Understanding Java Web Services

JAX-RPC, JAX-WS & JAX-RS

Mr. Sriman

As part of this we are going to cover Web Services and its related technologies like XML, XSD, DTD, JAX-P, JAX-B, JAX-RPC, JAX-WS, and JAX-RS etc. Provides an in-depth understanding of each including the examples.

Contents

Web service

1 Introduction

Web Service is the technology that allows us to build interoperable distributed applications. Before understanding anything about web services let's first of all try to understand what is distributed application and what does interoperability means.

1.1 Distributed Programming

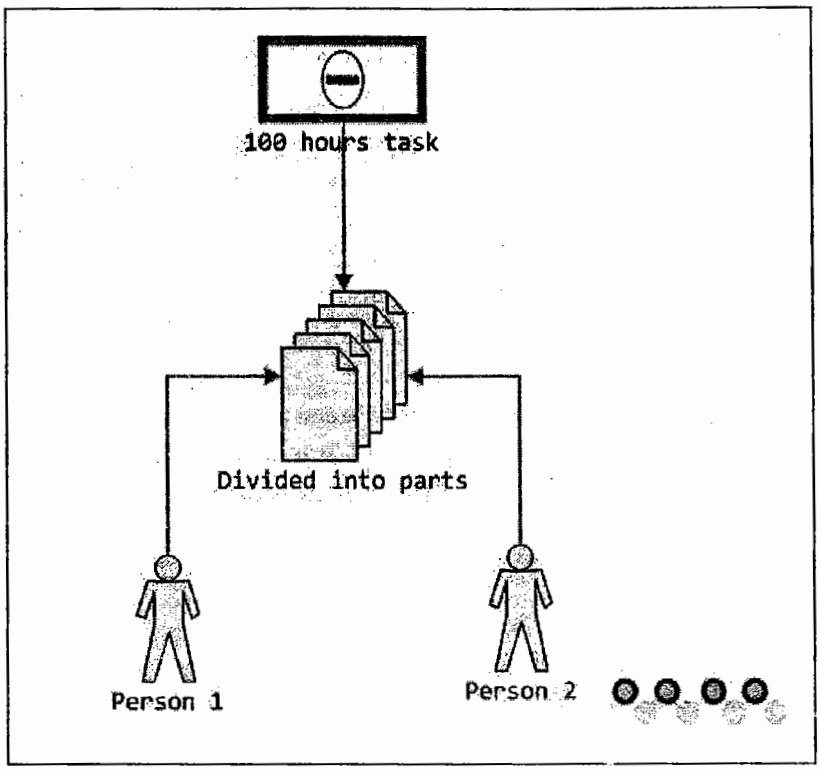
We have a piece of work or Job or Task, if it is assigned to one person he needs 100 hours of time to build the solution. But I want this job to be completed much lesser time than this. To reduce the amount of time needed, there are multiples ways, out of which few are:

a) Increase the energy efficiency levels of the resource, in other words scale up the hardware of the machine. Let's say if the machine has been configured with 2 GB RAM and takes 100 hours of time to compute the solution, increase the RAM from 2 GB to 4 GB, so that the Job would be computed quicker within 95 hours. Increasing the RAM size from 2 GB to 4 GB has improved the performance and reduced the processing time by 5 hours so, let's say if I increase the RAM from 4 to 8 or 8 to 16 or 16 to 32, the performance improvement might be observed but it would not be always relatively impacted. You might see performance improvement up to certain extent but beyond that they may not be any further effect in performance even you scale up the system.

b) Engineer the system. To build a solution for a job always there are multiple ways of doing it, but to complete it within less amount of time we should have a proficient skill set in working on it. When writing logic, we can produce the same output by writing 10 lines of code and even with 5 lines of code, to reduce the amount of code to build the solution; we need to engineer our system. By modularization and effective design we can always reduce the number of lines and tune the logic to complete it faster. The more you engineer/tune the less the time required to process it. But this would also not always relative, because even overdose of medicine will acts as poison, beyond certain extent if you try to re-use or reduce the amount of lines, rather than giving a positive effect; it might downgrade the performance of the application itself.

By now we understood that we can reduce the turnaround time that is required to execute a job by one person, but that is up to certain limit, may be could be 10% or 20% of the actual time, but we may not surprisingly bring down it to half or quarter of the original time by single person.

But there are use cases that demands high amount of throw put or less processing time, such kind of jobs cannot be handled by one person rather we need to distribute such jobs between more than one people. The same is shown in the below figure.



In the above diagram, it shows a 100 hours of job has been divided into multiple parts (let's say two equal 50 hours) and each part is independent with other part. The parts are assigned or shared across two or more people, so that each one would be able to execute on its own piece to build a partial solution, as there are working parallel the time required to complete 100 hours of job by two person's working parallel is 50 hours and the time required to combine their individual outputs to form the final output may be 5 hours. So, the total time required to build the final solution would be 55 hours which when compare with one person working on that job it's a great performance improvement and drastic change in turnaround time.

So, if you want greater performance improvements or quicker turnaround time such kind of jobs must be distributed across various people.

Initially we have programs which are designed to run on one Box/Machine. Like when we develop "C" language program, it will execute on a machine and can talk to other programs which are running on same machine. In this case if we run two instances of the program "P1" on the same machine both instances will execute simultaneously by sharing the totally hard of the same machine. In this case they won't be any performance improvement as they cannot scale up beyond the current machine configuration.

Now we are looking to build applications/programs which not only execute on a machine and talk to other programs on the same machine, these programs should also be able to access/communicate with other programs which are running on different machine. In order for a program to talk to other program running on different machines, both the machines should be connected over wire/network. So, we are trying to build applications which can talk over the network, which means network enabled or network aware applications. By now we understood distributed applications means the programs which are designed to talk to other programs which may be running on same or physically separated but connected via network machines as well.

1.1.1 Advantages of Distributed programming

There are several advantages of going for distributed programming few of them are as described below.

a) Higher throughput: - In distributed programming the no of jobs being computed within a particular time would be more when compared with one program running on one machine.

b) Quicker turnaround time: - If a job takes 100 hours of time to complete, if we distribute this job across multiple machines and executed parallely, it would be completed much faster than 100 hours. So, turnaround time would be faster when we distribute it.

c) High availability of resources: - If a job has been assigned to one person to execute, if he falls sick they won't be any other person who is going to back up the first person to complete it. In case if we distribute it across more than one resource and if one falls sick always there are other people/resources who can share the load of the person and can complete it.

d) High utilization of resources: - There are some resources which are very costly, let's say printer. If want every employee in the organization to have access to printer, instead of buying 100 printers for 100 employees, we can have one printer which can shared across 100 employees so that it would be utilized by everyone effectively.

1.2 Java role in Distributed Programming

Every programming languages added support to build distributed applications. Equally Java also has its contribution towards it. Java has various API's which allows us to build distributed/remote applications. Let's try to understand closely the API's in Java, their advantages and dis-advantages of each, and what are the factors driving us to Web services.

1.2.1 CORBA

CORBA stands for Common Object Request Broker architecture. After socket programming, the first API that has release by Java to support Distributed

programming is CORBA. In CORBA programmer has to code using IDL. IDL stands for "Interface Definition Language", it is a scripting language where CORBA developer writes the IDL file and gives it to CORBA compiler. The CORBA compiler will generate the language specific object (let's say Java or C or C++ or .net). In to the generated Object, the programmer has to write the business logic.

After writing the business logic, in order to expose the CORBA object over the network it has to be deployed on MOM's. MOM stands for Message Oriented Middleware and is nothing but a CORBA server. The purpose of MOM is to host and expose only the CORBA objects. But in order to use MOM, we need to acquire license and involves cost (not open source).

Considering all the above factors, like development will starts with IDL (seems to be different from general way of programming) and deploying requires a licensing server, at those times working would CORBA seems to be complicated and quite messy. This makes CORBA quickly vanish from the market.

1.2.2 RMI

RMI stands for Remote Method Invocation. After CORBA, SUN released RMI as an API to support distributed programming. It allows a Java Object to be exposed over the network. SUN designed RMI keeping in view of all the complexities and drawbacks in CORBA and ensured those will not be re-introduced in RMI. So, in RMI java developer instead of writing a different language script like IDL, he will start with Java Object development and once he finish writing the business logic in the POJO, he will give it as input to RMI Compiler, which will generates abstractions (a layer on top of it) to expose over the network.

Here the programmer need to worry to code a different language file or need not write the code in some other generated object, rather the development starts with POJO and once finishes will generate network abstractions to expose it.

To expose the generated object over network, it has to be deployed on a server called RMI server. The RMI server is open source and would be shipped as part of JDK and is very light weight server.

With this if you observe almost all the dis-advantages with CORBA have been fixed in RMI and it is light weight.

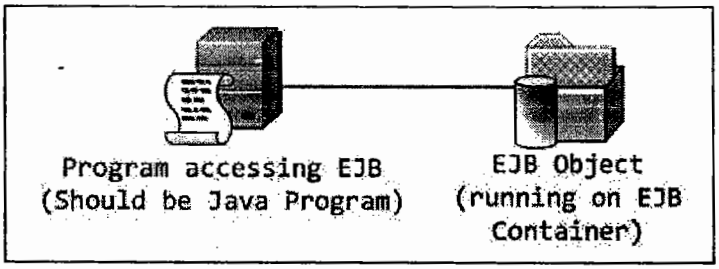
Out of its simplicity, it introduced certain dis-advantages as well. In RMI developer has to deploy the RMI object on RMI server and it is absolutely light weight, it is just a registry holding the RMI object with a name to locate it. Apart from this the server would not provide any infrastructural services like security or connection pooling or transactions and these has to be coded by developer itself, which would be so costly and huge amount of effort is required.

1.2.3 EJB

EJB stands for Enterprise Java Bean, which is next to RMI. From the EJB's onwards the concept of managed object came into picture. You will write an Object which acts as an EJB and is deployed on a managed server called EJB container. While your EJB object is executing, it might need some external infrastructural resource support, instead of programmer coding for it, like security or transaction or auditing or connection pooling, these would be configured, maintained and provided to the EJB object by EJB Container itself.

So, programmer need not worry about implementing these infrastructural resources and he cannot concentrate on his business logic leaving it to Container to provide them. This makes programmer life easier rather than coding each and everything out of box.

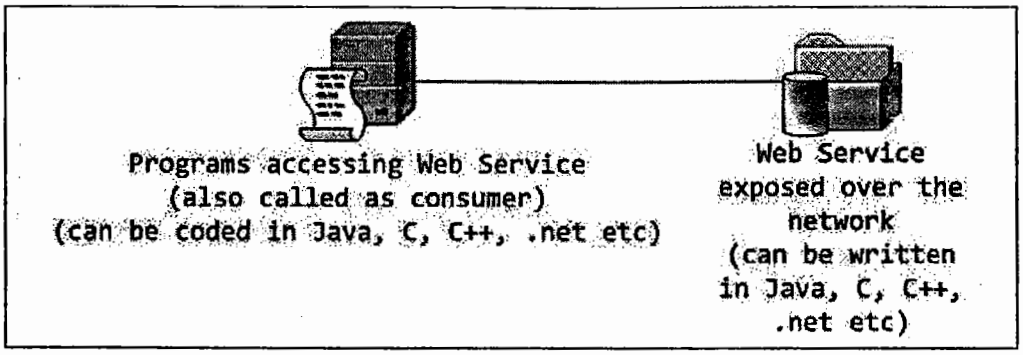
But the problem with EJB is when you expose an Object as EJB, it can be accessed over the network by other programs. The other programs which wants to access the EJB object should also be Java applications only and non-java programs cannot access EJB's as shown below.



1.2.4 Web services

Web services also allows a program to expose objects over the network, but the difference between other distributed objects (RMI or EJB) and Web Service distributed object is these are not only platform (OS) independent, these are even language independent. Which means you can write a Web service Object using C, C++, Perl, Phython, PHP, Java or .Net and can expose over the network. In order to access this, the other program could be written in any of the programming languages like C or C++or Perl or Java etc.

Anything that is accessible irrespective of Platform and programming language is called Interoperability. This mean we are building distributed interoperable programs using Web services.


Programs accessing Web Service
(also called as consumer)
(can be coded in Java, C, C++, .net etc)
Web Service
exposed over the
network
(can be written
in Java, C, C++,
.net etc)

2 Evolution

Initially every language can build programs which can run on one machine. There after every program language strived to build applications which can not only talk to the applications running on same machine but also they should be able to communicate with programs running on different machines but, connected over the network, which is nothing but distributed programming. With this the invention didn't end, they want to still find a way to build distributed programs which can talk over heterogeneous systems (irrespective of platforms and programming languages) is called interoperable distributed applications.

To build interoperable distributed applications, every software vendor in the industry started defining their own standards. Let us say SUN has defined its own standards, Microsoft defined its own standards and IBM came with its own standards etc.

If vendor has their own stack of standards for example as shown below

Vendor	SUN	Microsoft	IBM
Transport Protocol	HTTP	FTP	SMTP
Language for exchanging data	XML	MSXML	CSV
Description	JWSDL	MSWSDL	IBM4SDL

Java programs can only understand the data transmitted over HTTP, so if a Microsoft program is sending the data over FTP, it cannot be read/received by a Java application.

In this case Interoperability never will become a reality. Finally it was realized by every software vendor that we cannot build independently an interoperable distributed applications, unless everyone has agreed upon same standards.

There comes an organization WS-I, stands for Web Service Interoperability organization. It is a non-profitable organization which is formed to build open standards for building interoperable distributed applications. The members of this group are people/representatives from various vendors in the software industry.

WS-I has released BP 1.0 (Basic Profile) specification document. The specification document comprises of set of guidelines or standards that must be used to build an interoperable distributed programs. BP 1.0 doesn't talk about how to develop an interoperable distributed application in Java rather they will talk about the standards or guidelines to build web service in any programing language. After few years of BP 1.0, WS-I has again released one more specification document BP 1.1 which is the successor of BP 1.0 which contains some advancements than its earlier.

2.1 Java API's for WS-I Specification

As discussed above WS-I's BP 1.0 and BP 1.1 specification documents contains guidelines for building Web Services in any programming language, those are not specific for Java. SUN has to adopt those guidelines to facilitate building of Web services in Java.

