

An Introduction to

WEB SERVICES

- by Venu Kumar.S



Hi Friends,

Just go through the following small story.. (taken from **"You Can Win "** - by Shiv Khera)

Once upon a time a very strong woodcutter asks for a job in a timber merchant, and he got it. The paid was really good and so were the work conditions. For that reason, the woodcutter was determined to do his best. His boss gave him an axe and showed him the area where he was supposed to work. The first day, the woodcutter brought 18 trees "Congratulations," the boss said. "Go on that way!" Very motivated for the boss' words, the woodcutter try harder the next day, but he only could bring 15 trees. The third day he try even harder, but he only could bring 10 trees. Day after day he was bringing less and less trees. The woodcutter thought that "I must be losing my strength". He went to the boss and apologized, saying that he could not understand what was going on.

The boss asked, "When was the last time you sharpened your axe?"

"Sharpen? I had no time to sharpen my axe. I have been very busy trying to cut trees.

If we are just busy in applying for jobs & work, when we will sharpen our skills to chase the job selection process?

My aim is to provide good and quality content to readers to understand easily. All contents are written and practically tested by me before publishing. If you have any query or questions regarding any article feel free to leave a comment or you can get in touch with me on venus.kumaar@gmail.com.

This is just a start, I have achieved till now is just 0.000001n% .

With Warm Regards

Venu Kumaar.S
venus.kumaar@gmail.com

Understanding Traditional client-server approach: In traditional client-server approach the client application interact with business object of Server application directly.

If there is any change in the location of the Server application then we must modify the code of client application to interact with Server application that is changed to new Location this is called Location Dependency.

Ex: Socket programming based client-server applications are two-tier applications or traditional client-server applications.

Drawbacks of Traditional client-server Approach:

1. These applications are location dependent applications.
2. Keeping multiple components in one Server then application increases burden on the Server Application.

Distributed Technology:

- 1. The client-server application with Location Transparency utilizing Registry software as mediator that client-server application is called as Distributed application.
- 2. In Distributed Applications multiple Server applications will be there maintaining multiple business components to distribute burden/work.
- 3. In Distributed applications development more client applications will be there, single registry will be there and one or more Server applications will be there. Registry/JNDI register provides global visibility to object/object reference.
- 4. In one registry we can maintain one or more object/object references nick names or Alias names.
- 5. With respect to diagram multiple Server applications will be developed in multiple machine having business components.
- 6. These multiple business objects business objects reference will be bound with single Registry having nick names.
- 7. Client application gathers one or more business object reference from Registry through Lookup operation.

*Note:

- 1. In Distributed application Registry software must be there to find location but client-server applications may change their locations.
- 2. Distributed application's client talk with Server application only after taking with business object reference from Registry. So any change in location of Server application, we just need to inform to Registry software there is no need of informing to Client Application and modifying their code. This is called Location Transparency.

Advantages of Distributed Applications:

- 1. Distributed applications are Location Transparency applications.
- 2. Distributed Application allows distribute/share the burden in multiple Server applications.
- 3. Suitable for Large scale/complex applications which contains huge amount of Clients.

List of Distributed Technologies to develop Distributed Applications:

1. **RPC** → Open Community
2. **RMI** → SUN Micro Systems(Oracle)
3. **DCOM** → Microsoft
4. **EJB** → from SUN Microsystems
5. **CORBA** → from OML
6. **Remoting** → Microsoft
7. **Http Invoker** → from Interface
8. **Web services** → from Open Community

The plan of manufacturing computer is called as **Architecture of computers**.

There are 3 popular architectures to manufacture the computers

1. IBM Architecture
2. Apple Architecture
3. Sun Architecture

→ C, C++ is platform dependent and Architecture dependent because the .exe file contains platform(operating system) and Architecture specific instructions.

→ Java is platform independent and Architecture independent because .class file does not contain operating system and computer Architecture related instructions.

→ All our regular computers are IBM model computers.

RPC (Remote procedure Calls):

1. Applications must be developed in C, C++ technologies.
2. Platform dependent (operating system dependent).
3. Architecture dependent.
4. Language dependent. (Both client and server applications must be developed same language (C or C++))
5. Out dated technology.

RMI (Remote Method Invocation):

- From Sun ms (Java based)
- Platform independent.
- Architecture independent.
- Language dependent (Both client and server applications must be developed in java)
- Not suitable for large scale applications.
- Can't use internet network as communication channel.
- Does not provide built-in middle ware services.

DCOM (Distributed Component Object Model):

- From Microsoft (Microsoft Technology based)
- Platform dependent

- Architecture dependent
- Language independent
- Suitable for working with Microsoft dependent technologies.

CORBA (Common Object Request Broker Architecture)

- From OMC (Object management group) 800+ companies except Microsoft.
- CORBA is specification having rules and guidelines to develop distributed technologies.
- CORBA based technologies are IDL(Interface Definition Language) from SUN Micro Systems.
- According to CORBA specification
 - i. Applications are Platform, Architecture, Language independent
 - ii. The Allowed languages are C, C++, Java, Objective, C, etc.,
- CORBA is Strong in specification wise but filed in implementation

EJB:

- From SUN Micro systems (Java based)
- Enhancement of RMI having maximum features that CORBA
- Platform and architecture independent
- Language dependent (both client and server applications are should be developed in Java).
- Allows to use internet network as communication channels
- Allows to use the Application Server supplied middleware services
- EJB is a specification having rules and guidelines to develop EJB contains software.
- EJB is industry standard to develop business components in Java based applications.

Web services:

- Form open community
- It is web –based distributed technology(the Server application of Distributed application must be a web application)
- Language independent
- Platform independent and Architecture independent
- Allows to convert other technologies (any) based application to web service application.
- Uses http and SOAP application protocol(Simple Object Application protocol)
- In any technology base, distributed applications development the following resources are occurred.
 - i. Service Provider Application:** It is Server Application (one or more) having the business logic component and business objects.
In web service environment it can be Java/.NET application.
 - ii. Service Client:** The Client Application that calls business method of business components belonging to Server Application.
 - iii.** In Web Service environment Application can be developed in Java or .NET.

Service or Interface: The common understanding difference between Service provider and Service Client applications containing the declaration of methods.

Note: In web service environment this is WSDL ([Web Services Description Language](#))

Registry: The mediator between Service client and Service Provider having business component details in web service environment registry is UDDI.

If we want to pass data between to incompatible applications then pass that data in the form of XML Files.

If we want to see the interaction between to incompatible applications then they must be developed as “**web-services enabled applications**”.

In Web service environment

If we want to see the interaction between two incompatible applications then they must be developed as web service enabled application.

If we want to pass the data to incompatible applications then pass that data with the form of XML files.

Web Services:

- Supports Enterprise services
- Interoperable(Language independent & Platform independent)
- Web services is interoperable distributed technology

Web services is an XML, SOAP, HTTP based distributed technology that allows us to develop interoperable distributed applications by using technology or programmer choice.

- i. All software companies are interoperable works everywhere without working about compatibility.
- ii. To develop interoperable software components use web services that is we can use .NET components in Java and Vice-Versa

Web Services Advantages: Web Services offer many benefits over other types of distributed computing architectures.

Interoperability → This is the most important benefit of Web Services. Web Services typically work outside of private networks, offering developers a non-proprietary route to their solutions. Services developed are likely, therefore, to have a longer life-span, offering better return on investment of the developed service. Web Services also let developers use their preferred programming languages. In addition, thanks to the use of standards-based communications methods, Web Services are virtually platform-independent. **Enables Interoperability of legacy and heterogeneous applications:**

Usability → Web Services allow the business logic of many different systems to be exposed over the Web. This gives your applications the freedom to chose the Web Services that they need. Instead of re-inventing the wheel for each client, you need only include additional application-specific business logic on the client-side. This allows you to develop services and/or client-side code using the languages and

tools that you want.

Reusability → Web Services provide not a component-based model of application development, but the closest thing possible to zero-coding deployment of such services. This makes it easy to reuse Web Service components as appropriate in other services. It also makes it easy to deploy legacy code as a Web Service.

Enables Just-in-time integration → It means dynamic discovery and invocation(publish, find, bind).

Deployability → Web Services are deployed over standard Internet technologies. This makes it possible to deploy Web Services even over the fire wall to servers running on the Internet on the other side of the globe. Also thanks to the use of proven community standards, underlying security (such as SSL) is already built-in.

Two Applications Communication requirements:

- 1) Two applications
- 2) Medium
- 3) Language
- 4) Rules(Protocol)

Web Services architecture:

→ Two applications needs to communicate



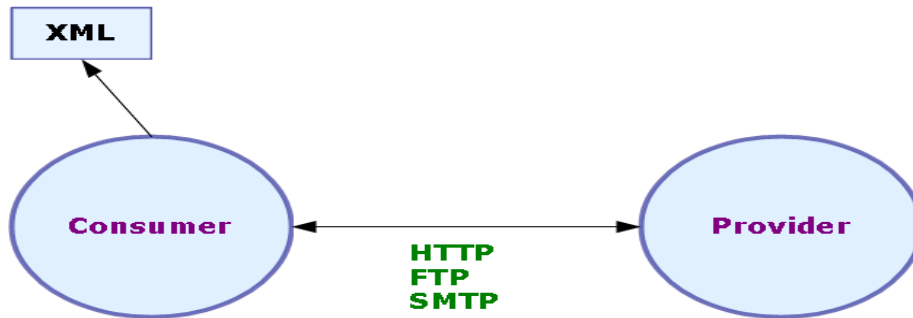
→ Both the applications should be connected(network)

→ Both applications needs to communicate by following some rules(protocols)

- ⌞ HTTP
- ⌞ FTP
- ⌞ SMTP.. etc.,

→ To communicate language is required(Neutral Language)

- ⌞ XML



→ In the internet world we prefer to use HTTP protocol, and we have so many technologies which supports HTTP protocol.

→ Lets assume that we are sending employee information from "Consumer" to "Provider" in the XML format as follows:

```

<employee>
  <id>Guru</id>
  <password>*****</password>
  <id>99999</id>
  <name>kumar</name>
  <salary> 52369</salary>
</employee>
  
```

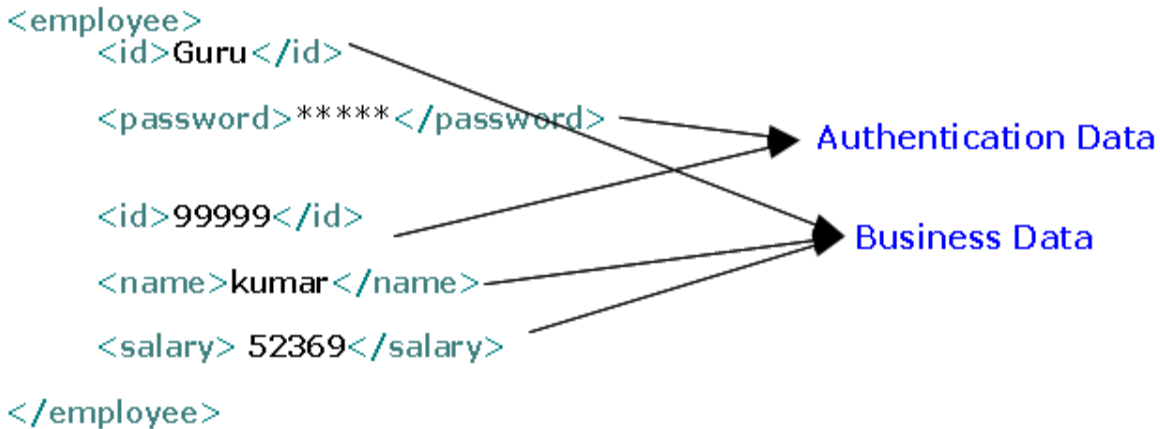
```

<employee>
  <id>Guru</id>
  <password>*****</password>
  <id>99999</id>
  <name>kumar</name>
  <salary> 52369</salary>
</employee>
  
```

Authentication Data

Business Data

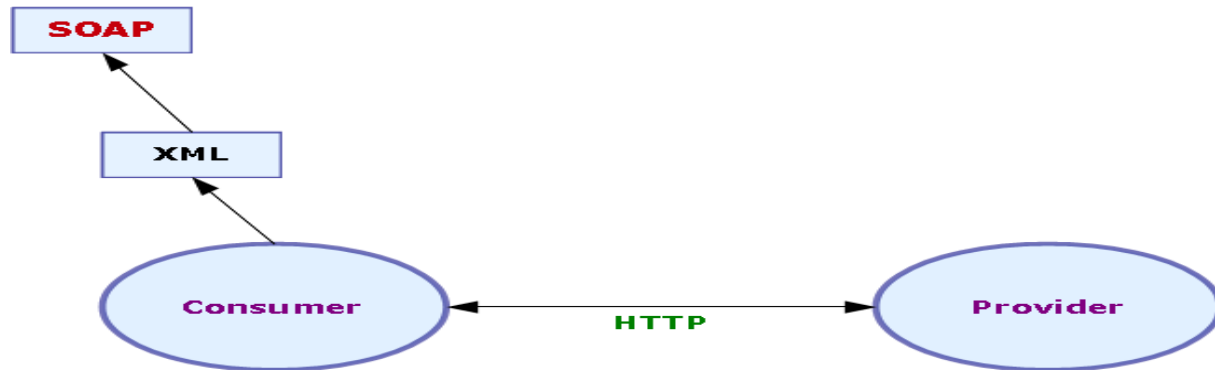
→ At "**Provider**" side it may be interpreted as follows:



- From this classification of the information is missing.
- We don't send raw XML doc (contains business data) to the providers.
- We need classification of the request data like
 - ⌆ Processing information/Authentication information
 - ⌆ Business information
 - ⌆
- To achieve this we will go for **SOAP**.

SOAP: Simple Object Access Protocol

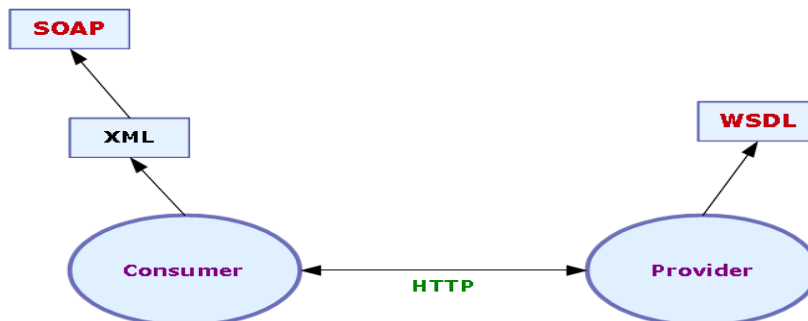
- It is a classification protocol
- It is a binding protocol (Processing data+business data)
- SOAP is an XML-based protocol for exchanging information between computers.
- SOAP is a communication protocol
- SOAP is for communication between applications
- SOAP is a format for sending messages
- SOAP is designed to communicate via Internet
- SOAP is platform independent
- SOAP is language independent
- SOAP is simple and extensible
- SOAP allows you to get around firewalls
- SOAP will be developed as a W3C standard



- But what services are providing by the **Provider** should be explained. To know the services of "**Provider**", "**Consumer**" cannot look into the code of "**Provider**".
- So "**Consumer**" needs the information like what are the services are providing, what they will take as input, and what the output they will return, what **url** has to call to get service., etc.
- "**Provider**" will explain all these information in one doc called "**WSDL**".
- "**WSDL**" is Web Service Description Language".

WSDL:Web Services Description Language WSDL an XML doc, because it should be language independent, so that any type of client can understand the services of the "**Provider**". WSDL is an XML based protocol for information exchange in decentralized and distributed environments.

- WSDL is the standard format for describing a web service.
- WSDL definition describes how to access a web service and what operations it will perform.
- WSDL is a language for describing how to interface with XML-based services.
- WSDL is an integral part of UDDI, an XML-based worldwide business registry.
- WSDL is the language that UDDI uses.
- WSDL was developed jointly by Microsoft and IBM.

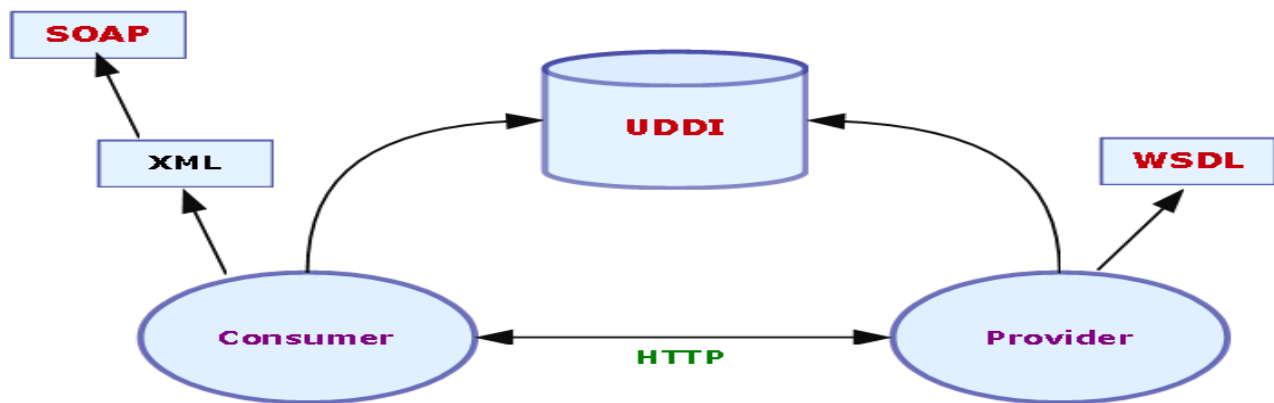


- **Client and server stubs are generated from the WSDL file**

- Using stubs simplifies our applications considerably
- The stubs are generally generated only once
- WSDL is the doc which explains the services information. But how can "**Consumer**" get that WSDL doc, from where "**Consumer**" get?
- So WSDL docs has to be placed in some location from where "**Consumer**" can access them that location is nothing but "**UDDI**".

UDDI: Universal Description, Discovery, and integration.

- It is registry, where all the WSDL docs are registered.
- It is interoperable, developed in XML.
- UDDI is platform independent, open framework.
- UDDI uses WSDL to describe interfaces to web services.
- UDDI is seen with SOAP and WSDL as one of the three foundation standards of web services.
- UDDI is an open industry initiative enabling businesses to discover each other and define how they interact over the Internet.
- UDDI can communicate via SOAP, CORBA, Java RMI Protocol.
- UDDI registry is also called XML registry.
- UDDI is a specification for a distributed registry of Web services.



The Web Services architecture :



Service Processes: This part of the architecture generally involves more than one Web service. For example, discovery belongs in this part of the architecture, since it allows us to locate one particular service from among a collection of Web services.

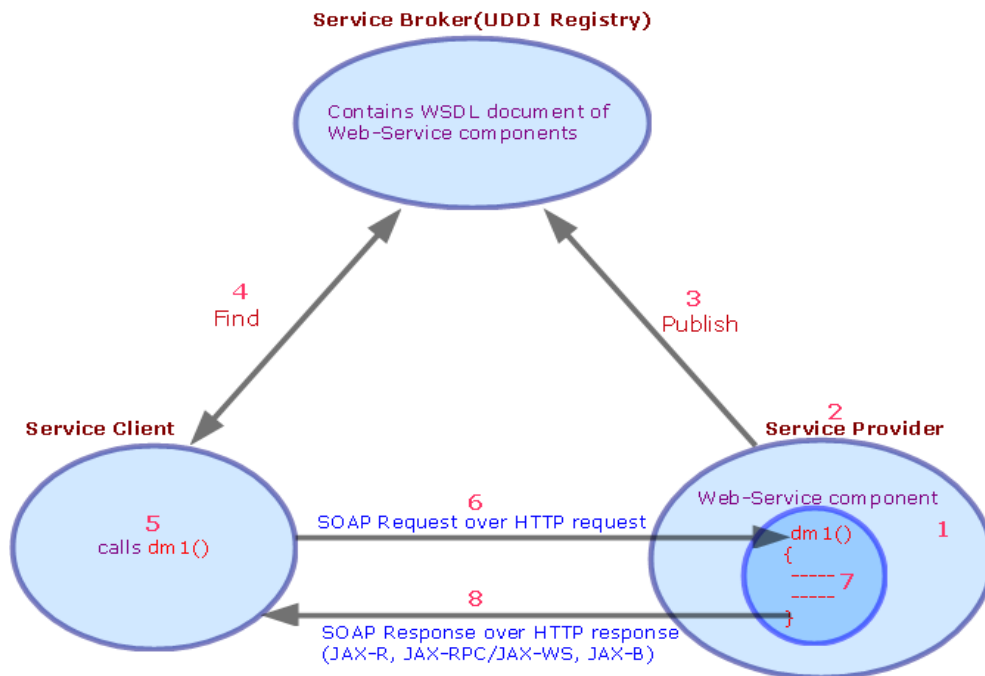
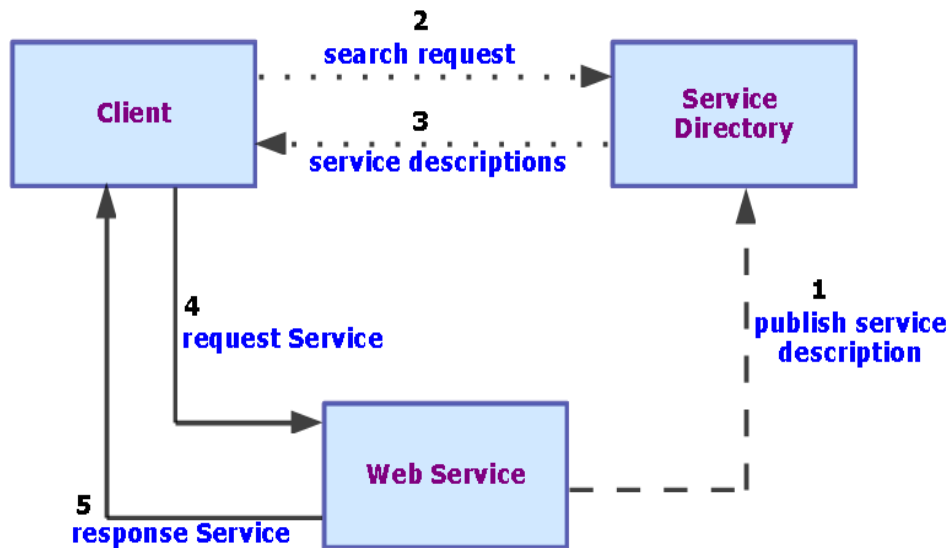
Service Description: One of the most interesting features of Web Services is that they are self-describing. This means that, once you've located a Web Service, you can ask it to 'describe itself' and tell you what operations it supports and how to invoke it. This is handled by the Web Services Description Language (WSDL).

Service Invocation: Invoking a Web Service (and, in general, any kind of distributed service such as a CORBA object or an Enterprise Java Bean) involves passing messages between the client and the server. SOAP (Simple Object Access Protocol) specifies how we should format requests to the server, and how the server should format its responses. In theory, we could use other service invocation languages (such as XML-RPC, or even some ad hoc XML language). However, SOAP is by far the most popular choice for Web Services.

Transport: Finally, all these messages must be transmitted somehow between the server and the client. The protocol of choice for this part of the architecture is HTTP (Hyper Text Transfer Protocol), the same protocol used to access conventional web pages on the Internet. Again, in theory we could be able to use other protocols, but HTTP is currently the most used one.

Overview of Web Service:





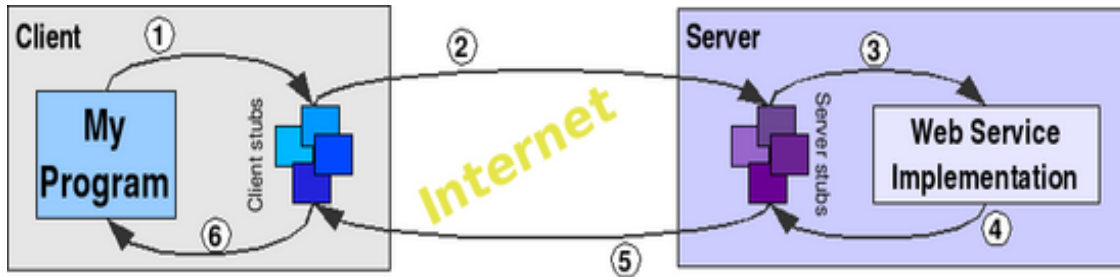
- 1) Service provider develops web service component as web application in his choice technology(Java, .net) having business methods.
- 2) Service provider generates web document for web service component having the details about components and business methods(XML)
- 3) Service provider publishes the web document in UDDI registry for global visibility.
- 4) Service client gathers the WSDL document in UDDI registry for global visibility.
- 5) Service Client understands WSDL document and develops client application in his choice technology.

This client application calls the business methods of web service component.

6) This business method call goes to web service computer as SOAP over HTTP.

7) & 8) Web service client application calls the business methods of the web services component and gathers the result by using SOAP over HTTP protocol.

Typical Web Service invocation :



1) Whenever the client application needs to invoke the Web Service, it will really call the client stub. The client stub will turn this 'local invocation' into a proper SOAP request. This is often called the marshaling or serializing process.

2) The SOAP request is sent over a network using the HTTP protocol. The server receives the SOAP requests and hands it to the server stub. The server stub will convert the SOAP request into something the service implementation can understand (this is usually called unmarshaling or deserializing)

3) Once the SOAP request has been deserialized, the server stub invokes the service implementation, which then carries out the work it has been asked to do.

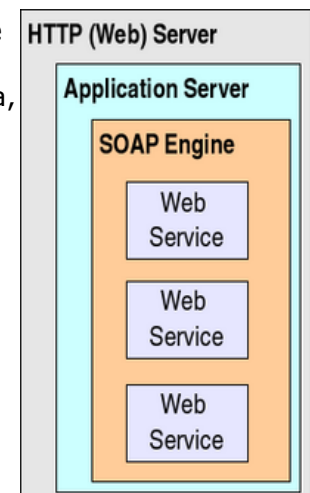
4) The result of the requested operation is handed to the server stub, which will turn it into a SOAP response.

5) The SOAP response is sent over a network using the HTTP protocol. The client stub receives the SOAP response and turns it into something the client application can understand.

6) Finally the application receives the result of the Web Service invocation and uses it.

Web service: First and foremost, we have our Web service. As we have seen, this is basically a piece of software that exposes a set of operations. For example, if we are implementing our Web service in Java, our service will be a Java class (and the operations will be implemented as Java methods). Obviously, we want a set of clients to be able to invoke those operations. However, our Web service implementation knows nothing about how to interpret SOAP requests and how to create SOAP responses. That's why we need a...

SOAP engine: This is a piece of software that knows how to handle SOAP requests and responses. In practice, it is more common to use a



generic SOAP engine than to actually generate server stubs for each individual Web service (note, however, that we still need client stubs for the client). Example of a SOAP engine is Apache Axis. However, the functionality of the SOAP engine is usually limited to manipulating SOAP. To actually function as a server that can receive requests from different clients, the SOAP engine usually runs within an...

Application server: This is a piece of software that provides a 'living space' for applications that must be accessed by different clients. The SOAP engine runs as an application inside the application server. Example is the Tomcat server, a Java Servlet and JSP container that is frequently used with Apache Axis and the Globus Toolkit.

Many application servers already include some HTTP functionality, so we can have Web services up and running by installing a SOAP engine and an application server. However, when an application server lacks HTTP functionality, we also need an...

HTTP Server: This is more commonly called a 'Web server'. It is a piece of software that knows how to handle HTTP messages. Example is the Apache HTTP Server, one of the most popular web servers in the Internet.

WS-I → Web Services Interoperability Organization is an association of IT industry companies, including IBM and Micro Soft, that aim to create Web Services specifications that all companies can use.

Different standards given by WS-I to implement Web Services are:

- ⤴ Basic Profile 1.0(BP 1.0)
- ⤴ Basic Profile 1.1(BP 1.1)
- ⤴ Basic Profile 2.0(BP 2.0)

Web Services:

Web Services is a software application that confirm that **WS-I**'s given specifications (BP1.0/BP1.1/BP2.0) is said to be Web Services.

→ Any program that is distributed in nature, language independent and plat form neutral is said to be web services.

There are so many API's implementations for the WS-I given standards:

- Sun has released "**JAX-RPC**" API that follows BP 1.0 standard.
- Sun has released "**JAX-WS**" API that follows BP 1.1 standard.

To develop Web services components and Client applications we can use the following APIs and Frameworks of Java Environment

- JAX-RPC** API from SUN Micro systems
- JAX-WS** API from SUN Micro systems
- AXIS** Framework Apache (based JAX-WS)
- Metro** Framework from open community (based on JAX-WS)

As of Now there are two methodologies to develop Web services (both Client and Component)

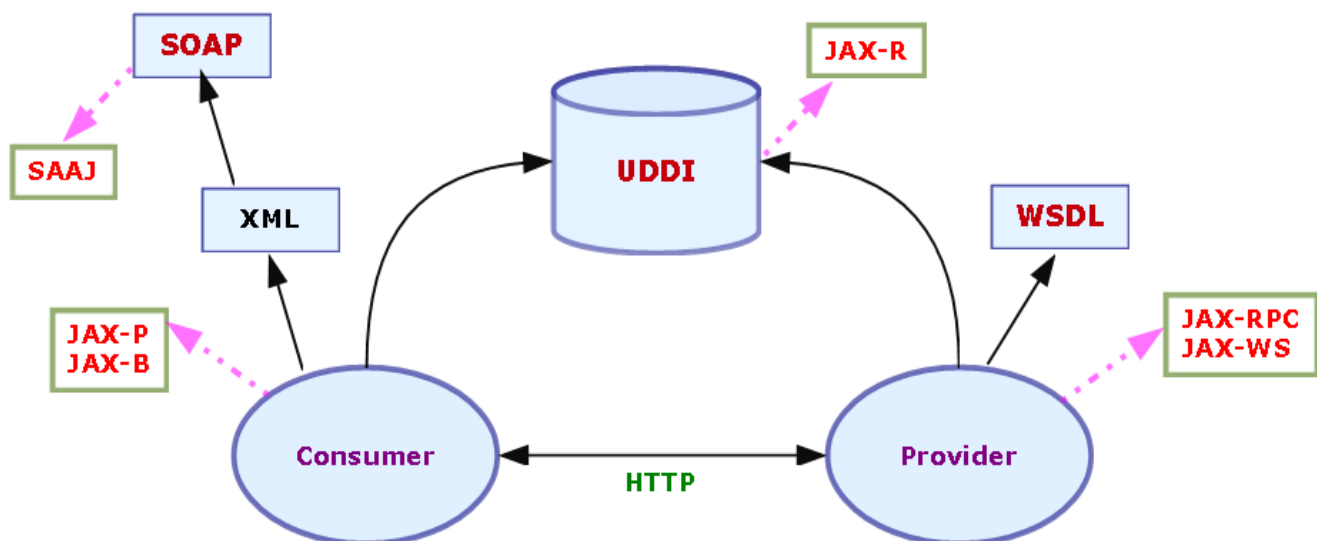
1. **SOAP** based web services

2. REST FULL Web services (REST - Representational state)

- In SOAP based Web services SOAP over HTTP protocol will be used.
- In RESTFULL Web services protocol HTTP will be used directly.
- REST is not a protocol it is methodology to develop web services without using SOAP protocol.

Difference between JAX-RPC & JAX-WEB

JAX-RPC	JAX-WEB
1. Uses SOAP 1.1 version	1. Uses SOAP 1.2 version and also 1.1
2. Supports WSDL 1.1	2. Supports WSDL 1.1, 1.1, 2.X
3. Given based on JDK1.4	3. Given based on JDK1.5
4. Given based on J2EE1.4	4. Given based on Java EE5
5. Does not support Annotations	5. Supports Annotations
6. Supports only Synchronous	6. Supports Synchronous, Asynchronous



Where :

JAX-RPC → Java API for XML-based Remote Procedure Call

JAX-WS → Java API for XML-based Web Services

JAX-R → Java API for XML Registries

JAX-P → Java API for XML Processing

JAX-B → Java Architecture for XML binding

SAAJ → SOAP with Attachments API for Java

To develop all the resources of Web services enabled Applications in JAVA:

- Use **JAXP** API makes service provider and Client interacting with UDDI Registry.
- Use **JAXRPC/JAX-WS** to develop web service component and clients.
- Use **JAXB** to get the interaction between Client and Component.
- JAXR** provides a convenient way to access standard business registries over the Internet.

