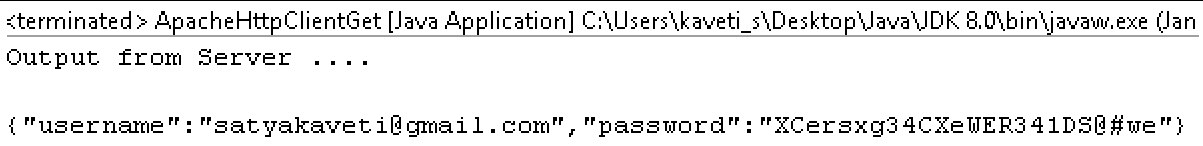
Apache HttpClient is available in Maven central repository, just declares it in your Maven pom.xml file.

1. Configure POM.xml with Apache HTTPClinet

```
<dependency>
        <groupId>org.apache.httpcomponents</groupId>
        <artifactId>httpclient</artifactId>
        <version>4.1.1</version>
</dependency
```

2.GET Request Example

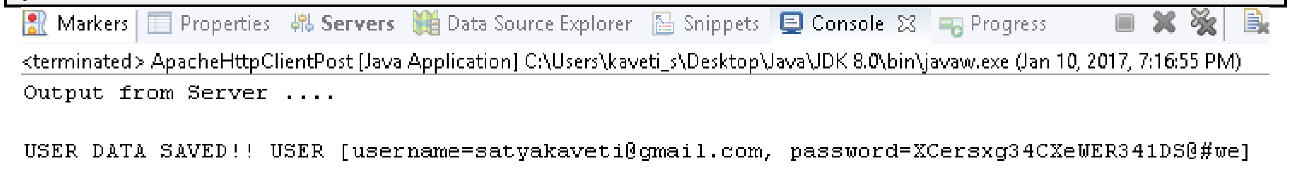
```
package rest.clients;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import org.apache.http.HttpResponse;
import org.apache.http.client.ClientProtocolException;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.DefaultHttpClient;

public class ApacheHttpClientGet {

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

                        DefaultHttpClient httpClient = new DefaultHttpClient();
                        HttpGet getRequest = new HttpGet("http://localhost:8080/JAXRS-JSON-JavaClients-
Example/rest/json/getjson");
                        getRequest.addHeader("accept", "application/json");
                        HttpResponse response = httpClient.execute(getRequest);

                        if (response.getStatusLine().getStatusCode() != 200) {
                                throw new RuntimeException("Failed : HTTP error code : " +
response.getStatusLine().getStatusCode());
                        }

                        BufferedReader br = new BufferedReader(new
InputStreamReader((response.getEntity().getContent())));

                        String output;
                        System.out.println("Output from Server .... \n");
                        while ((output = br.readLine()) != null) {
                                System.out.println(output);
                        }

                        httpClient.getConnectionManager().shutdown();

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



```
<terminated> ApacheHttpClientGet [Java Application] C:\Users\kaveti_s\Desktop\Java\JDK 8.0\bin\javaw.exe (Jan
Output from Server ....

{"username":"satyakaveti@gmail.com","password":"XCersxg34CXeWER341DS@#we"}
```

3.POST Request Example

```
package rest.clients;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.entity.StringEntity;
import org.apache.http.impl.client.DefaultHttpClient;

public class ApacheHttpClientPost {

        public static void main(String[] args) {

                try {

                        DefaultHttpClient httpClient = new DefaultHttpClient();
                        HttpPost postRequest = new HttpPost("http://localhost:8080/JAXRS-JSON-
JavaClients-Example/rest/json/postjson");

                        StringEntity input = new
StringEntity("{\"username\":\"satyakaveti@gmail.com\",\"password\":\"XCersxg34CXeWER341DS@#we\"}");
                        input.setContentType("application/json");
                        postRequest.setEntity(input);

                        HttpResponse response = httpClient.execute(postRequest);

                        if (response.getStatusLine().getStatusCode() != 201) {
                                throw new RuntimeException("Failed : HTTP error code : " +
response.getStatusLine().getStatusCode());
                        }

                        BufferedReader br = new BufferedReader(new
InputStreamReader((response.getEntity().getContent())));

                        String output;
                        System.out.println("Output from Server .... \n");
                        while ((output = br.readLine()) != null) {
                                System.out.println(output);
                        }

                        httpClient.getConnectionManager().shutdown();

                } catch (MalformedURLException e) {

                        e.printStackTrace();

                } catch (IOException e) {

                        e.printStackTrace();

                }

        }

}
```



```
<terminated> ApacheHttpClientPost [Java Application] C:\Users\kaveti_s\Desktop\Java\JDK 8.0\bin\javaw.exe (Jan 10, 2017, 7:16:55 PM)
Output from Server ....