To build WS-I's BP 1.0 complaint Web Services SUN has released JAX-RPC API and to build BP 1.1 complaint Web Services JAX-WS API has been released. As BP 1.1 is the successor of BP 1.0, similarly JAX-WS API is the successor of JAX-RPC API.

Always using Specifications and API's we cannot build programs rather for an API we need implementation. There a lot of Implementations available for both JAX-RPC API and JAX-WS API, few are open source and few other are commercial. Below is the table showing the implementations of JAX-RPC API and JAX-WS API.

API	Implementations	Description
JAX-RPC API	JAX-RPC SI	Java API for XML Remote Procedural calls Sun Implementation
	Apache AXIS	Apache group provided implementation
	Oracle Web logic Web Services	Oracle Corp
	IBM Web Sphere Web Services	IBM
JAX-WS API	JAX-WS RI	Java API for XML Web Services Reference Implementation
	Metro	One more SUN Implementation
	Apache AXIS2	Another Apache group implementation to support development of JAX-WS API complaint Web services.
	Apache CXF	Apache group
	Oracle Web logic Web Services	Supports JAX-WS API as well
	IBM Web Sphere Web Services	

3 Architecture

Let us try to understand what we need to build a Web Service in general. It's similar to block diagram of a computer like we need mouse, key board, monitor, CPU etc. Let us also identify here what are all the various components required to build a Web Service.

3.1 Medium

If two people want to exchange information between them, what they need? To propagate their wave lengths from one person to other, they need medium. Medium acts as a physical channel on which data would be exchanged. If two computers want to exchange data, first they should be connected over a wire or network.

3.2 Protocol

Having medium itself is enough? With medium we can propagate the wave lengths, but there is not guarantee of receiving the information intact and properly. Let's say two people are talking to each other, instead of one person listening to other both the people started talking to each other at the same time. It seems instead of talking there are barking at each other, would the communication be effective? Rather if there are set of rules that guards their communication, for e.g.. first person says am starting the conversation so that the other person would be in listening mode, once the first person finishes, he will say am done so, that the other person can start the conversation by following the same rules.

This indicates we need set of rules that guards the communication channel which is nothing but a protocol. If two computer systems want to exchange data, having network itself is not enough we need a protocol to guard the exchange of data over them. Protocol simulates that everyone has their independent bands of communication channel to exchange information, even though everyone is sharing the same physical channel. In the same way if two programs want to exchange the data over the network, they need physical data transport protocol like TCP/IP or RMI/IIOP or HTTP or SMTP etc. The recommended protocol in case of Web Service communication is HTTP.

3.3 Language

Now we have medium and protocol to transmit information between two people, but both the parties has to exchange the information in same language, which means a common language to exchange information in an understandable manner like English or French or Japanese etc.

If these two are computers, we can exchange the data between them in an ASCII or UNICODE encoded characters. For example it could be a CSV or a text file of information encoded in common encoding.

CSV files are good when your data is strictly tabular and you know its structure. If you have relationships between different levels of data, CSV are not competent to hold such information. CSV is not self-explanatory which means based on the position of fields user has to interpret the type or nature of information. Let's consider a simple example to understand the same.

There are two systems, one is a Bank system and the other is client who is trying to send the deposit data to deposit into his account. He has sent the information in a CSV format as shown below.

```
1, AC342, Suresh, 422, D, W
```

The field order in which he sent the information is Serial No, Account No, Name, Amount, Operation Type and Branch Code. The Bank system upon receiving the data has started interpreting the data in this order Amount, Account No, Name, Serial No, Branch Code and Operation Type.

The problem with the above representation is the data is not self-explanatory, which is nothing but semantics of data is missing; the receiver of the data also has to interpret the data in same order (which is not guarantee). This might lead to incorrect operations as above instead of deposit, it leads to withdrawl.

Instead of CSV, we can represent the data in MS-Word, MS-Excel etc. But the problem with these kinds of representations is those are recognized by only Windows and few other platforms, which means those are platform dependent formats. We are trying to build an interoperable solution, which means the data we carry between systems must also be interoperable.

Only way of carrying data between systems in an interoperable manner is XML. XML Stands for Extensible Markup Language. It is not a programming language; it is used for storing information in it. How we store the data in MS-Word or MS-Excel or a CSV file similarly XML is also a document which stores data in a XML Format. The data represented in XML has well-defined structure and has semantics attached to it (self-explanatory) as shown below.

```
<info>
  <sno>1</sno>
  <accno>AC324</accno>
  <name>Suresh</name>
  <amount>422</amount>
  <operationType>D</operationType>
  <branchCode>W</branchCode>
</info>
```

In the above XML it is clear that which data is conveying what type of information, so that the receiver of this will not have a chance of miss-interpreting the information. So, the defacto standard for exchanging information between computers is XML.

3.4 Binding Protocol

Having XML is not enough to exchange data between two systems. We need a layer on top of it to classify the information. For e.g. we not only send the business data for communication, along with that we might need to send transport specific or some other helper information. Using XML we cannot differentiate such kind of information. Consider the below example where the user wants to register his account online, so he is sending the registration information in a XML format to the banking system. Registration data will generally contains AccountNo, UserName, Password, Re-Password, Secret Password provided by bank to register.

```
<registration>
  <accountNo>AC222</accountNo>
  <password>533fd</password>        → Secret password
  <username>John</username>
  <password>33kk</password>         → Password with which want to register
</registration>
```

In the above XML we are trying to represent two different types of data one is business data nothing but AccountNo, username and password with which you want to register your account. The other one is helper data or processing data secret password given by bank with which you can be authorized to register.

As discussed it contains two types of data, it would be confusing for the person who is trying to read this XML, to identify which is processing data and which is business data. We need a wrapper on top of it to differentiate the same, which is nothing but SOAP.

SOAP stands for "Simple Object Access Protocol". It is also an XML, but SOAP XML has a rigid structure and has reserved grammar with which we need to write it. SOAP XML has root element as Envelope and has two sections, first one is header and second one is body as shown below.

```
<Envelope>
 <Header>
       <!—processing data or helper data -->
 </Header>
 <Body>
       <!—Business data -->
 </Body>
</Envelope>
```

All the processing data will be placed under <Header> element and the business data will be placed inside <Body> element. SOAP is also called as Binding Protocol, because it binds our business XML. It acts as an Application Specific Protocol, because the receiver of the SOAP XML will be able to identify and differentiate which is processing data and which is business XML. SOAP defines the rules in which other applications has to interpret the data so it is called Application Specific protocol.

Along with that SOAP is also called as carrier of the actual Payload. As our business XML is placed inside the SOAP XML and is carried by SOAP. It is also called as carrier of the actual payload.

3.5 WSDL (Web Service Description Language)

As we know in a Web Service two parts are involved, one is provider and the second one is consumer. Provider is the person who always listens for incoming requests from a consumer and serves data as part of request. Consumer is the requestor, who always seeks information from the Provider.

So, for a consumer in-order to get the data from the Provider, he needs to know the information about the provider. This means the information about the provider has to be document in some document like MS-Word or PDF etc. The problem with this is you are trying to build an Interoperable distributed application which can be accessible by any one literally, but the documentation about it is being documented in non-interoperable document, will it be understood by everyone. That means not only the program the documentation should also be interoperable, there comes WSDL. WSDL Stands for Web Service Description Language. It is also an XML type document, this makes it interoperable. WSDL has a pre-defined structure and has grammar, using which you need to write the WSDL document. WSDL document will provides the entire information about the provider like Service name, Number of Operations, their parameters and return types and Location of the Service etc.

The consumer can know the entire information about the provider by seeing its WSDL document.

Note: - WSDL is a static entity which describes information about the Provider. It is not a program to execute.

3.6 UDDI (Universal Description and Discovery Integration Registry)

After documenting the information about the service in WSDL, we need to distribute the WSDL document to the consumer's. Do we have one consumer for a Provider and multiple Consumers?

For a Provider it could have as many consumers as possible, so we need to circulate or distribute the WSDL document to each and every consumer. Will it be feasible to give WSDL document to each and every consumer, ideally speaking will not be possible that's where UDDI came into picture. UDDI stands for "Universal Description and Discovery Integration Registry", this is the registry which store's all the WSDL documents in it. Provider will publishes the WSDL documents to the UDDI registry and Consumers will browse for a WSDL document from the registry.

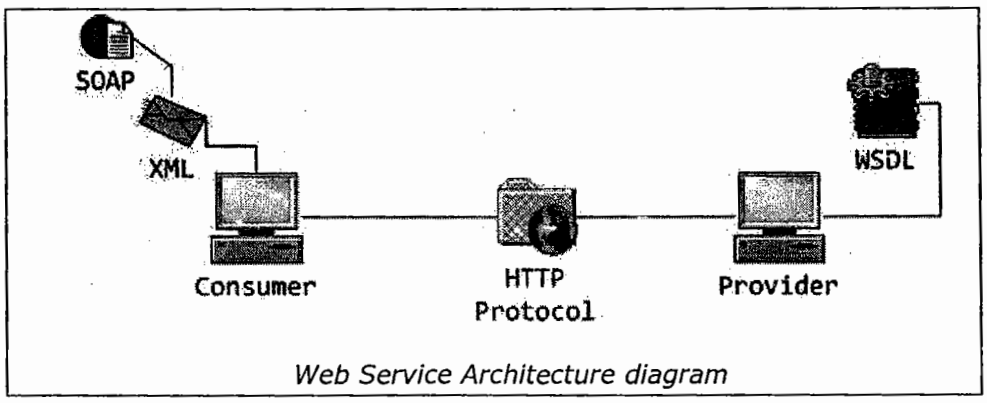
UDDI should be accessible irrespective of programming language which means these should also be interoperable, that's why those are built using XML technology, so those are also called as XML registries.

There are two types of UDDI registries

a) Private UDDI – These are maintained internal to an organization and would be accessible within the organizational n/w.

b) Public UDDI – These are accessible globally across over the network. But few would be open source hosting and few would be commercial.

From the above we understood what are the components required to build a Web Service, below is the diagram depicting the Architecture of a Web Service.



Web Service Architecture diagram

Explanation: - Consumer is the person who tries to access the information from the Provider, and Provider is the person who always services the Consumer.

Those should be often referred to as Consumer and Provider and should not call them as Client and Server in case of Web Services.

Consumer and Provider want to exchange information, so first they should be connected with a wire and needs a protocol to guard their communication. The recommended protocol in a web service communication is HTTP. HTTP protocol is firewall friendly.

They need a common language to exchange data, the defacto standard for exchanging information between computers is XML. But only XML is not enough, as a Web Service communication not only transmits business data, it might also transmit helper or processing information, so we need a layer on top of it called SOAP.

SOAP stands for "Simple Object Access Protocol" acts as a binding protocol to classify the information.

If the consumer wants to access the provider, he needs to know the information about the provider. So, the information about the provider is documented in a document called WSDL and is being published by the Provider in a registry called UDDI. The process of putting the document into the UDDI registry is called Publish and would be generally done by Provider.

Now the consumer has to connect to the registry and performs a search on the registry which is called Discovery to find an appropriate WSDL document. Downloads it locally and now starts understanding the information about provider.

He will builds a program called "Web Service Client" using the information in the WSDL document through which he will sends the request and gets the response back.

By the above we understood that language for exchanging the information is XML. Binding protocol is SOAP which is nothing but XML and Description of the Service is WSDL, is also XML. Registry in which we store the WSDL is UDDI, built on top of XML technologies, everything in Web service is built on XML and without XML Web Services would not be possible, so first we need to understand XML, let's move to XML.

XML

4 XML

XML stands for extensible markup language. Markup language is the language using which you can build other languages like, HTML, XML.

XML is defined and governed by W3Org. The first and final version of XML is XML 1.0. XML is the document which represents data. Unlike C, C++, Java etc. XML is not a programing language. It is the defacto standard for exchanging information between computer systems. When we represent data in an xml document it would be more structured and has well defined semantics attached to it. "Semantics" here represents what a particular data field is representing or stands for, so that any person reading the xml document would be able to interpret in the same manner.

Another significant feature of XML is, once written it is portable across the platforms (Windows, Linux etc). No changes are required to carry the data across platforms.

Every language has keywords. If you take example as C, it has keywords like (if, for, while, do, break, continue etc.) but when it comes to XML there are no keywords or reserved words. What you write will become the element of that XML document.

4.1 XML Element

Element in an XML is written in angular braces for e.g. <beans>. In XML there are two types of elements as follows

1) Start Element/Opening Tag: - Start element is the element which is written in <elementname> indicating the start of a block.
2) End Element/End Tag: - End element is the element which is written in </elementname> indicating the end of a block.

As everything is written in terms of start and end elements, XML is said to be more structured in nature. An XML Element may contain content or may contain other elements under it. So, XML elements which contain other elements in it are called as Compound elements or XML Containers.

4.2 XML Attribute

If we want to have supplementary information attach to an element, instead of having it as content or another element, we can write it as an Attribute of the element.

Example:-

```
<book type="entertainment">
  <isbn>isbn1001</isbn>
</book>
```

In the above example "book" is an element which contains one attributes type which acts as an supplementary information. Isbn is the sub-element of the book element.

4.3 Well-formness

As how any programing language has syntax in writing its code, Well-formness of XML document talks about how to write an XML document. Well-formness indicates the readability nature of an XML document. In other way if an XML document is said to be well-formed then it is readable in nature.

Following are the rules that describe the Well-formness of an XML Document.

1) Every XML document must start with **PROLOG**: - prolog stands for processing instruction, and typically used for understanding about the version of XML used and the data encoding used.

Example: - <?xml version="1.0" encoding="utf-8"?>

2) Root Element: - XML document must contain a root element, and should be the only one root element. All the other elements should be the children of root element.
3) Level constraint: - Every start element must have an end element and the level at which you open a start element, the same level you need to close your end element as well.

Example:-

```
<?xml version="1.0" encoding="utf-8"?>
<student>
        <info>
                <rollno>42</rollno>
                <name>John</info>  (-- info is closed in-correctly --)
        </name>
<student>
```

If any XML is said to follow the above defined rules then it is termed as well-formed.