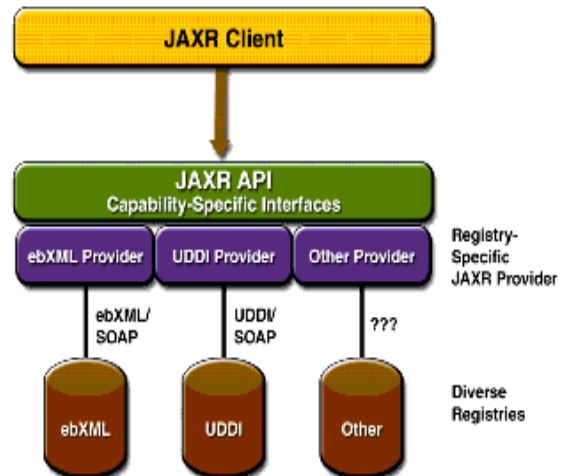
JAXR gives developers writing applications in the Java programming language a uniform way to use business registries that are based on industry consortium-led specifications (such as UDDI).

→ **JAXR client.** → This is a client application that uses the JAXR API to access a business registry through a JAXR provider.

→ **JAXR provider.** → This is a server-side implementation of the JAXR API that provides access to a one or more registry providers that are based on a common specification.

All JAXR applications have the following sequence:

1. Create a ConnectionFactory.
2. Create a Connection object from that factory to the registry.
3. Pass the Connection the appropriate user credentials (e.g., username and password) required by the registry operator.
4. Obtain the reference to the RegistryService from the connection.
5. Do some work with the RegistryService.



e. **SAAJ** → SOAP with Attachments API for Java - contains APIs for creating and populating SOAP messages which might or might not contain attachments. It also contains APIs for sending point to point, non-provider-based, request and response SOAP messages.

The SAAJ API provides the SOAPMessage class to represent a SOAP message, the SOAPPart class to represent the SOAP part, the SOAPEnvelope interface to represent the SOAP envelope, and so on.

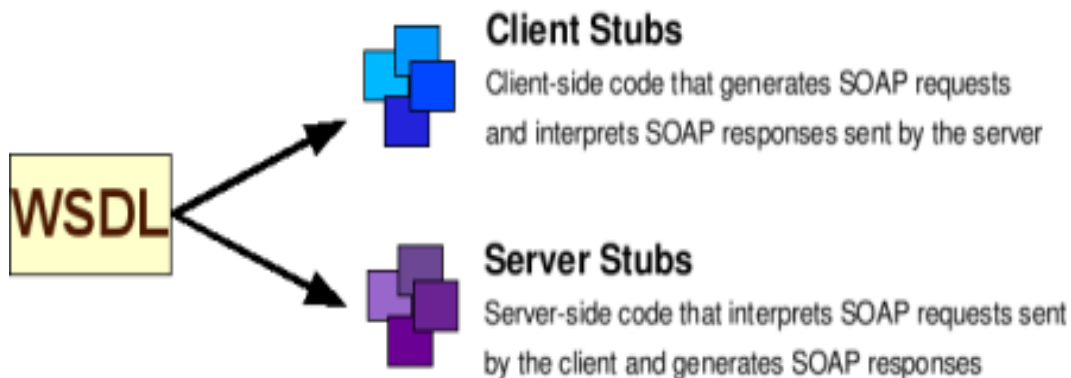
SAAJ is available as part of the Java Web Services Developer Pack

SAAJ → an offshoot of the Java API for XML Messaging (JAXM) -- automates many of the required steps, such as creating connections or creating and sending the actual messages. This tip chronicles the creation and sending of a synchronous SOAP message.

The process involves five steps:

1. Creating a SOAP connection
2. Creating a SOAP message
3. Populating the message
4. Sending the message
5. Retrieving the reply

Note: As of now Java/JEE, .NET are the two major technologies to work with web services.



Specification	API	Implementation
BP 1.0	JAX-RPC	JAX-RPC-SI(from sun), Apache-Axis(from ASF), etc.,
BP 1.1	JAX-WS	JAX-WS(from sun), Apache-Axis2(from ASF),Metro(from sun),CXF(fromASF),etc

Note 1: JAX-WS RI(Reference Implementation) was failed to access by .net, means failed in interoperability. Sun then released "metro" implementation of JAX-WS, which resolved interoperability problem and also added some more features.

Note 2: Apache **CXF** internally uses Spring.

Web Services Development:

The two ways of developing SOAP based Web services components:

1. Bottom-up Approach (Business first Applications): In this Approach first the Web service business components will be developed and they will be used to generate WSDL docs.

Services-WSDL

2. Top-Down Approach (Contract first): In this Approach WSDL document will be developed and that will be used as base document to develop Web service components.

WSDL – Services

- In large scale environment it is recommended to use Top-down Approach where multiple web service components can be developed based on single WSDL documents.
- In small and medium scale projects we can use Bottom-Up Approach where the components will be developed quickly.

Note:

- Services can be developed with any technology like java, .net, php...
- In Java, services can be developed using any API like JAX-RPC, JAX-WS, RESTful,...,etc.

- In Java, Services can be developed using any implementations like sun, apache,..etc.
- But, WSDL development is unique, it is not platform, language, technology, API, implementation specific.

We can say that we are developing Web Services using →
JAX-RPC, Contract First Approach
JAX-RPC, Contract Last Approach

Differences between Bottom-Up & Top-Down approach:

Bottom-Up Approach	Top-Down Approach
1. It is called as Java WSDL/business first approach.	1. It is called WSDL-Java/Contract first Approach.
2. The developed Web Service document is not stable and reliable document it contains the effect of underlying technology.	2. WSDL document is stable, reliable and does not contain underlying technology effect.
3. Good for converting existing components into Web Services.	3. Good to develop the Web Service components from scratch level.
4. Strong knowledge of XML Schema,WSDL is not required.	4. Strong knowledge of XML Schema,WSDL is required.
5. Good for exposing overloaded business components.	5. Complex for exposing overloaded business components.
6. Gives less control on WSDL document.	6. Gives more control on WSDL document.
7. After developing Web Service component we need to develop Client Application.	7. Based on WSDL document we can develop Client and Web Service components parallel.
8. We can take more support of tools	8. Tools support is limited.

Note: All components that we have developed so far comes under Bottom-Up Approach based Web Service component.

NOTE: In the internet world we prefer to use HTTP protocol, and we have so many technologies which supports HTTP protocol.

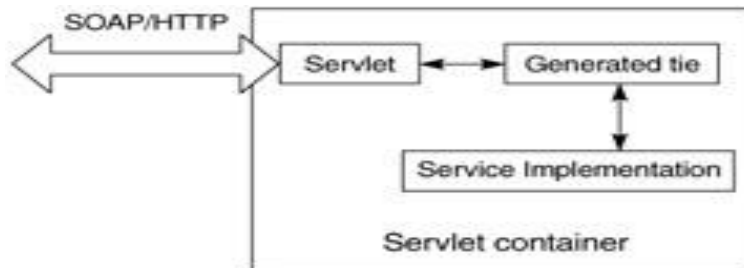
The HTTP protocols implemented API's are → 1) **Servlet API**, 2) **EJB API**

EndPoint: A *service endpoint* is the perimeter where the SOAP message is received and the response dispatched. It is the physical entity exposed to service consumers that essentially services client requests. An endpoint is provided by the runtime and is not written by developers. An endpoint is bound to the transport protocol. Because a runtime is required to support an HTTP transport, JAX-RPC also defines the behavior of an endpoint for this protocol as a Java servlet,

It is a component which receives "**Consumer**" request.

- In general we use either **servlet** or **ejb** as an endpoint.

So, we can say we are developing web services using -->
JAX-RPC, Servlet end point url, Contract First Approach
JAX-RPC, EJB end point url, Contract Last Approach

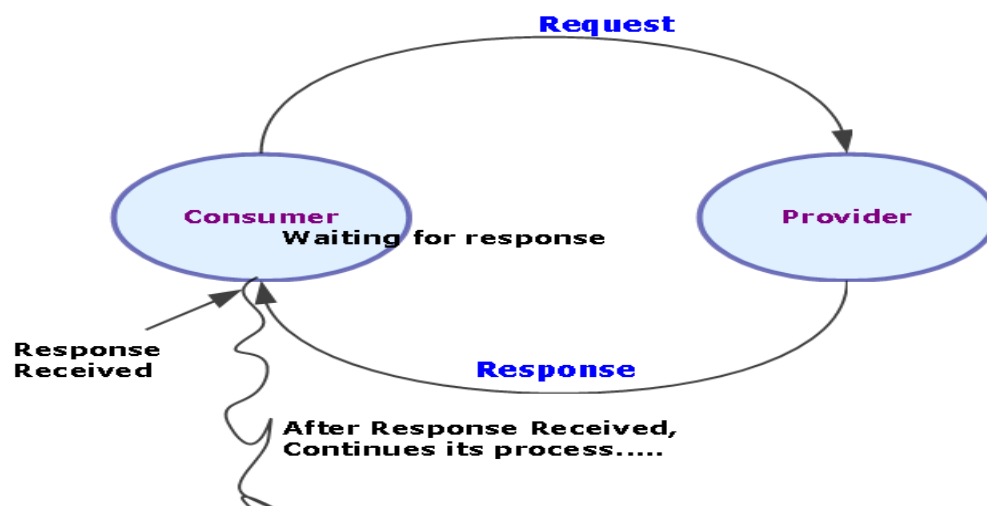


MEP: Message Exchanging Patterns

→ "**Consumer**" can communicate with "**Provider**" in three ways

- 1) Synchronous request-reply
- 2) Asynchronous request-reply
- 3) Fire and Forget

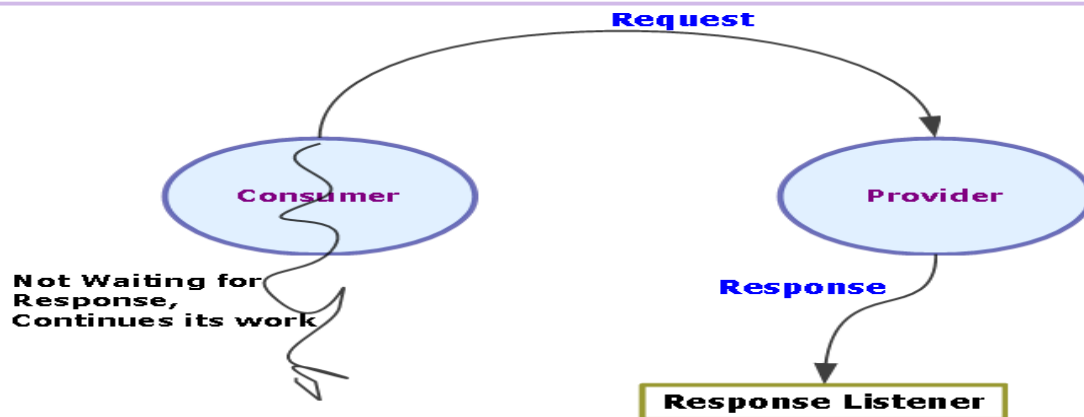
1) **Synchronous request-reply** : In this "**Consumer**" sends request to "**Provider**", and "**Consumer**" blocks until response comes back from the "**Provider**", until response comes back "**Consumer**" can't proceed.



2) **Asynchronous request-reply** : In this "**Consumer**" sends the request to "**Provider**", and "**Consumer**" will not wait until it gets the response.

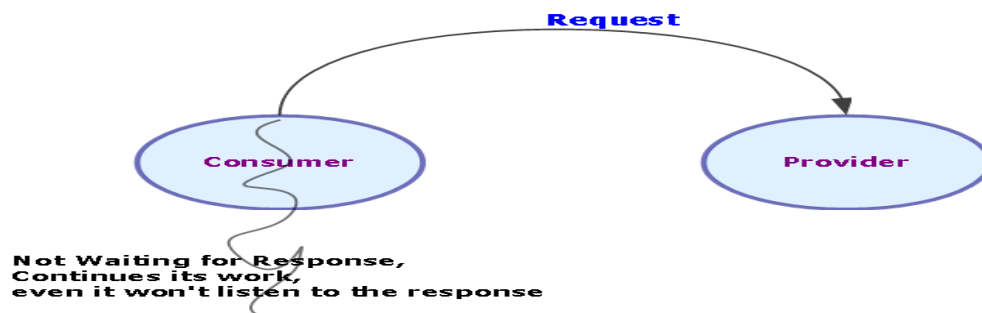
After request sent to the "**Provider**" it continues its flow of execution.

"**Consumer**" will have "**Response Listener**", which listens the response from the "**Provider**".



3) Fire and Forget: In this "Consumer" sends the request to "Provider", and "Consumer" will not wait until it gets the response. Even it won't have any Response Listener.

→ Once the request is sent to the "Provider" it continues its process, doesn't bother about response.



So, We can say we are developing web services using

- JAX-RPC, Servlet end point url, Contract First Approach, Synchronous request-reply
- JAX-RPC, Servlet end point url, Contract First Approach, Asynchronous request-reply
- JAX-RPC, Servlet end point url, Contract First Approach, fire and forget

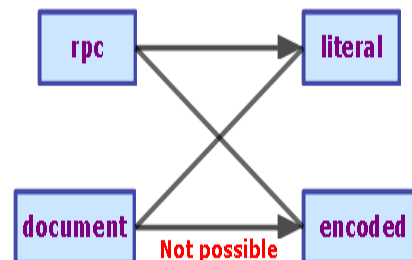
MEF: Message Exchanging Formats

→ When "Consumer", "Provider" exchanging the information, that information can happens in the following formats:

- ⤴ document-literal
- ⤴ rpc-literal
- ⤴ rpc-encoded

→ Default **MEF** is "rpc-encoded"

literal → as it is XML is exchanging
encoded → **UTF-8, UTF-16**

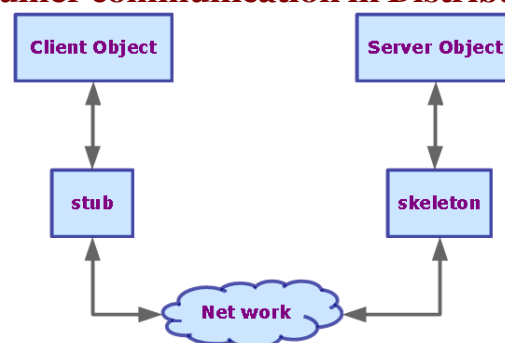


So, we can say we are developing web services using -->

- JAX-RPC, Servlet end point url, Contract First Approach, Synchronous request-reply, document-literal
- JAX-RPC, Servlet end point url, Contract First Approach, Synchronous request-reply, rpc-literal
- JAX-RPC, Servlet end point url, Contract First Approach, Synchronous request-reply, rpc-encoded



Provider and Consumer communication in Distributed Technology:



Server Object side request:

- 1) Interface for the service object
- 2) Service object
- 3) Skeleton

Skeleton:

- It is a server side proxy for the service object
- it is a server socket program
- it is generated by tools

Skeleton functions:

- Receiving the method call from the stub
- Unmarshaling the method input args
- Invoking actual method call on the service object
- Receiving the result from the service object
- Marshaling the result.
- Returning the marshaled result to the stub.

Client side requirement:

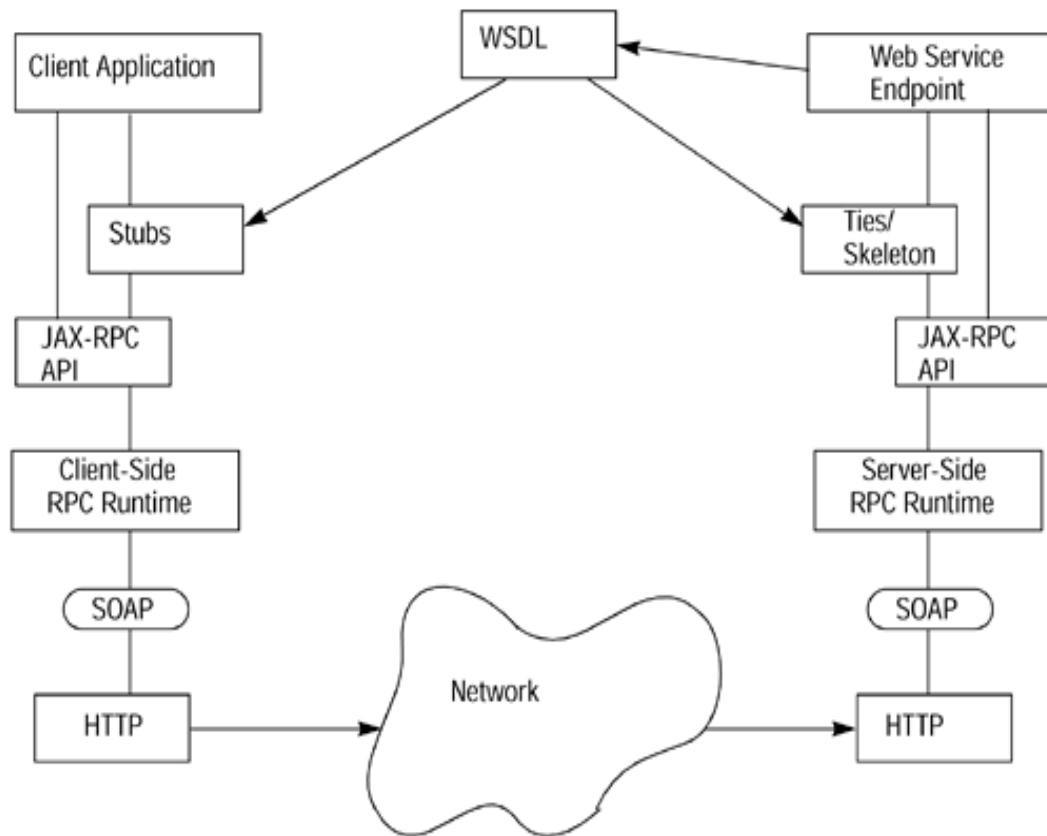
- 1) Interface for the service object
- 2) Stub

Stub:

- A stub is a client side proxy for the service object
- It is a java object that implements the same interface which the service object impls
- It is client socket program
- It is generated by a tools

Stub functions:

- Client makes a method call on stub
- Performing marshaling of the input args.
- Passing the methods call along with marshaled input args to the skeleton
- Receiving the result from the skeleton
- Un-marshaling the result returned by the skeleton
- Returning un-marshaled result to the client



Web services development with JAX-RPC:

1) Download **jwsdp-2_0-windows-i586.exe** (**jwsdp** → Java Web Services Developer Pack)

<http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-jwsdp-419428.html>

select → Java Web Services Developer Pack 2.0 22.69 MB → jwsdp-2_0-windows-i586.exe

The file **jwsdp-2_0-windows-i586.exe** is the java WSDP installer. After downloading, double click on the installer icon and click on run button to install.

2) After installation, we need to set the path to jax-rpc.

Path → ;C:\Sun\jwsdp-2.0\jaxrpc\bin;

Now check by typing "**wscompile -help**" in command prompt, it shows list of options.

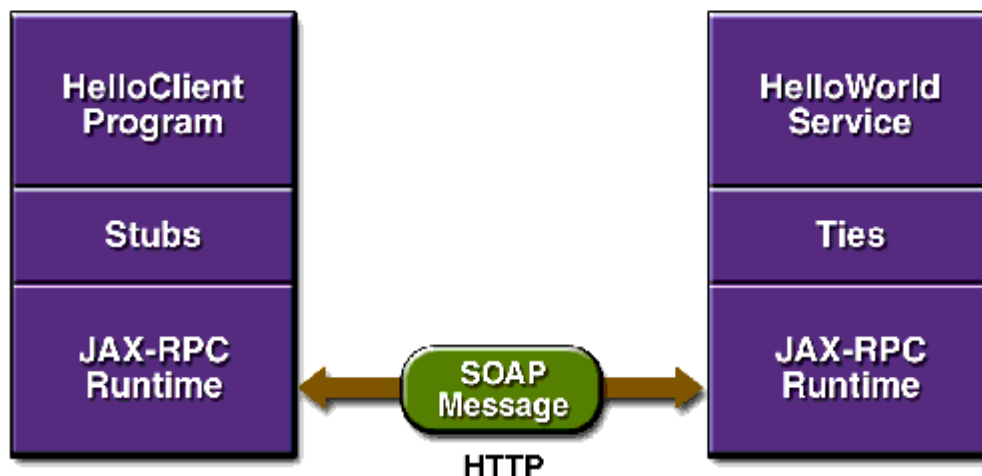
Bottom-up approach (Business First Approach)

Project in Eclipse: Example1

JAX-RPC Web service Server:

The basic steps for developing a JAX-RPC Web service are as follows.

- 1.Code the service definition interface and implementation class.
- 2.Compile the service definition code .
- 3.Package the code in a WAR file.
- 4.Generate the ties and the WSDL file.
- 5.Deploy the service.



Detailed description of what happens at runtime:

- 1.To call a remote procedure, the HelloClient program invokes a method on a stub, a local object that represents the remote service.
- 2.The stub invokes routines in the JAX-RPC runtime system.
- 3.The runtime system converts the remote method call into a SOAP message and then transmits the message as an HTTP request.
- 4.When the server receives the HTTP request, the JAX-RPC runtime system extracts the SOAP message from the request and translates it into a method call.
- 5.The JAX-RPC runtime system invokes the method on the tie object.

- 6.The tie object invokes the method on the implementation of the HelloWorld service.
- 7.The runtime system on the server converts the method's response into a SOAP message and then transmits the message back to the client as an HTTP response.
- 8.On the client, the JAX-RPC runtime system extracts the SOAP message from the HTTP response and then translates it into a method response for the HelloClient program.

Service Side:

HelloIF.java - the service definition interface

HelloImpl.java - the service definition implementation class, it implements the HelloIF interface

HelloClient.java - the remote client that contacts the service and then invokes the sayHello method

config.xml - a configuration file read by the wscompile tool

jaxrpc-ri.xml - a configuration file read by the wsdeploy tool

web.xml - a deployment descriptor for the Web component (a servlet) that dispatches to the service

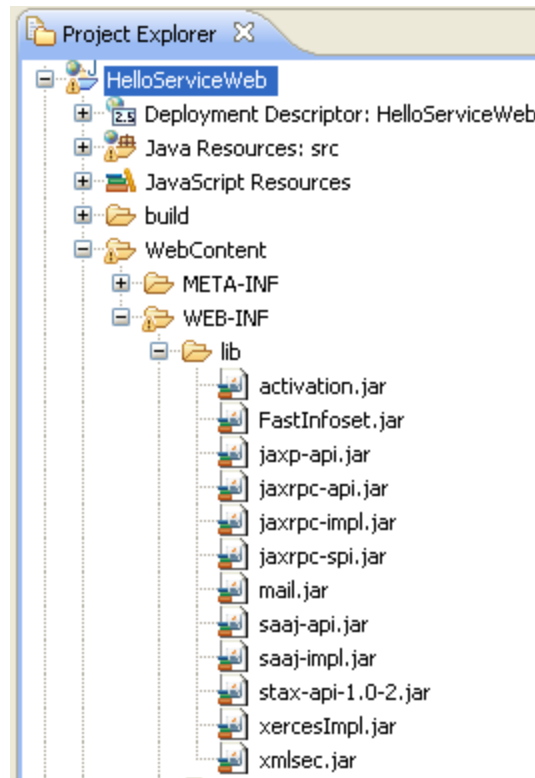
I) Project set up:

1) Select Dynamic Web Project--> name WeatherServiceWeb

2) Copy all the required jar bundles in to project

```
WebContent/WEB-INF/lib
activation.jar
dom.jar
FastInfoset.jar
jaxp-api.jar
jaxrpc-api.jar
jaxrpc-impl.jar
jaxrpc-spi.jar
jsr173_api.jar
mail.jar
relaxngDatatype.jar
resolver.jar
saaj-api-1.3.jar
saaj-impl-1.3.2.jar
```

stax-api-1.0.2.jar
xercesImpl.jar
xmlsec.jar
xsdlib.jar

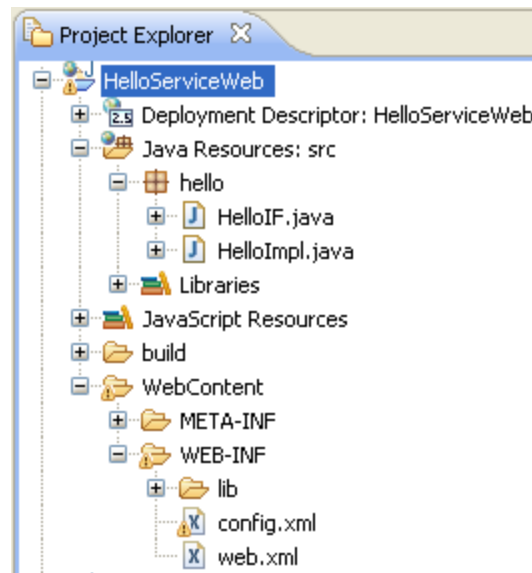


III) Java Code:

3) Create **hello**, under which, create a new **Service Endpoint interface** which acts as contract between "**Consumer**" and "**Provider**". Ex: **HelloIF.java**

Following are the rules while creating **SEI** interface:

- 1) Our Service Endpoint Interface should extend from **java.rmi.Remote** interface.
- 2) Declare all business methods that we want to expose as **webservice** methods.
- 3) All the methods that are declared in SEI interface should be public and should have parameters and return values in serializable in nature.
- 4) All our Web Service methods should be declared to throw **RemoteException**.

**HelloIF.java**

```
package hello;
```

```
import java.rmi.Remote;  
import java.rmi.RemoteException;
```

```
public interface HelloIF extends Remote {  
    public String sayHello(String s) throws RemoteException;  
}
```

4) Now create a java class under the same package, which implements the above defined **SEI interface** and override all the methods declared in the SEI interface. Implement the business logic in the Web Service methods.

Note: Web Service methods in the implementation class are not required to throw **RemoteException**.

HelloImpl.java

```
package hello;
```

```
public class HelloImpl implements HelloIF {  
    public String message ="Hello";  
    public String sayHello(String s) {  
        return message + s;  
    }  
}
```

IV) Configuration:

once, we are finished with our java part. Now write couple of XML files to generate

skeleton/stubs and binding information which performs the handshaking process.

5) Create a new **config.xml** configuration file under **/WEB-INF** dir, which contains the information that describe our web-service.

config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
<service name="HelloService"
  targetNamespace="http://com.test/wsd"
  typeNamespace="http://com.test/types"
  packageName="hello.service.binding">
  <interface name="hello.HelloIF"
    ServantName="hello.HelloImpl" />
</service>
</configuration>
```

→ In the configuration file we should modify the **targetnamespace**, **typenamespace**, **packageName** under which we want to generate binding classes.

→ In the **<interface>** subelement **name** attribute specifies the service endpoint **interface** and **servantName** attribute specifies the class that implements endpoint interface.

NOTE: The **<configuration>** may contain exactly one **<service>** or **<WSDL>** or **<modelfile>**

wscompile: The **wscompile** tool generates stubs, ties, serializers and WSDL files used in JAX-RPC clients and services. The tool reads a configuration file, which specifies either file, a compiled service endpoint interface.

syntax: **wscompile [options] configuration_file**

where [options] include:

- classpath <path> specify where to find input class files
- cp <path> same as -classpath <path>
- d <directory> specify where to place generated output files
- define define a service
- f:<features> enable the given features (see below)
- features:<features> same as -f:<features>
- g generate debugging info
- gen same as -gen:client
- gen:client generate client artifacts (stubs, etc.)
- gen:server generate server artifacts (ties, etc.)
- source <version> generate code for the specified JAXRPC SI version.
supported versions are: 1.0.1, 1.0.3, 1.1, 1.1.1 and 1.1.2(default)
- httpproxy:<host>:<port> specify a HTTP proxy server (port defaults to 8080)
- import generate interfaces and value types only
- keep keep generated files
- model <file> write the internal model to the given file
- nd <directory> specify where to place non-class generated files
- O optimize generated code

-s <directory>	specify where to place generated source files
-verbose	output messages about what the compiler is doing
-version	print version information
-mapping <file>	write the 109 mapping file to the given file
-security <file>	Security configuration file

Exactly one of the -import, -define, -gen options must be specified.

The -f option requires a comma-separated list of features.

Supported features (-f):

datahandleronly	always map attachments to the DataHandler type
documentliteral	use document literal encoding
rpcliteral	use rpc literal encoding
explicitcontext	turn on explicit service context mapping
infix:<name>	specify an infix to use for generated ties and serializers
infix=<name>	same as infix:<name> (not on Windows)
jaxbenumtype	map anonymous enumeration to its base type.
nodatabinding	turn off data binding for literal encoding
noencodedtypes	turn off encoding type information
nomultirefs	turn off support for multiple references
norpcstructures	do not generate RPC structures (-import only)
novalidation	turn off full validation of imported WSDL documents
resolveidref	resolve xsd:IDREF
searchschema	search schema aggressively for subtypes
serializeinterfaces	turn on direct serialization of interface types
strict	generate code strictly compliant with the JAXRPC 1.1 specification
useonewayoperations	allow generation of one-way operations
wsi	enable WSI-Basic Profile features (for document/literal and rpc/literal)
unwrap	enable unWrapping of document/literal wrapper elements in wsi mode
donotoverride	donot regenerate the classes
donotunwrap	disable unWrapping of document/literal wrapper elements in wsi mode (Default)

Internal options (unsupported):

-Xdebugmodel:<file>	write a readable version of the model to a file
-Xprintstacktrace	print exception stack traces
-Xserializable	generate value types that implement Serializable interface

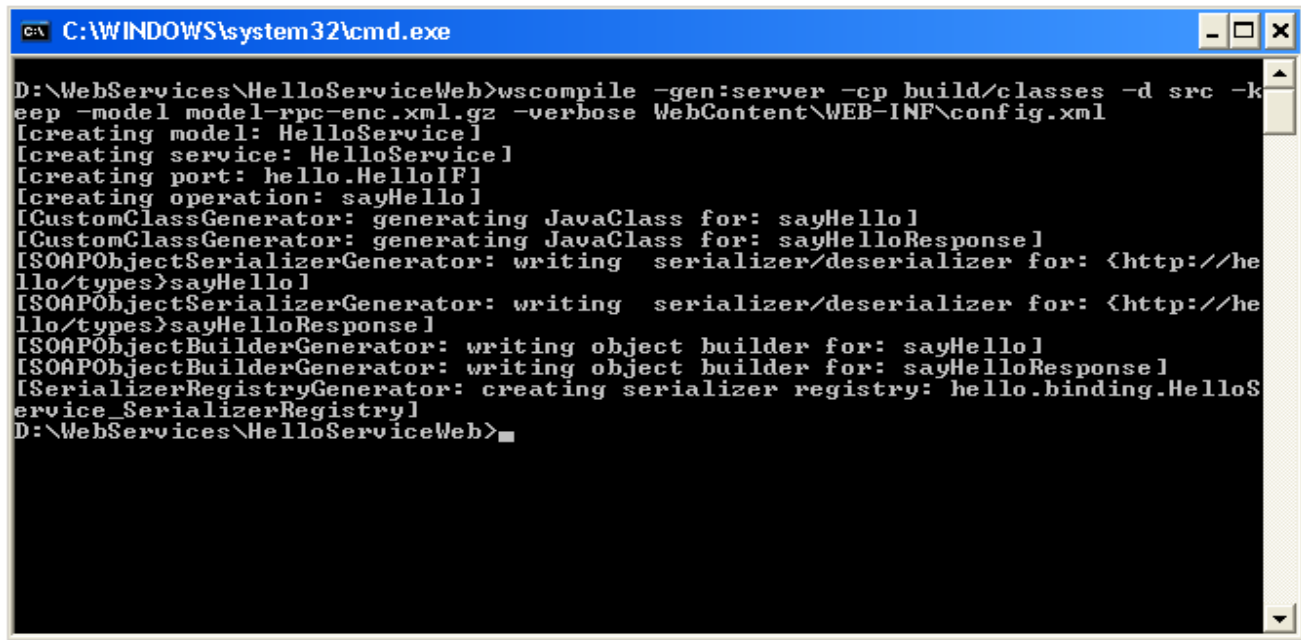
Examples:

```
wscompile -gen -classpath lib/foo.jar;lib/bar.jar -d generated config.xml
wscompile -gen -f:infix:Name -d generated config.xml
wscompile -define -f:nodatabinding -f:novalidation config.xml
wscompile -import -f:explicitcontext config.xml
```

6) Now navigate to the root directory of the project in the command prompt and execute the following command:

wscompile -gen:server -cp build/classes -d src -keep -model model-rpc-enc.xml.gz -verbose WebContent\WEB-INF\config.xml

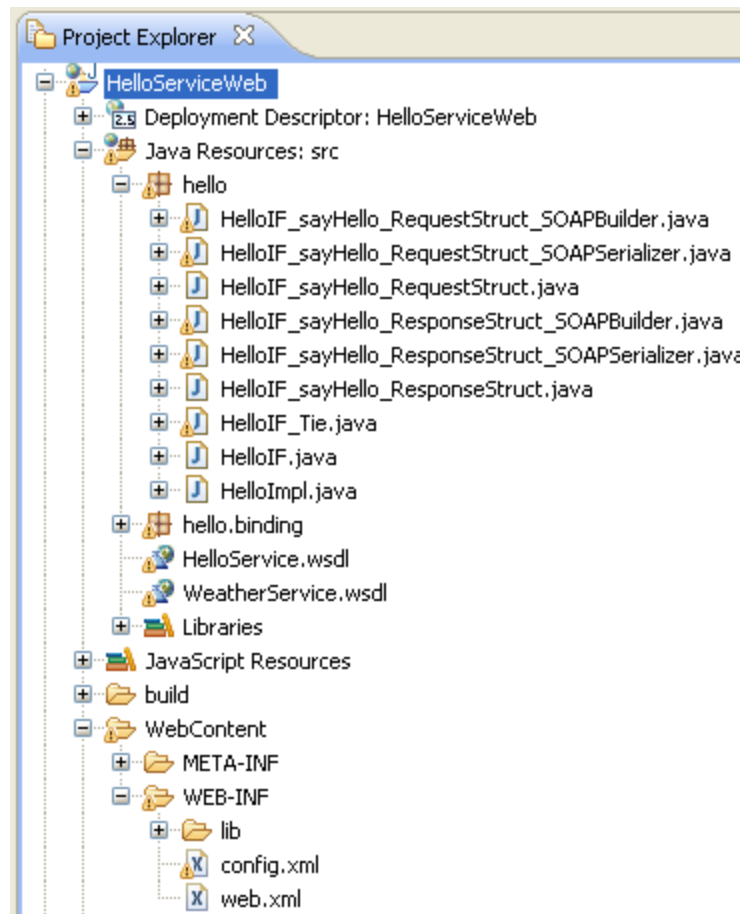
Ex: D:\WebServices\HelloServiceWeb>wscompile -gen:server -cp build/classes -d src -keep -model model-rpc-enc.xml.gz -verbose WebContent\WEB-INF\config.xml



```
C:\WINDOWS\system32\cmd.exe

D:\WebServices\HelloServiceWeb>wscompile -gen:server -cp build/classes -d src -keep -model model-rpc-enc.xml.gz -verbose WebContent\WEB-INF\config.xml
[creating model: HelloService]
[creating service: HelloService]
[creating port: hello.HelloIF]
[creating operation: sayHello]
[CustomClassGenerator: generating JavaClass for: sayHello]
[CustomClassGenerator: generating JavaClass for: sayHelloResponse]
[SOAPObjectSerializerGenerator: writing serializer/deserializer for: {http://hello/types}sayHello]
[SOAPObjectSerializerGenerator: writing serializer/deserializer for: {http://hello/types}sayHelloResponse]
[SOAPObjectBuilderGenerator: writing object builder for: sayHello]
[SOAPObjectBuilderGenerator: writing object builder for: sayHelloResponse]
[SerializerRegistryGenerator: creating serializer registry: hello.binding.HelloService_SerializerRegistry]
D:\WebServices\HelloServiceWeb>
```

7) Now go to the project and refresh, it shows generated stubs/skeleton files



→ This command along with generating binding classes, it will also generates a **model file** and **wsdl file** under source.

optional: If we want our own url patterns for our web service we can give that in the WSDL doc.