USER DATA SAVED!! USER [username=satyakaveti@gmail.com, password=XCersxg34CXeWER341DS@#we]
```

3.RESTEasy client

1. Configure POM.xml with

RESTEasy client framework is included in RESTEasy core module, so, you just need to declares the **“resteasy-jaxrs.jar”** in your pom.xml file

```
<dependency>
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>resteasy-jaxrs</artifactId>
        <version>2.2.1.GA</version>
</dependency>
```

2.GET Request Example

```
package rest.clients;

import java.io.BufferedReader;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStreamReader;

public class RESTEasyClientGet {

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

                ClientRequest request = new ClientRequest(
                                "http://localhost:8080/JAXRS-JSON-JavaClients-
Example/rest/json/getjson");
                request.accept("application/json");
                ClientResponse<String> response = request.get(String.class);

                if (response.getStatus() != 200) {
                        throw new RuntimeException("Failed : HTTP error code : "
                                + response.getStatus());
                }

                BufferedReader br = new BufferedReader(new InputStreamReader(
                        new ByteArrayInputStream(response.getEntity().getBytes())));

                String output;
                System.out.println("Output from Server .... \n");
                while ((output = br.readLine()) != null) {
                        System.out.println(output);
                }

          } catch (ClientProtocolException e) {

                e.printStackTrace();

          } catch (IOException e) {

                e.printStackTrace();

          } catch (Exception e) {

                e.printStackTrace();

          }

        }

}
```

3.POST Request Example

```
package rest.clients;

import java.io.InputStreamReader;
import java.net.MalformedURLException;
import org.jboss.resteasy.client.ClientRequest;
import org.jboss.resteasy.client.ClientResponse;

public class RESTEasyClientPost {

        public static void main(String[] args) {

          try {

                ClientRequest request = new ClientRequest(
                        "http://localhost:8080/JAXRS-JSON-JavaClients-Example/rest/json/postjson");
                request.accept("application/json");

                String input =
"{\"username\":\"satyakaveti@gmail.com\",\"password\":\"XCersxg34CXeWER341DS@#we\"}";
                request.body("application/json", input);

                ClientResponse<String> response = request.post(String.class);

                if (response.getStatus() != 201) {
                        throw new RuntimeException("Failed : HTTP error code : "
                                + response.getStatus());
                }

                BufferedReader br = new BufferedReader(new InputStreamReader(
                        new ByteArrayInputStream(response.getEntity().getBytes())));

                String output;
                System.out.println("Output from Server .... \n");
                while ((output = br.readLine()) != null) {
                        System.out.println(output);
                }

          } catch (MalformedURLException e) {

                e.printStackTrace();

          } catch (IOException e) {

                e.printStackTrace();

          } catch (Exception e) {

                e.printStackTrace();

          }

        }

}
```

4. Jersey client

1. Configure POM.xml with

To use Jersey client APIs, declares **“jersey-client.jar”** in your pom.xml file.

```
<dependency>
        <groupId>com.sun.jersey</groupId>
        <artifactId>jersey-client</artifactId>
        <version>1.8</version>
</dependency>
```

2.GET Request Example

```
package rest.clients;

import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.WebResource;

public class JerseyClientGet {

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

                        Client client = Client.create();

                        WebResource webResource = client
                                        .resource("http://localhost:8080/JAXRS-JSON-JavaClients-
Example/rest/json/getjson");

                        ClientResponse response =
webResource.accept("application/json").get(ClientResponse.class);

                        if (response.getStatus() != 200) {
                                throw new RuntimeException("Failed : HTTP error code : " +
response.getStatus());
                        }

                        String output = response.getEntity(String.class);

                        System.out.println("Output from Server .... \n");
                        System.out.println(output);

                } catch (Exception e) {

                        e.printStackTrace();

                }

        }
}
```

Output

```
Output from Server ....

{"username":"satyakaveti@gmail.com","password":"XCersxg34CXeWER341DS@#we"}
```

3.POST Request Example

```
package rest.clients;

import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.WebResource;

public class JerseyClientPost {

        public static void main(String[] args) {

                try {

                        Client client = Client.create();

                        WebResource webResource = client
                                        .resource("http://localhost:8080/JAXRS-JSON-JavaClients-
Example/rest/json/postjson");

                        String input =
"{\"username\":\"satyakaveti@gmail.com\",\"password\":\"XCersxg34CXeWER341DS@#we\"}";

                        ClientResponse response =
webResource.type("application/json").post(ClientResponse.class, input);

                        if (response.getStatus() != 201) {
                                throw new RuntimeException("Failed : HTTP error code : " +
response.getStatus());
                        }

                        System.out.println("Output from Server .... \n");
                        String output = response.getEntity(String.class);
                        System.out.println(output);