4.4 XML Usage

An XML document is used in two scenarios

1) **Used for transferring information**:- As said earlier XML is a document representing data and is used for carrying information between two computer systems in an Interoperable manner.
2) **Configurations**: - In J2EE world every component/resource you develop like Servlet or EJB, it has to be deployed into a Web Server. For e.g. in a Servlet application, the path with which the servlet has to be accessible should be specified to the Servlet container so that it can map the incoming request to

the Servlet. This is done in a web.xml. When we provide the configuration information in a XML file the main advantage is, the same xml document can be used in different platforms without changing anything.

4.5 Validatity

Every XML in-order to parse (read) should be well-formed in nature. As said earlier well-formness of an XML document indicates whether it is readable or not, it doesn't talks about whether the data contained in it is valid or not.

Validity of the XML document would be defined by the application which is going to process your XML document. Let's consider a scenario as follows.

You want to get driving license. In order to get a driving license you need to follow certain process like filling the RTA forms and signing them and submitting to the RTA department.

Instead of this can you write your own format of letter requesting the driving license from an RTA department, which seems to be not relevant because driving license is something that would be issued by the RTA department. So, the whole and sole authority of defining what should a person has to provide data to get a driving license will lies in the hands of RTA department rather than you.

In the same way when an application is going to process your xml document, the authority of defining what should be there as part of that xml is lies in the hands of the application which is going to process your document.

For example let's consider the below xml fragment.

Example:- PurchaseOrder xml document

```
<?xml version="1.0" encoding="utf-8"?>
<purchaseOrder>
 <orderItems>
        <item>
                <itemCode>IC323</itemCode>
                <quantity>24</quantity>
        </item>
        <item>
                <itemCode>IC324</itemCode>
                <quantity>abc</quantity>
        </item>
 </orderItems>
<purchaseOrder>
```

In the above xml even though it confirmes to all the well-formness rules, it cannot be used for business transaction, as the 2nd <quantity> element carries the data as "abc" which doesn't makes any sense.

So, in order to check for data validity we need to define the validation criteria of an XML document in either a DTD or XSD document.

5 DTD

DTD stands for Document Type Definition. It is the document which defines the structure of an XML document.

In an XML document we have two types of elements; elements which carry data are called simple elements. An element which contains sub-elements under it is called compound element.

In a language before using a variable we need to declare it. Similarly before using an element in XML first we need to declare it in DTD. While declaring the element in DTD, we need to tell whether it is a simple or compound element.

So, we need to understand the syntax of how to declare simple and compound elements of an xml in dtd.

5.1 Syntax for Declaring Simple Element of an XML

<!Element elementname (CONTENT-TYPE OR CONTENT MODEL)>

For e.g..

<!ELEMENT itemCode (#PCDATA)>

As per the above declaration, it means in my xml we should have an element whose name is itemCode and it contains content of type #PCDATA (parsable character data).

In an XML characters like "<", ">", '"', "'" and "&" are valid characters and if the text content contains those characters the parsers are going to parse or will expand them as further entities.

5.2 Syntax for Compound element of an XML

<!ELEMENT elementname (sub-elem1, sub-elem2...)

For e.g..

<!ELEMENT itemCode (#PCDATA)>

<!ELEMENT quantity (#PCDATA)>

<!ELEMENT item (itemCode, quantity)>

As per the above declaration, we declared an element itemCode in XML which contains child elements like itemCode, quantity.

In XML

```
<item>
   <itemCode>IC200</itemCode>
   <quantity>35</quantity>
</item>
```

We need to follow certain rules while declaring elements in DTD.

1) Element name should follow the java variable naming conventions.
2) Content of an XML is always case-sensitive. So, if we declare element names in small case in DTD, it should appear in small case in XML as well.
3) The child elements separator in the above example is used as "," comma. This is also called as sequence separator. This means in the item element, the first child should be itemCode followed by quantity.

Let's take a complete XML and derive how the DTD looks like for it.

Here are few of the guidelines in creating a DTD from XML.

1) Identify all the simple elements of XML from bottom of the XML and draft them in the DTD.

2) Identify the compound elements of XML from lowest level in the XML and draft them in the DTD.

XML Document

```
<?xml version="1.0" encoding="utf-8"?>
<purchaseOrder>
  <orderItems>
        <item>
              <itemCode>IC323</itemCode>
              <quantity>24</quantity>
        </item>
        <item>
              <itemCode>IC324</itemCode>
              <quantity>abc</quantity>
        </item>
  </orderItems>
<purchaseOrder>
```

For the above xml the DTD looks as shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<!Element purchaseOrder (orderItems)>
<!Element orderItems (item+)>
<!Element item (itemCode, quantity)>
<!Element itemCode (#PCDATA)>
<!Element quantity (#PCDATA)>
```

5.3 Occurrences of an Element under another element

In the above xml if you observe the orderItems element can contain any number of item elements in it, but atleast one item element must be there for a purchaseOrder. This is called occurrence of an element under another element, to indicate this we use three symbols.

? – Represents the sub element under a parent element can appear zero or one-time (0/1).

+ - indicates the sub element must appear at least once and can repeat any number of times (1 – N)

* - indicates the sub element is optional and can repeat any number of times (0 - N)

You will mark the occurrence of an element under another element as follows.

<!Element elementname (sub-elem1 (?/+/*), sub-elem2(?/+/*)>

Leaving any element without any symbol indicates it is mandatory and at max can repeat only once.

5.4 Elements with any contents

Elements declared with the content type as ANY, can contain any combination of parsable data.

<!ELEMENT elementname ANY>

<!ELEMENT mailBody ANY>

In an e-mail body part we have content which can be mixture of any parsable characters which can be declared as ANY type.

5.5 Elements with either/or content

Let's consider an example where in an email the following elements will be there to, from, subject and mailBody. In these elements to and from or mandatory elements and either subject or mailBody should be present but not both. To declare the same we need to use or content separator instead of sequence separator.

<!Element mail (to, from, (subject | mailBody)>

For the above declaration the xml looks as below.

```
<mail>
  <to>toaddr.com</to>
  <from>fromaddr.com<from>
  <subject>mysub</subject>
</mail>
```

(OR)

```
<mail>
  <to>toaddr.com</to>
  <from>fromaddr.com<from>
  <mailBody>mysub</mailBody>
</mail>
```

5.6 Declaring Mixed content

We can even declare an element with mixture of parsable data and elements called mixed content as follows.

<!ELEMENT mail (#PCDATA| to| from| subject| mailBody)*>

5.7 Declaring attribute for an element

Syntax:-

<!ATTLIST elementname attributename attributeType attributeValue>

As shown above to declare an attribute you need to use the tag ATTLIST and elementname stands for which element you want to declare the attribute and attributename stands for what is the attribute you want to have in that element.

The attributeType can be the following:

Type	Description
CDATA	The value is Character data
(en1|en2|..)	The value must be one from the enumerated list of values.
ID	The value is unique id.
IDREF	The value is the id of another element
NMTOKEN	The value is a valid XML element name

The attributeValue can be the following:

Value	Description
#REQUIRED	The attribute is required
#IMPLIED	The attribute is not required
#FIXED value	The attribute value is fixed.

5.7.1 Default Attribute Value

<!ELEMENT shippingAddress (addressLine1, addressLine2, city, state, zip, country)>

<!ATTLIST shippingAddress type CDATA "permanent">

In xml :- <shippingAddress type="permanent">....</shippingAddress>

5.7.2 #REQUIRED

<!ATTLIST shippingAddress type CDATA #REQUIRED> this indicates type attribute is mandatory in shipping address element.

5.7.3 #IMPLIED

<!ATTLIST shippingAddress type CDATA #IMPLIED>, this indicates the type attribute in shippingAddress element is optional.

5.7.4 #FIXED

<!ATTLIST shippingAddress type CDATA #FIXED "permanent">, this indicates the type attribute in shippingAddress element must contain only the value as permanent.

5.7.5 Enumerated Attribute Values

<!ATTLIST shippingAddress type (permanent|temporary) "permanent">

This indicates the type attribute in shippingAddress element should contain only two possible values either permanent or temporary.

5.8 Drawback with DTD's.

DTD are not typesafe, which means when we declare simple elements we indicate it should contain data of type (#PCDATA). #PCDATA means parsable character data means any data that is computer represented format. So it indicates an element can contain any type of data irrespective of whether it is int or float or string. You cannot impose stating my element should contain int type data or float. This is the limitation with DTD documents.

6 XML Schema Document (XSD)

XSD stands for XML schema document. XSD is also an XML document. It is owned by W3Org. The latest version of XSD is 1.1.

Even though we can use DTD's for defining the validity of an XML document, it is not type safe and is not flexibile. XSD is more powerful and is type strict in nature.

XSD document is used for defining the structure of an XML document; it declares elements and the type of data that elements carry.

As XSD is also an XML document so, it also starts with prolog and has one and only one root element. In case of XSD document the root element is <schema>. All the subsequent elements will be declared under this root element.

An XML contains two types of elements. A) Simple elements (these contains content as data) B) Compound or Container's (these contains sub-elements under it).

So, while writing an XSD documents we need to represent two types of elements simple or compound elements. The syntax's for representing a simple and compound elements of XML in a XSD document are shown below.

Syntax for representing simple elements of XML document

```
<xs:element name="elementname" type="datatype"/>
```

Syntax for representing compound element of XML document

In order to represent compound element of an XML document in an XSD, first we need to create a type declaration representing structure of that xml element. Then we need to declare element of that user-defined type, shown below.

```
<xs:complexType name="typeName">
  <xs:sequence> or <xs:all>
        <xs:element name=".." type=".."/>
        <xs:element name=".." type=".."/>
        <xs:element name=".." type=".."/>
  </xs:sequence> or </xs:all>
</xs:complexType>
```

Declaring a complex type is equal to declaring a class in java. In java we define user-defined data types using class declaration. When we declare a class it represents the structure of our data type, but it doesn't allocates any memory. Once a class has been declared you can create any number of objects of that class.

The same can be applied in case of complex type declaration in an XSD document. In order to create user-defined data type in XSD document you need to declare a complex type, creating a complex indicates you just declared the class structure. Once you create your own data type, now you can define as many elements you

need of that type. Let's say for example I declared a complex type whose name is AddressType as follows

```
<xs:complexType name="AddressType">
  <xs:sequence>
        <xs:element name="addressLine1" type="xs:string"/>
        <xs:element name="addressLine2" type="xs:string"/>
        <xs:element name="city" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Now the above fragment indicates you created your own user-defined data type whose name is "AddressType". Now you can declare any number of elements representing that type as shown below.

```
<xs:element name="myAddress" type="AddressType"/>
```

Let's take an XML document for which we will show how its XSD looks like

XML Document – Courier consignment information

```
<consignment>
  <id>C4242</id>
  <bookedby>durga</bookedby>
  <deliveredTo>Sriman</deliveredTo>
  <shippingAddress>
        <addressLine1>S.R Nagar</addressLine1>
        <addressLine2>Opp Chaitanya</addressLine2>
        <city>Hyderabad</city>
        <state>Andhra Pradesh</state>
        <zip>353</zip>
        <country>India</country>
  </shippingAddress>
</consignment>
```

XSD Document – Courier consignment information

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="consignment" type="consignmentType"/>
   <xs:complexType name="consignmentType">
         <xs:sequence>
               <xs:element name="id" type="xs:string"/>
               <xs:element name="bookedby" type="xs:string"/>
               <xs:element name="deliveredTo" type="xs:string"/>
               <xs:element name="shippingAddress"
type="shippingAddressType"/>
         </xs:sequence>
   </xs:complexType>
   <xs:complexType name="shippingAddressType">
         <xs:sequence>
               <xs:element name="addressLine1" type="xs:string">
               <xs:element name="addressLine2" type="xs:string">
               <xs:element name="city" type="xs:string">
               <xs:element name="state" type="xs:string">
               <xs:element name="zip" type="xs:int">
               <xs:element name="country" type="xs:string">
         </xs:sequence>
   </xs:complexType>
</xs:schema>
```

6.1 Sequence VS All

When declaring a complex type, following the <xs:complexType> tag we see a tag <xs:sequence> or <xs:all>. When we use <xs:sequence> under <xs:complexType> tag, what it indicates is all the elements that are declared in that complex type must appear in the same order in xml document. For example let's take a complex type declaration "ItemType" which uses <xs:sequence> tag in its declaration.

```
<xs:element name="item" type="ItemType"/>
<xs:complexType name="ItemType">
   <xs:sequence>
         <xs:element name="itemCode" type="xs:string"/>
         <xs:element name="quantity" type="xs:int"/>
   </xs:sequence>
</xs:complexType>
```

So while using the item element in the xml we need to declare the itemCode and quantity sub-elements under item in the same order as shown below.

```
<item>
   <itemCode>IC303</itemCode>
   <quantity>35</quantity>
</item>
```

When we use <xs:all>, under item element the itemCode and quantity may not appear in the same order. First quantity might appear then itemCode can present in the xml. When we use <xs:all> all the elements under it should appear atleast 0 and at max once.

6.2 Extending Complex Types

XSD along with allowing you to declare your own data types, it will also allow you to extend your own types by the means of inheritance as shown below.

```
<xs:complexType name="USShippingAddressType">
   <xs:complexContent>
         <xs:extension base="shippingAddressType">
               <xs:sequence>
                     <xs:element name="county" type="xs:int"/>
               </xs:sequence>
         </xs:extension>
   </xs:complexContent>
</xs:complexType>
```

6.3 Imposing restrictions on SimpleTypes

XSD's allows you to define data validations on the data an element is going to carry. In order to do this you can define your own simpleType by extending the in-built types and can impose restrictions on it as shown below.

```
<xs:simpleType name="zipType">
   <xs:restriction base="xs:int">
         <xs:totalDigits value="4"/>
   </xs:restriction>
</xs:simpleType>
```

6.4 Declaring attributes for complex elements

In XML we can place supplementary information to an element in an attribute. Let's understand how to add an attribute to an element.

In our example shippingAddress is an compound element. Now we want to add one attribute "type" indicating whether the shipping address is permanent or temporary. In order to do this we need to modify the existing complex type and should add the attribute type to it.

In case of attributes, we should declare the attributes in a complex type at the last after the <xs:sequence> tag, because sequence doesn't apply for attributes.