```
<service name="HelloService">
  <port name="HelloPort" binding="tns:HelloInfoBinding">
    <soap:address location="http://localhost:8081/HelloServiceWeb/hello"/>
  </port>
</service>
```

8) Drag and drop **model and wsdl files** into **/WEB-INF** dir.

10) Create **jaxrpc-ri.xml** configuration file, under **/WEB-INF** dir.

jaxrpc-ri.xml

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

<webServices
```



```
xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/dd"
version="1.0"
targetNamespaceBase="http://com.test/wsdl"
typeNamespaceBase="http://com.test/types"
urlPatternBase="/ws">

<endpoint
  name="MyHello"
  displayName="HelloWorld Web Service"
  description="A Simple Web Service"
  wsdl="/WEB-INF/HelloService.wsdl"
  interface="hello.HelloIF"
  implementation="hello.HelloImpl"
  model="/WEB-INF/model-rpc-enc.xml.gz" />

<endpointMapping
  endpointName="MyHello"
  urlPattern="/hello" />

</webServices>
```

→ **<webservices>** element must contain one or more **<endpoint>** elements.

In this example, the interface and implementing attributes of **<endpoint>** specifies the service's interface and implementation class. The **<endpointMapping>** element associates the service name with a URL pattern.

9) Export the project as **WAR** file onto the desktop. (using File-->Export-->Web-->WAR file)

→ **wsdeploy** → This tool reads a WAR file and then generates another **WAR** file that is capable for deployment. Basically this tool updates content in **web.xml** file.

10) Navigate to the directory where we exported our project and execute the following command:

wsdeploy -verbose -o target.war HelloServiceWeb.war

Ex: C:\Documents and Settings\venukumars\Desktop>wsdeploy -verbose -o target.war
HelloServiceWeb.war

```

C:\WINDOWS\system32\cmd.exe

C:\Documents and Settings\venukumars\Desktop>wsdeploy -verbose -o target.war HelloWorldServiceWeb.war
info: created temporary directory: C:\DOCUMENTS~1\VENUKU~1\LOCALS~1\Temp\jaxrpc-deploy-26bf37
error: modeler error: java.io.FileNotFoundException: C:\DOCUMENTS~1\VENUKU~1\LOCALS~1\Temp\jaxrpc-deploy-26bf37\WEB-INF\model-rpc-enc.xml.gz (The system cannot find the file specified)
info: created output war file: C:\Documents and Settings\venukumars\Desktop\target.war
info: removed temporary directory: C:\DOCUMENTS~1\VENUKU~1\LOCALS~1\Temp\jaxrpc-deploy-26bf37
C:\Documents and Settings\venukumars\Desktop>

```

11) Unzip the **target.war**, we should be able to find **jaxrpc-ri-runtime.xml** and **web.xml**. copy both these files into **/WEB-INF** dir. Deploy the project and start server.

We can find the following changes in **web.xml** is **<listener-class>**, **<servlet-class>**, **<url-pattern>**

web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="WebApp_ID"
  version="2.5"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <display-name>HelloServiceWeb</display-name>

  <listener>
    <listener-class>com.sun.xml.rpc.server.http.JAXRPCContextListener
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>MyHello</servlet-name>
    <servlet-class>com.sun.xml.rpc.server.http.JAXRPCServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

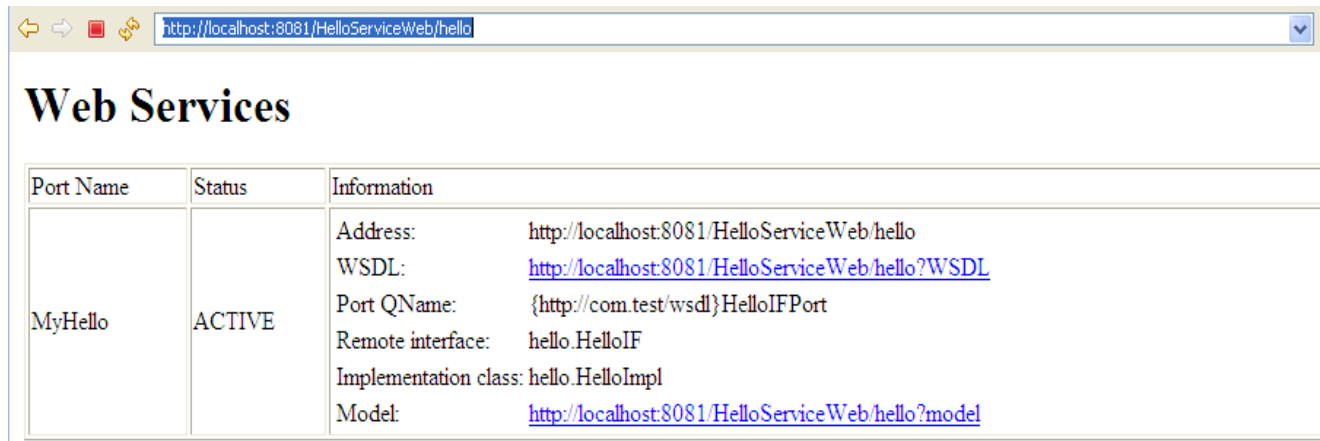
  <servlet-mapping>
    <servlet-name>MyHello</servlet-name>
    <url-pattern>/hello</url-pattern>
  </servlet-mapping>

</web-app>

```

NOTE: Open **jaxrpc-ri-runtime.xml** file and make sure that, **tie="hello.HelloIF_Tie"** is available.

12) → Now deploy the project into servlet container like tomcat and start server.



Port Name	Status	Information
MyHello	ACTIVE	<p>Address: http://localhost:8081/HelloServiceWeb/hello</p> <p>WSDL: http://localhost:8081/HelloServiceWeb/hello?WSDL</p> <p>Port QName: {http://com.test/wsdl}HelloIFPort</p> <p>Remote interface: hello.HelloIF</p> <p>Implementation class: hello.HelloImpl</p> <p>Model: http://localhost:8081/HelloServiceWeb/hello?model</p>

13) Access the web service **wsdl** with the url format in web browser:

Syntax: *http://<domain>:<port>/<context-root>/<servletpath/url-pattern of service>?wsdl*

<http://localhost:8081/HelloServiceWeb/hello?WSDL>

```

http://localhost:8081/HelloServiceWeb/hello?WSDL
- <definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http://com.test/wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/"
  name="HelloService" targetNamespace="http://com.test/wsdl">
  <types />
  - <message name="HelloIF_sayHello">
    <part name="String_1" type="xsd:string" />
  </message>
  - <message name="HelloIF_sayHelloResponse">
    <part name="result" type="xsd:string" />
  </message>
  - <portType name="HelloIF">
    - <operation name="sayHello" parameterOrder="String_1">
      <input message="tns:HelloIF_sayHello" />
      <output message="tns:HelloIF_sayHelloResponse" />
    </operation>
  </portType>
  - <binding name="HelloIFBinding" type="tns:HelloIF">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc" />
    - <operation name="sayHello">
      <soap:operation soapAction="" />
      - <input>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" use="encoded"
          namespace="http://com.test/wsdl" />
      </input>
      - <output>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" use="encoded"
          namespace="http://com.test/wsdl" />
      </output>
    </operation>
  </binding>
  - <service name="HelloService">
    - <port name="HelloIFPort" binding="tns:HelloIFBinding">
      <soap:address location="http://localhost:8081/HelloServiceWeb/hello"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" />
    </port>
  </service>
</definitions>

```

JAX-RPC Web service client:

To develop a JAX-RPC client, you follow these steps:

1. Generate the stubs.
2. Code the client.
3. Compile the client code. Package the client classes into a JAR file.
5. Run the client.

- 1) Create a java project, and add the required jars
- 2) Create a source folder with name "**resources**" and create a **config.xml**.

config.xml

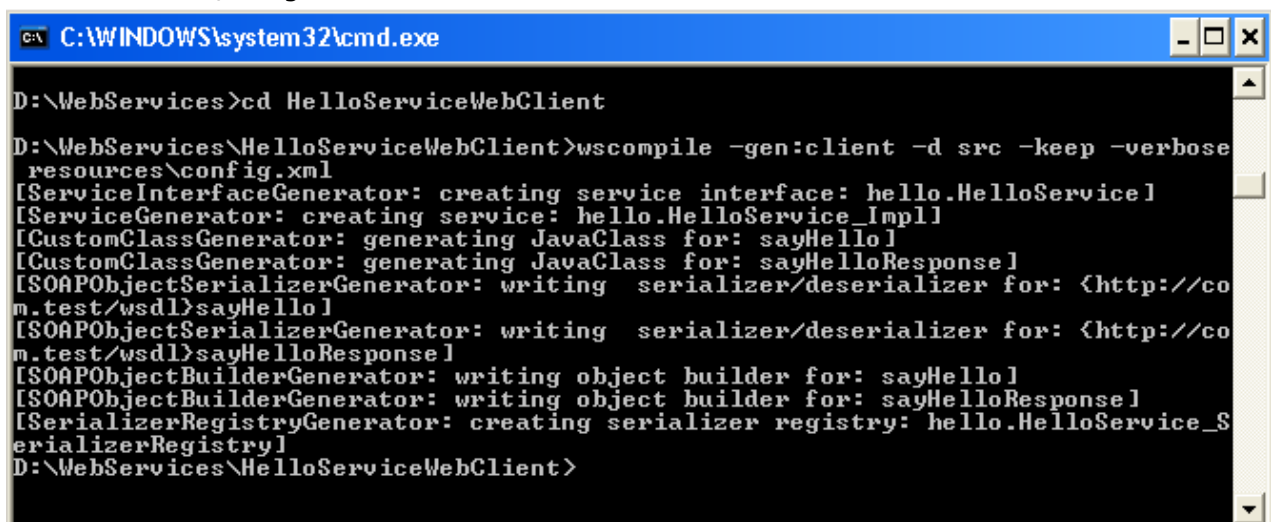
```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <wsdl location="http://localhost:8081/HelloServiceWeb/hello?WSDL"
    packageName="hello" />
</configuration>
```

3) Open command prompt and navigate to our project location .

4) Use **wscompile** tool with **-gen:client** option to generate client side stubs.

wscompile -gen:client -d src -keep -verbose resources\config.xml

D:\WebServices\HelloServiceWebClient>wscompile -gen:client -d src -keep -verbose
resources\config.xml

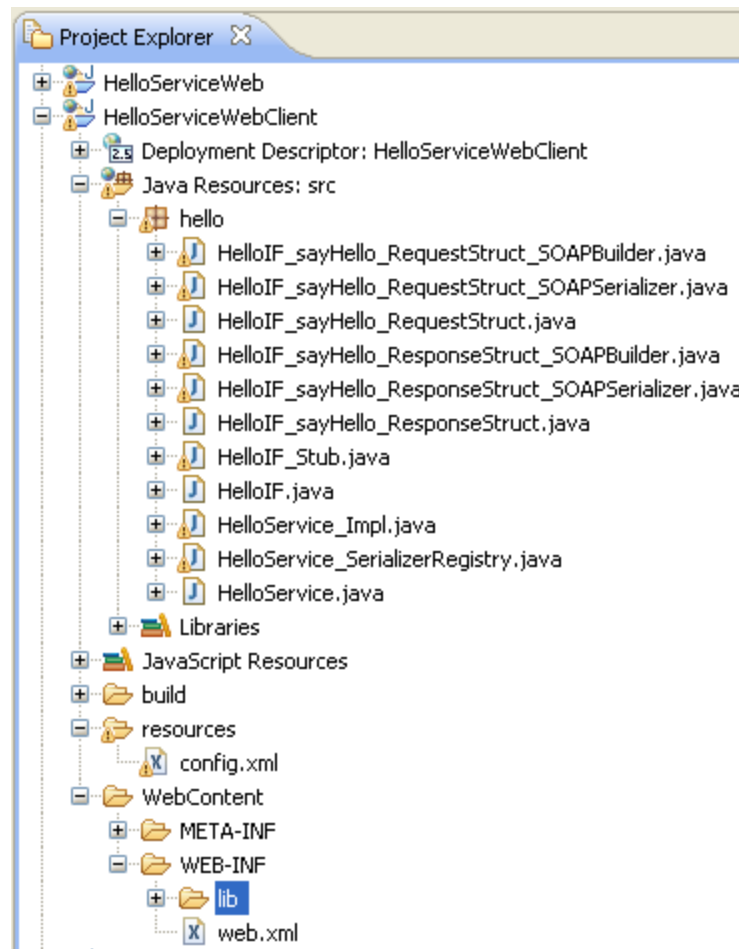


```
C:\WINDOWS\system32\cmd.exe

D:\WebServices>cd HelloServiceWebClient

D:\WebServices\HelloServiceWebClient>wscompile -gen:client -d src -keep -verbose
resources\config.xml
[ServiceInterfaceGenerator: creating service interface: hello.HelloService]
[ServiceGenerator: creating service: hello.HelloService_Impl]
[CustomClassGenerator: generating JavaClass for: sayHello]
[CustomClassGenerator: generating JavaClass for: sayHelloResponse]
[SOAPObjectSerializerGenerator: writing serializer/deserializer for: {http://co
m.test/wsdl}sayHello]
[SOAPObjectSerializerGenerator: writing serializer/deserializer for: {http://co
m.test/wsdl}sayHelloResponse]
[SOAPObjectBuilderGenerator: writing object builder for: sayHello]
[SOAPObjectBuilderGenerator: writing object builder for: sayHelloResponse]
[SerializerRegistryGenerator: creating serializer registry: hello.HelloService_S
erializerRegistry]
D:\WebServices\HelloServiceWebClient>
```

1.5) Now refresh the project, we can see generated stub classes in
hello package.



6) create **hello.client** and create "**HelloServiceClient**" class in that package.

```
package hello.client;

import hello.HelloIF;
import hello.HelloService_Impl;

import javax.xml.rpc.Stub;

public class HelloClient {
    public static void main(String[] args) {
        try {
            Stub stub = createProxy();
            HelloIF hello = (HelloIF) stub;
            System.out.println(hello.sayHello(" Raam!"));
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

```
        private static Stub createProxy() {  
            // Note: MyHello_Impl is implementation-specific.  
            return (Stub) (new HelloService_Impl().getHelloIFPort());  
        }  
    }  
}
```

7) Run **HelloServiceClient** program

Output:

Hello Raam!

Example2:

JAX-RPC Web service Server:

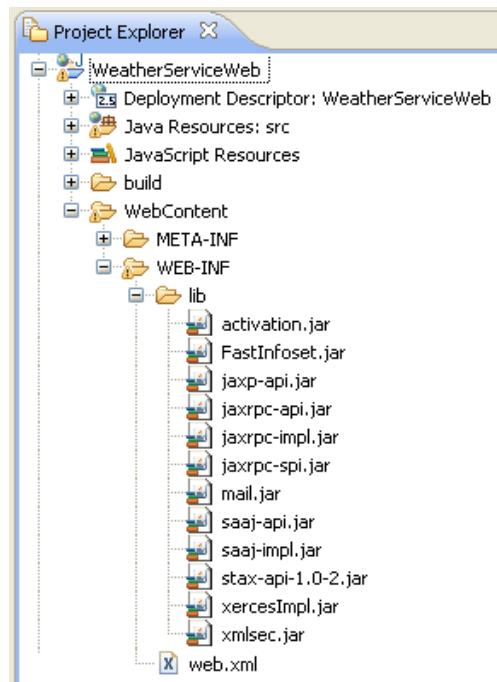
I) Project set up:

1) Select Dynamic Web Project--> name WeatherServiceWeb

2) Copy all the required jar bundles in to project

WebContent/WEB-INF/lib

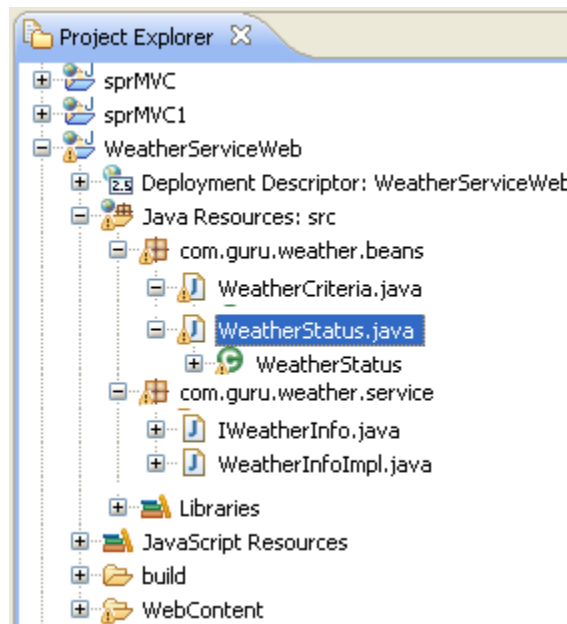
- activation.jar
- dom.jar
- FastInfoset.jar
- jaxp-api.jar
- jaxrpc-api.jar
- jaxrpc-impl.jar
- jaxrpc-spi.jar
- jsr173_api.jar
- mail.jar
- relaxngDatatype.jar
- resolver.jar
- saaj-api-1.3.jar
- saaj-impl-1.3.2.jar
- stax-api-1.0.2.jar
- xercesImpl.jar
- xmlsec.jar
- xsdlib.jar



III) Java Code:

3) Create new package --> **com.guru.weather.beans**, under which create two java beans, which acts as a request and response objects for web service --> **WeatherCriteria** and **WeatherStatus** respectively.

The request and response objects should abide to the rules of Standard Java Beans convention and , the objects should be serializable.

**WeatherCriteria.java****package** com.guru.weather.beans;**import** java.io.Serializable;**public class** WeatherCriteria **implements** Serializable { **private** String city; **private** String state; **private** String country; **public** String getCity() {
 return city;
 } **public void** setCity(String city) {
 this.city = city;
 } **public** String getState() {
 return state;
 } **public void** setState(String state) {
 this.state = state;
 } **public** String getCountry() {
 return country;
 }

```
        public void setCountry(String country) {  
            this.country = country;  
        }  
    }  
}
```

WeatherStatus.java

```
package com.guru.weather.beans;  
  
import java.io.Serializable;  
  
public class WeatherStatus implements Serializable {  
    private int temperature;  
    private double humidity;  
    private String description;  
  
    public int getTemperature() {  
        return temperature;  
    }  
  
    public void setTemperature(int temperature) {  
        this.temperature = temperature;  
    }  
  
    public double getHumidity() {  
        return humidity;  
    }  
  
    public void setHumidity(double humidity) {  
        this.humidity = humidity;  
    }  
  
    public String getDescription() {  
        return description;  
    }  
  
    public void setDescription(String description) {  
        this.description = description;  
    }  
}
```

4) Create **com.guru.weather.service**, under which, create a new Service Endpoint interface which acts as contract between "**Consumer**" and "**Provider**". **Ex: IWeatherInfo.java**

Following are the rules while creating **SEI** interface:

- 1) Our Service Endpoint Interface should extend from **java.rmi.Remote** interface.
- 2) Declare all business methods that we want to expose as **webservice** methods.

- 3) All the methods that are declared in SEI interface should be public and should have parameters and return values in serializable in nature.
- 4) All our Web Service methods should be declared to throw **RemoteException**.

IweatherInfo.java

```
package com.guru.weather.service;
```

```
import java.rmi.Remote;
```

```
import java.rmi.RemoteException;
```

```
import com.guru.weather.beans.WeatherCriteria;
```

```
import com.guru.weather.beans.WeatherStatus;
```

```
public interface IWeatherInfo extends Remote {  
    public WeatherStatus getWeatherInfo(WeatherCriteria weatherCriteria)  
        throws RemoteException;  
}
```

5) Now create a java class under the same package, which implements the above defined SEI interface and override all the methods declared in the SEI interface. Implement the business logic in the Web Service methods.

Note: Web Service methods in the implementation class are not required to throw **RemoteException**.

WeatherInfoImpl.java

```
package com.guru.weather.service;
```

```
import com.guru.weather.beans.WeatherCriteria;
```

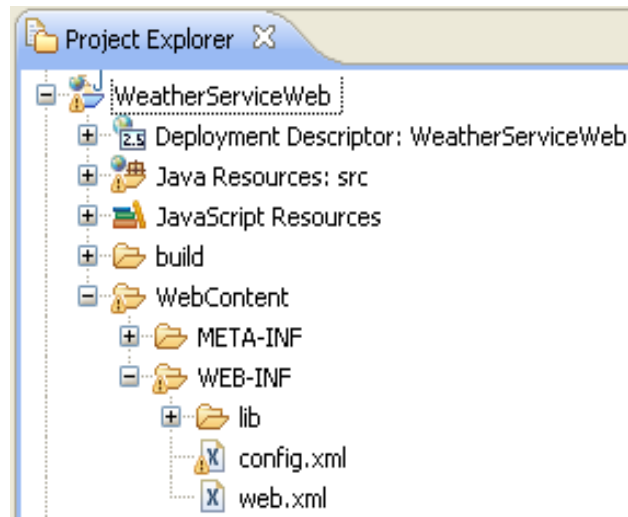
```
import com.guru.weather.beans.WeatherStatus;
```

```
public class WeatherInfoImpl implements IWeatherInfo {  
    public WeatherStatus getWeatherInfo(WeatherCriteria weatherCriteria) {  
  
        if (weatherCriteria != null) {  
            System.out.println(" City : " + weatherCriteria.getCity());  
            System.out.println(" State : " + weatherCriteria.getState());  
            System.out.println(" Country : " + weatherCriteria.getCountry());  
        }  
        WeatherStatus status = new WeatherStatus();  
        status.setTemperature(101);  
        status.setHumidity(12.5);  
        status.setDescription("Today may be rained or may not be");  
        return status;  
    }  
}
```

IV) Configuration:

once, we are finished with our java part. Now write couple of XML files to generate skeleton/stubs and binding information which performs the handshaking process.

6) Create a new **config.xml** configuration file under **/WEB-INF** dir, which contains the information that describe our web-service.



config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <service name="WeatherService" targetNamespace="http://guru.com/weather/wsdl"
    typeNamespace="http://guru.com/weather/types"
    packageName="com.guru.weather.service.binding">
    <interface name="com.guru.weather.service.IWeatherInfo"
      ServantName="com.guru.weather.service.WeatherInfoImpl" />
  </service>
</configuration>
```

→ In the configuration file we should modify the **targetNamespace**, **typeNamespace**, **packageName** under which we want to generate binding classes.

→ In the **<interface>** subelement **name** attribute specifies the service endpoint **interface** and **servantName** attribute specifies the class that implements endpoint interface.

NOTE: The **<configuration>** may contain exactly one **<service>** or **<WSDL>** or **<modelFile>**

wscompile: The **wscompile** tool generates stubs, ties, serializers and WSDL files used in JAX-RPC clients and services. The tool reads a configuration file, which specifies either file, a compiled service endpoint interface.

syntax: **wscompile [options] configuration_file**

where [options] include:

- classpath <path> specify where to find input class files
- cp <path> same as -classpath <path>
- d <directory> specify where to place generated output files
- define define a service
- f: <features> enable the given features (see below)
- features: <features> same as -f: <features>

-g generate debugging info
 -gen same as -gen:client
 -gen:client generate client artifacts (stubs, etc.)
 -gen:server generate server artifacts (ties, etc.)

 -source <version> generate code for the specified JAXRPC SI version.
 supported versions are: 1.0.1, 1.0.3, 1.1, 1.1.1 and 1.1.2(default)
 -httpproxy:<host>:<port> specify a HTTP proxy server (port defaults to 8080)
 -import generate interfaces and value types only
 -keep keep generated files
 -model <file> write the internal model to the given file
 -nd <directory> specify where to place non-class generated files
 -O optimize generated code
 -s <directory> specify where to place generated source files
 -verbose output messages about what the compiler is doing
 -version print version information
 -mapping <file> write the 109 mapping file to the given file
 -security <file> Security configuration file

Exactly one of the -import, -define, -gen options must be specified.

The -f option requires a comma-separated list of features.

Supported features (-f):

datahandleronly always map attachments to the DataHandler type
 documentliteral use document literal encoding
 rpcliteral use rpc literal encoding
 explicitcontext turn on explicit service context mapping
 infix:<name> specify an infix to use for generated ties
 and serializers
 infix=<name> same as infix:<name> (not on Windows)
 jaxbenumtype map anonymous enumeration to its base type.
 nodatabinding turn off data binding for literal encoding
 noencodedtypes turn off encoding type information
 nomultirefs turn off support for multiple references
 norpcstructures do not generate RPC structures (-import only)
 novalidation turn off full validation of imported WSDL documents
 resolveidref resolve xsd:IDREF
 searchschema search schema aggressively for subtypes
 serializeinterfaces turn on direct serialization of interface types
 strict generate code strictly compliant with the JAXRPC
 1.1 specification
 useonewayoperations allow generation of one-way operations
 wsi enable WSI-Basic Profile features (for
 document/literal and rpc/literal)
 unwrap enable unWrapping of document/literal
 wrapper elements in wsi mode
 donotoverride donot regenerate the classes
 donotunwrap disable unWrapping of document/literal
 wrapper elements in wsi mode (Default)

Internal options (unsupported):

- Xdebugmodel:<file> write a readable version of the model to a file
- Xprintstacktrace print exception stack traces
- Xserializable generate value types that implement Serializable interface

Examples:

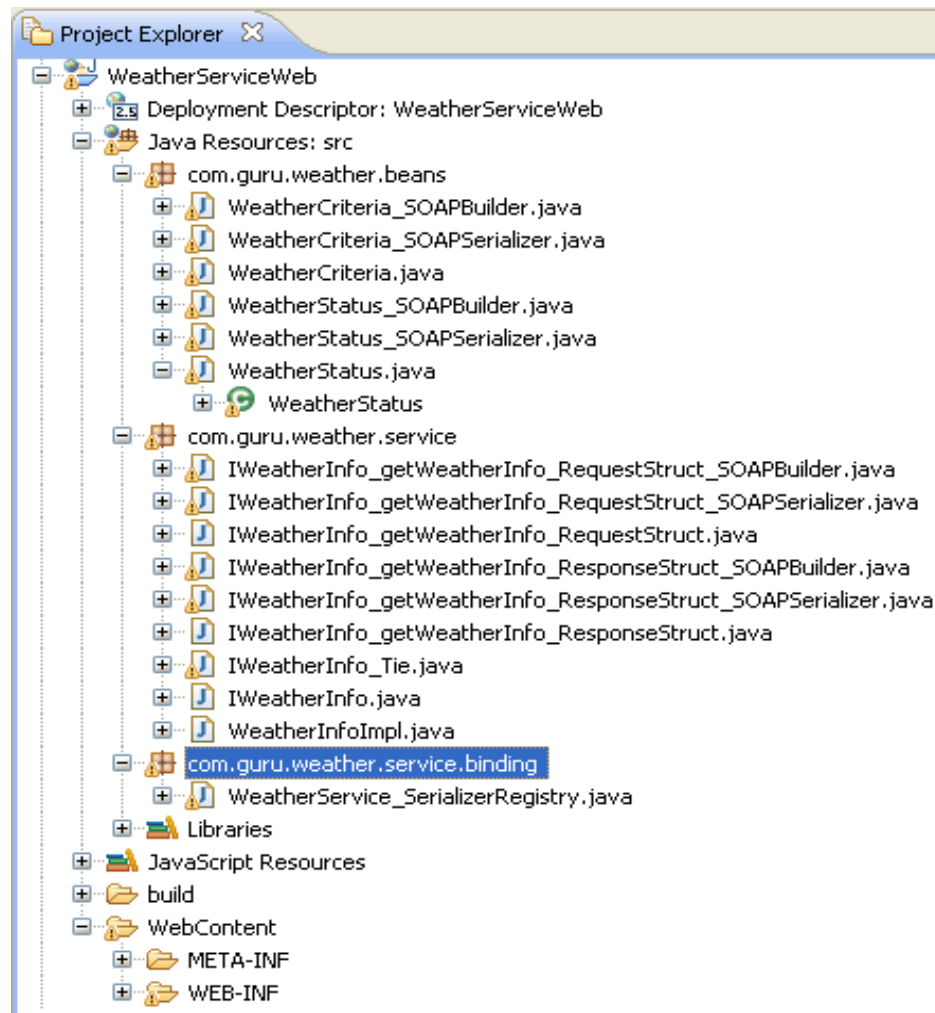
```
wscompile -gen -classpath lib/foo.jar;lib/bar.jar -d generated config.xml
wscompile -gen -f:infix:Name -d generated config.xml
wscompile -define -f:nodatabinding -f:novalidation config.xml
wscompile -import -f:explicitcontext config.xml
```

7) Now navigate to the root directory of the project in the command prompt and execute the following command:

wscompile -gen:server -cp build/classes -d src -keep -model model-rpc-enc.xml.gz -verbose WebContent\WEB-INF\config.xml

Ex: D:\WebServices\WeatherServiceWeb>wscompile -gen:server -cp build/classes -d src -keep -model model-rpc-enc.xml.gz -verbose WebContent\WEB-INF\config.xml

8) Now go to the project and refresh, it shows generated stubs/skeleton files



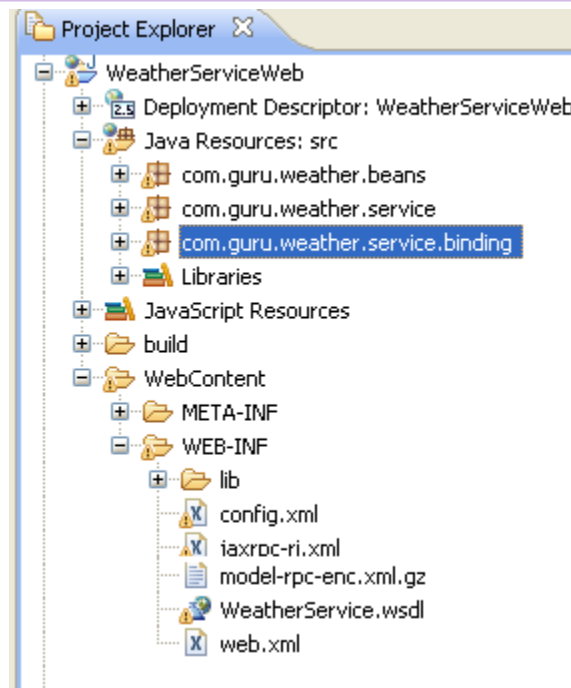
→ This command along with generating binding classes, it will also generates a **model file** and **wsdl file** under source.

optional: If we want our own url patterns for our web service we can give that in the WSDL doc.