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

Output

```
Output from Server ....

USER DATA SAVED!! USER [username=satyakaveti@gmail.com, password=XCersxg34CXeWER341DS@#we]
```

14.How to Test (JAX-RS) RESTful Web Services

in real time projects we will use different tools to test RESTful web services

1.Browser Addons

- Postman [Chrome Extension]
- REST Client [Chrome Extension]
- Advanced REST Client [Chrome Extension]
- Rest Client [Firefox Add-On]

2.JAX-RS Local System Tools

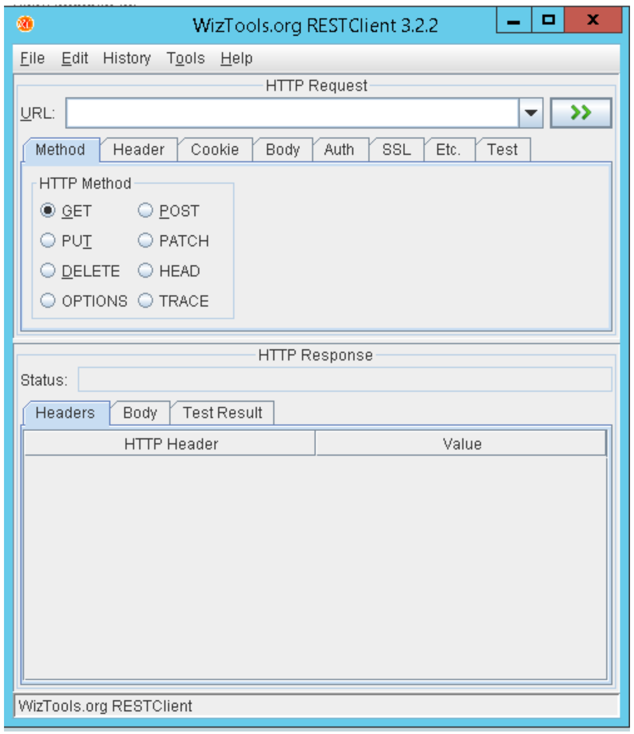
- RESTClient UI
- SoupUi

RESTClient UI

1. Download .jar file from here https://code.google.com/archive/p/rest-client/downloads

2.Run jar by giving >`java -jar restclient-ui-3.2.2-jar-with-dependencies.jar`

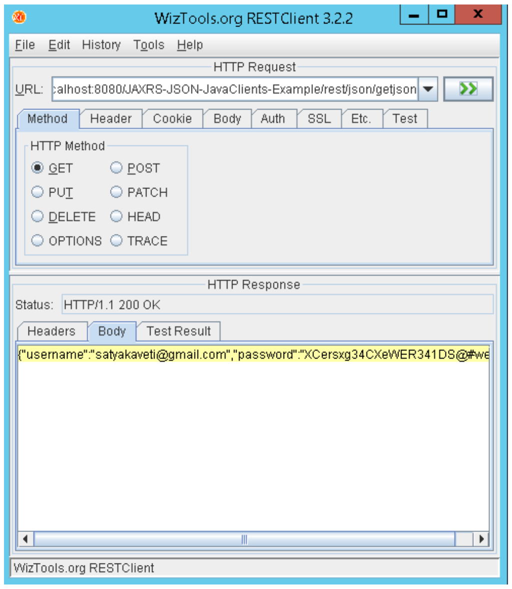
3.It will Opens the window as follows



HTTP Method: [✓ GET]

4.Test Your application by proving any running web service URL

Ex: http://localhost:8080/JAXRS-JSON-JavaClients-Example/rest/json/getjson



URL: :alhost:8080/JAXRS-JSON-JavaClients-Example/rest/json/getjson

HTTP Method: [✓ GET]

Status: HTTP/1.1 200 OK

```
{"username":"satyakaveti@gmail.com","password":"XCersxg34CXeWER341DS@#we
```

Similarly we can work with SoapUI also

References

JAX-WS

http://www.java2blog.com/2013/03/soap-web-service-tutorial.html

https://examples.smlcodes.com/enterprise-java/jws/jax-ws-annotations-example/

http://cxf.apache.org/docs/developing-a-service.html

https://docs.oracle.com/cd/E13222_01/wls/docs92/webserv/annotations.html

http://docs.oracle.com/javaee/5/api/javax/jws/WebService.html

https://jax-ws.java.net/jax-ws-ea3/docs/annotations.html

JAX-RS

http://www.java4s.com/web-services/restful-web-services-jax-rs-annotations/

http://www.mkyong.com/tutorials/jax-rs-tutorials/

http://www.mkyong.com/tutorials/jax-rs-tutorials/

http://www.java4s.com/web-services/