```
<xs:complexType name="shippingAddressType">
   <xs:sequence>
         <xs:element name="addressLine1" type="xs:string"/>
         <xs:element name="addressLine2" type="xs:string"/>
         <xs:element name="city" type="xs:string"/>
         <xs:element name="state" type="xs:string"/>
         <xs:element name="zip" type="xs:int"/>
         <xs:element name="country" type="xs:string"/>
   </xs:sequence>
   <xs:attribute name="type" type="xs:string" use="required"/>
</xs:complexType>
```

6.5 Declaring Attributes for simple elements

```
<xs:element name="zip type="zipType"/>
<xs:complexType name="zipType">
   <xs:simpleContent>
         <xs:extension base="xs:int">
               <xs:attribute name="zone" type="xs:string"/>
         </xs:extension>
   </xs:simpleContent>
</xs:complexType>
```

For the above element declaration the xml equivalent looks like below.

```
<zip zone="local">353</zip>
```

6.6 Working with Choice

Choice is used for specifying the either or condition in XSD's. For example we have purchaseOrder element. In a purchaseOrder element we have orderItems as first element. Now when it comes to second one we may have shippingAddress or usShippingAddress element.

It is not necessary to present both but either this or the other is mandatory in impose such kind of constraint we need to use Choice element as shown below.

```
<xs:complexType name="purchaseOrderType">
 <xs:sequence>
        <xs:element name="orderItems" type="orderItemsType"/>
        <xs:choice>
                <xs:element name="shippingAddress"
type="shippingAddressType"/>
                <xs:element name="usShippingAddress"
type="usShippingAddressType"/>
        </xs:choice>
 </xs:sequence>
</xs:complexType>
```

6.7 Using ref

Instead declaring an element directly under complexType we can declare it elsewhere in the XSD document and we can refer it in complexType as shown below.

```
<xs:schema....>
 <xs:element name="itemCode" type="xs:string"/>
 <xs:complexType name="itemType">
        <xs:sequence>
                <xs:element name="itemCode" ref="itemCode"/>
                <xs:element name="quantity" type="xs:int"/>
        </xs:sequence>
 <xs:complexType>
</xs:schema>
```

6.8 Import VS Include

In you want to refer one XSD in another XSD there are two options available. First one is Import and the other is Include

6.8.1 Import

Import has to be used when the namespaces of both the XSD documents are different for e.g..

Po1.xsd

```
<xs:schema targetNamespace="http://ebay1.in/sales/types">

</xs:schema>
```

Po2.xsd

```
<xs:schema targetNamespace="http://ebay2.in/sales/types">

        <xs:import namespace="ttp://ebay1.in/sales/types"
        location="c:\po1.xsd"/>

</xs:schema>
```

If you observe in the above scenario po2.xsd want's to use few elements declared in po1.xsd. In order to do that first po2.xsd should refer the po1.xsd. as both the namespaces of these documents are different we should use import rather than include.

6.8.2 Include

Let's say we have a XSD document with more than 100 elements. Writing or managing the elements in one single XSD is not so easy and is error prone as well. Instead we can split it in to two XSD's and can include one into another.

Which means if two XSD's are having the same namespace we need to use include tag to include one in the other, which is similar to combining two into one.

Po1.xsd

```
<xs:schema targetNamespace="http://ebay1.in/sales/types">

</xs:schema>
```

Po2.xsd

```
<xs:schema targetNamespace="http://ebay1.in/sales/types">

        <xs:include schemaLocation="po1.xsd"/>

<xs:schema>
```

7 XSD Namespace

Every programming language one or in another way allows you to declare user-defined data types. Let's consider the case of "C" language it allows you to declare your own types using Structure. In case of "C++" Class declaration allows you to declare your own type, same in case of Java as well.

When it comes to XSD you can declare your element type using XSD complex type declaration.

As how any language allows you to create your own types, they allow you to resolve the naming collision between their types. Let's consider the case of Java; it allows you to declare your own types by class declarations. But when a programmer is given a choice of declaring their own types, language has to provide a means to resolve type names collision declared by several programmers. Java allows resolving those type naming conflicts using packages.

Packages are the means of grouping together the related types. It will allow you to uniquely identify a type by prefixing the package name. In the same way XSD also allows you to declare your own data type by using Complex Type declaration and in the same way it allows you to resolve the type naming conflicts by means of Namespace declaration.

XSD Namespaces has two faces,

1) Declaring the namespace in the XSD document using Targetnamespace declaration

2) Using the elements that are declared under a namespace in xml document.

7.1 XSD Targetnamespace

Targetnamespace declaration is similar to a package declaration in java. You will bind your classes to a package so, that while using them you will refer with fully qualified name. Similarly when you create a complexType or an Element you will bind them to a namespace, so that while referring them you need to use qName.

In order to declare a package we use package keyword followed by name of the package. To declare a namespace in XSD we need to use targetNameSpace attribute at the Schema level followed by targetnamespace label as shown below.

Example:-

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetnamespace="http://durgasoft.com/training/calendar/types">
 <xs:complexType name="courseType">
        <xs:element name="courseId" type="xs:string"/>
        <xs:element name="courseName" type="xs:string"/>
        <xs:element name="duration" type="xs:datetime"/>
 </xs:complexType>
</xs:schema>
```

The courseType by default is binded to the namespace http://durgasoft.com/training/calendar/types

So, while creating an element "course" of type courseType you should use the qName of the courseType rather simple name. (qName means namespace:element/type name).

But the namespace labels could be any of the characters in length, so if we prefix the entire namespace label to element/type names it would become tough to read. So, to avoid this problem XSD has introduced a concept called short name. Instead of referring to namespace labels you can define a short name for that namespace label using xmlns declaration at the <schema> level and you can use the short name as prefix instead of the complete namespace label as shown below.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetnamespace=http://durgasoft.com/training/calendar/types
xmlns:dt="http://durgasoft.com/training/calendar/types">
 <xs:element name="course" type="dt:courseType"/>
 <xs:complexType name="courseType">
        <xs:element name="courseId" type="xs:string"/>
        <xs:element name="courseName" type="xs:string"/>
        <xs:element name="duration" type="xs:datetime"/>
 </xs:complexType>
</xs:schema>
```

In java we can have only one package declaration at a class level. In the same way we can have only one targetNamespace declaration at an XSD document.

7.2 Using elements from an xml namespace (xmlns)

While writing an XML document, you need to link it to XSD to indicate it is following the structure defined in that XSD. In order to link an XML to an XSD we use an attribute "schemaLocation".

schemaLocation attribute declaration has two pieces of information. First part is representing he namespace from which you are using the elements. Second is the document which contains this namespace, shown below.

```
<?xml version="1.0" encoded="utf-8"?>

<course xsi:schemaLocation="http://durgasoft.com/training/calendar/types
file:///c:\folder1\folder2\courseInfo.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-Instance">

</course>
```

In the above xml we stated this xml is using xsd document courseInfo.xsd whose namespace is http://durgasoft.com/training/calendar/types

If you want to include two XSD documents in the xml then you need to declare in schemaLocation tag <namespace1> <schemalocation1> <namespace2> <schemalocation2>.
For example:-

```
<?xml version="1.0" encoded="utf-8"?>

<course xsi:schemaLocation="http://durgasoft.com/training/calendar/types
file:///c:\folder1\folder2\courseInfo.xsd
http://durgasoft.com/training/vacation/types
file:///c:\folder1\folder2\vacation.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-Instance">

</course>
```

Now in the above xml we are using two XSD documents courseInfo.xsd and vacation.xsd. With this declaration we will not be able to find whether the course element is used from courseInfo.xsd or vacation.xsd. To indicate it we should prefix the namespace to the course element.

But the namespace labels can be arbitrary string of characters in any length. So, we need to define short name, so that we can prefix shortname while referring the elements as shown below.

```
<?xml version="1.0" encoded="utf-8"?>

<dc:course xsi:schemaLocation="http://durgasoft.com/training/calendar/types
file:///c:\folder1\folder2\courseInfo.xsd
http://durgasoft.com/training/vacation/types
file:///c:\folder1\folder2\vacation.xsd"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-Instance
xmlns:dc="http://durgasoft.com/training/calendar/types">

</dc:course>
```

7.3 Importing Multiple XSD's in XML

Here is the example showing how to work with two XSD documents which has different namespaces in one XML.

Po1.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" targetNamespace="http://ebay1.in/sales/types"
xmlns:et1="http://ebay1.in/sales/types">
    <xs:element name="item" type="et1:itemType"/>
    <xs:complexType name="itemType">
        <xs:sequence>
            <xs:element name="itemCode" type="xs:string"/>
            <xs:element name="quantity" type="xs:int"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Po2.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" targetNamespace="http://ebay2.in/sales/types"
xmlns:et2="http://ebay2.in/sales/types"
xmlns:et1="http://ebay1.in/sales/types">
    <xs:import namespace="http://ebay1.in/sales/types"
schemaLocation="c:\po1.xsd"/>
    <xs:element name="orderItems" type="et2:orderItemsType"/>
    <xs:complexType name="orderItemsType">
        <xs:sequence>
            <xs:element ref="et1:item"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Po.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<et2:orderItems xmlns:et1="http://ebay1.in/sales/types"
xmlns:et2="http://ebay2.in/sales/types"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ebay2.in/sales/types file:///C:/po2.xsd">
    <et1:item>
        <et1:itemCode>IC1001</et1:itemCode>
        <et1:quantity>12</et1:quantity>
    </et1:item>
</et2:orderItems>
```

7.4 Difference between DTD and XSD

DTD	XSD
• DTD stands for document type definition • DTD's are not XML Type documents • DTD's are tough to learn as those are not XML documents. So, an XML programmer has to under new syntaxes in coding DTD • DTD's are not type safe, these will represents all the elements with data type as (#PCDATA). • DTD's don't allow you to create use-defined types.	• XSD stands for XML Schema Documents • XSD's are XML Type documents • As XSD's are XML type documents it is easy for any programmer to work with XSD. • XSD's are strictly typed, where the language has list of pre-defined data types in it. While declaring the element you need tell whether this element is of what type. • XSD's allows you to create user-defined data types using complexType declaration and using that type you can create any number of elements.

JAX-P

8 JAX-P

JAX-P stands for Java API for XML Processing. As we discussed earlier, XML is also a document which holds data in an XML format. Initially in Java there is no API that allows reading the contents of an XML document in an XML format. Java programmers has to live with Java IO programming to read them in a text nature or need to build their own logic to read them in XML format.

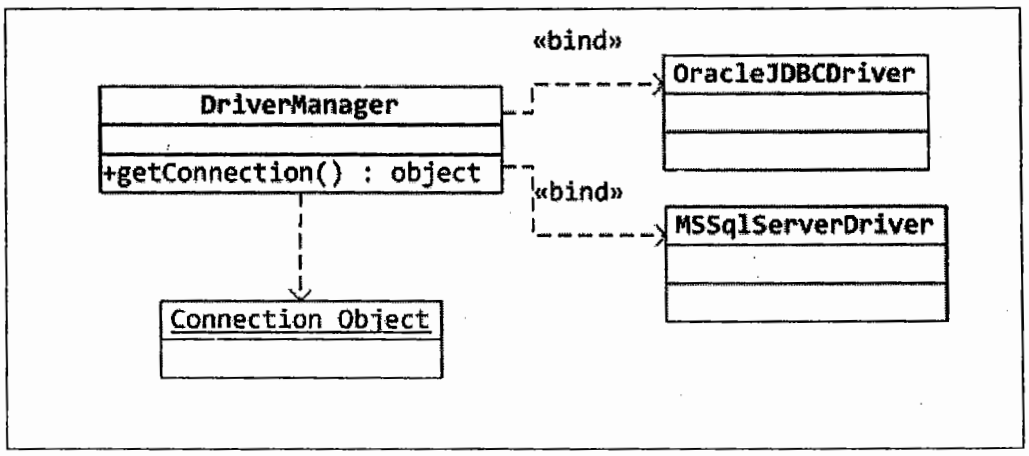
By looking at this lot of Java software vendors in the market has started building their own libraries/API's that allows processing an XML documents in XML nature. Few of them are DOM4J, JDOM.

But the problem with third party libraries are you need to stick with vendor specific classes, as there is no common API which facilities switching between them.

For example if we want to work with database programming in Java, we need to use JDBC API provided by Sun. But JDBC API itself is not enough, as any API contains only set of interfaces and abstract classes and only few concrete implementations. So we need an implementation to work with a specific database. JDBC Drivers are the implementations of the JDBC API which allows you to deal with specific database.

Advantage of using standard JDBC API is your API classes never allow your code to expose to the implementation, so that your code would be flexible enough to change between different databases by simply switching the drivers.

Let's say in order to execute a sql query against a database you need Connection. But Connection is an Interface provided by JDBC API, and we need to find the implementation class of Connection Interface and should instantiate it. Instead of us finding the Implementation class, JDBC API has provided a class DriverManager which acts as a factory class who knows how to find the appropriate implementation of Connection Interface and instantiate it.



The advantage of going for Factory class like "DriverManager" is by passing different url, username and password, you would be able to switch between one database vendor to another database vendor.

After many software vendors started developing their own libraries, sun has finally released JAX-P API which allows us to work with XML documents. As indicated JAX-P is an API which means not complete in nature, so we need implementations of JAX-P API and there are many implementations of JAX-P API available for e.g.. Crimson, Xerces2, Oracle V2 Parser etc.

Xerces2 is the default parser that would be shipped as part of JDK1.5+. This indicates if you are working on JDK1.5+ you don't need any separate Jar's to work with JAX-P API.

8.1 XML Processing Methodologies

Let's park aside JAX-P for now and try to understand the ways of processing an XML document. If you are asked to read notes, each one has their own preference of reading the notes like one will start reading it from middle and other from the first chapter etc. In the same way there are multiple methods of reading the XML documents exists, those are SAX and DOM.

SAX and DOM are the universal methodologies of processing (reading) XML documents irrespective of a particular technology. Few programming languages/API's supports only SAX way of reading documents, few other supports only DOM and quite a few number support both the ways of reading.

Let us try to understand what these methodologies are and how to read XML documents following these methodologies.

8.1.1 SAX (Simple Access for XML)

SAX stands for Simple Access for XML, it is a methodology that allows reading XML documents in an event based processing model. Any event based processing model contains three actors as described below.