```
<service name="WeatherService">
  <port name="IWeatherInfoPort" binding="tns:IWeatherInfoBinding">
    <soap:address location="http://localhost:8081/WeatherServiceWeb/getWeather"/>
  </port>
</service>
```

9) Drag and drop **model and wsdl files** into **/WEB-INF** dir.

10) Create **jaxrpc-ri.xml** configuration file, under **/WEB-INF** dir.

**jaxrpc-ri.xml**

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

```
<webServices xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/dd"
  version="1.0" targetNamespaceBase="http://guru.com/weather/wsdl"
  typeNamespaceBase="http://guru.com/weather/types" urlPatternBase="/ws">
  <endpoint name="IWeatherInfo" displayName="Weather Web Service"
    description="Get Weather Web Service" wsdl="/WEB-INF/WeatherService.wsdl"
    interface="com.guru.weather.service.IWeatherInfo"
    implementation="com.guru.weather.service.WeatherInfoImpl"
    model="/WEB-INF/model-rpc-enc.xml.gz" />

    <endpointMapping endpointName="Weather" urlPattern="/getWeather" />
</webServices>
```

→ **<webServices>** element must contain one or more **<endpoint>** elements.

In this example, the interface and implementing attributes of **<endpoint>** specifies the service's interface and implementation class. The **<endpointMapping>** element associates the service name with a URL pattern.

10) Export the project as **WAR** file onto the desktop. (using File-->Export-->Web-->WAR file)

→ **wsdeploy** → This tool reads a WAR file and then generates another **WAR** file that is capable for deployment. Basically this tool updates content in **web.xml** file.

11) Navigate to the directory where we exported our project and execute the following command:

wsdeploy -verbose -o target.war WeatherServiceWeb.war

Ex: C:\Documents and Settings\venukumars\Desktop>wsdeploy -verbose -o target.war WeatherServiceWeb.war

12) Unzip the **target.war**, we should be able to find **jaxrpc-ri-runtime.xml** and **web.xml**. copy both these files into **/WEB-INF** dir. Deploy the project and start server.

We can find the following changes in **web.xml** is **<listener-class>**, **<servlet-class>**, **<url-pattern>**

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="WebApp_ID"
  version="2.5"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <display-name>WeatherServiceWeb</display-name>
  <listener>
    <listener-class> com.sun.xml.rpc.server.http.JAXRPCContextListener </listener-class>
  </listener>
  <servlet>
    <servlet-name>IWeatherInfo</servlet-name>
    <servlet-class> com.sun.xml.rpc.server.http.JAXRPCServlet </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>IWeatherInfo</servlet-name>
    <url-pattern>/ws/IWeatherInfo</url-pattern>
  </servlet-mapping>
</web-app>
```

NOTE: Open **jaxrpc-ri-runtime.xml** file and make sure that, **tie="com.guru.weather.service.IWeatherInfo_Tie"** is available.

→ Now deploy the project into servlet container like tomcat and start server.

13) Access the web service **wsdl** with the url format in web browser:

Syntax: `http://<domain>:<port>/<context-root>/<servletpath>/url-pattern of service?wsdl`

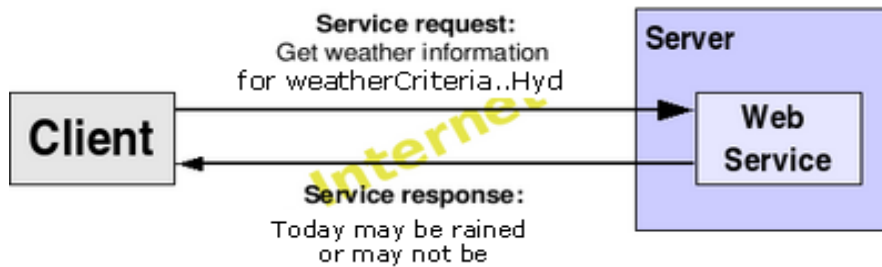
<http://localhost:8081/WeatherServiceWeb/ws/IWeatherInfo?WSDL>

```
<?xml version="1.0" encoding="UTF-8" ?>
- <definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http://guru.com/weather/wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:ns2="http://guru.com/weather/types" name="WeatherService" targetNamespace="http://guru.com/weather/wsdl">
- <types>
- <schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://guru.com/weather/types" xmlns:soap11-
  enc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" targetNamespace="http://guru.com/weather/types">
  <import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
- <complexType name="WeatherCriteria">
- <sequence>
  <element name="city" type="string" />
  <element name="country" type="string" />
  <element name="state" type="string" />
  </sequence>
</complexType>
- <complexType name="WeatherStatus">
- <sequence>
  <element name="description" type="string" />
  <element name="humidity" type="double" />
  <element name="temperature" type="int" />
  </sequence>
</complexType>
</schema>
</types>
- <message name="IWeatherInfo_getWeatherInfo">
  <part name="WeatherCriteria_1" type="ns2:WeatherCriteria" />
</message>
- <message name="IWeatherInfo_getWeatherInfoResponse">
  <part name="result" type="ns2:WeatherStatus" />
</message>
- <portType name="IWeatherInfo">
- <operation name="getWeatherInfo" parameterOrder="WeatherCriteria_1">
  <input message="tns:IWeatherInfo_getWeatherInfo" />
```

JAX-RPC Web service client:

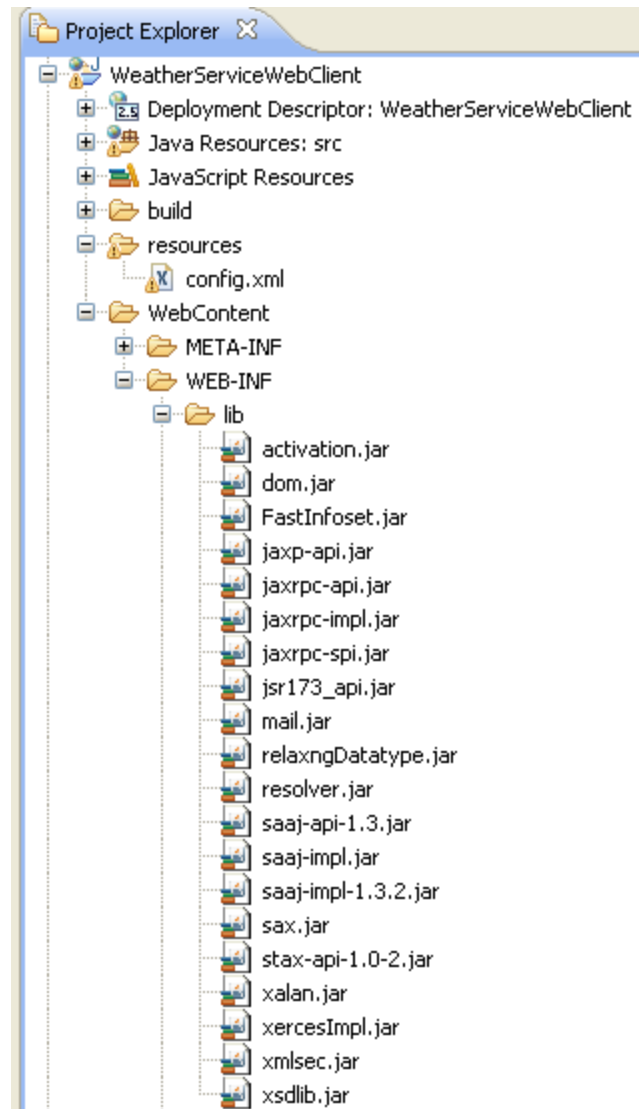
We can develop Web service client using JAX-RPC in three ways:

- 1) **Stub-based** → by creating stubs at client side
- 2) **Dynamic-proxy based** → without stubs at client side
- 3) **Dynamic-Invocation-Interface** → Our own serializer, deserilizer



1) stub-based client:

- 1.1) Create a java project, and add the required jars
- 1.2) Create a source folder with name "**resources**" and create a **config.xml**.



config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
    <wsdl location="http://localhost:8081/WeatherServiceWeb/ws/IWeatherInfo?WSDL"
           packageName="com.guru.weather.ws.stubs">
    </wsdl>
</configuration>
```

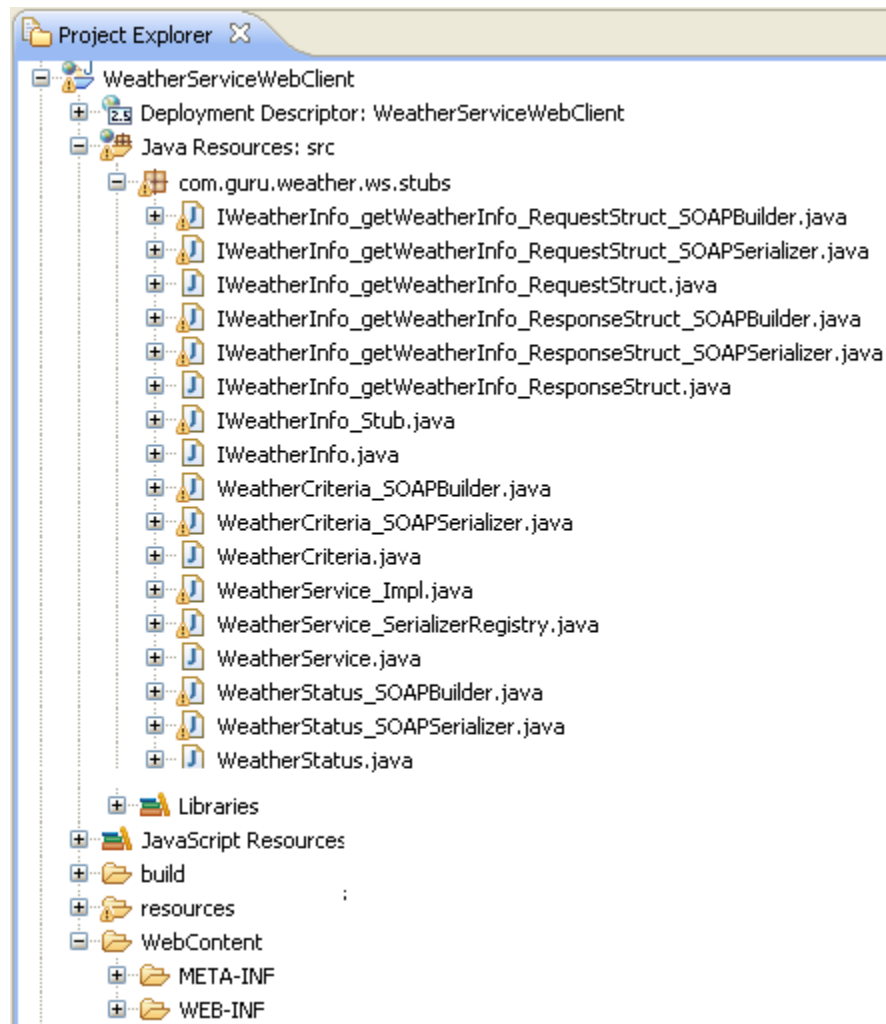
1.3) Open command prompt and navigate to our project location .

1.4) Use **wscompile** tool with **-gen:client** option to generate client side stubs.

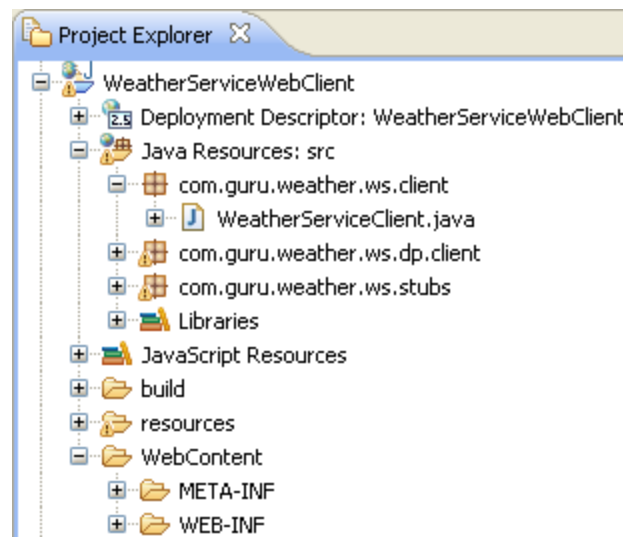
wscompile -gen:client -d src -keep -verbose resources\config.xml

D:\WebServices\WeatherServiceWebClient>wscompile -gen:client -d src -keep -verbose
resources\config.xml

1.5) Now refresh the project, we can see generated stub classes in
com.guru.weather.ws.stubs package.



1.6) create **com.guru.weather.client** and create "**WeatherServiceClient**" class in that package.



```
package com.guru.weather.ws.client;

import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;

import com.guru.weather.ws.stubs.IWeatherInfo;
import com.guru.weather.ws.stubs.WeatherCriteria;
import com.guru.weather.ws.stubs.WeatherService;
import com.guru.weather.ws.stubs.WeatherService_Impl;
import com.guru.weather.ws.stubs.WeatherStatus;

public class WeatherServiceClient {
    public static void main(String args[]) throws ServiceException,
        RemoteException {
        WeatherService weatherService = new WeatherService_Impl();
        IWeatherInfo weatherInfo = (IWeatherInfo) weatherService
            .getIWeatherInfoPort();
        WeatherCriteria weatherCriteria = new WeatherCriteria();
        weatherCriteria.setCity("Hyderabad");
        weatherCriteria.setState("Telangana");
        weatherCriteria.setCountry("Hindusthan");

        WeatherStatus weatherStatus = weatherInfo
            .getWeatherInfo(weatherCriteria);
        System.out.println("Weather Details...");
        System.out.println("Temperature : " + weatherStatus.getTemperature());
        System.out.println("Humidity : " + weatherStatus.getHumidity());
        System.out.println("Description : " + weatherStatus.getDescription());
    }
}

/*Output:
Weather Details...
Temperature : 101
Humidity : 12.5
Description : Today may be rained or may not be
*/
```

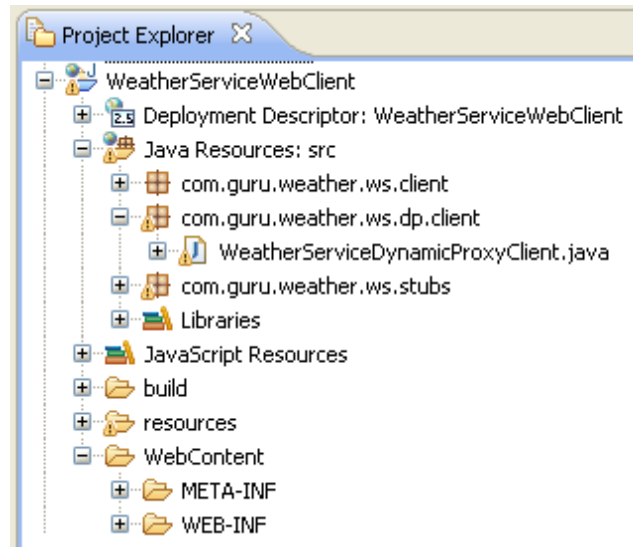
7) Run **WeatherServiceClient** program

Output:

```
Weather Details...
Temperature : 101
Humidity : 12.5
Description : Today may be rained or may not be
```

2) Dynamic Proxy based client:

2.1) In the above project create another package "**com.guru.weather.ws.dp.client**", **WeatherServiceDynamicProxyClient.java**



```
package com.guru.weather.ws.dp.client;
```

```
import java.net.MalformedURLException;  
import java.net.URL;  
import java.rmi.RemoteException;  
import javax.xml.rpc.ServiceException;  
import javax.xml.rpc.ServiceFactory;
```

```
import com.guru.weather.ws.stubs.IWeatherInfo;  
import com.guru.weather.ws.stubs.WeatherCriteria;  
import com.guru.weather.ws.stubs.WeatherService;  
import com.guru.weather.ws.stubs.WeatherService_Impl;  
import com.guru.weather.ws.stubs.WeatherStatus;  
import javax.xml.namespace.QName;  
import javax.xml.rpc.Service;
```

```
public class WeatherServiceDynamicProxyClient {  
    final static String WSDL_URL =  
        "http://localhost:8081/WeatherServiceWeb/ws/IWeatherInfo?WSDL";  
    final static String SERVICE_NM = "WeatherService";  
    final static String PORT_NM="IWeatherInfoPort";  
    final static String NAME_SPACE="http://guru.com/weather/wsdl";  
  
    public static void main(String args[]) throws ServiceException, RemoteException,  
        MalformedURLException {  
        ServiceFactory factory=ServiceFactory.newInstance();
```

```
Service service=factory.createService(new URL(WSDL_URL), new
QName(NAME_SPACE,SERVICE_NM));
IWeatherInfo weatherInfo=(IWeatherInfo)service.getPort(new QName(NAME_SPACE,
PORT_NM), IWeatherInfo.class);
```

```
WeatherCriteria weatherCriteria = new WeatherCriteria();
weatherCriteria.setCity("Hyderabad");
weatherCriteria.setState("Telangana");
weatherCriteria.setCountry("Hindusthan");
```

```
WeatherStatus weatherStatus=weatherInfo.getWeatherInfo(weatherCriteria);
System.out.println("Weather Details...");
System.out.println("Temperature : "+weatherStatus.getTemperature());
System.out.println("Humidity : "+weatherStatus.getHumidity());
System.out.println("Description : "+weatherStatus.getDescription());
```

```
}
}
```

```
/*Output:
Weather Details...
Temperature : 101
Humidity : 12.5
Description : Today may be rained or may not be
```

```
*/
```

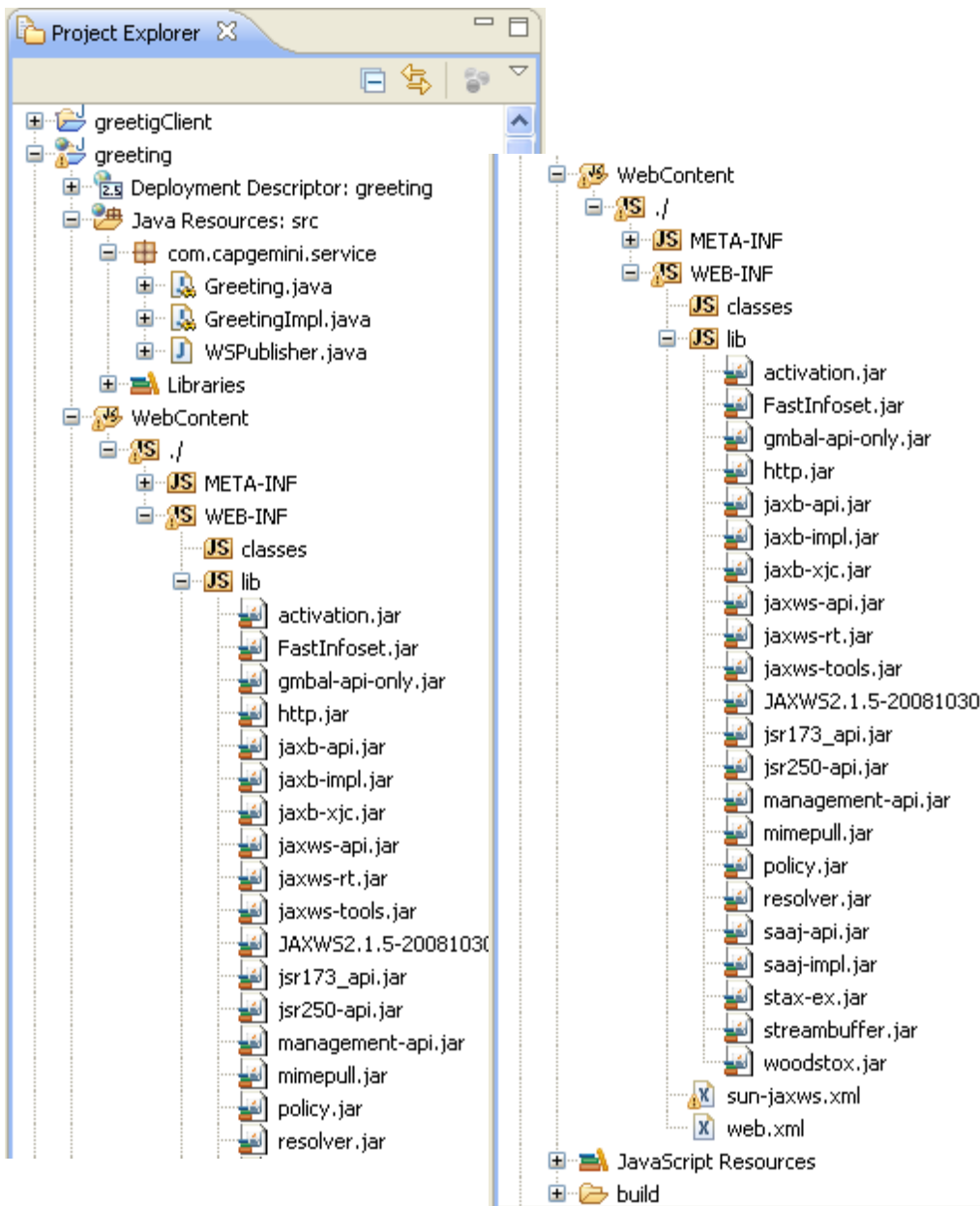
2.2) Run the program

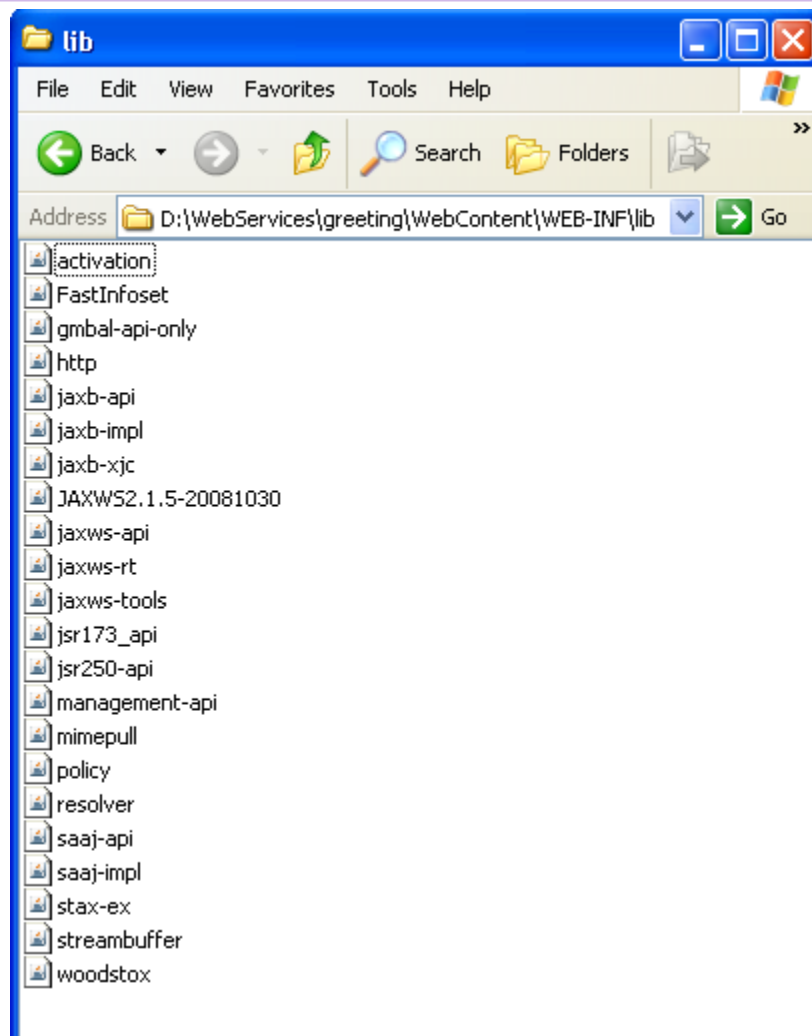
JAX-WS

Web Services – JAX-WS

Greeting

Directory Structure:





1) Greeting.java

```
/**
 * These are assignment topics
 * @Jan 23, 2013 @11:15:03 AM
 */
package com.cap.service;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
@WebService
@SOAPBinding(style = SOAPBinding.Style.DOCUMENT, use=SOAPBinding.Use.LITERAL)
interface Greeting {
    @WebMethod
    String sayHello(String name);
}
```

2) GreetingImpl.java

```
/**
 * These are assignment topics
 * @Jan 23, 2013 @11:15:05 AM
 */
package com.cap.service;

/**
 * @author Venu Kumar for assignments
 * @version 1.0
 * @date Jan 23, 2013
 */

import javax.jws.WebService;
@WebService(endpointInterface = "com.cap.service.Greeting")
public class GreetingImpl implements Greeting {
    public String sayHello(String name) {
        return "Hello, Welcom to jax-ws " + name;
    }
}
```

3) WSPublisher.java

```
/**
 * These are assignment topics
 * @Jan 23, 2013 @11:14:15 AM
 */
package com.cap.service;

/**
 * @author Venu Kumar for assignments
 * @version 1.0
 * @date Jan 23, 2013
 */

import javax.xml.ws.Endpoint;
import com.cap.service.GreetingImpl;
public class WSPublisher {
    public static void main(String[] args) {
        Endpoint.publish("http://localhost:8080/greeting/Greeting",
            new GreetingImpl());
    }
}
```

4) sun-jaxws.xml

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

```

xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
version="2.0">
<endpoint
  name="Greeting"
  implementation="com.cap.service.GreetingImpl"
  url-pattern="/greet"/>
</endpoints>

```

⤴ web.xml

```

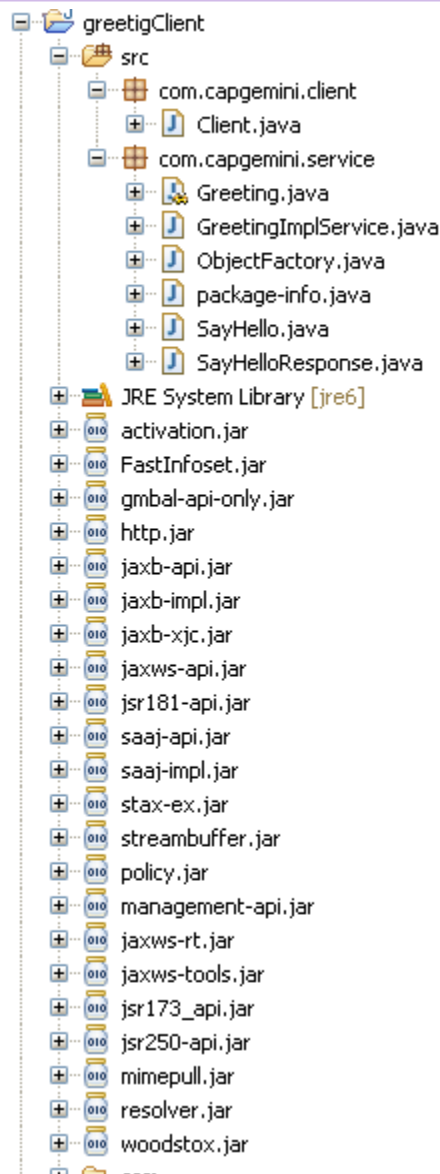
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
  <display-name>greeting</display-name>
  <listener>
    <listener-class>
      com.sun.xml.ws.transport.http.servlet.WSServletContextListener
    </listener-class>
  </listener>
  <servlet>
    <servlet-name>greet</servlet-name>
    <servlet-class>
      com.sun.xml.ws.transport.http.servlet.WSServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>greet</servlet-name>
    <url-pattern>/greet</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>120</session-timeout>
  </session-config>
</web-app>

```

Greetingclient :

To Generate client side Stub files in cmd prompt execute the following:

D:\WebServices\Greetingclient\src
 wsimport -keep -p com.cap.Service <http://localhost:8081/WSHandlerServe/myService?wsdl>



Write Client.java

```
/**
 * These are assignment topics
 * @Jun 3, 2011 @12:03:04 PM
 */
package com.cap.client;

import com.cap.service.Greeting;
import com.cap.service.GreetingImplService;

/**
 * @author Venu Kumar for assignments
 * @version 1.0
```

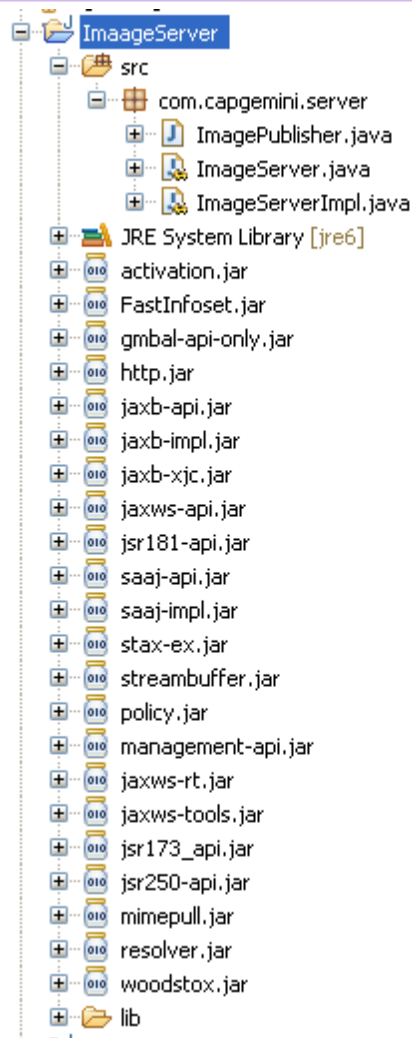
```
* @date Jan 23, 2013
*/
```