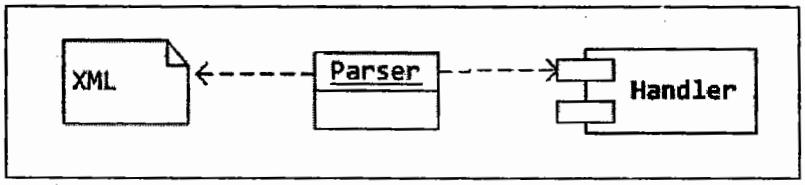
Source: - is the originator of event, who can raise several types of events. Events could be of multiple types if we consider AWT, we can consider source as a button, which is capable of raising several types of events like button click, mouse move, key pressed etc.

Listener: - Listener is the person who will listens for an event from the source. Listener listens for a specific type of event from the source and triggers an even handling method on the Event handler to process it.

Event Handler: - Once a listener captures an event, in order to process it, it calls a method on the event handler class. As there are different types of events, to handle them the event handler contains several methods each target to handle and process a specific type of event.

In addition SAX is a sequential processing model, which process XML elements sequentially from top to the bottom. While processing the document, it places a pointer to the document and increments sequentially from the top. Based on the type of element it is pointing to, it raises a respective event, which will notify the listener to handle and process it.

So here source is the XML document which can raise several types of events based on the type of element it is pointing to (e.g... START_DOCUMENT, START_ELEMENT etc...). Listener is the parser who will reads the XML document and triggers a method call on the Handler. Event handler contains several methods to handle various types of events as shown below.



SAX is very fast in processing XML documents when compared with DOM and consumes less amount of memory, as at any given point of time it loads only one element into the memory and process. Here one key thing to note is using SAX we can only read the XML document, we cannot modify/create a document.

Using JAX-P API we can process XML documents in SAX model, here also source is the XML document, who is capable of raising multiple types of events likes startDocument, startElement, endDocument etc. In order to read the elements of XML and process them by raising events, we need a parser and here in SAX it is SAXParser.

As indicated earlier JAX-P is an API which contains interfaces/abstract classes and few concrete implementations, SAXParser is also an interface, in order to instantiate it, we need to find the respective implementation of it, instead of we finding, the API has provided a factory, SAXParserFactory which would be able to Instantiate the implementation of SAXParser. The following two lines shows how to instantiate the SAXParser.

```
SAXParserFactory saxParserFactory = SAXParserFactory.newInstance();
SAXParser parser = saxParserFactory.newSAXParser();
```

Along with the parser, we need a Handler class to handle the event and process it. A handler class would be written by extending it from DefaultHandler, we can override the methods to process respective events. Below is the example showing a class extending from DefaultHandler.

```
package com.saxparser.handler;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class POHandler extends DefaultHandler {
	@Override
	public void characters(char[] xml, int offSet, int len) throws
	SAXException{
		String data = new String(xml, offSet, len);
		System.out.print(data);
	}

	@Override
	public void endDocument() throws SAXException {
		System.out.println("END DOCUMENT");
	}

	@Override
	public void endElement(String arg0, String arg1, String localName)
			throws SAXException {
		System.out.println("</"+localName+">");
	}

	@Override
	public void startDocument() throws SAXException {
		System.out.println("START DOCUMENT");
	}

	@Override
	public void startElement(String arg0, String arg1, String localName,
			Attributes arg3) throws SAXException {
		System.out.print("<"+localName+">");
	}
}
```

In the above class we are overriding five methods which would be triggered based on the type of elements the parser is pointing to on the source XML document.

a) startDocument – would be triggered at the start of the XML document
b) startElement – Whenever the parser encounters the starting element, it raises this method call by passing the entire element information to the method.
c) characters – This method would be invoked by the parser, whenever it encounters data portion between the XML elements. To this method it passes the entire XML as character array along with two other integers one indicating the position in the array from which the current data portion begins and the second integer representing the number of characters the data span to.
d) endElement – would be triggered by the parser, when it encounters a closing element or ending element.

e) endDocument – would be triggered by the parser once it reaches end of the document.

Once the handler has been created, we need to call the parser class parse() method by passing the source as XML and the handler as the object of handler class which we created just now. So, that the parser will reads the XML elements and triggers a respective method call on the handler object provided.

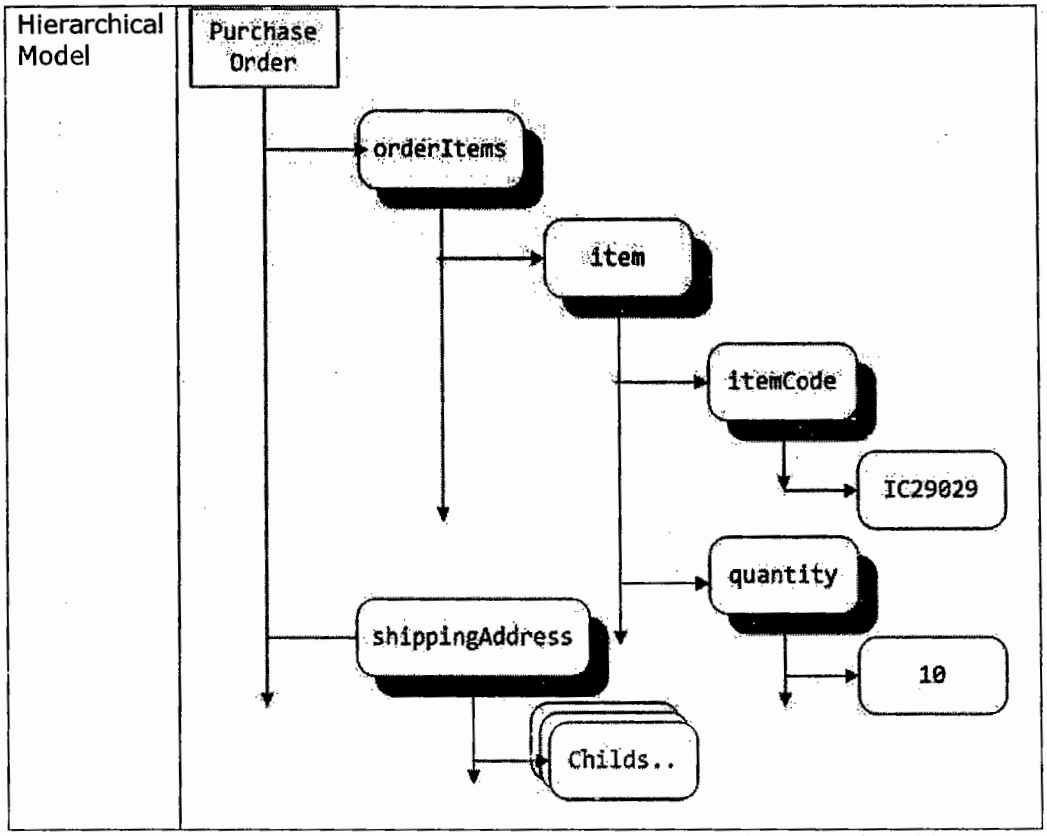
The below program will reads the XML document sequentially and prints it on to the console.

```
package com.saxparser.parser;

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

import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;

import org.xml.sax.SAXException;

import com.saxparser.handler.POHandler;

public class POParser {
	public static void main(String args[]) throws
ParserConfigurationException,
			SAXException, IOException {
		SAXParserFactory factory = SAXParserFactory.newInstance();
		SAXParser parser = factory.newSAXParser();
		parser.parse(new File(
					"D:\\WorkShop\\Trainings\\March,
2012\\Web Services\\SAXParser\\resources\\po.xml"), new POHandler());
		}
}
```

8.1.2 DOM (Document Object Model)

DOM stands for Document Object model, it allows us to read the XML documents in a Hierarchical fashion. Anything which is expressed in terms of parent and child which depicts tree like structure is called Hierarchical. So, DOM when given an XML document to it, it immediate reads the **entire** XML into the memory, loads it and builds a content tree (tree like structure containing elements/data) representing elements as parent and child. As it builds a tree structure in the memory, in DOM we can navigate between various elements in **random order** rather than sequential.

Input XML	<?xml version="1.0" encoding="UTF-8"?> <purchaseOrder> <orderItems> <item> <itemCode>IC29029</itemCode> <quantity>10</quantity> </item> </orderItems> <shippingAddress type="primary"> <addressLine1>Durga Soft Solutions Pvt Ltd</addressLine1> <addressLine2>S R Nagar</addressLine2> <city>Hyderabad</city> <state>AP</state> <zip>500038</zip> <country>India</country> </shippingAddress> </purchaseOrder>



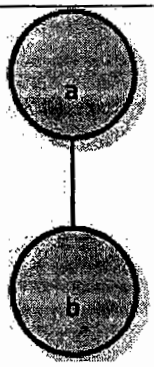
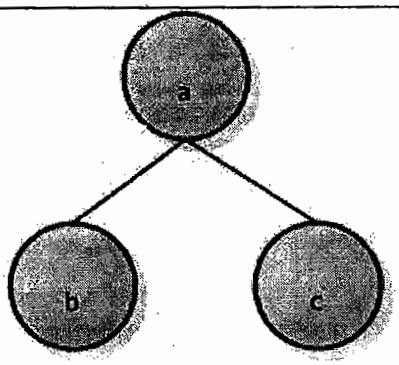
The above diagram shows for an Input XML how DOM loads and builds a tree structure representing elements and data.

As DOM loads the entire XML into memory at one shot, to load it takes some time and as it holds the complete XML in-memory, it consumes more amount of memory. That's the reason DOM is relatively slow when compared with SAX. The main advantage of DOM is unlike SAX, it is a read/write API which allows you to create/modify a document.

Anything in DOM is called a Node. There are several types of Node like Document Node, Element Node, Text Node and Empty node etc. The top of the document is called the Document Node, if an element which is enclosed in angular braces (<>) then it is called element node and if it just represents data then it is called Text node.

When we give XML document to DOM, it loads the XML and builds tree structure representing elements as nodes as described above. Then it places a pointer to the top level of the tree which is a Document Node.

To navigate between various elements in DOM, it has provided convenient methods by calling which we can traverse between them.

• **a** – is parent • **b** – is child of parent **a**	
• **a** – is parent • **b** and **c** – are children of **a** • **b** and **c** are called siblings, as those are children of same parent **a**	

Relation between nodes in DOM

As explained above except the root element all the other elements have a parent node. Children of same parent are called brother/sister in other words siblings.

JAX-P provides a way to work with these nodes in Java, JAX-P has provided methods like getFirstChild(), getChildNodes etc to access children of a parent. In the same way to access the siblings it has provided methods like getNextSiblings().

Before using the above methods, first of all we need to load the XML document as described earlier. An XML document in JAX-P DOM is represented in Document class object. As said earlier JAX-P is an API and org.w2c.dom.Document is an interface. So, we need to provide implementation for Document object and should instantiate by loading the XML into it.

So, we need a Builder which will read the existing XML, loads and places in the Document object, this would be done by DocumentBuilder, he is the person who knows how to read an existing XML document and loads in memory, along with that using DocumentBuilder we can create or modify existing documents as well (DOM is a read/write API so to support writing we need builder to build/modify document).

Unfortunately DocumentBuilder is an abstract class, so to instantiate it we need DocumentBuilderFactory. So, to summarize DocumentBuilderFactory knows → creating DocumentBuilder → Knows creating Document. The below piece of code shows how to load an existing XML document and represent in Document Object.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse(new File("po.xml"));
```

After creating the Document object, it points to the top of the Document which is called Document Node. Now we need to use methods to navigate between the elements of it.

Below sample program shows how to navigate to an element orderItems in the earlier XML.

Note: - we need to set factory.setIgnoringElementContentWhitespace(true) to ignore newlines and whitespaces in the document while parsing.

```
package com.domparser.parser;

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

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.xml.sax.SAXException;

public class PODomParser {
	public static void main(String args[]) throws
ParserConfigurationException,
			SAXException, IOException {
		DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
		DocumentBuilder builder = factory.newDocumentBuilder();
		Document poDocument = builder
				.parse(new File(
					"D:\\WorkShop\\Trainings\\March,
2012\\Web Services\\DOMParser\\resources\\po.xml"));

	System.out.println(poDocument.getFirstChild().getFirstChild().getNodeNa
me());
	}
}
```

Instead to print the entire XML document on to the console we need to traverse all the elements recursively. As you know DOM loads the XML document represents it in a Hierarchical fashion (Tree structure), any Hierarchical representations needs recursion technic as the things would be in parent and child nested format. If we take Binary tree, the tree traversal algorithms are In-order, pre-order and post-order which are recursive technic based traversal algorithms.

The below programs shows how to traverse between all the DOM nodes recursively.

```
package com.po.parser;

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

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

public class PODOMParser {
	private static final String XML_PATH =
"C:\\Workshop\\Trainings\\WebServices\\Batch12092012\\Web
Services\\PODOM\\resources\\po.xml";

	public static void main(String args[]) throws
ParserConfigurationException,
			SAXException, IOException {
		DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
		factory.setIgnoringElementContentWhitespace(true);
		DocumentBuilder builder = factory.newDocumentBuilder();
		Document doc = builder.parse(new File(XML_PATH));

	//System.out.println(doc.getFirstChild().getFirstChild().getNodeName());
		printNode(doc);
	}
```

Contd.....

```
        private static void printNode(Node node) {
                if (node != null) {
                        int type = node.getNodeType();
                        switch (type) {
                        case Node.DOCUMENT_NODE:
                                System.out.println("<?xml version='1.0'?>");
                                printNode(node.getFirstChild());
                                break;
                        case Node.ELEMENT_NODE:
                                System.out.print("<"+node.getNodeName()+">");
                                NodeList childrens = node.getChildNodes();
                                if(childrens != null) {
                                        for(int i=0;i<childrens.getLength();i++) {
                                                Node child = childrens.item(i);
                                                printNode(child);
                                        }
                                }
                                System.out.println("</"+node.getNodeName()+">");
                                break;
                        case Node.TEXT_NODE:
                                System.out.print(node.getNodeValue());
                                break;
                        }
                }
        }
}
```

8.1.3 Validating XML with an XSD using JAX-P

While parsing an XML, the parser only checks for well-formness of an XML it never checks whether it is valid or invalid. But in a typical business transaction we should first validate the XML before parsing it.

To validate an XML we need either a DTD/XSD. Now we are trying to understand how to validate an XML with an XSD in java using JAX-P API.

To validate an XML first of all we need to load the contents of XSD and keep in a java object called Schema. Schema is coming from JAX-P API which means it is an abstract class. The implementation for the Abstract class Schema will be there with resp implementation, in our case with Xerces2.

As we don't know who is the implementation for Schema class we need one more factory class called SchemaFactory. While creating the SchemaFactory we need to pass the input as W3C XML Schema Namespace URI as input, to indicate we are going to read the contents of XML Schema compliant XSD.

Once we create the SchemaFactory we can call newSchema method on it by passing File pointing to XSD as input.

After creating the Schema object call a method newValidator() will gives you the validator object. On this object you can call the method called validate by passing XML as input.

If the XML passed is invalid, it throws exception otherwise will not return anything.

```
SchemaFactory schemaFactory =
SchemaFactory.newInstance(XMLConstants.W3C_XML_NS_URI);

Schema poSchema = schemaFactory.newSchema(new File(XSD_PATH));

Validator v = poSchema.newValidator();

v.validate(new StreamSource(new File(XML_PATH));
```

8.1.4 Difference between SAX and DOM

Till now we understood what is SAX and DOM and how to process them using JAX-P. Let us understand what the differences between them are.

SAX	DOM
• SAX stands for Simple Access for XML API • It is an event based processing model • It process the XML sequentially one after the other from top to bottom • It consumes less amount of memory as at any point of time loads only one element into the memory • It is faster when compared with DOM as it doesn't need to load or form anything. • SAX is an readonly API, which means we can just read the XML documents, but we cannot modify or create new.	• DOM stands for [illegible]ent Object Model. • It is a [illegible]essing model, where [illegible] the entire XML in memory and represents in tree structure. • DOM supports random access of elements. • It consumes huge amount of memory as it loads the entire XML and forms tree structure (Generally for 1kb of xml document it takes 10kb memory to hold in memory, which means it multiplies by 10X) • It is slow when compared with SAX as it has to represent the XML by loading it. • DOM is a read and write API using which we can read/modify/create an XML document.

In order to overcome the both the dis-advantages in SAX and DOM new processing model came which is STAX, stands for Streaming API for XML. This is unlike SAX event based or DOM hierarchical based, it is a pull based event parser.

JAX-B

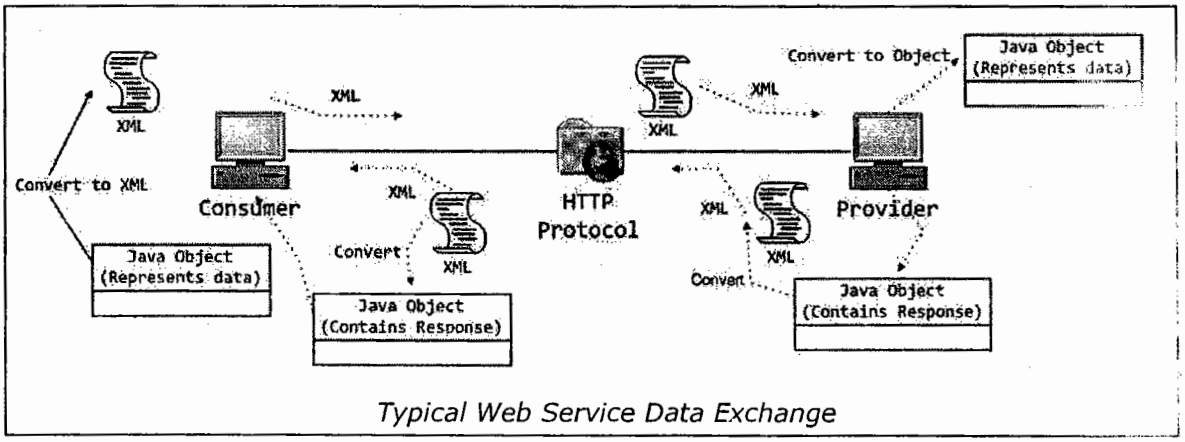
9 JAX-B

JAX-B stands for Java Architecture for XML Binding; it is the API that is used for Binding Java Object to XML document and XML documents to Java Object back.

Let us try to understand, where we can use JAX-B in a typical Web service communication. As we know in Web service communication always they involve two actors Consumer and Provider. Consumer could be written in any language and even the Provider can also be coded in any language.

If we consider a scenario where consumer and provider both are coded in Java Language and now the consumer wants to send some information to the provider. As consumer is a java program he holds the information in-terms of Java Object, so in order to send the Java Object data over the network, he needs to serialize the contents of that object into bits and bytes and has to send to Provider. Up on receiving the serialized stream of data, the Provider will de-serialize and build back the object for processing. The problem with this approach is Serialization is the protocol proprietary for Java language and non-Java programs cannot understand serialized data send by java application. So, this breaks the thumb rule Interoperability.

So, if the consumer wants to send some information to the provider rather than converting it into serialized stream of data, he should convert the Java Object data to XML representation and should send to Provider. Now provider upon receiving the XML he has to convert it to Java Object and process it. Again Provider instead of returning Java Object as response, he should convert the response Object to XML and should send it to consumer as response. The consumer has to convert that incoming XML data to Object to use it. This is depicted in the below diagram.



Typical Web Service Data Exchange

By now we understood we should not use Object serialization for exchanging data in web services rather we should convert them to XML. In-order to work with XML java has already provided API's for dealing with XML data which is JAX-P. In JAX-P we can use either SAX or DOM parser to convert Java to XML and wise versa.

Even though we can achieve this with JAX-P, a programmer has to write manually the parser's dealing with this, and more over there are limitations with JAX-P.

Let's say if a Java Program wants to modify a Purchase Order XML document data, it contains several things like OrderItems, Items and ShippingAddress information. Rather than dealing them as elements, attributes or Text node, it would be easy for a Java Programmer to look at them in-terms of Java Objects with data and modify them than modifying them as XML. This can be accomplished by using the XML Binding API's.

Initially Java doesn't have XML Binding supported API's; there are lot of third party XML binding libraries are released by software vendors in Java like JibX, Castor, XML Beans, Apache Data binding API, Javolution etc. Then SUN has realized and released its own API for dealing with XML Binding in Java "JAX-B API". SUN has even released an implementations for it JAX-B RI (Reference Implementation) along with that we have one open source implementations for JAX-B API, JaxMe.

Basically XML Binding API's are categorized into two types of tools.

a) Design Time tools – In these type of tools, there will be an Binding compiler which will take a Schema or DTD as input and generated supported classes for converting the XML to Object and Object to XML

b) Runtime tools – These will not generate any classes rather on fly will take the XML as an input and based the elements names, matches them with corresponding attribute names of the class and will convert XML into Object of the class.

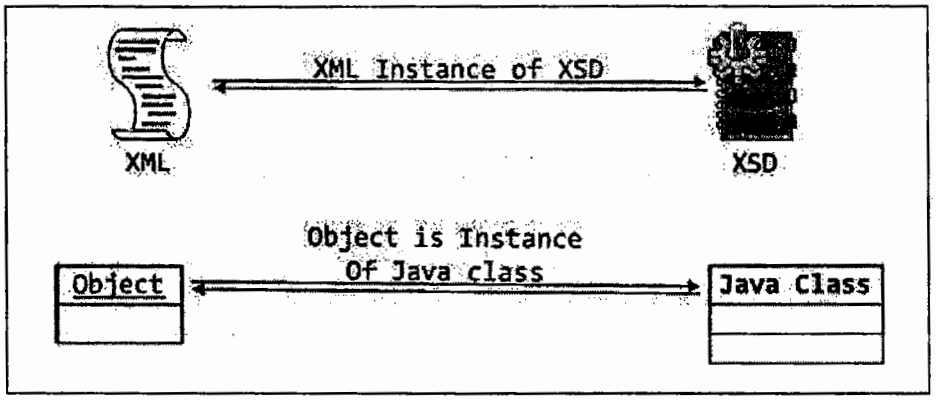
When it comes to JAX-B API it is a design time tool, where it will come up with its own compiler to generated classes to support the binding process. Before proceeding further on JAX-B let us first try to understand its architecture and its offerings.

9.1 Architecture

Most of the XML Binding API's architectures are similar to, what we are discussing now.

If a Java Object is following the structure of a Java class then that Object is called Instance of that class. If XML document is following the structure of an XSD document can I call the XML as an Instance of that XSD or not? Certainly the answer for this would be yes.

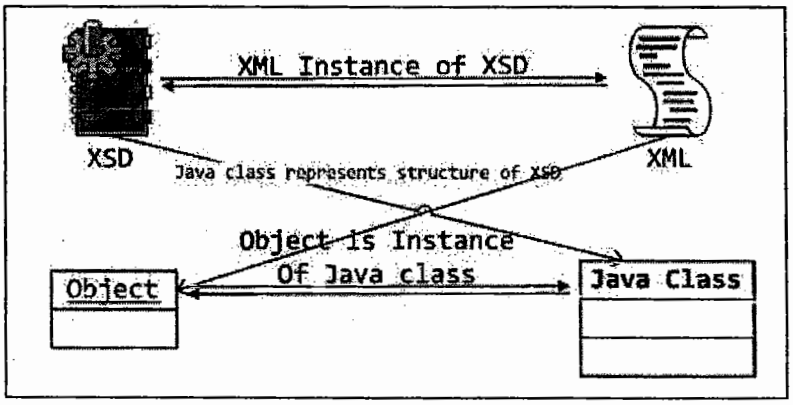
In Java, Object structure would be defined by a class. In XML, structure of an XML is defined by an XSD document as shown below.



If we try to derive a logical conclusion from the above, we can understand that Classes and XSD documents both try to represent the structure, so we can create set of Java classes which represents the structure of an XSD document, then the XML document (holds data) representing the structure of that XSD can be converted into its associated Java class Objects. Even the wise versa would also be possible.

Let us try to deep down into the above point. How to created user defined data type in Java? Using class declaration. How to created user defined data type in XSD? By using ComplexType declaration. Which means a ComplexType in XSD is equal to a Java class in Java. As XSD document means it contains elements and multiple ComplexType's so, to represent the structure of an XSD document in Java, it is nothing but composition of various Java classes (one Java class → one XSD ComplexType) representing its structure.

The below diagram shows a pictorial representation of the above description.



Let us take the example of PurchaseOrder XSD to understand better.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="purchaseOrder" type="at:purchaseOrderType"/>
   <xs:complexType name="purchaseOrderType">
         <xs:sequence>
               <xs:element name="orderItems" type="at:orderItemsType"/>
               <xs:element name="shippingAddress"
type="at:shippingAddressType"/>
         </xs:sequence>
   </xs:complexType>
   <xs:complexType name="orderItemsType">
         <xs:sequence>
               <xs:element name="item" type="at:itemType" minOccurs="1"
maxOccurs="unbounded"/>
         </xs:sequence>
   </xs:complexType>
   <xs:complexType name="itemType">
         <xs:sequence>
               <xs:element name="itemCode" type="xs:string"/>
               <xs:element name="quantity" type="xs:int"/>
         </xs:sequence>
   </xs:complexType>
   <xs:complexType name="shippingAddressType">
         <xs:sequence>
               <xs:element name="addressLine1" type="xs:string"/>
               <xs:element name="addressLine2" type="xs:string"/>
               <xs:element name="city" type="xs:string"/>
               <xs:element name="state" type="xs:string"/>
               <xs:element name="zip" type="xs:int"/>
               <xs:element name="country" type="xs:string"/>
         </xs:sequence>
   </xs:complexType>
</xs:schema>
```

In the above XSD we have one element and four ComplexType declarations, as we discussed earlier ComplexType declarations in XSD are equal to a Java class in Java. So from the above we can derive four Java classes as shown below.

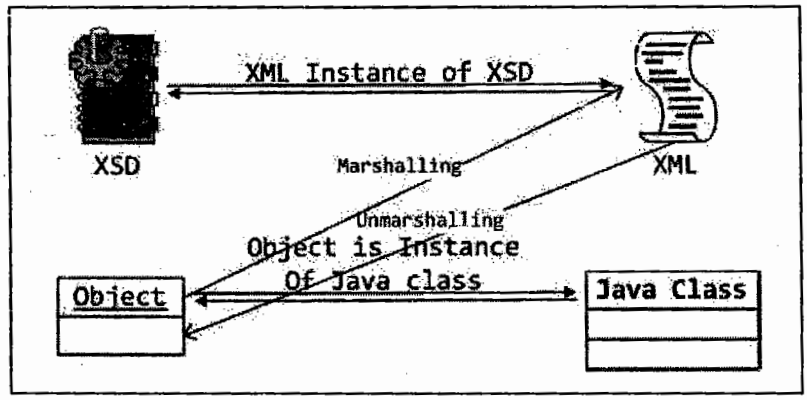
class ShippingAddressType {
String addressLine1;
String addressLine2;
String city;
String state;
int zip;
String country;
}
	class ItemType {
	String itemCode;
	int quantity;
	}

class OrderItemsType {
List<ItemType> item;
}
	class PurchaseOrderType {
	OrderItemsType orderItems;
	ShippingAddressType shippingAddress;
	}

From the above we can create the PurchaseOrderType object representing the root element of XSD, can I hold the PurchaseOrder XML document data into this instance or not? Absolutely the answer would be yes, because PurchaseOrderType is representing the structure of the XSD document, so its instance can hold its XML data.

With this we understood that we can created the Java classes representing the structure of an XSD document, hence those classes are called as Binding Classes as those are bonded to the XSD document structure and the objects out of those classes are called binding objects.

Now we know that we can convert the XML document data into a Binding Class Object, so the process of converting an XML to a Binding class Object is called Un-Marshalling. And the process of converting a Binding Object back into XML representation is called Marshalling and both the types of operations are supported by any of the XML Binding API's including JAX-B.



The above architecture is the generalized architecture which can be applied to any XML Binding API's including JAX-B.

With our earlier discussions we understood that a typical XML Binding API should support Un-Marshalling and Marshalling operations. But to perform Un-Marshalling and Marshalling Operations do we need binding classes or not?

So, JAX-B supports two types of operations 1) One-Time operations 2) Runtime operations. Let us discuss them in detail below.

9.2 One-Time Operation

One-Time operations are the operations which would be done only once within the lifetime of an XSD or Binding class.