```
public class Client {
    public static void main(String[] args) {
        GreetingImplService service = new GreetingImplService();
        Greeting greet = service.getGreetingImplPort();
        String mymsg=greet.sayHello("Kumar!!!!!!!!!!");
        System.out.println("Messsssssage : "+mymsg);
        System.out.println("----->> Call Started");
        System.out.println(greet.sayHello("Kumar!!!!!!!!!!"));
        System.out.println("----->> Call Ended");
    }
}
```

Then Execute the Client.java

JAX-WS

2)ImageServer:



```
/**
 * These are assignment topics
 * @Jan 23, 2013 @11:31:26 AM
 */
package com.cap.server;

import javax.xml.ws.Endpoint;
import com.cap.server.ImageServerImpl;

//Endpoint publisher
public class ImagePublisher{

    public static void main(String[] args) {

        Endpoint.publish("http://localhost:1234/ImageServer/myimage", new ImageServerImpl());

        System.out.println("Server is published!");
    }
}
```

```
}  
  
}  
  
/**  
 * These are assignment topics  
 * @Jan 23, 2013 @11:31:10 AM  
 */  
package com.cap.server;  
  
import java.awt.Image;  
  
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 ImageServer{  
  
    //download a image from server  
    @WebMethod Image downloadImage(String name);  
  
    //update image to server  
    @WebMethod String uploadImage(Image data);  
  
}  
  
  
/**  
 * These are assignment topics  
 * @Jan 23, 2013 @11:31:43 AM  
 */  
package com.cap.server;  
  
import java.awt.Image;  
import java.io.File;  
import java.io.IOException;  
  
import javax.imageio.ImageIO;  
import javax.jws.WebService;  
import javax.xml.ws.WebServiceException;  
import javax.xml.ws.soap.MTOM;  
  
//Service Implementation Bean  
@MTOM
```



```
@WebService(endpointInterface = "com.cap.server.ImageServer")
public class ImageServerImpl implements ImageServer{

    public Image downloadImage(String name) {

        try {

            File image = new File("D:\\naren\\images\\" + name);
            return ImageIO.read(image);

        } catch (IOException e) {

            e.printStackTrace();
            return null;

        }

    }

    public String uploadImage(Image data) {

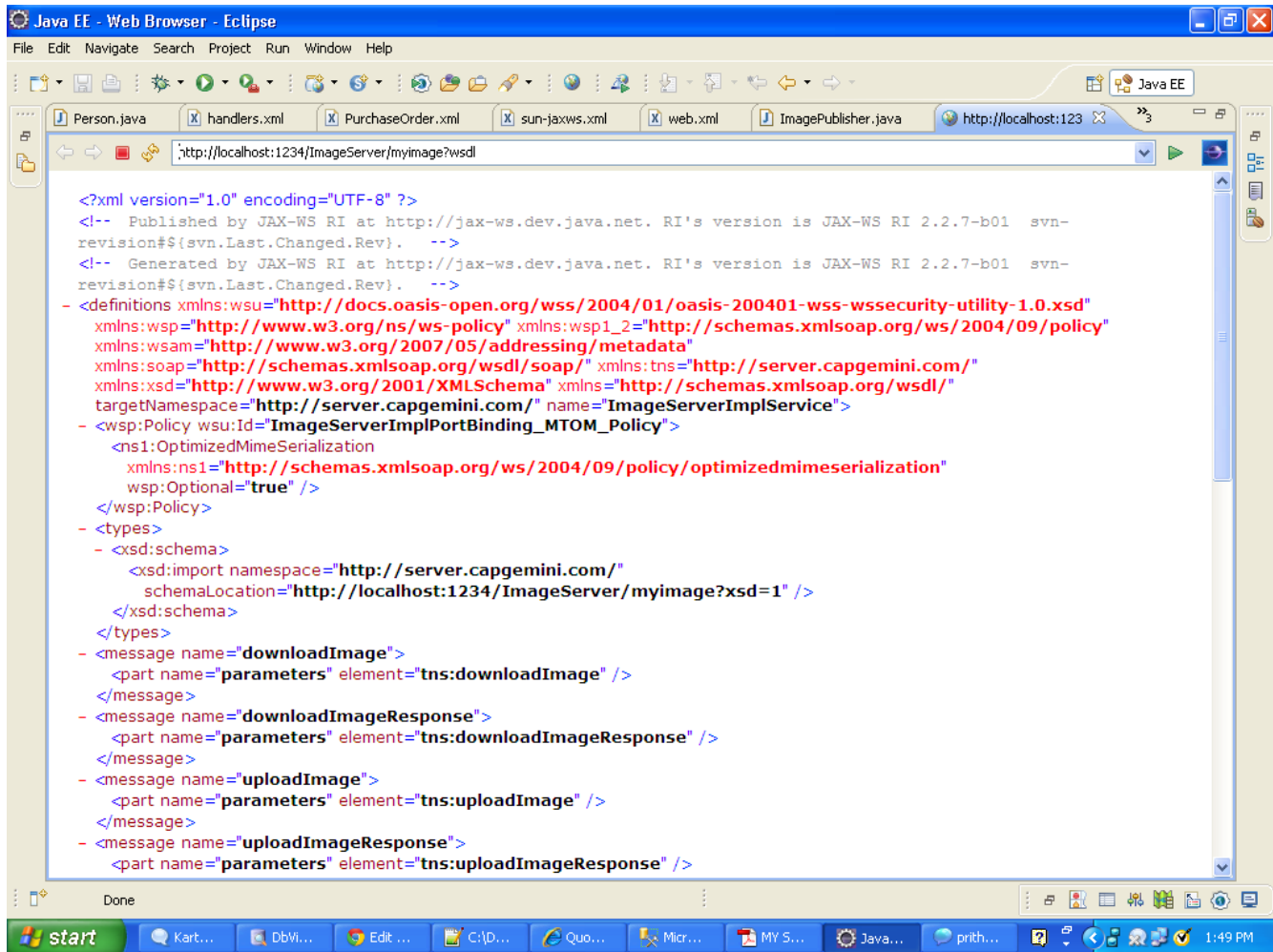
        if(data!=null){
            //store somewhere
            return "Upload Successful";
        }

        throw new WebServiceException("Upload Failed!");

    }

}
```

ImageClient:



D:\WebServices\ImageServer\src
wsimport -keep -p com.cap.server <http://localhost:8081/ImageServer/myimage?wsdl>

It Creates 6 Client side stub files and classes.

ImageClient.java

```
/**
 * These are assignment topics
 * @Jun 7, 2011 @10:38:41 AM
 */
package com.cap.client;

import java.awt.Image;
import java.io.File;
import java.net.URL;
import javax.imageio.ImageIO;
```

```
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.Service;
import javax.xml.ws.soap.MTOMFeature;
import javax.xml.ws.soap.SOAPBinding;

import com.cap.server.ImageServer;

public class ImageClient{

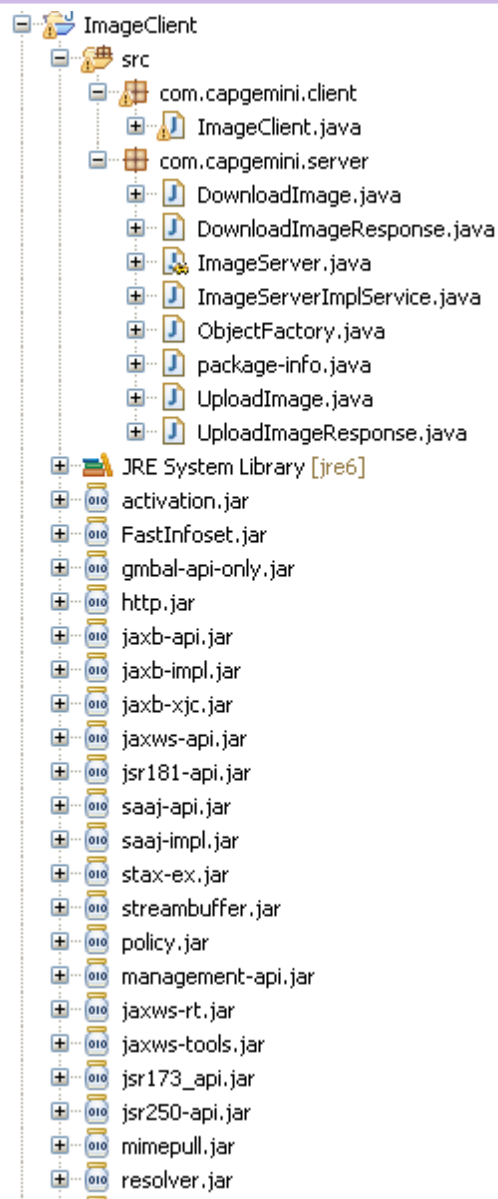
    public static void main(String[] args) throws Exception {

        URL url = new URL("http://localhost:1234/ImageServer/myimage?wsdl");
        QName qname = new QName("http://server.cap.com/", "ImageServerImplService");

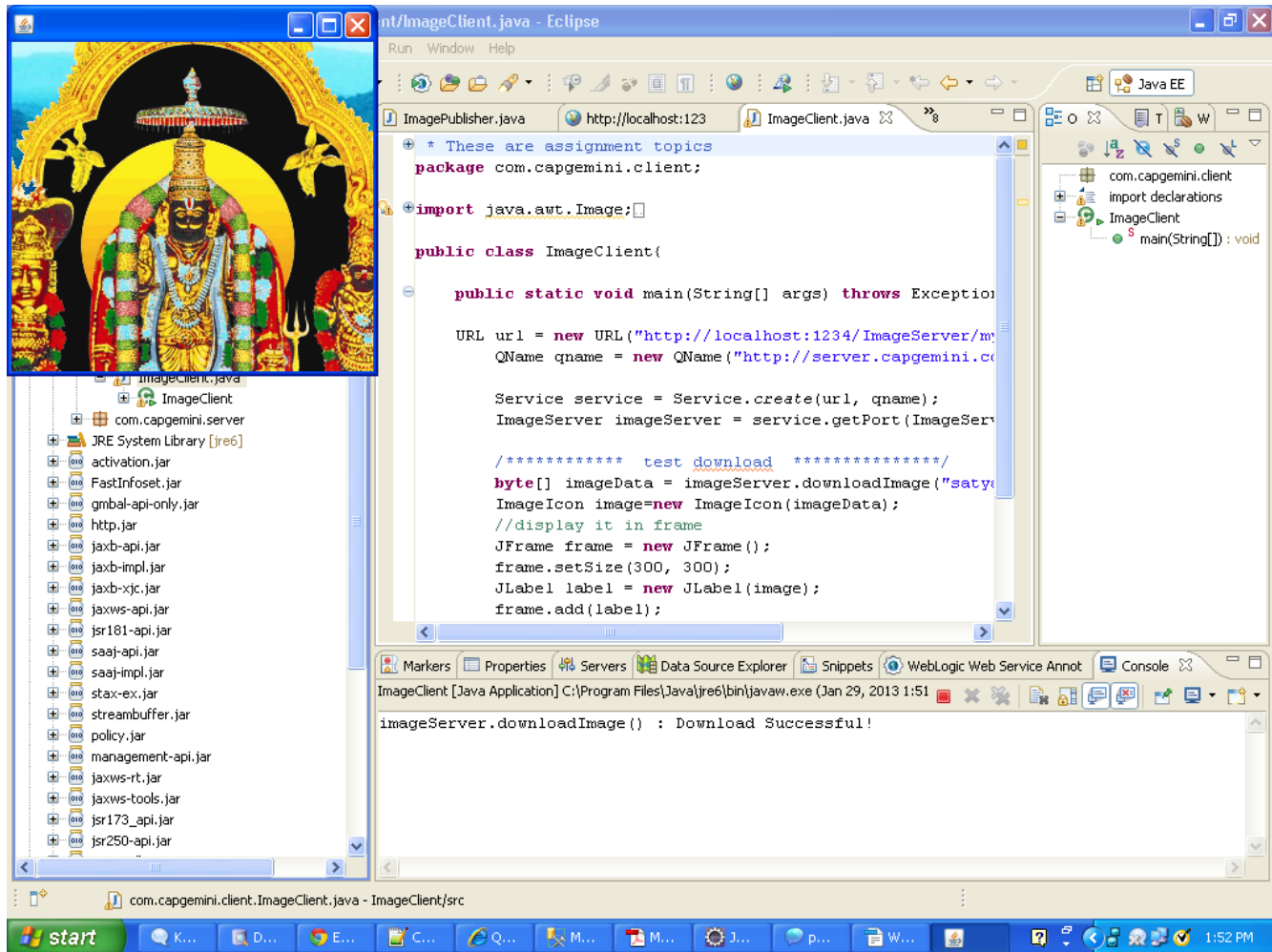
        Service service = Service.create(url, qname);
        ImageServer imageServer = service.getPort(ImageServer.class);

        /***** test download *****/
        byte[] imageData = imageServer.downloadImage("satyadev.png");
        ImageIcon image=new ImageIcon(imageData);
        //display it in frame
        JFrame frame = new JFrame();
        frame.setSize(300, 300);
        JLabel label = new JLabel(image);
        frame.add(label);
        frame.setVisible(true);

        System.out.println("imageServer.downloadImage() : Download Successful!");
    }
}
```

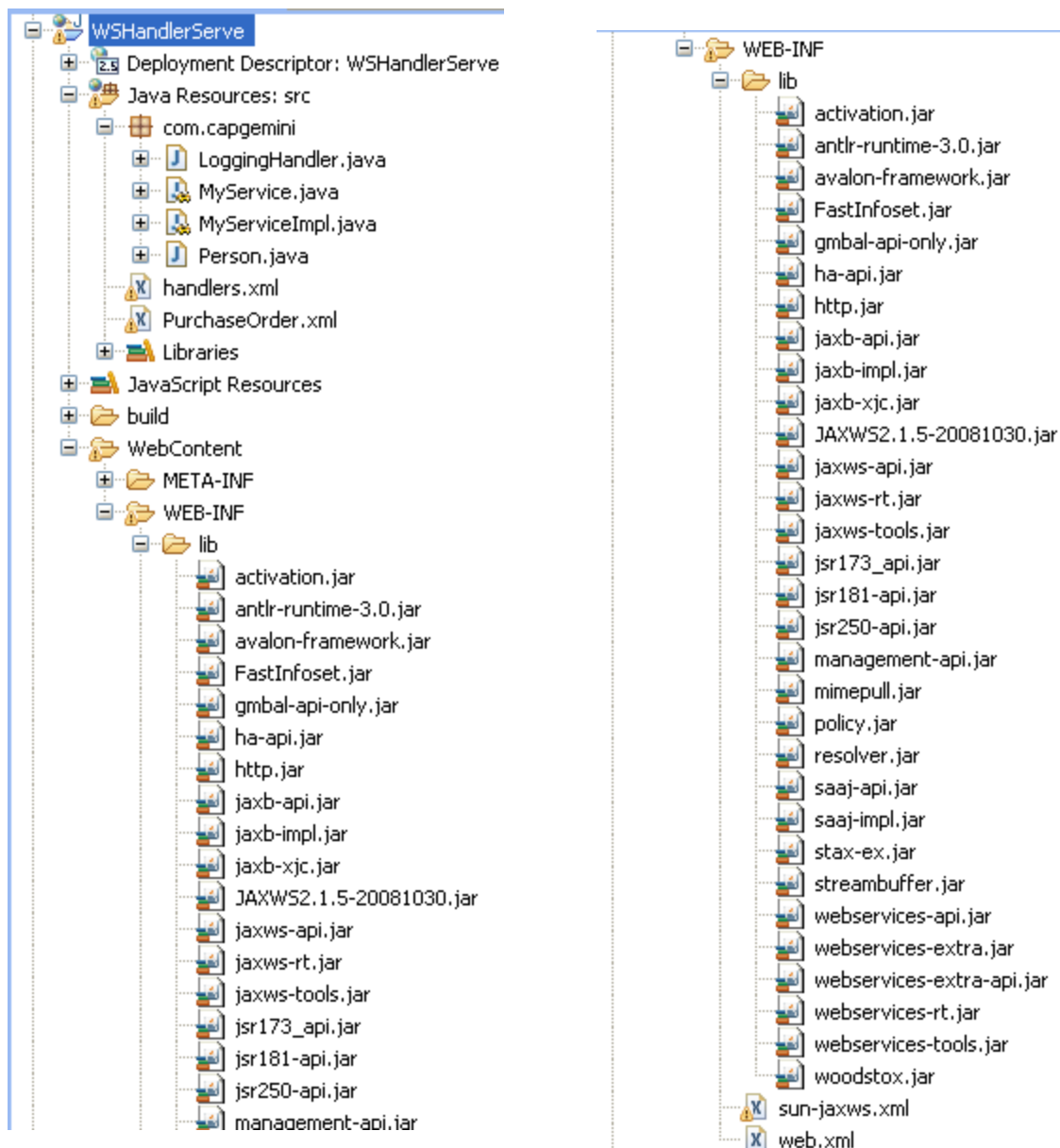


Execute this ImageClient.java file.



JAX-WS

WSHandlerServer:



Directory Structure:

LoggingHandler.java

package com.cap;

import java.util.Set;

import javax.xml.namespace.QName;

import javax.xml.soap.SOAPMessage;

import javax.xml.ws.handler.MessageContext;

```

import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;

public class LoggingHandler implements SOAPHandler<SOAPMessageContext> {

    public Set<QName> getHeaders() {
        return null;
    }

    public void close(MessageContext context) {
    }

    public boolean handleFault(SOAPMessageContext context) {
        logToSystemOut(context);
        return true;
    }

    public boolean handleMessage(SOAPMessageContext context) {
        logToSystemOut(context);
        return true;
    }

    private void logToSystemOut(SOAPMessageContext smc) {
        Boolean outboundProperty = (Boolean)
smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);

        if (outboundProperty.booleanValue()) {
            System.out.println("\nOutgoing message:");
        } else {
            System.out.println("\nIncoming message:");
        }

        SOAPMessage message = smc.getMessage();
        try {
            message.writeTo(System.out);
            System.out.println("=== DONE ===");
        } catch (Exception e) {
            System.out.println("Exception in handler: " + e);
        }
    }
}
MyService.java
package com.cap;

import javax.jws.WebParam;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

@WebService
@SOAPBinding(style = SOAPBinding.Style.DOCUMENT, use = SOAPBinding.Use.LITERAL)

```



```
public interface MyService {

    String sayHello(@WebParam(name = "name") String name);

    Person getPerson(@WebParam(name = "id") long id);

    String xmlData(@WebParam(name = "data") String data);

    /** Returns a purchase order in xml format. */
    String getPurchaseOrder();
}
MyServiceImpl.java
package com.cap;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.sql.Date;

import javax.jws.HandlerChain;
import javax.jws.WebService;

@WebService(endpointInterface = "com.cap.MyService")
@HandlerChain(file = "handlers.xml")
public class MyServiceImpl implements MyService {

    public String sayHello(String name) {
        if (name == null || name.trim().length() <= 0)
            throw new IllegalArgumentException("name");

        System.out.println("Going to say Hellow to " + name);
        return "Hello " + name + "!";
    }

    @SuppressWarnings("deprecation")
    public Person getPerson(long id) {
        Person person = new Person();
        person.setId(id);
        person.setName("Vinod Singh");
        person.setDateOfBirth(new Date(1978, 4, 9));

        System.out.println("Serving getPerson(" + id + "): " + person);

        return person;
    }

    public String xmlData(String data) {
        System.out.println("Received XML data from client: " + data);
        return "<greeting>" + "Hello! " + data.substring(data.indexOf(">") + 1,
data.lastIndexOf("<")) + "</greeting>";
    }
}
```

```
public String getPurchaseOrder() {
    StringBuilder data = new StringBuilder(2048);
    try {
        BufferedReader fReader = new BufferedReader(new
InputStreamReader(MyServiceImpl.class
.getResourceAsStream("/PurchaseOrder.xml")));
        String line;
        do {
            line = fReader.readLine();
            if (line != null)
                data.append(line).append("\n");
        } while (line != null);
    } catch (Exception e) {
        e.printStackTrace();
        data.append(e.getMessage());
    }
    return data.toString();
}
```

Person.java

```
package com.cap;
```

```
import java.util.Date;
```

```
public class Person {

    private long id;

    private String name;

    private Date dateOfBirth;

    public Person() {

    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```

    public Date getDateOfBirth() {
        return dateOfBirth;
    }

    public void setDateOfBirth(Date dateOfBirth) {
        this.dateOfBirth = dateOfBirth;
    }
}

```

handlers.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
    <handler-chain>
        <handler>
            <handler-class>com.cap.LoggingHandler</handler-class>
        </handler>
    </handler-chain>
</handler-chains>

```

purchaseOrder.xml

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<PurchaseOrder>
    <Company>
        <Name>ACME Company</Name>
        <Address1>ADDRESS LINE 1</Address1>
        <Address2 />
        <City>Ghaziabad</City>
        <State>UP</State>
        <PostCode>201010</PostCode>
        <Phone>0120 99999999</Phone>
        <Fax>0120 99999999</Fax>
        <SalesTaxCertificate>E9999999</SalesTaxCertificate>
    </Company>
    <Supplier>
        <SupplierCode>9999</SupplierCode>
        <Name>SUPPLIER XYZ</Name>
        <Address1>ADDRESS LINE 1</Address1>
        <Address2 />
        <City>Ghaziabad</City>
        <State>UP</State>
        <PostCode>201010</PostCode>
        <Contact>J BOCKS</Contact>
    </Supplier>
    <OrderHeader>
        <OrderType>ST</OrderType>
        <OrderNumber>00099999</OrderNumber>
        <OrderDate>2009-06-26T00:00:00</OrderDate>
        <ReceivingSite>01</ReceivingSite>
        <ReceivingWarehouse>01</ReceivingWarehouse>
        <Cancelled>Y</Cancelled>
        <Reprint>N</Reprint>
    </OrderHeader>
</PurchaseOrder>

```

```

<BuyerName>Buyer Name</BuyerName>
<BuyerPhone>0120 99999999</BuyerPhone>
<Currency>INR</Currency>
<PrintDate>2009-06-26T14:00:42</PrintDate>
<DeliveryInstructionCode>9</DeliveryInstructionCode>
<DeliveryInstructions>NOTE: On completion, Please fax a copy
  of the invoice (Include Order No) to
</DeliveryInstructions>
<DeliveryLocationCode>4</DeliveryLocationCode>
<DeliveryLocation>OCS CENTRAL WAREHOUSE ADDRESS
</DeliveryLocation>
<SpecialInstructionCode />
<SpecialInstructions />
<OrderTerms>NETT FROM STATEMENT 30 DAYS</OrderTerms>
<Revision>0</Revision>
<Remarks />
</OrderHeader>
<OrderLines>
  <Line>
    <LineNumber>0001</LineNumber>
    <Action />
    <DestinationSite>01</DestinationSite>
    <DestinationWarehouse>01</DestinationWarehouse>
    <DueDate>2009-06-30T00:00:00</DueDate>
    <Agreement />
    <Quantity>1.0000</Quantity>
    <UnitPurchase>EA</UnitPurchase>
    <Requisition />
    <Status>Active</Status>
    <Priority>U</Priority>
    <UnitPrice>21.0000</UnitPrice>
    <LineValue>21.00</LineValue>
    <GSTValue>2.10</GSTValue>
    <DiscountValue>0.00</DiscountValue>
    <SalesTaxDescription>TAX EXEMPT</SalesTaxDescription>
    <SalesTaxValue>0.00</SalesTaxValue>
    <GSTExempt>N</GSTExempt>
    <FreightValue>0.00</FreightValue>
    <OnCostValue>0.00</OnCostValue>
    <StockNumber>00004004</StockNumber>
    <SubStock>01</SubStock>
    <ItemDescription>FLANGE PIPE BLACK 250MM NB BST D 10IN
      ABC</ItemDescription>
    <PatternedDescription />
    <UnpatternedDescription />
    <Manufacturer>BLKWOODS</Manufacturer>
    <PartNumber>03230505</PartNumber>
    <TBACode />
    <TBADetails />
    <LineRemarks />
    <PackingInstructions />
  </Line>
</OrderLines>

```

```

</Line>
<Line>
  <LineNumber>0002</LineNumber>
  <Action />
  <DestinationSite>01</DestinationSite>
  <DestinationWarehouse>01</DestinationWarehouse>
  <DueDate>2009-06-30T00:00:00</DueDate>
  <Agreement />
  <Quantity>50.0000</Quantity>
  <UnitPurchase>ROLL</UnitPurchase>
  <Requisition />
  <Status>Active</Status>
  <Priority>U</Priority>
  <UnitPrice>0.3700</UnitPrice>
  <LineValue>18.50</LineValue>
  <GSTValue>1.85</GSTValue>
  <DiscountValue>0.00</DiscountValue>
  <SalesTaxDescription>TAX EXEMPT</SalesTaxDescription>
  <SalesTaxValue>0.00</SalesTaxValue>
  <GSTExempt>N</GSTExempt>
  <FreightValue>0.00</FreightValue>
  <OnCostValue>0.00</OnCostValue>
  <StockNumber>00006219</StockNumber>
  <SubStock>01</SubStock>
  <ItemDescription>TAPE THREAD SEALING 12MM X 10M TEFLON
    PTFE</ItemDescription>
  <PatternedDescription />
  <UnpatternedDescription />
  <Manufacturer>BLKWOODS</Manufacturer>
  <PartNumber>05122404</PartNumber>
  <TBACode />
  <TBADetails />
  <LineRemarks />
  <PackingInstructions />
</Line>
</OrderLines>
<OrderTotals>
  <FreightTotal>0.00</FreightTotal>
  <OnCostTotal>0.00</OnCostTotal>
  <OrderTotal>43.45</OrderTotal>
</OrderTotals>
</PurchaseOrder>

```

sun-jaxws.xml

```

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

<endpoints
  xmlns='http://java.sun.com/xml/ns/jax-ws/ri/runtime'
  version='2.0'>
  <endpoint
    name='myService'

```

```

        implementation='com.cap.MyServiceImpl'
        url-pattern='/myService' />
</endpoints>

```

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/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
  <display-name>WSHandlerServer</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
  <listener>
    <listener-class>
      com.sun.xml.ws.transport.http.servlet.WSServletContextListener
    </listener-class>
  </listener>
  <servlet>
    <description>JAX-WS endpoint</description>
    <display-name>The JAX-WS servlet</display-name>
    <servlet-name>jaxws</servlet-name>
    <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>jaxws</servlet-name>
    <url-pattern>/myService</url-pattern>
  </servlet-mapping>
</web-app>

```

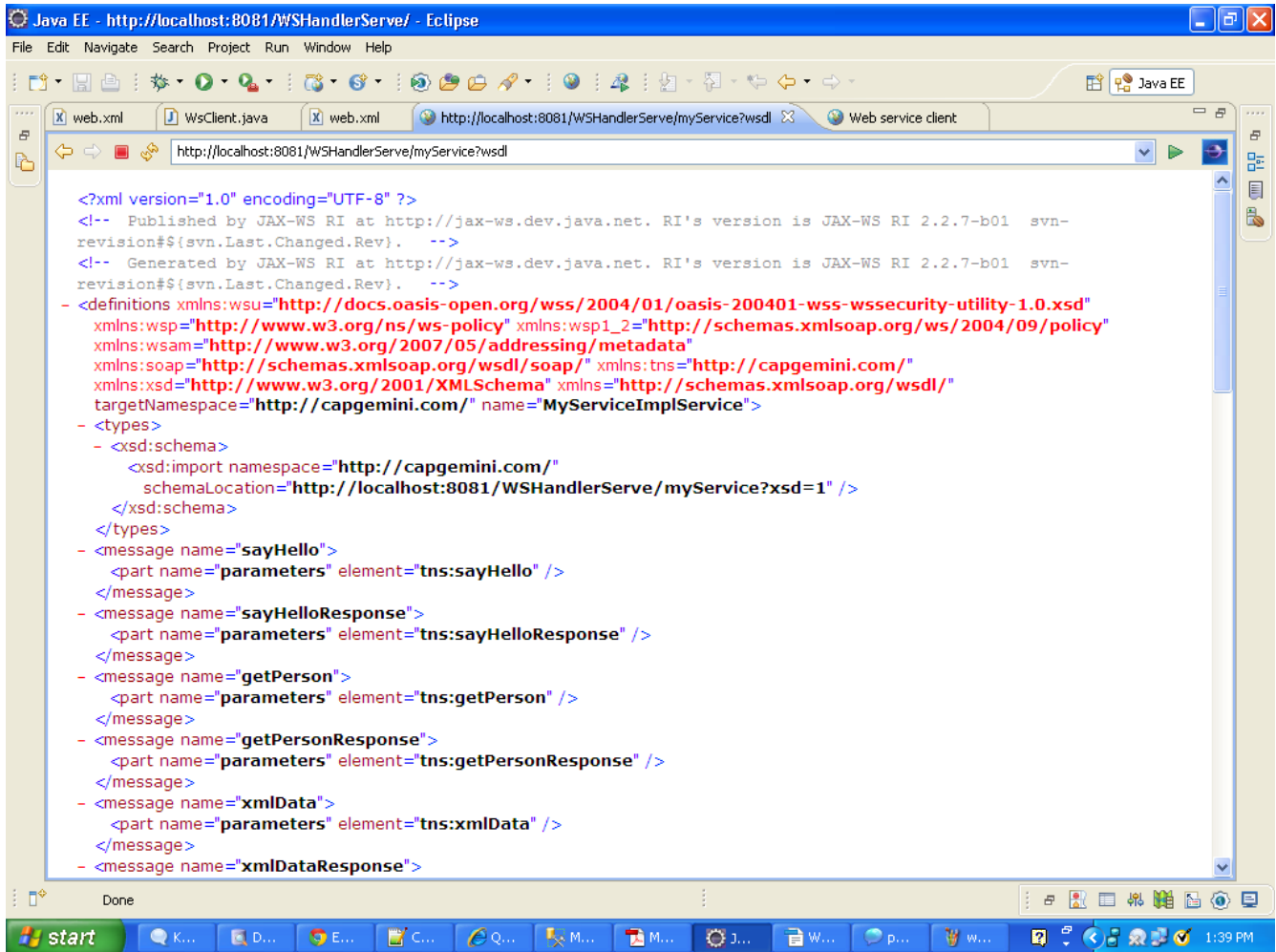
WSHandlerClient:

Run the following through cmd prompt to generate client side stubs

```

D:\WebServices\WSHandlerClient\src wsimport -keep -p com.cap
http://localhost:8081/WSHandlerServe/myService?wsdl

```



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

- <!--

Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.2.7-b01 svn-revision#\$(svn.Last.Changed.Rev).

-->

- <!--

Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.2.7-b01 svn-revision#\$(svn.Last.Changed.Rev).