Let's say I want to convert the Purchase Order XML document to Object, this is Un-Marshalling. To perform Un-Marshalling do we need Binding classes representing the structure of the Purchase Order XSD or not? So the programmer has to read the contents of the XSD and based on the number of complex types he needs to create equivalent Java classes representing their structure. As our purchase order XSD document contains only four complex types it would be easy for the developer to determine the relations between the classes and can create. Let's say I have an XSD document in which 100 ComplexType declarations are there, is it easy for a developer to develop 100 Java classes representing the structure of 100 ComplexTypes. Even though it seems to be feasible, but it is a time taking task.

So, instead of writing the Binding classes manually, JAX-B has provided a tool to generated Binding classes from XSD called XJC. And to generate XSD from a Binding class there is another tool called Schemagen.

If I have an XSD document, how many times do I need to generate Java classes to hold the data of its XML? It would be only once, because classes will not hold data they just represents structure, so instances of those classes holds data of an XML. Even the reverse would also be same. As we need to generate the Binding classes or XSD only once this process is called one-time operations. And there are two types of one-time operations.

a) XJC – XSD to Java compiler – Generates Binding classes from XSD document

b) Schemagen – Schema generator – Generated XSD document from an Binding class

Point to recollect: - We said JAX-B is a design time tool, if you see now it has a binding compiler which will generates classes at design time.

9.2.1 How to use XJC or Schemagen?

XJC or Schemagen are the tools that will come when we install JWSDP. JWSDP stands for Java Web Service Developer Pack and it is software which offers development tools and libraries with which we can work on Web Services related technology applications (for example JAX-P, JAX-B, SAAJ, FastInfoset, JAX-RPC and JAX-WS etc).

When we install JWSDP by default it would be installed under c:\Sun\JWSDP-2.0. Under this directory it will have one more directory called JAXB this is the directory under which all the JAXB related tools and jars would be located.

```
➢ C:\Sun\JWSDP-2.0
   |-- JAXB
        | -- lib
        | -- bin
             | -- xjc.bat, xjc.sh, schemagen.bat and schemagen.sh
```

From this directory I can run xjc.bat or schemagen.bat to generate Binding classes or XSD document respectively. Most of the times we use xjc tool rather than schemagen tool. Because we will start with XSD most of the time to work on XML binding API's. So let us first try to understand how to execute XJC, the same can be even carried for schemagen as well.

XJC compiler will take input as XSD and will generates set of binding classes. Now where should I generate these binding classes, do I need to generate them in the JAXB\bin directory or by Project Folder "src" directory? The answer would be should generated in your project source directory. So to generate under source directory of your project you need to run the XJC.bat from the Project directory rather that the JAXB\bin directory. To run it from project directory we need to set the system path pointing to it as shown below.

```
➢ C:> set path=%path%;c:\sun\jwsdp-2.0\jaxb\bin
```

After setting the path, we need to switch to the project directory. For example if your project is under D:\ drive and the folder is Sample you need to switch to the Sample directory under D:\ drive and should execute the XJC.bat by giving XML as input as shown below.

```
➢ C:> cd D:\Sample
➢ Xjc –d src resources\po.xsd
```

In the above command we are saying take the po.xsd as an input under resources directory (from the current directory) and generate the binding classes under (-d represents the directory) src directory in my Sample Project.

9.2.2 What does XJC generates?

When we run XJC.bat by giving XSD as an input along with generating binding classes it generates few other things as explained below.

a) Generates Package based on the Namespace – In an XSD document, it would contain an XSD namespace declaration. Namespaces in XSD similar to packages in Java. So when XJC is generating Binding class for every ComplexType declaration. It would have to even consider mapping the XSD Namespace to package name and should generate the classes under that mapped package (because the ComplexType declared under targetnamespace are bonded to Namespace, so those classes have to be bonded to packages). XJC by default will uses a mechanism to map a Namespace to package in Java as shown below.

TargetNamespace in XSD – http://ebay.in/sales/types
Package Name in Java –

i. ignore http:// or www
ii. Find the first "/" forward slash and extract the substring (ebay.in)
iii. Now delimit by "." (dot) and extract tokens in reverse order in.ebay
iv. In the remaining string "sales/types" delimit by "/" forward slash and append them to the first constructed package name
v. The resulted package name would become in.ebay.sales.types.

All the binding classes would be generated into the above package

b) Package-info.java – XJC after mapping the namespace to a package name, it would store this mapping information into a class called package-info.java. This is also generated under the above mapped package only.
c) ObjectFactory.java – These Class Acts as factory class, which knows how to create objects of all the Binding classes generated. It contains methods like createXXX, upon calling these methods it would return respective binding class objects.
d) Set of Binding classes – With the above it generates Binding classes representing the structure of each ComplexType in XSD. Can we call any java class as Binding class? No, the binding classes are the classes which are bonded to XSD complexType declarations and to denote this relationship, those are marked with JAX-B annotations as described below.

i. @XMLType – To the top of the binding class it would be marked with @XMLType annotation indicated this class is representing an XML Complex Type
ii. @XMLAccessorType(AccessType.FIELD) – In conjunction with @XMLType, one more annotation @XMLAccessorType(AccessType.FIELD) also would be generated. It denotes how to map XML elements to Binding class attributes. With the AccessType.FIELD we are saying the elements names must be matched with binding class attributes names to port the data.
iii. @XMLElement – This annotation is marked at the attributes of the Binding classes to indicates these attributes are representing elements of XML.

So, by the above we understood that not every Java class is called as Binding class, only the classes which are annotated with JAX-B annotations are called binging classes.

So from the above binding classes we can even generated XSD document by running the Schemagen tool as well, this is left to you as exercise.

With this we understood how to perform one-time operations. Now we can proceed on runtime operations.

9.3 Runtime Operation

Till now we discussed about how to perform one-time operations. Let us try to understand Runtime operations. JAX-B supports three runtime operations.

a) Un-Marshalling – The process of converting XML to Java Object

b) Marshalling – The process of converting Java Object with data to its XML equivalent

c) In-Memory Validation – Before converting an XML document to Java Object first we need to validate whether the XML conforms to XSD or not would be done using In-Memory Validation.

If we want to perform runtime operations, first you need to do one-time operation. Unless you have binding classes or XSD equivalent for any existing binding class, you can't work on runtime operations.

By following the things described in earlier section we assume you already completed the One-Time operations and you have binding classes with you. Let us try to understand how to work on each of the Runtime operations in the following section.

9.3.1 Un-Marshalling

As described earlier, the process of converting an XML document into Java Object is called Un-Marshalling. In order to perform Un-Marshalling, you need to have Binding Classes with you. You can generate these by running XJC compiler. If you run XJC compiler on the po.xsd which was described earlier, the following binding classes will be generated shown below.

```
package org.amazon.types.sales;

@XmlAccessorType(AccessType.FIELD)
@XmlType(name = "shippingAddressType", propOrder = {
    "addressLine1",
    "addressLine2",
    "city",
    "state",
    "zip",
    "country"
})
public class ShippingAddressType {

    @XmlElement(namespace = "http://amazon.org/types/sales")
    protected String addressLine1;
    @XmlElement(namespace = "http://amazon.org/types/sales")
    protected String addressLine2;
    @XmlElement(namespace = "http://amazon.org/types/sales")
    protected String city;
    @XmlElement(namespace = "http://amazon.org/types/sales")
    protected String state;
    @XmlElement(namespace = "http://amazon.org/types/sales", type =
Integer.class)
    protected int zip;
    @XmlElement(namespace = "http://amazon.org/types/sales")
    protected String country;

    // setters and getters
}
```

```
package org.amazon.types.sales;

@XmlAccessorType(AccessType.FIELD)
@XmlType(name = "itemType", propOrder = {
    "itemCode",
    "quantity"
})
public class ItemType {

    @XmlElement(namespace = "http://amazon.org/types/sales")
    protected String itemCode;
    @XmlElement(namespace = "http://amazon.org/types/sales", type =
Integer.class)
    protected int quantity;

    // setters and getters
}
```

```
package org.amazon.types.sales;
@XmlAccessorType(AccessType.FIELD)
```

```
@XmlType(name = "orderItemsType", propOrder = {
    "item"
})
public class OrderItemsType {

    @XmlElement(namespace = "http://amazon.org/types/sales")
    protected List<ItemType> item;

    public List<ItemType> getItem() {
        if (item == null) {
            item = new ArrayList<ItemType>();
        }
        return this.item;
    }

}
```

```
package org.amazon.types.sales;

@XmlRootElement
@XmlAccessorType(AccessType.FIELD)
@XmlType(name = "purchaseOrderType", propOrder = {
    "orderItems",
    "shippingAddress"
})
public class PurchaseOrderType {

    @XmlElement(namespace = "http://amazon.org/types/sales")
    protected OrderItemsType orderItems;
    @XmlElement(namespace = "http://amazon.org/types/sales")
    protected ShippingAddressType shippingAddress;


}
```

Package-Info.java

```
@javax.xml.bind.annotation.XmlSchema(namespace =
"http://amazon.org/types/sales")
package org.amazon.types.sales;
```

```
package org.amazon.types.sales;
@XmlRegistry
public class ObjectFactory {

   private final static QName _PurchaseOrder_QNAME = new
QName("http://amazon.org/types/sales", "purchaseOrder");

   public ObjectFactory() {
   }
   public ShippingAddressType createShippingAddressType() {
      return new ShippingAddressType();
   }
   public OrderItemsType createOrderItemsType() {
      return new OrderItemsType();
   }
   public PurchaseOrderType createPurchaseOrderType() {
      return new PurchaseOrderType();
   }

   public ItemType createItemType() {
      return new ItemType();
   }

   @XmlElementDecl(namespace = "http://amazon.org/types/sales",
name = "purchaseOrder")
   public JAXBElement<PurchaseOrderType>
createPurchaseOrder(PurchaseOrderType value) {
      return new
JAXBElement<PurchaseOrderType>(_PurchaseOrder_QNAME,
PurchaseOrderType.class, null, value);
   }

}
```

Having the binding classes will not automatically converts the XML document into Object of these classes, rather we need a mediator who would be able to read the contents of the XML, and identify the respective binding class for that and creates objects to populate data. This will be done by a class called "Unmarshaller". As JAX-B is an API, so Unmarshaller is an interface defined by JAX-B API and its JAX-B API implementations (JAX-B RI or JaxMe) contains the implementation class for this interface.

The object of Unmarshaller has a method unmarshal() , this method takes the input as XML document and first check for Well-formness. Once the document is well-formed, its reads the elements of the XML and tries to identify the binding classes for those respective elements based on @XMLType and @XMLElement annotations and converts them into the Objects.

We have two problems here

1) How to created Un-marshaller as we don't know implementation class

2) Where does un-marshaller should search for identifying a binding class (is it in entire JVM memory or classpath?)

As we don't know how to identify the implementation of Unmarshaller interface, we have a factory class called JAXBContext who will finds the implementation class for Unmarshaller interface and would instantiate. It has a method createUnmarshaller() which returns the object of unmarshaller. This addressed the concern #1.

While creating the JAXBContext, we need to supply any of the three inputs stated below.

1) Package name containing the binding classes

2) ObjectFactory.class reference as it knows the binding classes.

3) Individual Binding classes as inputs to the JAXBContext.

```
JAXBContext jaxbContext = JAXBContext.newInstance("packagename" or
"ObjectFactory.class" or "BC1, BC2, BC3...");
```

With the above statement, the JAXBContext would be created and loads the classes that are specified in the package or referred by ObjectFactory or the individual binding classes specified and creates a in-memory context holding these.

Now when we call the Unmarshaller.unmarshal() method, the Unmarshaller will searches with the classes loaded in the JAXBContext and tries to identifies the classes for an XML element based on their annotations. The code for UnMarshalling has been shown below.

```
package com.po.unmarshall;

public class POUnmarshaller {
	public static void main(String args[]) throws SAXException {
		try {
			JAXBContext jContext = JAXBContext
					.newInstance("org.amazon.types.sales");
			Unmarshaller poUnmarshaller =
jContext.createUnmarshaller();
			JAXBElement<PurchaseOrderType> jElement =
(JAXBElement<PurchaseOrderType>) poUnmarshaller
					.unmarshal(new File(
							"D:\\WorkShop\\Trainings\\Aug,
2011\\Web Services\\POJaxB\\resources\\po.xml"));
			PurchaseOrderType poType = jElement.getValue();

	System.out.println(poType.getShippingAddress().getAddressLine1());
			System.out.println(poType.getOrderItems().getItem().get(0)
					.getItemCode());
		} catch (JAXBException e) {
			e.printStackTrace();
		}

	}
}
```

9.3.2 Marshalling

Marshalling is the process of converting the Object data to XML. For performing Marshalling also we need to first of all have the Binding classes. Once we have the Binding classes, we need to create the Object of these classes and should populate data in them.

These objects can be marshaled to XML using the Marshaller. Again Marshaller is an interface, to instantiate it we need JAXBContext. Then we need to call a method marshall by giving input as XML and Outputstream where we want to generate the output to. The below program depicts the same.

```
package com.po.unmarshall;

public class POMarshaller {
	public static void main(String args[]) {
		try {
			JAXBContext jContext = JAXBContext
					.newInstance("org.amazon.types.sales");
			Marshaller poMarshaller = jContext.createMarshaller();

			ShippingAddressType shippingAddress = new
ShippingAddressType();
			shippingAddress.setAddressLine1("121/A");
			shippingAddress.setAddressLine2("Punjagutta PS");
			shippingAddress.setCity("Hyd");
			shippingAddress.setState("AP");
			shippingAddress.setCountry("India");
			shippingAddress.setZip(24244);

			ItemType item1 = new ItemType();
			item1.setItemCode("IC2002");
			item1.setQuantity(34);

			ItemType item2 = new ItemType();
			item2.setItemCode("IC2003");
			item2.setQuantity(1);

			OrderItemsType oit = new OrderItemsType();
			oit.getItem().add(item1);
			oit.getItem().add(item2);

			PurchaseOrderType pot = new PurchaseOrderType();
			pot.setOrderItems(oit);
			pot.setShippingAddress(shippingAddress);

			poMarshaller.marshal(pot, System.out);
		} catch (JAXBException e) {
			e.printStackTrace();
		}
	}
}
```

9.3.3 In-Memory Validation

While performing the Unmarshalling or Marshalling, we need to first of all validate whether the XML is valid against the XSD document or not, and we will validate using the In-Memory validation.

An XML can be validated against an XSD document, so we need to read the XSD document and should represent it in a Java Object called Schema.