-->

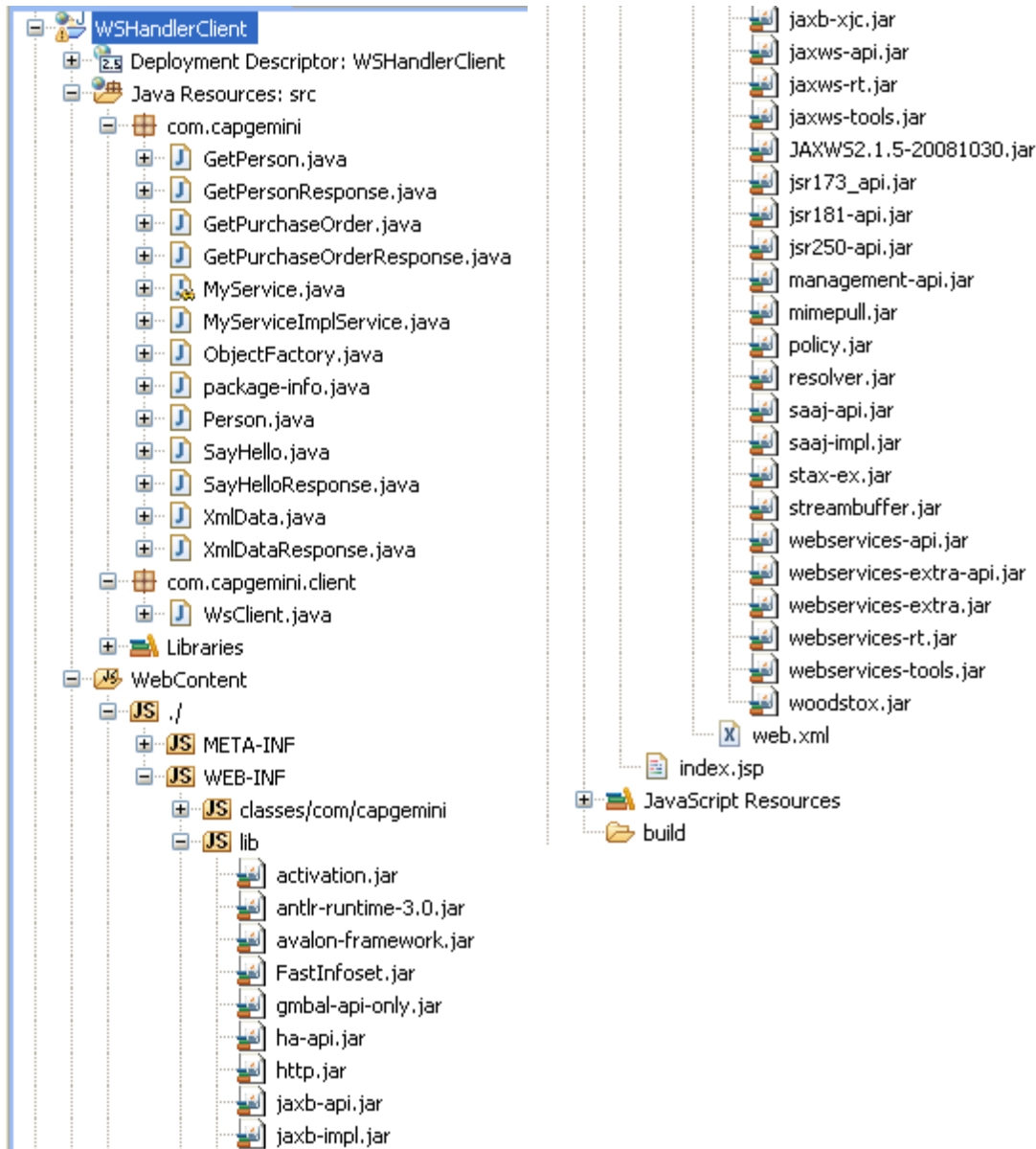
```
<definitions xmlns:wsu="http://docs.oasis-  
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"  
xmlns:wsp="http://www.w3.org/ns/ws-policy"  
xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"  
xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"  
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"  
xmlns:tns="http://cap.com/"  
xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
xmlns="http://schemas.xmlsoap.org/wsdl/"  
targetNamespace="http://cap.com/" name="MyServiceImplService">  
  <types>  
    <xsd:schema>  
      <xsd:import namespace="http://cap.com/"  
schemaLocation="http://localhost:8081/WSHandlerServe/myService?xsd  
=1" />  
    </xsd:schema>  
  </types>  
  <message name="sayHello">  
    <part name="parameters" element="tns:sayHello" />  
  </message>  
  <message name="sayHelloResponse">  
    <part name="parameters" element="tns:sayHelloResponse" />  
  </message>  
  <message name="getPerson">  
    <part name="parameters" element="tns:getPerson" />  
  </message>  
  <message name="getPersonResponse">  
    <part name="parameters" element="tns:getPersonResponse" />  
  </message>  
  <message name="xmlData">  
    <part name="parameters" element="tns:xmlData" />  
  </message>  
  <message name="xmlDataResponse">  
    <part name="parameters" element="tns:xmlDataResponse" />  
  </message>  
  <message name="getPurchaseOrder">  
    <part name="parameters" element="tns:getPurchaseOrder" />  
  </message>  
  <message name="getPurchaseOrderResponse">  
    <part name="parameters" element="tns:getPurchaseOrderResponse" />  
  </message>  
  <portType name="MyService">  
    <operation name="sayHello">
```



```
<input wsam:Action="http://cap.com/MyService/sayHelloRequest"
message="tns:sayHello" />
<output wsam:Action="http://cap.com/MyService/sayHelloResponse"
message="tns:sayHelloResponse" />
</operation>
_ <operation name="getPerson">
  <input wsam:Action="http://cap.com/MyService/getPersonRequest"
message="tns:getPerson" />
  <output wsam:Action="http://cap.com/MyService/getPersonResponse"
message="tns:getPersonResponse" />
  </operation>
_ <operation name="xmlData">
  <input wsam:Action="http://cap.com/MyService/xmlDataRequest"
message="tns:xmlData" />
  <output wsam:Action="http://cap.com/MyService/xmlDataResponse"
message="tns:xmlDataResponse" />
  </operation>
_ <operation name="getPurchaseOrder">
  <input
wsam:Action="http://cap.com/MyService/getPurchaseOrderRequest"
message="tns:getPurchaseOrder" />
  <output
wsam:Action="http://cap.com/MyService/getPurchaseOrderResponse"
message="tns:getPurchaseOrderResponse" />
  </operation>
  </portType>
_ <binding name="MyServiceImplPortBinding" type="tns:MyService">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
style="document" />
_ <operation name="sayHello">
  <soap:operation soapAction="" />
_ <input>
  <soap:body use="literal" />
  </input>
_ <output>
  <soap:body use="literal" />
  </output>
  </operation>
_ <operation name="getPerson">
  <soap:operation soapAction="" />
_ <input>
  <soap:body use="literal" />
  </input>
```

```
- <output>
  <soap:body use="literal" />
</output>
</operation>
- <operation name="xmlData">
  <soap:operation soapAction="" />
- <input>
  <soap:body use="literal" />
</input>
- <output>
  <soap:body use="literal" />
</output>
</operation>
- <operation name="getPurchaseOrder">
  <soap:operation soapAction="" />
- <input>
  <soap:body use="literal" />
</input>
- <output>
  <soap:body use="literal" />
</output>
</operation>
</binding>
- <service name="MyServiceImplService">
- <port name="MyServiceImplPort"
binding="tns:MyServiceImplPortBinding">
  <soap:address
location="http://localhost:8081/WSHandlerServe/myService" />
  </port>
</service>
</definitions>
```

Directory Structure:



WsClient.java

```

package com.cap.client;

import java.io.IOException;
import java.net.URL;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.namespace.QName;
import javax.xml.ws.WebServiceClient;

```

```

import com.cap.MyService;
import com.cap.MyServiceImplService;
import com.cap.Person;

/**
 * Servlet implementation class WsClient
 */
public class WsClient extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public WsClient() {
        super();
    }

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        String name = request.getParameter("name");
        String id = request.getParameter("id");

        try {
            // Get a handle to web service client interface
            WebServiceClient ann =
MyServiceImplService.class.getAnnotation(WebServiceClient.class);
            MyServiceImplService service = new MyServiceImplService(new
URL("http://localhost:8081/WSHandlerServe/myService"),
                new QName(ann.targetNamespace(), ann.name()));
            MyService myService = service.getMyServiceImplPort();

            // Invoke methods
            if (name != null) {
                String hello = myService.sayHello(name);
                request.setAttribute("nameResult", hello);
            }

            if (id != null) {
                Person person = myService.getPerson(Long.parseLong(id));
                String idResult = "Person [id: " + person.getId() + ", name: " + person.getName() +
", Date of Birth: "
                    + person.getDateOfBirth() + "];";
                request.setAttribute("idResult", idResult);
            }
        } catch (Exception e) {
            if (name != null) {
                request.setAttribute("nameResult", e.getMessage());
            }
        }
    }
}

```

```

    }

    if (id != null) {
        request.setAttribute("idResult", e.getMessage());
    }
    e.printStackTrace();
}

RequestDispatcher dispatcher = getServletContext().getRequestDispatcher("/index.jsp");
dispatcher.forward(request, response);
}

public static void main(String[] args) {
    String name = "Vinod Singh";
    String id = "0031";
    try {
        // Get a handle to web service client interface
        WebServiceClient ann =
MyServiceImplService.class.getAnnotation(WebServiceClient.class);
        MyServiceImplService service = new MyServiceImplService(new
URL("http://localhost:8081/WSHandlerExample/myService"),
        new QName(ann.targetNamespace(), ann.name()));
        MyService myService = service.getMyServiceImplPort();

        // Invoke methods
        if (name != null) {
            String hello = myService.sayHello(name);
            System.out.println("nameResult" + hello);
        }

        if (id != null) {
            Person person = myService.getPerson(Long.parseLong(id));
            String idResult = "Person [id: " + person.getId() + ", name: " + person.getName() +
", Date of Birth: "
            + person.getDateOfBirth() + "];";
            System.out.println("idResult " + idResult);
        }

        // now try to send XML String
        String xml = "<name>vinod</name>";
        String returnXml = myService.xmlData(xml);
        System.out.println(returnXml);

        // Now receive a big XML from service
        String po = myService.getPurchaseOrder();
        System.out.println("PURCHASE ORDER\n" + po);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
}

```

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/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
  <display-name>WSHandlerClient</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>

  <servlet>
    <description></description>
    <display-name>WsClient</display-name>
    <servlet-name>WsClient</servlet-name>
    <servlet-class>com.cap.client.WsClient</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>WsClient</servlet-name>
    <url-pattern>/WsClient</url-pattern>
  </servlet-mapping>
</web-app>
```

index.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Web service client</title>
</head>
<body>
  <%
    String nameResult = (String) request.getAttribute("nameResult");
    nameResult = nameResult == null ? "" : nameResult;
    String idResult = (String) request.getAttribute("idResult");
    idResult = idResult == null ? "" : idResult;
  %>
  <h1>Say Hello</h1>
  <form action="/WSHandlerClient/WsClient">
    Name: <input type="text" name="name"> <input type="submit" title="Submit">
  </form>
  Result: <%=nameResult %>
  <hr />
```

```
<h1>Get Person</h1>
<form action="/WSHandlerClient/WsClient">
  Id: <input type="text" name="id"> <input type="submit" title="Submit">
</form>
Result: <%=idResult %>
</body>
</html>
```

run the client side **web.xml** file

output:

web.xml http://localhost:808 Web service client

http://localhost:8081/WSHandlerClient/

Say Hello

Name:

Result:

Get Person

Id:

Result:

when we give id as 0031, and select submitQuery, it shows the following in the console:

Incoming message:

```
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:wsa="http://www.w3.org/2005/08/addressing"><soap-
env:Header><wsa:Action>http://schemas.xmlsoap.org/ws/2004/09/transfer/Get</wsa:Action><
wsa:To>http://localhost:8081/WSHandlerServe/myService</wsa:To><wsa:ReplyTo><wsa:Addre
ss>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address></wsa:ReplyTo><wsa:M
```

```
essageID>uuid:778b135f-3fdf-44b2-b53e-ebaab7441e40</wsa:MessageID></soap-  
env:Header><soap-env:Body/></soap-env:Envelope>=== DONE ===
```

Outgoing message:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Body><ns2:Fault  
xmlns:ns2="http://schemas.xmlsoap.org/soap/envelope/"  
xmlns:ns3="http://www.w3.org/2003/05/soap-  
envelope"><faultcode>ns2:Client</faultcode><faultstring>Cannot find dispatch method for  
{}</faultstring></ns2:Fault></S:Body></S:Envelope>=== DONE ===
```

Incoming message:

```
<S:Envelope  
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Header/><S:Body><ns2:getPerson  
xmlns:ns2="http://cap.com/"><id>31</id></ns2:getPerson></S:Body></S:Envelope>===  
DONE ===  
Serving getPerson(31): com.cap.Person@18fee4f
```

Outgoing message:

```
<S:Envelope  
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Body><ns2:getPersonResponse  
xmlns:ns2="http://cap.com/"><return><dateOfBirth>3878-05-  
09T00:00:00+05:30</dateOfBirth><id>31</id><name>Vinod  
Singh</name></return></ns2:getPersonResponse></S:Body></S:Envelope>=== DONE ===
```


The screenshot shows a web browser window at the URL `http://localhost:8081/WSHandlerClient/WSClient?id=0031`. The page has two sections: "Say Hello" and "Get Person".

Say Hello

Name:

Result: _____

Get Person

Id:

Result: Person [id: 31, name: Vinod Singh, Date of Birth: 3878-05-09T00:00:00+05:30]

The bottom part of the image shows the IDE's console window for Tomcat v6.0 Server at localhost. It displays the following messages:

```
Incoming message:
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Header/><S:Body><ns2:getPerson xmlns:r
Serving getPerson(31): com.caggemini.Person@1890c67

Outgoing message:
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Body><ns2:getPersonResponse xmlns:ns2=
```

when we give Name as Vinod Singh, and select submitQuery, it shows the following in the console:



The screenshot shows a web browser window with the URL `http://localhost:8081/WSHandlerClient/WSClient?name=Vinod+Singh`. The browser displays two sections: "Say Hello" and "Get Person".

Say Hello Section:

Name: Submit Query

Result: Hello Vinod Singh!

Get Person Section:

Id: Submit Query

Result:

Below the browser window is a console window showing SOAP messages. The console title is "Tomcat v6.0 Server at localhost [Apache Tomcat] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 29, 2013 1:27:25 PM)".

Incoming message:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Header/><S:Body><ns2:sayHello xmlns:ns2="http://schemas.xmlsoap.org/soap/envelope/"><name>Vinod Singh</name></ns2:sayHello></S:Body></S:Envelope>
```

Going to say Hellow to Vinod Singh

Outgoing message:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Body><ns2:sayHelloResponse xmlns:ns2="http://schemas.xmlsoap.org/soap/envelope/"><name>Vinod Singh</name></ns2:sayHelloResponse></S:Body></S:Envelope>
```

Incoming message:

```
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:wsa="http://www.w3.org/2005/08/addressing"><soap-
env:Header><wsa:Action>http://schemas.xmlsoap.org/ws/2004/09/transfer/Get</wsa:Action><
wsa:To>http://localhost:8081/WSHandlerServe/myService</wsa:To><wsa:ReplyTo><wsa:Addre
ss>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address></wsa:ReplyTo><wsa:M
essageID>uuid:778b135f-3fdf-44b2-b53e-ebaab7441e40</wsa:MessageID></wsa:Header><soap-
env:Body/></soap-env:Envelope>=== DONE ===
```

Outgoing message:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Body><ns2:Fault
xmlns:ns2="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ns3="http://www.w3.org/2003/05/soap-
envelope"><faultcode>ns2:Client</faultcode><faultstring>Cannot find dispatch method for
{</faultstring></ns2:Fault></S:Body></S:Envelope>=== DONE ===
```

Incoming message:

```
<S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Header/><S:Body><ns2:sayHello
xmlns:ns2="http://cap.com/"><name>Vinod
Singh</name></ns2:sayHello></S:Body></S:Envelope>=== DONE ===
Going to say Hellow to Vinod Singh
```

Outgoing message:

<S:Envelope

xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Body><ns2:sayHelloResponse

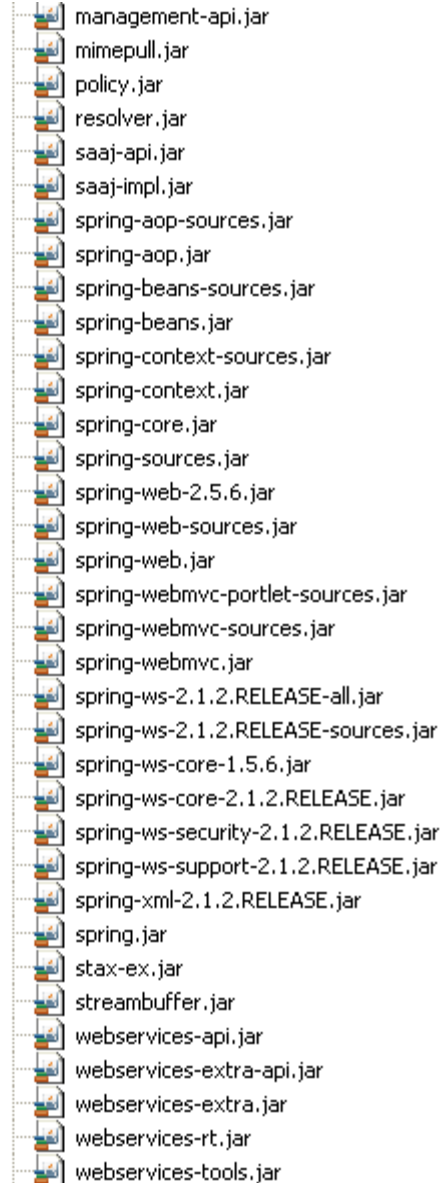
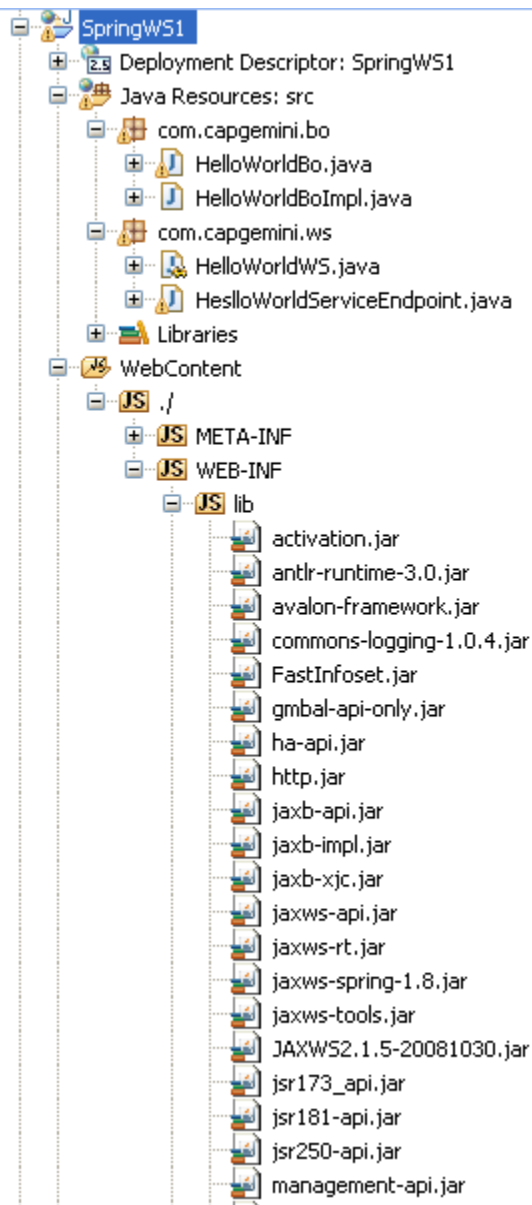
xmlns:ns2="http://cap.com/"><return>Hello Vinod

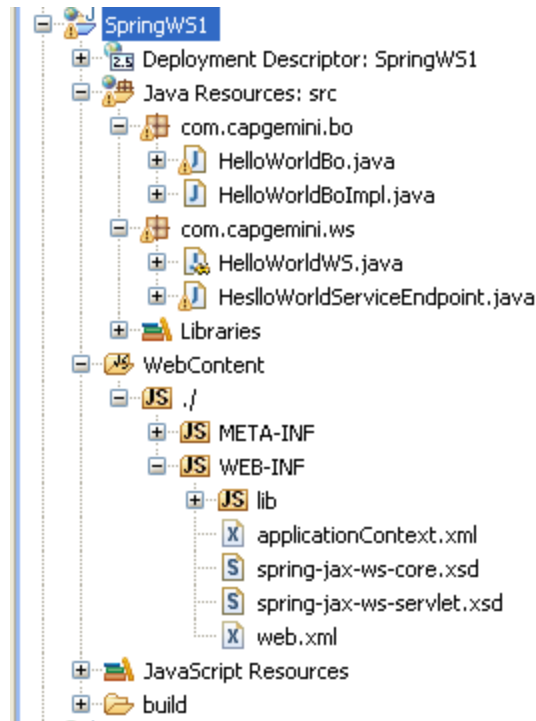
Singh!</return></ns2:sayHelloResponse></S:Body></S:Envelope>=== DONE ===

JAX-WS

SpringWSServer

Directory Structure:





1) HelloBo.java

```
/**
 * These are assignment topics
 * @Jan 23, 2013 @1:00:34 PM
 */
package com.capgemini.bo;

import javax.jws.WebService;

/**
 * @author Venu Kumar for assignments
 * @version 1.0
 * @date Jan 23, 2013
 */

public interface HelloWorldBo{

    String getHelloWorld();

}
```

2) HelloWorldBoImpl

```
/**
 * These are assignment topics
 * @Jun 3, 2011 @1:02:31 PM
 */
package com.capgemini.bo;

/**
 * @author markumar for assignments
 * @version 1.0
 * @date Jun 3, 2011
 */
public class HelloWorldBoImpl implements HelloWorldBo{

    public String getHelloWorld(){
        return "JAX-WS BO Layer..... + Spring!";
    }

}
```

HelloWorldWS.java

```
/**
 * These are assignment topics
 * @Jan 23, 2013 @12:58:36 PM
 */
package com.capgemini.ws;

import javax.jws.WebMethod;
import javax.jws.WebService;

import com.capgemini.bo.HelloWorldBo;

@WebService
public class HelloWorldWS {
    HelloWorldBo helloWorldBo;

    @WebMethod(exclude = true)
    public void setHelloWorldBo(HelloWorldBo helloWorldBo) {
        this.helloWorldBo = helloWorldBo;
    }

    @WebMethod(operationName = "getHelloWorld")
    public String getHelloWorld() {
        return helloWorldBo.getHelloWorld();
    }

}
HeslloWorldServiceEndpoint.java
/**
```

* These are assignment topics
 * @Jun 3, 2011 @3:25:36 PM
 */

```
package com.capgemini.ws;
```

```
import javax.jws.WebMethod;
```

```
import javax.jws.WebService;
```

```
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.web.context.support.SpringBeanAutowiringSupport;
```

```
import com.capgemini.bo.HelloWorldBo;
```

```
public class HeslloWorldServiceEndpoint extends SpringBeanAutowiringSupport {
```

```
}
```

```
applicationContext.xml
```

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

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ws="http://jax-ws.dev.java.net/spring/core"
  xmlns:wss="http://jax-ws.dev.java.net/spring/servlet"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://jax-ws.dev.java.net/spring/core classpath:spring-jax-ws-core.xsd
    http://jax-ws.dev.java.net/spring/servlet classpath:spring-jax-ws-servlet.xsd" >
```

```
<wss:binding url="/hello">
  <wss:service>
    <ws:service bean="#helloWs"/>
  </wss:service>
</wss:binding>
```

```
<!-- Web service methods -->
<bean id="helloWs" class="com.capgemini.ws.HelloWorldWS">
  <property name="helloWorldBo" ref="HelloWorldBo" />
</bean>
```

```
<bean id="HelloWorldBo" class="com.capgemini.bo.HelloWorldBoImpl" />
```

```
</beans>
```

```
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/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5">
  <display-name>SpringWS1</display-name>
  <servlet>
```



```

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

    <servlet-mapping>
        <servlet-name>hello-servlet</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>

    <!-- Register Spring Listener -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
    </listener>

</web-app>

```

SpringWSClient:

Run the following through Cmd prompt

D:\WebServices\SpringWSClient\src wsimport -keep -p com.capgemini.ws
<http://localhost:8081/SpringWS1/hello?wsdl>

```

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

```

Published by JAX-WS RI at <http://jax-ws.dev.java.net>. RI's version is JAX-WS RI 2.2.7-b01 svn-revision#`{svn.Last.Changed.Rev}`.

```

-->
- <!--

```

Generated by JAX-WS RI at <http://jax-ws.dev.java.net>. RI's version is JAX-WS RI 2.2.7-b01 svn-revision#`{svn.Last.Changed.Rev}`.

```

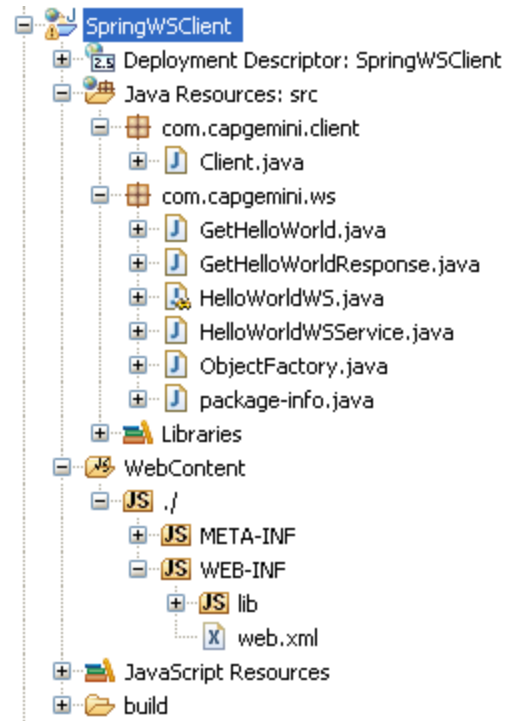
-->
_ <definitions xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-utility-1.0.xsd" xmlns:wsp="http://www.w3.org/ns/ws-policy"
xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://ws.capgemini.com/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://ws.capgemini.com/" name="HelloWorldWSService">
_ <types>

```

```

- <xsd:schema>
  <xsd:import namespace="http://ws.cappgemini.com/"
schemaLocation="http://localhost:8081/SpringWS1/hello?xsd=1" />
  </xsd:schema>
  </types>
- <message name="getHelloWorld">
  <part name="parameters" element="tns:getHelloWorld" />
  </message>
- <message name="getHelloWorldResponse">
  <part name="parameters" element="tns:getHelloWorldResponse" />
  </message>
- <portType name="HelloWorldWS">
- <operation name="getHelloWorld">
  <input wsam:Action="http://ws.cappgemini.com/HelloWorldWS/getHelloWorldRequest"
message="tns:getHelloWorld" />
  <output
wsam:Action="http://ws.cappgemini.com/HelloWorldWS/getHelloWorldResponse"
message="tns:getHelloWorldResponse" />
  </operation>
  </portType>
- <binding name="HelloWorldWSPortBinding" type="tns:HelloWorldWS">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
- <operation name="getHelloWorld">
  <soap:operation soapAction="" />
- <input>
  <soap:body use="literal" />
  </input>
- <output>
  <soap:body use="literal" />
  </output>
  </operation>
  </binding>
- <service name="HelloWorldWSService">
- <port name="HelloWorldWSPort" binding="tns:HelloWorldWSPortBinding">
  <soap:address location="http://localhost:8081/SpringWS1/hello" />
  </port>
  </service>
</definitions>

```



Client.java

```

/**
 * These are assignment topics
 * @Jan 23, 2013 @7:18:59 PM
 */
package com.capgemini.client;

import com.capgemini.ws.HelloWorldWS;
import com.capgemini.ws.HelloWorldWSService;

/**
 * @author VenuKumar for assignments
 * @version 1.0
 * @date Jan 23, 2013
 */

public class Client {

    public static void main(String[] args) {
        HelloWorldWSService service = new HelloWorldWSService();
        HelloWorldWS hellow = service.getHelloWorldWSPort();
        String mymsg=hellow.getHelloWorld();
        System.out.println("Messssssage : "+mymsg);
        System.out.println("----->> Call Started");
        System.out.println(hellow.getHelloWorld());
    }
}

```

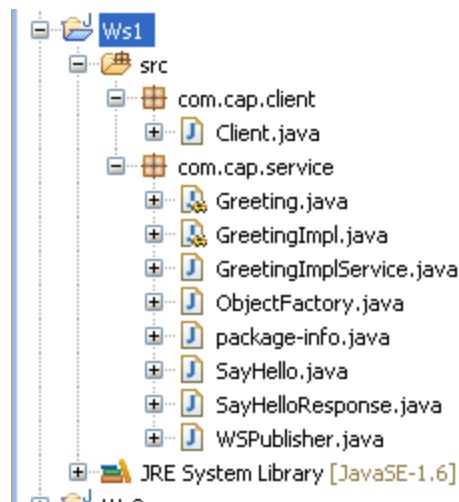
```

        System.out.println("----->> Call Ended");
    }

}
/* OUTPUT:
Messssssage : JAX-WS BO Layer..... + Spring!
----->> Call Started
JAX-WS BO Layer..... + Spring!
----->> Call Ended */

```

JAX-WS → web Services simple creation:



Server side:

Greeting.java

```
package com.cap.service;
```

```

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.bind.annotation.XmlSeeAlso;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;

```

```

/**
 * This class was generated by the JAX-WS RI.
 * JAX-WS RI 2.1.6 in JDK 6
 * Generated source version: 2.1
 *
 */

```

```

@WebService(name = "Greeting", targetNamespace = "http://service.cap.com/")
@XmlSeeAlso({
    ObjectFactory.class
})
public interface Greeting {

    /**
     *
     * @param arg0
     * @return
     * returns java.lang.String
     */
    @WebMethod
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "sayHello", targetNamespace = "http://service.cap.com/",
        className = "com.cap.service.SayHello")
    @ResponseWrapper(localName = "sayHelloResponse", targetNamespace =
        "http://service.cap.com/", className = "com.cap.service.SayHelloResponse")
    public String sayHello(
        @WebParam(name = "arg0", targetNamespace = "")
        String arg0);

}

```

GreetingImpl.java

```

/**
 * These are assignment topics
 * @Jan 18, 2013 @03:00:04 PM
 */
package com.cap.service;

/**
 * @author venu Kumar for assignments
 * @version 1.0
 * @date Jan 18, 2013
 */

import javax.jws.WebService;
@WebService(endpointInterface = "com.cap.service.Greeting")
public class GreetingImpl implements Greeting {
    public String sayHello(String name) {
        return "Hello, Welcom to jax-ws " + name;
    }
}

```

WSPublisher.java

```

/**

```

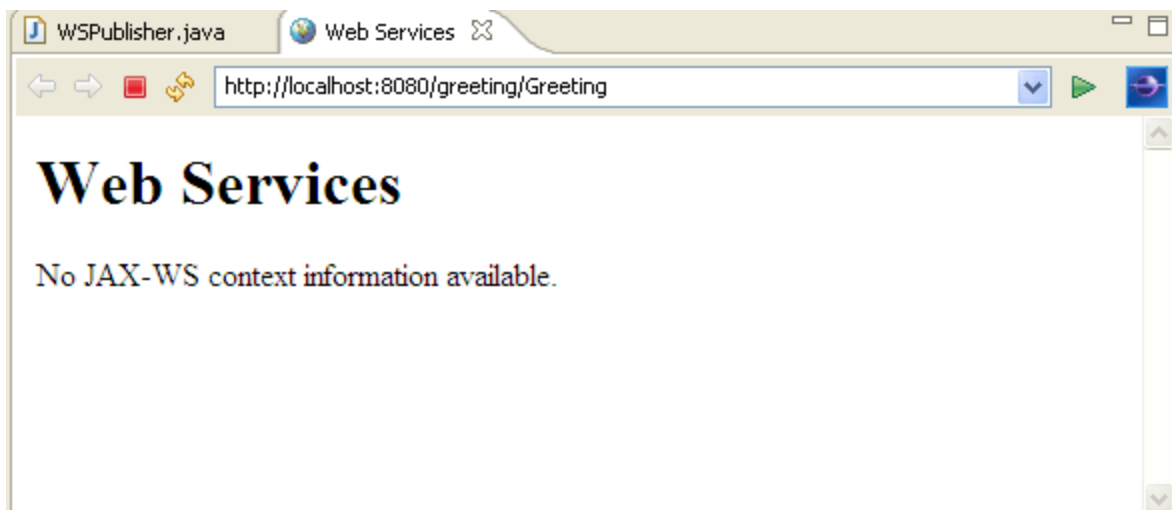
```
* These are assignment topics
* @Jan 18, 2013 @03:06:04 PM
*/
package com.cap.service;

/**
 * @author venu Kumar for assignments
 * @version 1.0
 * @date Jan 18, 2013
 */

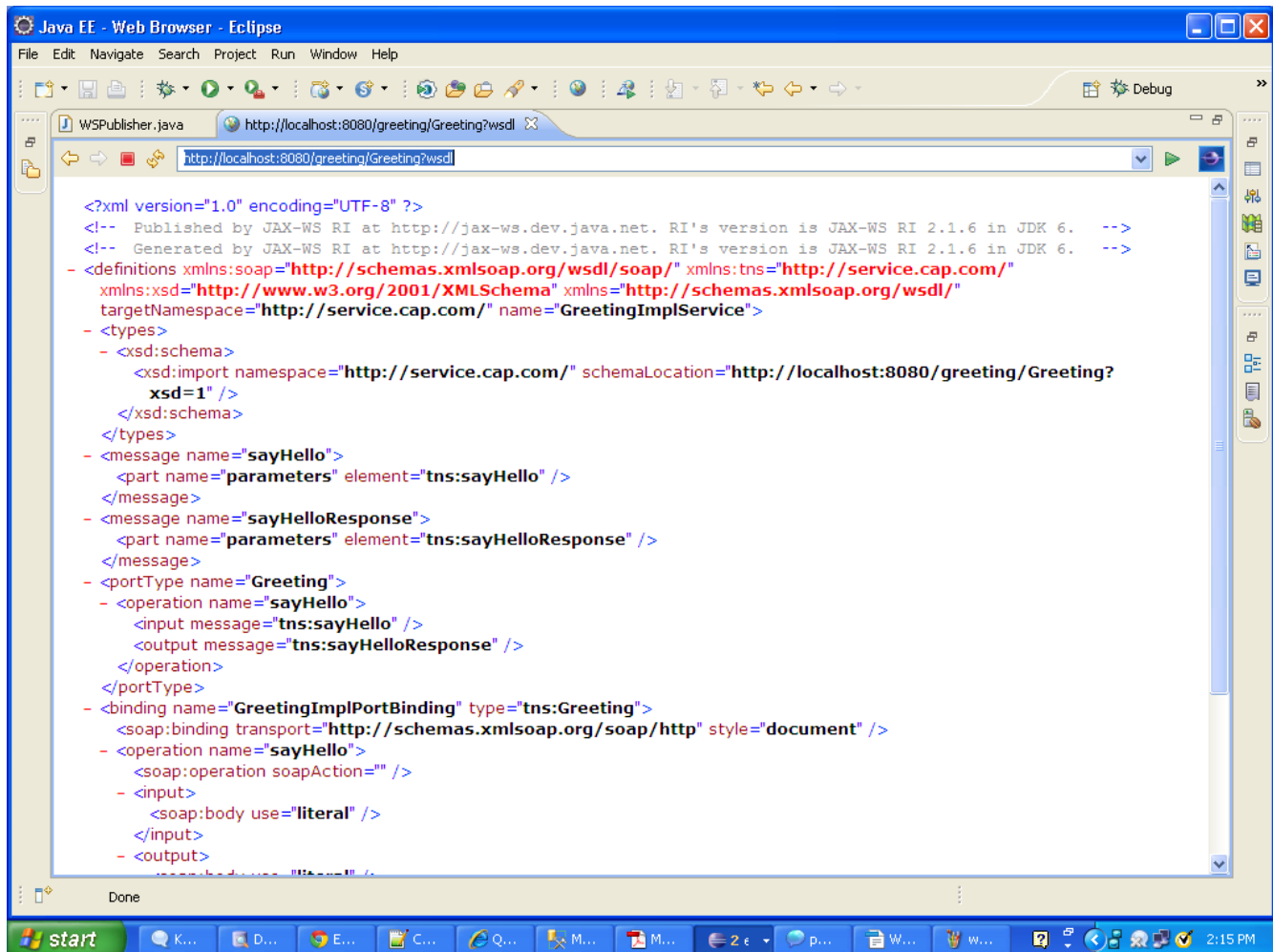
import javax.xml.ws.Endpoint;

import com.cap.service.GreetingImpl;
public class WSPublisher {
    public static void main(String[] args) {
        Endpoint.publish("http://localhost:8080/greeting/Greeting",
            new GreetingImpl());
    }
}
```

Run WSPublisher.java



<http://localhost:8080/greeting/Greeting?wsdl>



Client Side:

To generate client side stubs :

D:\WebServices\WS1\src wsimport -keep -p com.cap.client
<http://localhost:8080/greeting/Greeting?wsdl>

Client.java

```
/**
 * These are assignment topics
 * @Jan 18, 2013 @03:08:04 PM
 */
package com.cap.client;
```

```
import com.cap.service.Greeting;  
import com.cap.service.GreetingImplService;  
  
/**  
 * @author VenuKumar.S for assignments  
 * @version 1.0  
 * @date Jan 18, 2013  
 */  
  
public class Client {  
    public static void main(String[] args) {  
        GreetingImplService service = new GreetingImplService();  
        Greeting greet = service.getGreetingImplPort();  
        String mymsg=greet.sayHello("Kumar!!!!!!!!!!");  
        System.out.println("Messssssage : "+mysmsg);  
        System.out.println("----->> Call Started");  
        System.out.println(greet.sayHello("Kumar!!!!!!!!!!"));  
        System.out.println("----->> Call Ended");  
    }  
}  
  
/*Output:  
Messssssage : Hello, Welcom to jax-ws Kumar!!!!!!!!!!  
----->> Call Started  
Hello, Welcom to jax-ws Kumar!!!!!!!!!!  
----->> Call Ended  
*/
```

Run Client.java

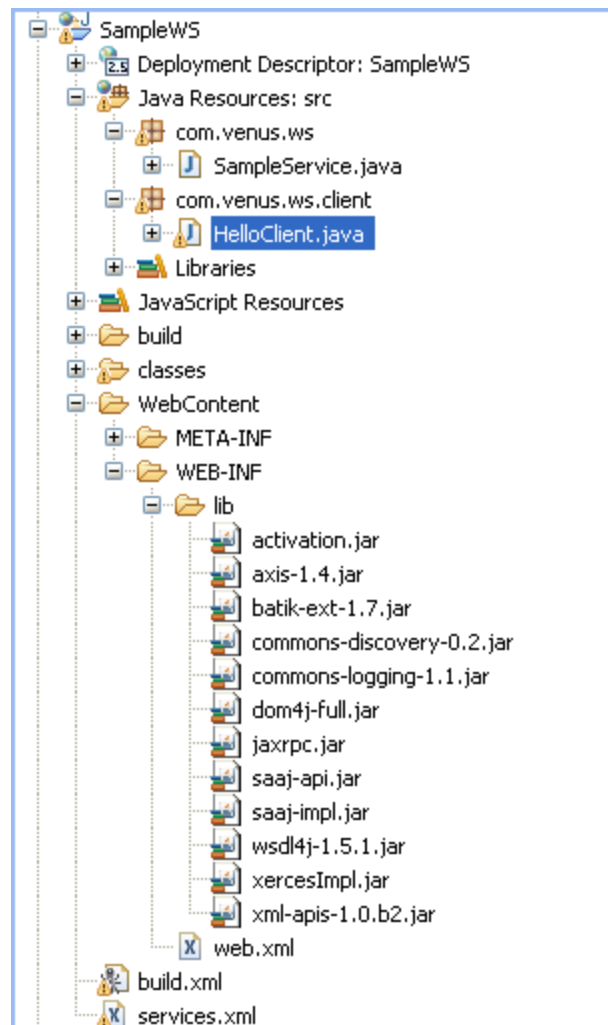
OutPut:

```
Messssssage : Hello, Welcom to jax-ws Kumar!!!!!!!!!!  
----->> Call Started  
Hello, Welcom to jax-ws Kumar!!!!!!!!!!  
----->> Call Ended
```

AXIS2

AXIS2 Example :

1) Install Axis1.4.1



2) Create a new Java project :Create a new Java project titled 'SampleWS '

Server Side Code:

3)Create a new java class called 'SampleService'

```
package com.venus.ws;  
  
public class SampleService {  
    public String getOSName() {
```

```

    return System.getProperty("os.name");
}
}

```

We will be exposing the 'getOSName()' method which is returning the name of the operating system, as a web service.

4) **Configure the web service methods** : Create a new 'services.xml' file

```

<?xml version="1.0" encoding="UTF-8"?>
<serviceGroup>
  <service name="SampleService">
    <Description> Sample Web Service</Description>
    <messageReceivers>
      <messageReceiver mep="http://www.w3.org/2004/08/wsd/in-only"
        class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver" />
      <messageReceiver mep="http://www.w3.org/2004/08/wsd/in-out"
        class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />
    </messageReceivers>
    <parameter name="ServiceClass" locked="false">
      com.venus.ws.SampleService
    </parameter>
  </service>
</serviceGroup>

```

In the above xml, we are supposed to declare all the methods which we want to expose as web services.

5) **Package & deploy the component** : Create a new build.xml in the project path

```

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

<project name="SampleWS" default="all" basedir=".">

  <description>
    Builds the Sample Web Service component.
  </description>

  <!-- Reads the system environment variables and stores them in properties prefixed with "env" -->
  <property environment="env" />

  <!-- Information about the application -->
  <property name="app.title" value="Sample Web Service" />

  <!-- Common directory locations -->
  <property name="DeploymentDir" value="${env.AXIS2_HOME}/repository/services" />
  <property name="SampleWSDir" value="../SampleWS" />

```

```

<property name="classesSampleWS" value="./classes/SampleWS" />

<target name="clean">
  <delete dir="${classesSampleWS}" />
</target>

<target name="compile" depends="clean">

  <mkdir dir="${classesSampleWS}" />

  <echo message="Compiling Sample Web Service ..." />
  <javac destdir="${classesSampleWS}" debug="true" deprecation="false">
    <src path="${SampleWSDir}/src" />
    <include name="**/*.java" />
  </javac>
  <echo message="Completed Sample Web Service compiling." />

</target>

<target name="all" depends="compile">

  <mkdir dir="${classesSampleWS}/META-INF" />
  <copy file="${SampleWSDir}/services.xml" todir="${classesSampleWS}/META-INF" />

  <zip destfile="${DeploymentDir}/samplews.aar" filesonly="false" whenempty="skip"
basedir="${classesSampleWS}" />

  <echo message="Completed packaging sample service" />

</target>

</project>

```

Right click on the build file and select 'Run As – Ant Build'. The ant script will create an archive file named 'samplews.aar' inside the Axis2 deployment directory (%AXIS2_HOME%/repository/services)

6) **start the Axis2 runtime** :Axis2 service can be started by running the 'axis2server.bat' present within the '%AXIS2_HOME%/bin' directory. Thats it, your web service will be up & running.

7) We can access the web service end point (WSDL) by pointing the browser to the <http://localhost:8080/axis2/services/SampleService?wsdl> URL. The web service methods can be tested by creating a proxy of the service using a language of your choice.

Client Side Code:

8) **HelloClient.java** is the client program to access web service.
package com.venus.ws.client;

```
import org.apache.axis.AxisFault;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import org.apache.axis.utils.Options;

import javax.xml.namespace.QName;
import javax.xml.rpc.ParameterMode;
import java.net.URL;

public class HelloClient
{
    public static void main(String args[]) throws Exception
    {
        Service service = new Service();
        Call call = (Call) service.createCall();
        call.setTargetEndpointAddress( "http://localhost:8080/axis2/services/SampleService?wsdl");
        call.setOperationName("getOSName");

        Object ret = call.invoke( new Object[] { "Srikanth" } );
        String res = (String) ret;
        System.out.println(res);
    }
}
```

OUTPUT:

Windows XP

OR

8) Generate Stub files in client side using the following commands:

8.1) WSDL2Java -uri http://localhost:8080/axis2/services/SampleService?wsdl -p com.venus.ws.client -d adb -s

8.2) wsdl2java -uri http://localhost:8080/axis2/services/SampleService?wsdl -ss -sd -d xmlbeans -o ../SamplesWs -p com.venus.ws.client

8.3) Create TestSampleWS1Client.java to test server side code

```
package com.venus.ws.client;
```

```
import java.rmi.RemoteException;
import com.venus.ws.client.SampleServiceStub;
import com.venus.ws.client.SampleServiceStub.GetOSNameResponse;;
```

```
public class TestClient {  
  
    public static void main(String[] args) throws RemoteException {  
  
        SampleServiceStub stub = new SampleServiceStub();  
        GetOSNameResponse res = stub.getOSName();  
        System.out.println(res.get_return());  
        //System.out.println(stub.getOSName());  
  
    }  
}
```

OUTPUT:
Windows XP

Once service is deployed successfully then run the client by first setting the CLASSPATH and PATH as follows:

```
set AXIS_HOME=d:\axis-1_3  
set AXIS_LIB=%AXIS_HOME%\lib  
set AXISCLASSPATH=.;%AXIS_LIB%\axis.jar;%AXIS_LIB%\commons-discovery-  
0.2.jar;%AXIS_LIB%\commons-logging-  
1.0.4.jar;%AXIS_LIB%\jaxrpc.jar;%AXIS_LIB%\saaj.jar;%AXIS_LIB%\log4j-  
1.2.8.jar;%AXIS_LIB%\xml-apis.jar;%AXIS_LIB%\xercesImpl.jar  
  
set classpaht=.;%classpath%;%AXISCLASSPATH%  
  
path c:\jdk1.5.0\bin
```

Creating and consuming a simple Web service

The following procedure explains how to create and invoke a simple web service, which has a single method **sayHello()**.

▲ Create the following class and place it in **axis** directory under the name **Hello.java**.

```
public class Hello  
{  
  
    public String sayHello( String name)  
    {  
        return "Hello," + name;  
    }  
}
```

⤴ Test the web service by giving the url

http://localhost:8080/axis/Hello.jws.

You must see the message saying that there is a web service installed. Click on **wsdl** hyperlink to see WSDL for the service.

⤴ Once service is deployed successfully then run the client by first setting the CLASSPATH and PATH as follows:

```
set AXIS_HOME=d:\axis-1_3
set AXIS_LIB=%AXIS_HOME%\lib
set AXISCLASSPATH=.;%AXIS_LIB%\axis.jar;%AXIS_LIB%\commons-discovery-
0.2.jar;%AXIS_LIB%\commons-logging-
1.0.4.jar;%AXIS_LIB%\jaxrpc.jar;%AXIS_LIB%\saaj.jar;%AXIS_LIB%\log4j-
1.2.8.jar;%AXIS_LIB%\xml-apis.jar;%AXIS_LIB%\xercesImpl.jar

set classpaht=.;%classpath%;%AXISCLASSPATH%

path c:\jdk1.5.0\bin
```

You have to copy **xercesimpl.jar** and **xml-apis.xml** or something similar to it must be copied into **lib** directory of **d:\axis-1_3**.

HelloClient.java is the client program to access web serice.

```
import org.apache.axis.AxisFault;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import org.apache.axis.utils.Options;

import javax.xml.namespace.QName;
import javax.xml.rpc.ParameterMode;
import java.net.URL;

public class HelloClient
{
    public static void main(String args[]) throws Exception
    {
        Service service = new Service();
        Call call = (Call) service.createCall();
        call.setTargetEndpointAddress( "http://localhost:8080/axis/Hello.jws");
        call.setOperationName("sayHello");
        Object ret = call.invoke( new Object[] { "Srikanth" } );
        String res = (String) ret;
        System.out.println(res);
    }
}
```

⤴ Then compile and run the following client.

javac HelloClient.java

java HelloClient

You must see message saying **Hello,Srikanth**. If you get any warning regarding Log4j, ignore them.

AXIS2 - 2nd Example to find sum of two numbers**1)FirstWebService.java**

package com.venus.ws;

```
public class FirstWebService {
    public int addTwoNumbers(int firstNumber, int secondNumber) {
        return firstNumber + secondNumber;
    }
}
```

2)Services.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<serviceGroup>
  <service name="SampleService1">
    <Description> Sample Web Service</Description>
    <messageReceivers>
      <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-only"
        class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver" />
      <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
        class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />
    </messageReceivers>
    <parameter name="ServiceClass" locked="false">
      com.venus.ws.FirstWebService
    </parameter>
  </service>
</serviceGroup>
```

3)build.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="SampleWS1" default="all" basedir=".">
  <description>
    Builds the Sample Web Service component.
  </description>
  <!-- Reads the system environment variables and stores them in properties prefixed with "env" -->
  <property environment="env"/>
  <!-- Information about the application -->
  <property name="app.title" value="Sample Web Service"/>
```

```

<!-- Common directory locations -->
<property name="DeploymentDir" value="${env.AXIS2_HOME}/repository/services"/>
<property name="SampleWSDir" value="../SampleWS1"/>
<property name="classesSampleWS" value="./classes/SampleWS1"/>
<target name="clean">
  <delete dir="${classesSampleWS}"/>
</target>
<target name="compile" depends="clean">
  <mkdir dir="${classesSampleWS}"/>
  <echo message="Compiling Sample Web Service1 ..."/>
  <javac destdir="${classesSampleWS}" debug="true" deprecation="false">
    <src path="${SampleWSDir}/src"/>
    <include name="**/*.java"/>
  </javac>
  <echo message="Completed Sample Web Service compiling."/>
</target>
<target name="all" depends="compile">
  <mkdir dir="${classesSampleWS}/META-INF"/>
  <copy file="${SampleWSDir}/services.xml" todir="${classesSampleWS}/META-INF"/>
  <zip destfile="${DeploymentDir}/samples1.aar" filesonly="false" whenempty="skip"
basedir="${classesSampleWS}"/>
  <echo message="Completed packaging sample service"/>
</target>
</project>

```

Right click on the build file and select 'Run As – Ant Build'. The ant script will create an archive file named 'samples1.aar' inside the Axis2 deployment directory (%AXIS2_HOME%/repository/services)

4) **start the Axis2 runtime** :Axis2 service can be started by running the 'axis2server.bat' present within the '%AXIS2_HOME%/bin' directory. Thats it, your web service will be up & running.

5) We can access the web service end point (WSDL) by pointing the browser to the <http://localhost:8080/axis2/services/SampleService1?wsdl> URL. The web service methods can be tested by creating a proxy of the service using a language of your choice.

<http://localhost:8080/axis2/services/SampleService1?wsdl>

```

<?xml version="1.0" encoding="UTF-8" ?>
_ <wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:ns1="http://org.apache.axis2/xsd" xmlns:ns="http://ws.venus.com"
xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"

```



```

xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
targetNamespace="http://ws.venus.com">
  <wsdl:documentation>Sample Web Service</wsdl:documentation>
  <wsdl:types>
  <xs:schema attributeFormDefault="qualified" elementFormDefault="qualified"
targetNamespace="http://ws.venus.com">
  <xs:element name="addTwoNumbers">
  <xs:complexType>
  <xs:sequence>
    <xs:element minOccurs="0" name="firstNumber" type="xs:int" />
    <xs:element minOccurs="0" name="secondNumber" type="xs:int" />
  </xs:sequence>
  </xs:complexType>
  </xs:element>
  <xs:element name="addTwoNumbersResponse">
  <xs:complexType>
  <xs:sequence>
    <xs:element minOccurs="0" name="return" type="xs:int" />
  </xs:sequence>
  </xs:complexType>
  </xs:element>
  </xs:schema>
  </wsdl:types>
  <wsdl:message name="addTwoNumbersRequest">
    <wsdl:part name="parameters" element="ns:addTwoNumbers" />
  </wsdl:message>
  <wsdl:message name="addTwoNumbersResponse">
    <wsdl:part name="parameters" element="ns:addTwoNumbersResponse" />
  </wsdl:message>
  <wsdl:portType name="SampleService1PortType">
  <wsdl:operation name="addTwoNumbers">
    <wsdl:input message="ns:addTwoNumbersRequest" wsaw:Action="urn:addTwoNumbers"
/>
    <wsdl:output message="ns:addTwoNumbersResponse"
wsaw:Action="urn:addTwoNumbersResponse" />
  </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="SampleService1Soap11Binding"
type="ns:SampleService1PortType">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
  <wsdl:operation name="addTwoNumbers">
    <soap:operation soapAction="urn:addTwoNumbers" style="document" />
  <wsdl:input>
    <soap:body use="literal" />
  </wsdl:input>
  <wsdl:output>
    <soap:body use="literal" />
  </wsdl:output>

```

```

</wsdl:operation>
</wsdl:binding>
<wsdl:binding name="SampleService1Soap12Binding"
type="ns:SampleService1PortType">
  <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"
/>
  <wsdl:operation name="addTwoNumbers">
    <soap12:operation soapAction="urn:addTwoNumbers" style="document" />
  <wsdl:input>
    <soap12:body use="literal" />
  </wsdl:input>
  <wsdl:output>
    <soap12:body use="literal" />
  </wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:binding name="SampleService1HttpBinding" type="ns:SampleService1PortType">
  <http:binding verb="POST" />
  <wsdl:operation name="addTwoNumbers">
    <http:operation location="SampleService1/addTwoNumbers" />
  <wsdl:input>
    <mime:content type="text/xml" part="addTwoNumbers" />
  </wsdl:input>
  <wsdl:output>
    <mime:content type="text/xml" part="addTwoNumbers" />
  </wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="SampleService1">
  <wsdl:port name="SampleService1HttpSoap11Endpoint"
binding="ns:SampleService1Soap11Binding">
    <soap:address
location="http://192.168.32.42:8080/axis2/services/SampleService1.SampleService1H
ttpSoap11Endpoint/" />
  </wsdl:port>
  <wsdl:port name="SampleService1HttpSoap12Endpoint"
binding="ns:SampleService1Soap12Binding">
    <soap12:address
location="http://192.168.32.42:8080/axis2/services/SampleService1.SampleService1H
ttpSoap12Endpoint/" />
  </wsdl:port>
  <wsdl:port name="SampleService1HttpEndpoint"
binding="ns:SampleService1HttpBinding">
    <http:address
location="http://192.168.32.42:8080/axis2/services/SampleService1.SampleService1H
ttpEndpoint/" />
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>

```

Client Side code for SimpleWS1

6)Generate Stub files in client side using the following commands:

6.1)WSDL2Java -uri http://localhost:8080/axis2/services/SampleService1?wsdl -p com.venus.ws.SampleWS1client -d adb -s

6.2) wsdl2java -uri http://localhost:8080/axis2/services/SampleService1?wsdl -ss -sd -d xmlbeans -o ../SamplesWS1 -p com.venus.ws.SampleWSclient

6.3)Create TestSampleWS1Client.java to test server side code

```
package com.venus.ws.clientSampleWS1.test;

import java.rmi.RemoteException;

import com.venus.ws.SampleWS1client.SampleService1Stub;
import com.venus.ws.SampleWS1client.SampleService1Stub.AddTwoNumbers;
import com.venus.ws.SampleWS1client.SampleService1Stub.AddTwoNumbersResponse;

public class TestSampleWS1Client {

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

        SampleService1Stub stub = new SampleService1Stub();
        AddTwoNumbers atn = new AddTwoNumbers();
        atn.setFirstNumber(5);
        atn.setSecondNumber(6);
        AddTwoNumbersResponse res = stub.addTwoNumbers(atn);
        System.out.println("RESPONSE " + res.get_return());

    }
}
```

RESTful-JAX-RS

RESTful (Representational State Transfer) Webservices with Java (Jersey / JAX-RS)

This tutorial explains how to develop RESTful web services in Java with the JAX-RS reference implementation Jersey.

1.REST - Representational State Transfer

1.1.Overview

REST is an architectural style which is based on web-standards and the HTTP protocol. REST was first described by Roy Fielding in 2000.

In a REST based architecture everything is a resource. A resource is accessed via a common interface based on the HTTP standard methods.

In a REST based architecture you typically have a REST server which provides access to the resources and a REST client which accesses and modify the REST resources.

Every resource should support the HTTP common operations. Resources are identified by global IDs (which are typically URIs).

REST allows that resources have different representations, e.g. text, xml, json etc. The rest client can ask for specific representation via the HTTP protocol (content negotiation).

1.2. HTTP methods

The PUT, GET, POST and DELETE methods are typical used in REST based architectures.

The following table gives an explanation of these operations.

- ⤴ GET defines a reading access of the resource without side-effects. The resource is never changed via a GET request, e.g. the request has no side effects (idempotent).
- ⤴ PUT creates a new resource, must also be idempotent.
- ⤴ DELETE removes the resources. The operations are idempotent, they can get repeated without leading to different results.
- ⤴ POST updates an existing resource or creates a new resource.

1.3.RESTFul webservises

A RESTFul webservises are based on the HTTP methods and the concept of REST. A RESTFul webservice typically defines the base URI for the services, the supported MIME-types (XML, Text, JSON, user-defined,...) and the set of operations (POST, GET, PUT, DELETE) which are supported.

2.JAX-RS with Jersey

2.1. JAX-RS and Jersey

Java defines REST support via the Java Specification Request 311 (JSR). This specification is called JAX-RS (The Java API for RESTful Web Services). JAX-RS uses annotations to define the REST relevance of Java classes.

Jersey is the reference implementation for this specification. Jersey contains basically a REST server and a REST client. The core client can be used provides a library to communicate with the server.

On the server side Jersey uses a servlet which scans predefined classes to identify RESTful resources. Via the *web.xml* configuration file for your web application, registers this servlet which is provided by the Jersey distribution

The base URL of this servlet is:

http://your_domain:port/display-name/url-pattern/path_from_rest_class

This servlet analyzes the incoming HTTP request and selects the correct class and method to respond to this request. This selection is based on annotations in the class and methods.

A REST web application consists therefore out of data classes (resources) and services. These two types are typically maintained in different packages as the Jersey servlet will be instructed via the *web.xml* to scan certain packages for data classes.

JAX-RS supports the creation of XML and JSON via the Java Architecture for XML Binding (JAXB).

2.2.JAX-RS annotations

The most important annotations in JAX-RS are listed in the following table.

Table1.JAX-RS annotations

Annotation	Description
@PATH(your_path)	Sets the path to base URL + /your_path. The base URL is based on your application name, the servlet and the URL pattern from the web.xml" configuration file.
@POST	Indicates that the following method will answer to a HTTP POST request
@GET	Indicates that the following method will answer to a HTTP GET request
@PUT	Indicates that the following method will answer to a HTTP PUT request

Annotation	Description
@DELETE	Indicates that the following method will answer to a HTTP DELETE request
@Produces(MediaType.TEXT_PLAIN [, more-types])	@Produces defines which MIME type is delivered by a method annotated with @GET. In the example text ("text/plain") is produced. Other examples would be "application/xml" or "application/json".
@Consumes(type [, more-types])	@Consumes defines which MIME type is consumed by this method.
@PathParam	Used to inject values from the URL into a method parameter. This way you inject for example the ID of a resource into the method to get the correct object.

The complete path to a resource is based on the base URL and the @PATH annotation in your class.

http://your_domain:port/display-name/url-pattern/path_from_rest_class

3.Installation

3.1.Jersey

Download Jersey from the Jersey Homepage.

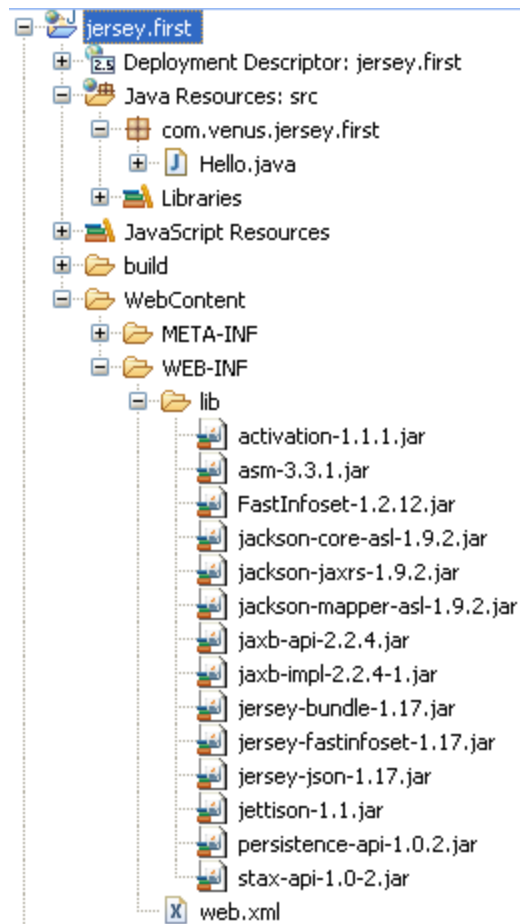
<http://jersey.java.net/>

As of the time of writing the file is called A zip of Jersey containing the Jersey jars, core dependencies (it does not provide dependencies for third party jars beyond the those for JSON support) and JavaDoc.Download this zip; it contains the jar files required for the REST functionality.5. Create your first RESTful Webservice

5.1. Create project with Jersey libraries

Create a new Dynamic Web Project called **jersey.first**.

Copy all jars from your Jersey download into the *WEB-INF/lib* folder



5.2. Java Class

Create the following class.

```
package com.venus.jersey.first;
```

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

```
// Plain old Java Object it does not extend as class or implements  
// an interface
```

```
// The class registers its methods for the HTTP GET request using the @GET annotation.  
// Using the @Produces annotation, it defines that it can deliver several MIME types,  
// text, XML and HTML.
```

```
// The browser requests per default the HTML MIME type.
```

```
//Sets the path to base URL + /hello
@Path("/hello")
public class Hello {

    // This method is called if TEXT_PLAIN is request
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sayPlainTextHello() {
        return "Hello Jersey";
    }

    // This method is called if XML is request
    @GET
    @Produces(MediaType.TEXT_XML)
    public String sayXMLHello() {
        return "<?xml version='1.0'?'>" + "<hello> Hello Jersey" + "</hello>";
    }

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

This class register itself as a get resource via the @GET annotation. Via the @Produces annotation it defines that it delivers the text and the HTML MIME types. It also defines via the @Path annotation that its service is available under the hello URL.

The browser will always request the html MIME type.

5.3. Define Jersey Servlet dispatcher

You need to register Jersey as the servlet dispatcher for REST requests. Open the file *web.xml* and modify the file to the following.

```
<?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/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
```



```
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
<display-name>jersey.first</display-name>
<servlet>
  <servlet-name>Jersey REST Service</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>com.venus.jersey.first</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Jersey REST Service</servlet-name>
  <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
</web-app>
```

The parameter "**com.sun.jersey.config.property.package**" defines in which package jersey will look for the web service classes. This property must point to your resources classes. The URL pattern defines part of the base URL your application will be placed.

5.4. Run the rest service

Run you web application in Eclipse.

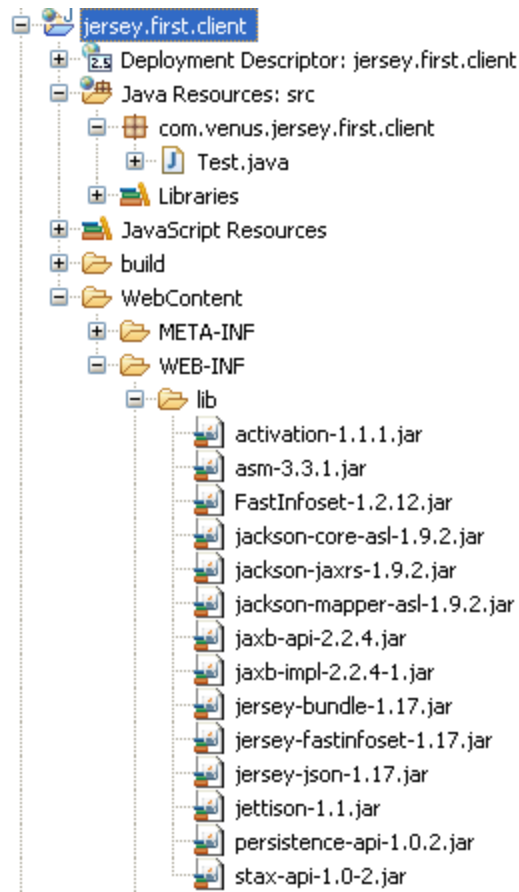
Test your REST service under: "**http://localhost:8080/jersey.first/rest/hello**". This name is derived from the "**display-name**" defined in the **web.xml** file, augmented with the **servlet-mapping url-pattern** and the "**hello**" @Path annotation from your class file. You should get the message "**Hello Jersey**".

The browser requests the HTML representation of your resource. In the next chapter we are going to write a client which will read the XML representation.

5.5. Create a client

Jersey contains a REST client library which can be used for testing or to build a real client in Java.

Create a new Java "**jersey.first.client**" and add the jersey jars to the project and the project build path.



Create the following test class.

```
package com.venus.jersey.first.client;
```

```
import java.net.URI;
```

```
import javax.ws.rs.core.MediaType;
```

```
import javax.ws.rs.core.UriBuilder;
```

```
import com.sun.jersey.api.client.Client;
```

```
import com.sun.jersey.api.client.ClientResponse;
```

```
import com.sun.jersey.api.client.WebResource;
```

```
import com.sun.jersey.api.client.config.ClientConfig;
```

```
import com.sun.jersey.api.client.config.DefaultClientConfig;
```

```
public class Test {
```

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

```
        ClientConfig config = new DefaultClientConfig();
```

```
        Client client = Client.create(config);
```

```
        WebResource service = client.resource(getBaseURI());
```

```
        // Fluent interfaces
```

```
System.out.println(service.path("rest").path("hello").accept(MediaType.TEXT_PLAIN).get(ClientResponse.class).toString());
    // Get plain text

System.out.println(service.path("rest").path("hello").accept(MediaType.TEXT_PLAIN).get(String.class));
    // Get XML

System.out.println(service.path("rest").path("hello").accept(MediaType.TEXT_XML).get(String.class));
    // The HTML

System.out.println(service.path("rest").path("hello").accept(MediaType.TEXT_HTML).get(String.class));

}

private static URI getBaseURI() {
    return UriBuilder.fromUri("http://localhost:8080/jersey.first").build();
}

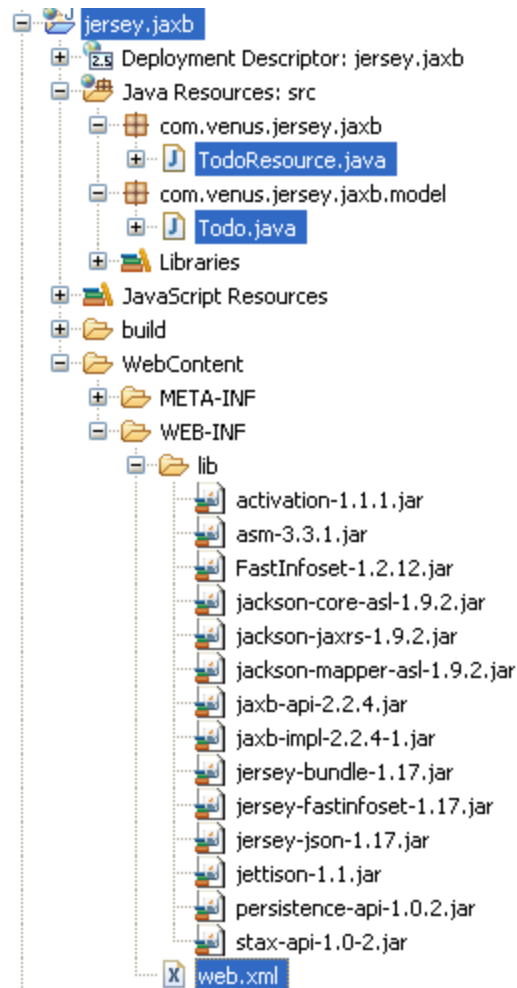
}
```

6. Restful webservices and JAXB

JAX-RS supports the automatic creation of XML and JSON via JAXB.

6.1. Create project

Create a new Dynamic Web Project called com.venus.jersey.jaxb and copy all jersey jars into the WEB-INF/lib folder.



Create your domain class.

```
package com.venus.jersey.jaxb.model;
```

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

```
@XmlRootElement
```

```
// JAX-RS supports an automatic mapping from JAXB annotated class to XML and JSON
```

```
// Isn't that cool?
```

```
public class Todo {  
    private String summary;  
    private String description;  
    public String getSummary() {  
        return summary;  
    }  
    public void setSummary(String summary) {  
        this.summary = summary;  
    }  
    public String getDescription() {
```

```

    return description;
}
public void setDescription(String description) {
    this.description = description;
}
}

```

Create the following resource class. This class simply return an instance of the Todo class.

```

package com.venus.jersey.jaxb;

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

import com.venus.jersey.jaxb.model.Todo;

@Path("/todo")
public class TodoResource {
    // This method is called if XML is request
    @GET
    @Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
    public Todo getXML() {
        Todo todo = new Todo();
        todo.setSummary("This is my first todo");
        todo.setDescription("This is my first todo");
        return todo;
    }

    // This can be used to test the integration with the browser
    @GET
    @Produces({ MediaType.TEXT_XML })
    public Todo getHTML() {
        Todo todo = new Todo();
        todo.setSummary("This is my first todo");
        todo.setDescription("This is my first todo");
        return todo;
    }
}

```

Change *web.xml* to the following.

```

<?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/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
id="WebApp_ID" version="2.5">
<display-name>jersey.jaxb</display-name>
<servlet>
  <servlet-name>Jersey REST Service</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>com.venus.jersey.jaxb</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Jersey REST Service</servlet-name>
  <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
</web-app>
```

Run you web application in Eclipse and validate that you can access your service. Your application should be available under the following URL.

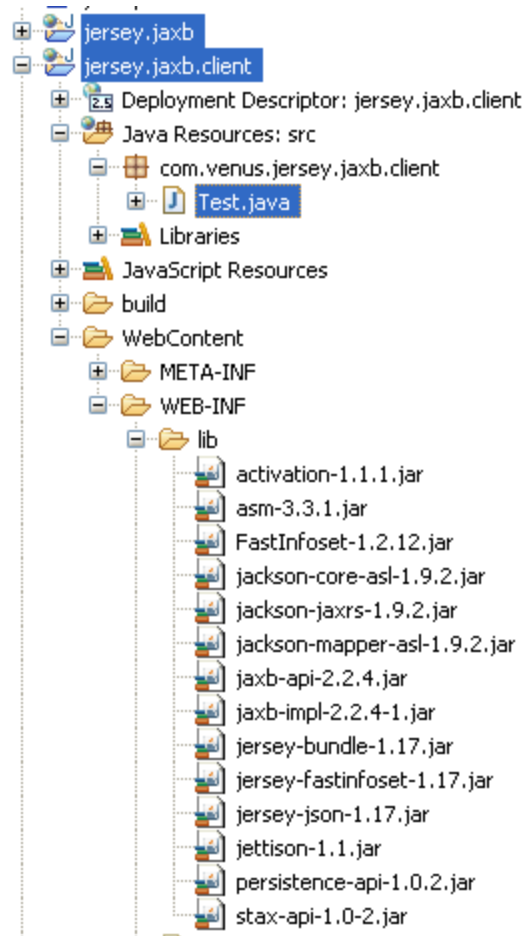
<http://localhost:8080/jersey.jaxb/rest/todo>

OUTPUT:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
- <todo>
  <description>This is my first todo</description>
  <summary>This is my first todo</summary>
</todo>
```

6.2.Create a client

Create a new Java "com.venus.jersey.jaxb.client" and add the jersey jars to the project and the project build path.



Create the following test class.

```
package com.venus.jersey.jaxb.client;
```

```
import java.net.URI;
```

```
import javax.ws.rs.core.MediaType;
```

```
import javax.ws.rs.core.UriBuilder;
```

```
import com.sun.jersey.api.client.Client;
```

```
import com.sun.jersey.api.client.WebResource;
```

```
import com.sun.jersey.api.client.config.ClientConfig;
```

```
import com.sun.jersey.api.client.config.DefaultClientConfig;
```

```
public class Test {
```

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

```
        ClientConfig config = new DefaultClientConfig();
```

```
        Client client = Client.create(config);
```

```

WebResource service = client.resource(getBaseURI());
// Get XML

System.out.println(service.path("rest").path("todo").accept(MediaType.TEXT_XML).get(String.class));
// Get XML for application

System.out.println(service.path("rest").path("todo").accept(MediaType.APPLICATION_JSON).get(String.class));
// Get JSON for application

System.out.println(service.path("rest").path("todo").accept(MediaType.APPLICATION_XML).get(String.class));
}

private static URI getBaseURI() {
    return UriBuilder.fromUri("http://localhost:8080/jersey.jaxb").build();
}

}

/*
OUTPUT:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><todo><description>This is my first
todo</description><summary>This is my first todo</summary></todo>
{"description":"This is my first todo","summary":"This is my first todo"}
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><todo><description>This is my first
todo</description><summary>This is my first todo</summary></todo>
*/

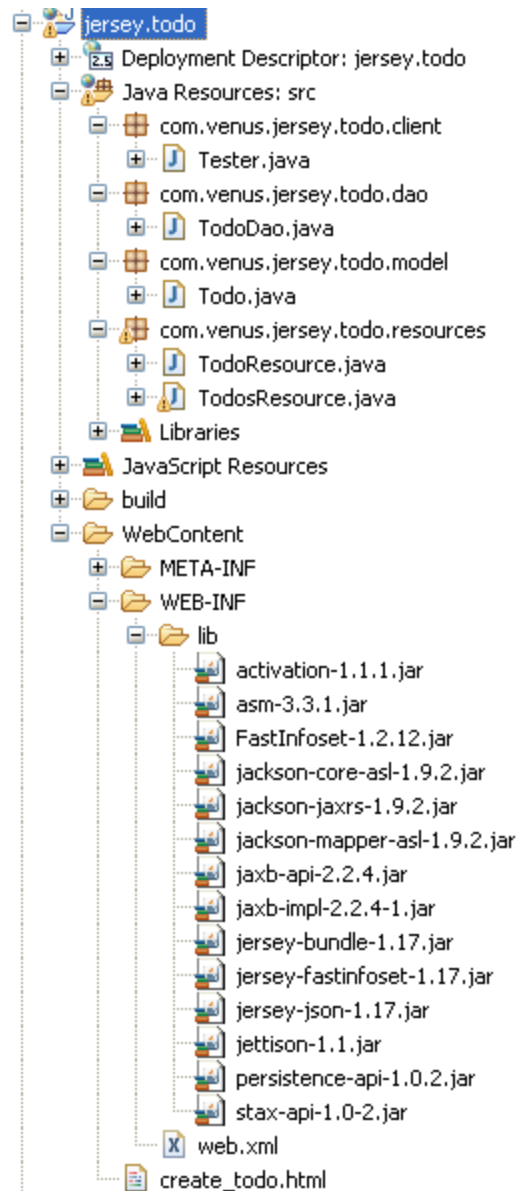
```

7.CRUD RESTful webservice

This section creates a CRUD (Create, Read, Update, Delete) restful web service. It will allow to maintain a list of todos in your web application via HTTP calls.

7.1.Project

Create a new dynamic project called d com.venus.jersey.todo and add the jersey libs.



Change the *web.xml* file to the following.

```
<?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/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
  <display-name>jersey.todo</display-name>
```

```
<servlet>
  <servlet-name>Jersey REST Service</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>com.venus.jersey.todo.resources</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Jersey REST Service</servlet-name>
  <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
</web-app>
```

Create the following data model and a [Singleton](#) which serves as the data provider for the model. We use the implementation based on an enumeration. Please see the link for details. The Todo class is annotated with a JAXB annotation.

```
package com.venus.jersey.todo.model;
```

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

```
@XmlRootElement
```

```
public class Todo {
  private String id;
  private String summary;
  private String description;

  public Todo(){

  }
  public Todo (String id, String summary){
    this.id = id;
    this.summary = summary;
  }
  public String getId() {
    return id;
  }
  public void setId(String id) {
    this.id = id;
  }
  public String getSummary() {
    return summary;
  }
  public void setSummary(String summary) {
    this.summary = summary;
  }
  public String getDescription() {
    return description;
  }
}
```

```

public void setDescription(String description) {
    this.description = description;
}

}

package com.venus.jersey.todo.dao;

import java.util.HashMap;
import java.util.Map;

import com.venus.jersey.todo.model.TODO;

public enum TODODao {
    instance;

    private Map<String, TODO> contentProvider = new HashMap<String, TODO>();

    private TODODao() {

        TODO todo = new TODO("1", "Learn REST");
        todo.setDescription("Read http://www.venus.com/articles/REST/article.html");
        contentProvider.put("1", todo);
        todo = new TODO("2", "Do something");
        todo.setDescription("Read complete http://www.venus.com");
        contentProvider.put("2", todo);

    }

    public Map<String, TODO> getModel(){
        return contentProvider;
    }

}

```

7.2. Create a simple HTML form

The rest service can be used via HTML forms. The following HTML form will allow to post new data to the service. Create the following page called ***create_todo.html*** in the ***WebContent*** folder.

```

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Form to create a new resource</title>
</head>
<body>

```

```

<form action=" ../jersey.todo/rest/todos" method="POST">
<label for="id">ID</label>
<input name="id" />
<br/>
<label for="summary">Summary</label>
<input name="summary" />
<br/>
Description:
<TEXTAREA NAME="description" COLS=40 ROWS=6></TEXTAREA>
<br/>
<input type="submit" value="Submit" />
</form>
</body>
</html>

```

7.3.Rest Service

Create the following classes which will be used as REST resources.

```

package com.venus.jersey.todo.resources;

import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.PUT;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;
import javax.xml.bind.JAXBElement;

import com.venus.jersey.todo.dao.TODOdao;
import com.venus.jersey.todo.model.TODO;

public class TODOResource {
    @Context
    UriInfo uriInfo;
    @Context
    Request request;
    String id;
    public TODOResource(UriInfo uriInfo, Request request, String id) {
        this.uriInfo = uriInfo;
        this.request = request;
        this.id = id;
    }

    //Application integration
    @GET

```

```
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public Todo getTodo() {
    Todo todo = TodoDao.instance.getModel().get(id);
    if(todo==null)
        throw new RuntimeException("Get: Todo with " + id + " not found");
    return todo;
}

// For the browser
@GET
@Produces(MediaType.TEXT_XML)
public Todo getTodoHTML() {
    Todo todo = TodoDao.instance.getModel().get(id);
    if(todo==null)
        throw new RuntimeException("Get: Todo with " + id + " not found");
    return todo;
}

@PUT
@Consumes(MediaType.APPLICATION_XML)
public Response putTodo(JAXBElement<Todo> todo) {
    Todo c = todo.getValue();
    return putAndGetResponse(c);
}

@DELETE
public void deleteTodo() {
    Todo c = TodoDao.instance.getModel().remove(id);
    if(c==null)
        throw new RuntimeException("Delete: Todo with " + id + " not found");
}

private Response putAndGetResponse(Todo todo) {
    Response res;
    if(TodoDao.instance.getModel().containsKey(todo.getId())) {
        res = Response.noContent().build();
    } else {
        res = Response.created(uriInfo.getAbsolutePath()).build();
    }
    TodoDao.instance.getModel().put(todo.getId(), todo);
    return res;
}

}

package com.venus.jersey.todo.resources;

import java.io.IOException;
import java.net.URI;
```

```
import java.util.ArrayList;
import java.util.List;

import javax.servlet.http.HttpServletResponse;
import javax.ws.rs.Consumes;
import javax.ws.rs.FormParam;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;

import com.venus.jersey.todo.dao.TODODao;
import com.venus.jersey.todo.model.TODO;

// Will map the resource to the URL todos
@Path("/todos")
public class TodosResource {

    // Allows to insert contextual objects into the class,
    // e.g. ServletContext, Request, Response, UriInfo
    @Context
    UriInfo uriInfo;
    @Context
    Request request;

    // Return the list of todos to the user in the browser
    @GET
    @Produces(MediaType.TEXT_XML)
    public List<TODO> getTodosBrowser() {
        List<TODO> todos = new ArrayList<TODO>();
        todos.addAll(TODODao.instance.getModel().values());
        return todos;
    }

    // Return the list of todos for applications
    @GET
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public List<TODO> getTodos() {
        List<TODO> todos = new ArrayList<TODO>();
        todos.addAll(TODODao.instance.getModel().values());
        return todos;
    }
}
```

```
// returns the number of todos
// Use http://localhost:8080/de.venus.jersey.todo/rest/todos/count
// to get the total number of records
@GET
@Path("count")
@Produces(MediaType.TEXT_PLAIN)
public String getCount() {
    int count = TodoDao.instance.getModel().size();
    return String.valueOf(count);
}

@POST
@Produces(MediaType.TEXT_HTML)
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
public void newTodo(@FormParam("id") String id,
    @FormParam("summary") String summary,
    @FormParam("description") String description,
    @Context HttpServletResponse servletResponse) throws IOException {
    Todo todo = new Todo(id,summary);
    if (description!=null){
        todo.setDescription(description);
    }
    TodoDao.instance.getModel().put(id, todo);

    servletResponse.sendRedirect("../create_todo.html");
}

// Defines that the next path parameter after todos is
// treated as a parameter and passed to the TodoResources
// Allows to type http://localhost:8080/de.venus.jersey.todo/rest/todos/1
// 1 will be treated as parameter todo and passed to TodoResource
@Path("/{todo}")
public TodoResource getTodo(@PathParam("todo") String id) {
    return new TodoResource(uriInfo, request, id);
}
}
```

7.4. Run

Run your web application in Eclipse and test the availability of your REST service under: "<http://localhost:8080/jersey.todo/rest/todos>". You should see the XML representation of your Todo items.

<http://localhost:8080/jersey.todo/rest/todos>

output:

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

- <todoes>

- <todo>

  <description>Read complete http://www.venus.com</description>

  <id>2</id>

  <summary>Do something</summary>

</todo>

- <todo>

  <description>Read http://www.venus.com/articles/REST/article.html</description>

  <id>1</id>

  <summary>Learn REST</summary>

</todo>

</todoes>
```

Todo items.

To see the count of Todo items use "http://localhost:8080/jersey.todo/rest/todos/count"

to see an exiting todo use "http://localhost:8080/jersey.todo/rest/todos/{id}", e.g.

"http://localhost:8080/jersey.todo/rest/todos/1" to see the todo with ID 1.

We currently have only todos with the id's 1 and 2, all other requests will result an HTTP error code.

Please note that with the browser you can only issue HTTP GET requests. The next chapter will use the jersey client libraries to issue get, post and delete.

7.5.Create a client

Create a new Java project called com.venus.jersey.todo.client . Create a *lib* folder and place all jersey libs in this folder. Add the jars to the classpath of the project.

Create the following class.

```
package com.venus.jersey.todo.client;
```



```
import java.net.URI;

import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.UriBuilder;

import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.config.ClientConfig;
import com.sun.jersey.api.client.config.DefaultClientConfig;
import com.sun.jersey.api.representation.Form;

import com.venus.jersey.todo.model.TODO;

public class Tester {
    public static void main(String[] args) {
        ClientConfig config = new DefaultClientConfig();
        Client client = Client.create(config);
        WebResource service = client.resource(getBaseURI());
        // Create one TODO
        TODO todo = new TODO("3", "Blabla");
        ClientResponse response = service.path("rest").path("todos")
            .path(todo.getId()).accept(MediaType.APPLICATION_XML)
            .put(ClientResponse.class, todo);
        // Return code should be 201 == created resource
        System.out.println(response.getStatus());
        // Get the Todos
        System.out.println(service.path("rest").path("todos")
            .accept(MediaType.TEXT_XML).get(String.class));
        // Get JSON for application
        System.out.println(service.path("rest").path("todos")
            .accept(MediaType.APPLICATION_JSON).get(String.class));
        // Get XML for application
        System.out.println(service.path("rest").path("todos")
            .accept(MediaType.APPLICATION_XML).get(String.class));

        // Get the TODO with id 1
        System.out.println(service.path("rest").path("todos/1")
            .accept(MediaType.APPLICATION_XML).get(String.class));
        // get TODO with id 1
        service.path("rest").path("todos/1").delete();
        // Get the all todos, id 1 should be deleted
        System.out.println(service.path("rest").path("todos")
            .accept(MediaType.APPLICATION_XML).get(String.class));

        // Create a TODO
        Form form = new Form();
        form.add("id", "4");
        form.add("summary", "Demonstration of the client lib for forms");
        response = service.path("rest").path("todos")
```

```

        .type(MediaType.APPLICATION_FORM_URLENCODED)
        .post(ClientResponse.class, form);
System.out.println("Form response " + response.getEntity(String.class));
// Get the all todos, id 4 should be created
System.out.println(service.path("rest").path("todos")
        .accept(MediaType.APPLICATION_XML).get(String.class));
    }

    private static URI getBaseURI() {
        return UriBuilder.fromUri("http://localhost:8080/jersey.todo").build();
    }
}

/*
OUTPUT:
201
    <?xml version="1.0" encoding="UTF-8"
standalone="yes"?><todoes><todo><id>3</id><summary>Blabla</summary></todo>
><todo><description>Read complete
http://www.venus.com</description><id>2</id><summary>Do
something</summary></todo><todo><description>Read
http://www.venus.com/articles/REST/article.html</description><id>1</id><summary>
Learn REST</summary></todo></todoes>
    {"todo":[{"id":"3","summary":"Blabla"}, {"description":"Read complete
http://www.venus.com","id":"2","summary":"Do something"}, {"description":"Read
http://www.venus.com/articles/REST/article.html","id":"1","summary":"Learn REST"}]}
    <?xml version="1.0" encoding="UTF-8"
standalone="yes"?><todoes><todo><id>3</id><summary>Blabla</summary></todo>
><todo><description>Read complete
http://www.venus.com</description><id>2</id><summary>Do
something</summary></todo><todo><description>Read
http://www.venus.com/articles/REST/article.html</description><id>1</id><summary>
Learn REST</summary></todo></todoes>
    <?xml version="1.0" encoding="UTF-8"
standalone="yes"?><todo><description>Read
http://www.venus.com/articles/REST/article.html</description><id>1</id><summary>
Learn REST</summary></todo>
    <?xml version="1.0" encoding="UTF-8"
standalone="yes"?><todoes><todo><id>3</id><summary>Blabla</summary></todo>
><todo><description>Read complete
http://www.venus.com</description><id>2</id><summary>Do
something</summary></todo></todoes>

    Form response <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
    <html>
    <head>
    <title>Form to create a new resource</title>

```

```

</head>
<body>
  <form action="../jersey.todo/rest/todos" method="POST">
    <label for="id">ID</label>
    <input name="id" />
    <br/>
    <label for="summary">Summary</label>
    <input name="summary" />
    <br/>
    Description:
    <TEXTAREA NAME="description" COLS=40 ROWS=6></TEXTAREA>
    <br/>
    <input type="submit" value="Submit" />
  </form>
</body>
</html>

```

```

<?xml version="1.0" encoding="UTF-8"
standalone="yes"?><todoes><todo><id>3</id><summary>Blabla</summary></todo>
<todo><description>Read complete
http://www.venus.com</description><id>2</id><summary>Do
something</summary></todo><todo><id>4</id><summary>Demonstration of the
client lib for forms</summary></todo></todoes>
*/

```

7.6.Using the rest service via HTML page

The above example contains a form which calls a post method of your rest service.

7.7.Using the rest service via X

Usually every programming language provide somewhere libraries for creating HTTP get, post, put and delete requests.

Annotations in RESTful JAX-RS

RESTful Web Service - JAX-RS Annotations - Contents:

Annotation	Package Detail/Import statement
@GET	import javax.ws.rs.GET;
@Produces	import javax.ws.rs.Produces;
@Path	import javax.ws.rs.Path;
@PathParam	import javax.ws.rs.PathParam;
@QueryParam	import javax.ws.rs.QueryParam;

@POST import javax.ws.rs.POST;
@Consumes import javax.ws.rs.Consumes;
@FormParam import javax.ws.rs.FormParam;
@PUT import javax.ws.rs.PUT;
@DELETE import javax.ws.rs.DELETE;

As stated earlier in [Example Application](#), we are using Jersey for RESTful Web services and JAX-RS annotations.

REST follows one-to-one mapping between create, read, update, and delete (CRUD) operations and HTTP methods.

- ⤴ To create a resource on the server, use POST.
- ⤴ To retrieve a resource, use GET.
- ⤴ To change the state of a resource or to update it, use PUT.
- ⤴ To remove or delete a resource, use DELETE.

@GET

Annotate your Get request methods with @GET.

```
1 @GET
2 public String getHTML() {
3     ...
4 }
```

@Produces

@Produces annotation specifies the type of output this method (or web service) will produce.

```
1 @GET
2 @Produces("application/xml")
3 public Contact getXML() {
4     ...
5 }
1 @GET
2 @Produces("application/json")
3 public Contact getJSON() {
4     ...
5 }
```

@Path

@Path annotation specify the URL path on which this method will be invoked.

```
1 @GET
2 @Produces("application/xml")
3 @Path("xml/{firstName}")
4 public Contact getXML() {
5     ...
6 }
```

@PathParam

We can bind REST-style URL parameters to method arguments using @PathParam annotation as shown below.

```
1 @GET
2 @Produces("application/xml")
3 @Path("xml/{firstName}")
4 public Contact getXML(@PathParam("firstName") String firstName) {
5     Contact contact = contactService.findByName(firstName);
6     return contact;
7 }
1 @GET
2 @Produces("application/json")
3 @Path("json/{firstName}")
4 public Contact getJSON(@PathParam("firstName") String firstName) {
5     Contact contact = contactService.findByName(firstName);
6     return contact;
7 }
```

@QueryParam

Request parameters in query string can be accessed using @QueryParam annotation as shown below.

```
1 @GET
2 @Produces("application/json")
3 @Path("json/companyList")
4 public CompanyList getJSON(@QueryParam("start") int start, @QueryParam("limit") int
5 limit) {
6     CompanyList list = new CompanyList(companyService.listCompanies(start, limit));
7     return list;
8 }
```

The example above returns a list of companies (with server side pagination) which can be displayed with rich clients implemented using Ext-js or jQuery. You can read more more about setting up [ExtJS grid panel with remote sorting and pagination using Hibernate](#).

@POST

Annotate POST request methods with @POST.

```
1 @POST
2 @Consumes("application/json")
3 @Produces("application/json")
4 public RestResponse<Contact> create(Contact contact) {
5 ...
6 }
```

@Consumes

The @Consumes annotation is used to specify the MIME media types a REST resource can consume.

```
1 @PUT
```

```
2@Consumes("application/json")
3@Produces("application/json")
4@Path("/{contactId}")
5public RestResponse<Contact> update(Contact contact) {
6...
7}
```

@FormParam

The REST resources will usually consume XML/JSON for the complete Entity Bean. Sometimes, you may want to read parameters sent in POST requests directly and you can do that using @FormParam annotation. GET Request query parameters can be accessed using [@QueryParam](#) annotation.

```
1@POST
2public String save(@FormParam("firstName") String firstName,
3    @FormParam("lastName") String lastName) {
4    ...
5 }
```

@PUT

Annotate PUT request methods with @PUT.

```
1@PUT
2@Consumes("application/json")
3@Produces("application/json")
4@Path("/{contactId}")
5public RestResponse<Contact> update(Contact contact) {
6...
7}
```

@DELETE

Annotate DELETE request methods with @DELETE.

```
1@DELETE
2@Produces("application/json")
3@Path("/{contactId}")
4public RestResponse<Contact> delete(@PathParam("contactId") int contactId) {
5...
6}
```

RESTful Second Example:

RESTful Service: Representational State Transfer (REST) has gained widespread acceptance across the Web as a simpler alternative to SOAP- and Web Services Description Language (WSDL)-based Web services. REST defines a set of architectural principles by which you can design Web services that focus on a system's resources, including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages. If measured by the number of Web services that use it, REST has emerged in the last few years alone as a predominant Web service design model. In fact, REST has had such a large impact on the Web that it has mostly displaced SOAP- and WSDL-based interface

design because it's a considerably simpler style to use.

JAX-RS: Java API for RESTful Web Services (JAX-RS), is a set of APIs to develop REST service. JAX-RS is part of the Java EE6, and make developers to develop REST web application easily.

Jersey: Jersey is the open source, production quality, JAX-RS (JSR 311) Reference Implementation for building RESTful Web services. But, it is also more than the Reference Implementation. Jersey provides an API so that developers may extend Jersey to suit their needs.

Lets start building simple RESTful API with below steps:

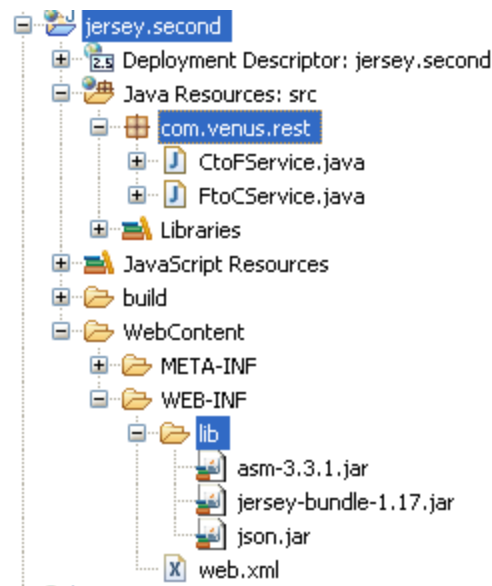
Step 1) In Eclipse => File => New => Dynamic Web Project. Name it as "RESTERJerseyExample"

Step 2) If you don't see web.xml (deployment descriptor) under WebContent\WEB-INF\ then follow [these steps](#).

Step 3) Open web.xml file and add below code just above </web-app>:

```
<?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/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5">
  <display-name>jersey.jaxb</display-name>
  <servlet>
    <servlet-name>Jersey Web Application</servlet-name>
    <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Jersey Web Application</servlet-name>
    <url-pattern>/venus/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Step 4) You need three .jar files as visible in below image. Put it under **WebContents\WEB-INF\lib** folder.



Step 6) Create .java file **CtoFService.java**

package com.venus.rest;

```
/**
 * @author Venu Kumaar.S
 */
```

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
```

```
@Path("/ctofservice")
public class CtoFService {
    @GET
    @Produces("application/xml")
    public String showEmployees() {
```

```
        Double fahrenheit;
        Double celsius = 36.8;
        fahrenheit = ((celsius*9)/5)+32;
```

```
        String result = "@Produces(\"application/xml\") Output: \n\nC to F Converter Output: \n\n"
+ fahrenheit;
        return "<ctofservice>"
            + "<celsius>" + celsius + "</celsius>"
            + "<ctofoutput>" + result + "</ctofoutput>"
            + "</ctofservice>";
    }
```

```
@Path("/{c}")
@GET
```



```

@Produces("application/xml")
public String showEmployee(@PathParam("c") Double c) {
    Double fahrenheit;
    Double celsius = c;
    fahrenheit = ((celsius*9)/5)+32;

    String result = "@Produces(\"application/xml\") Output: \n\nC to F Converter Output:
\n\n" + fahrenheit;
    return "<ctofservice>"
        + "<celsius>" + celsius + "</celsius>"
        + "<ctofoutput>" + result + "</ctofoutput>"
        + "</ctofservice>";
}
}

```

Step 7) Create .java file **FtoCService.java**

```
package com.venus.rest;
```

```

/**
 * @author Venu Kumar.S
 */

```

```

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.Response;
import org.json.JSONException;
import org.json.JSONObject;

```

```

@Path("/ftocservice")
public class FtoCService {

    @GET
    @Produces("application/json")
    public Response convertFtoC() throws JSONException {

```

```

        JSONObject jsonObject = new JSONObject();
        Double fahrenheit = 98.24;
        Double celsius;
        celsius = (fahrenheit - 32)*5/9;
        jsonObject.put("F Value", fahrenheit);
        jsonObject.put("C Value", celsius);

```

```

        String result = "@Produces(\"application/json\") Output: \n\nF to C Converter Output: \n\n"
+ jsonObject;
        return Response.status(200).entity(result).build();
    }
}

```

```
@Path("{f}")
@GET
@Produces("application/json")
public Response convertFtoCfromInput(@PathParam("f") float f) throws JSONException {

    JSONObject jsonObject = new JSONObject();
    float celsius;
    celsius = (f - 32)*5/9;
    jsonObject.put("F Value", f);
    jsonObject.put("C Value", celsius);

    String result = "@Produces(\"application/json\") Output: \n\nF to C Converter Output: \n\n"
+ jsonObject;
    return Response.status(200).entity(result).build();
}
```

Step 8) Deploy project **jersey.second** on Tomcat. Web project should be deployed without any exception.

Step 9) All set. Now test it.

9.1) <http://localhost:8080/jersey.second/venus/ctofservice/>



9.2) <http://localhost:8080/jersey.second/venus/ctofservice/40>

9.3) <http://localhost:8080/jersey.second/venus/ftocservice/>

9.3) <http://localhost:8080/RESTJerseyExample/icrunched/ftocservice/>

9.4) <http://localhost:8080/RESTJerseyExample/icrunched/ftocservice/90>

|| Jai Hind ||