But Schema is an interface and to create the Object of interface, we need factory called SchemaFactory. But schema factory can created various types of XSD documents, so we need to provide the grammar of XSD document which you want to load or created and would be done as show below.

```
SchemaFactory sFactory =
SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_URI);
```

With the above we created a SchemaFactory which is capable of creating or loading W3C XML Schema type documents.

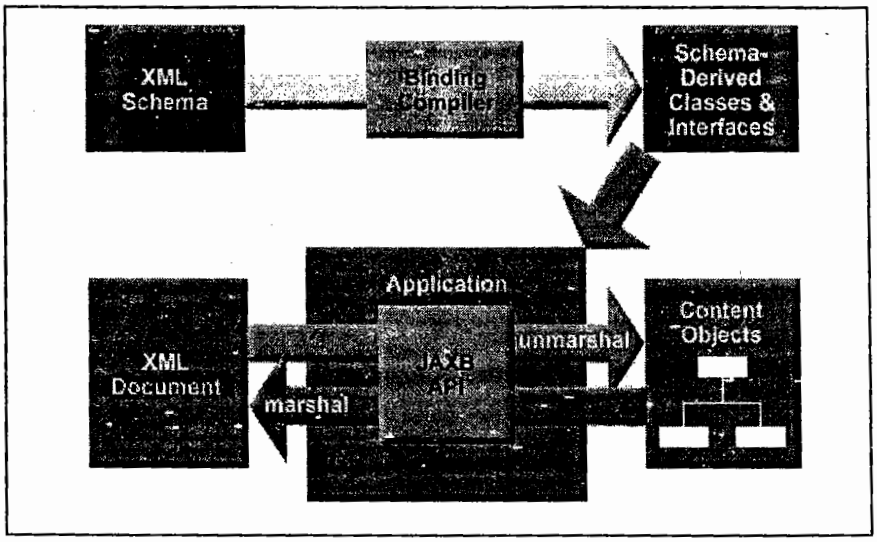
Now we need to load the XSD document into the Schema class object as shown below.

```
Schema poSchema = sFactory.newSchema(new File("po.xsd"));
```

Once the schema object is created holding the structure of XSD, we need to give it to Unmarshaller before calling the unmarshal(). So, that Unmarshaller will validate the input XML with this supplied Schema Object and then performs Unmarshalling. Otherwise Unmarshaller will throw exception, indicating the type of validation failure. The below programs shows how to perform in-memory validation.

```
package com.po.unmarshall;

public class POUnmarshaller {
        public static void main(String args[]) throws SAXException {
                SchemaFactory sFactory = SchemaFactory

        .newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
                Schema poSchema = sFactory
                                .newSchema(new File("po.xsd"));

                JAXBContext jContext = JAXBContext
                                .newInstance("org.amazon.types.sales");
                Unmarshaller poUnmarshaller = jContext.createUnmarshaller();
                poUnmarshaller.setSchema(poSchema);
                JAXBElement<PurchaseOrderType> jElement =
(JAXBElement<PurchaseOrderType>) poUnmarshaller
                                .unmarshal(new File("po.xml"));
                PurchaseOrderType poType = jElement.getValue();

        System.out.println(poType.getShippingAddress().getAddressLine1());
                System.out.println(poType.getOrderItems().getItem().get(0)
                                .getItemCode());
        }
}
```

Finally if you look at the JAX-B Architecture if would be see as below.


XML Schema
Binding Compiler
Schema-Derived Classes & Interfaces
Application
XML Document
JAXB API
marshal
unmarshal
Content Objects

JAX-RPC

10 Getting Started With Web Services

Before discussing about JAX-RPC API based Web Service development, first we need to understand certain aspects related to Web Services. The points we are discussing below are common for both JAX-RPC and JAX-WS as well.

10.1 Types of Web Services

As discussed earlier, we have two types of Web Services JAX-RPC API adhering the BP 1.0, and JAX-WS API adhering to BP 1.1. Now while developing a Web service, first of we need to choose a type. After choosing the type, we need to select an appropriate implementation and various other aspects described in following sections.

10.2 Web Service Development parts

Always a Web Service development involves two parts. Provider and Consumer. Provider is the actual Web service, who will handle the incoming requests from the consumer. The Consumer is the Client Program which we develop to access the Provider. Now both JAX-RPC and JAX-WS API's has provided classes and interfaces to develop Consumer as well as Provider programs.

10.3 Ways of developing a Web Service

There are two ways of developing a Web service

10.3.1 Contract First approach

In Web Services, WSDL acts as a contract between Consumer and Provider. If consumer wants to know the information about the provider, he needs to see the WSDL document. In a Contract First approach, the development of the provider will starts with WSDL, from which the necessary Java classes required to build the service would be generated. As the development starts from WSDL, hence it is called Contract First approach.

10.3.2 Contract Last approach

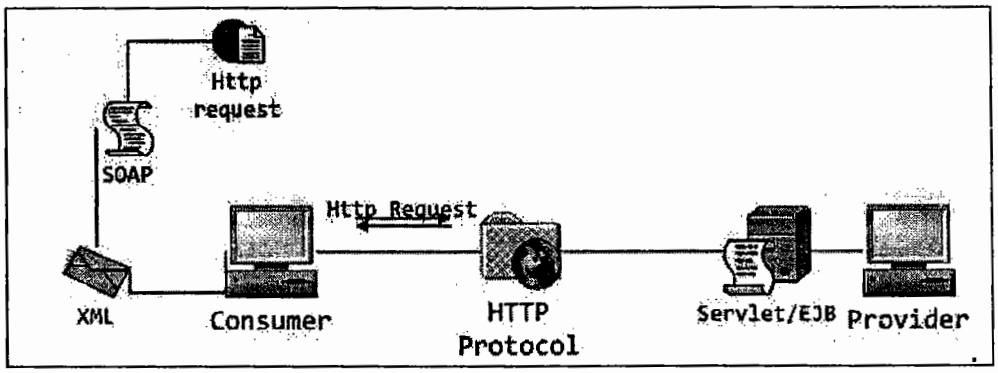
Here the developer first starts developing the service by writing the Java code, once done then he will generates the WSDL contract to expose it as a Web Service. As the contract is being generated at the end after the development is done, this is called Contract Last approach.

10.4 Choosing an Endpoint

In Web services the provider is often referred as Endpoint, as he is the ultimate point from whom we request data or information.

Now in a typical Web Service communication, the consumer will sends the initial request with some data to the Provider. The consumer cannot send Object data over the network; rather he needs to convert it into XML and needs to place the XML into SOAP XML. Then the SOAP XML is being transported over the HTTP protocol. This means the consumer will send a HTTP request, in order to receive the HTTP request being sent by the consumer, at the Provider end we need to

have a HTTP Listener. The listener will listens for the incoming request and forwards it for further processing as shown below



So we understood we need a listener at the provider side to handle the incoming request. Do we have any existing API's in Java which are capable of handling incoming HTTP Requests? Those are nothing but Servlet or EJB.

There are two types of Endpoints, where a developer can choose while developing a Provider.

10.4.1 Servlet Endpoint

Here at the provider end, the servlet acts as a listener in listening for the incoming request from the client and would process.

10.4.2 EJB Endpoint

In place of Servlet even we can substitute it with EJB, and we can make EJB's to handle even HTTP Request as well.

Most of the people prefer to use Servlet endpoint rather than EJB endpoint as EJB seems to be heavy and costly component, and for the sake of developing Web services, the application needs to be hosted on an EJB container.

10.5 Message Exchange Patterns

Message exchange format talks about how a consumer does exchanges information with the provider. This could happen in three ways as described below.

10.5.1 Synchronous request/reply

In this pattern, the consumer sends the request with some data and will waits till the provider finishes processing request and returns the response to the consumer.

For e.g. let's take two classes A and B as shown below

```
public class A{
   public void do() {
      B b = new B();
      int I b.execute(); // blocked
      // some other processing
   }
}
```

```
public B{
   public int execute() {
      // some logic
      return 33;
   }
}
```

Class A is calling a method execute() on B, until the method execute() finishes execution and returns value, the do() method will be blocked on the execute() method call, as it is blocked on method call, it is called blocking request/reply or synchronous request reply.

In this case also the Consumer program would be blocked on the Providers method call (request) and will not proceeds execution until the Provider returns response. Which means the Consumer opened HTTP Connection would be open till the Provider returns response.

10.5.2 Asynchronous request/reply or Delayed Response

In this pattern, the Consumer will not be blocked on the Providers method call/request, rather after Consumer sending the request; he disconnects/closes the HTTP Connection. The provider upon receiving the request will hold it and process it after certain amount of time. Once the response is ready will opens a new connection back to the Consumer and dispatches the response.

For e.g. let us consider an example like you are sending a letter to your friend, when you post the letter it would be carried by the postmaster to your friend. Once the letter has been received, your friend may not immediately read the letter, rather when he finds leisure he will read. Till your friend reads the letter and give the reply the postmaster can't wait at your friends door rather he moves on. Now when your friend reads your letter, then he will reply to your letter by identifying the from address on the letter and post back's you the reply.

As the response here is not immediate and is being sent after sometime, this is also called as Delayed response pattern.

10.5.3 One way Invoke or Fire and forget

In this case, the consumer will send the request with some data, but never expects a response from the provider. As the communication happens in only one direction it is called One-Way invoke as well.

10.6 Message Exchange formats

This describes about how the consumer will prepares/formats the xml message that should be exchanged with the provider.

For e.g. a lecturer is dictating the notes, every student has their own way of writing the notes. Someone may right the notes from bottom to top, others might right from top to bottom.

In the same way, in order to send information, the consumer has to prepare his data in XML format. How does consumer represents the data in the XML document is called Message Exchange format.

The message exchange format would be derived by two parts 1) Style and other is 2) Use.

a) Style – The possible values the Style part can carry is rpc and document

b) Use – This could be encoded and literal

With the above it results in four combinations as rpc-encoded, rpc-literal, document-literal and document-encoded. Out of which document-encoded is not supported by any web service specification.

We will discuss the remaining three in later sections.

11 JAXRPC API (SI Implementation)

11.1 Building Provider

In this section we will try to understand how to work with JAX-RPC API Web services using JAX-RPC SI (reference implementation provided by SUN) Implementation.

By default any JAX-RPC API based web service uses the default message exchange pattern as RPC-Encoded.

Before developing a consumer, first they should be a provider to access it. So, let us start first with Provider Development. As you know there are multiple things you need to consider while building provider, we are developing here JAX-RPC SI – Contract Last – Servlet Endpoint – Sync req/repl – RPC-Encoded Service.

11.1.1 Contract Last (Servlet Endpoint, Sync req/reply with rpc-encoded)

As you know always from architecture point of view, the contract between provider and consumer is WSDL. Here always the service development would be started with Java and at the end, contract would generated, as the contract is being generated at the end, it is called contract last approach.

As we are Java programmers, it would be easy for us to start with Java rather than with WSDL, so we are trying to understand development of the provider using Contract Last.

Let us try to under step by step, how to develop a JAX-RPC SI service in contract last approach.

a) **Write SEI Interface** – Always from an architecture point of view, the contract between the consumer and the provider is WSDL. From Java point of view, the contract between consumer and the provider is Interface. How does the consumer knows the number of operations exposed by the provider, declaratively he can see it in WSDL and from program perspective he can refer the Interface.

   The Web service development always starts with SEI interface, stands for Service Endpoint Interface. As it describes the entire information about the Provider (service), it is called Service Endpoint Interface. It acts as a contract between Consumer and Provider. All the methods declared in the SEI interface are exposed as Web Service methods. There are certain rules you need to follow while writing SEI interface in JAX-RPC API described below.

   i) SEI interface must extend from java.rmi.Remote interface – The methods declared in this interface are by default exposed over the network and can be accessed remotely, to indicate that, it has to be extended from java.rmi.Remote. In addition to this, to differentiate a normal interface to a remotely accessible interface, it has to extend from Remote.

ii) Declare all the methods which you want to expose as Remote methods – If you want to expose five methods as Web service methods, all those five methods has to be declared in SEI interface. Only the methods declared in SEI interface are exposed as Web Service methods or remote accessible methods, others would be treated as local methods.

iii) Web Service methods should take parameters and return returnvalues as Serializable – The methods we declare in SEI interface should take parameters as Serializable, and even return values must be serializable.

iv) Should throw RemoteException – SEI interface methods should throw RemoteException. Let us say a program along with performing some operation; he is trying to access the Web service. Refer to the below snippet.

```
class BookInfoRetreiver {
    public void findBook(int isbn) {
        try{
          BookInfo info = // call service and get Book info
          // perform something using bookInfo
        } catch(RemoteException re) {
        } catch(NullPointerException npe){
        }
    }
}
```

Now in the above code, we are trying to get the book information by calling the service, after getting the BookInfo, we are computing something with the data. While executing the findBook method an exception was occurred, how do I need to find whether the exception was caused due to Web service call or the local code has thrown the error? If the type of Exception being caught is RemoteException, it indicates a Remote program has caused the error. Otherwise would be considering as local code has raised the exception.

In order to differentiate between local exceptions and Web service exceptions, my SEI interface Web service method should always throw RemoteException.

In addition to this, even your method is declared to throw RemoteException, the code inside your method may not be throwing exception at all. In such condition, why should I declare it to throw? Your code may not be throwing the exception, but while client sending a request to the provider, they could be some network failures, in such cases the Web service runtime will propagates such failures to the client by throwing RemoteException to the User, so by declaring RemoteException you are always preparing the consumer to handle.

With the above rules, let us try to write an SEI Interface as shown below.

```
package com.store.service;

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

// SEI
public interface Store extends Remote {
    float getBookPrice(String isbn) throws RemoteException;
}
```

b) **Provide Implementation class** – Write a class which implements from SEI interface, your implementation class must provide implementation from all the methods which are declared in SEI interface. Implementation class could contain some other methods as well, but those will not be exposed as Web Service methods. Your web service methods overridden from SEI interface may or may not throws Remote Exception.

The implementation class methods will contain business logic to handle and process the incoming requests, but the logic you wrote here may or may not raise any exceptions. In a case where your logic is causing any exceptions, you don't need to declare your method to throw RemoteException.

But if your business logic is raising a NullPointerException or SqlException, the class should not throw NullPointerException or SqlException rather those exceptions has to be caught in try/catch block and should wrap those exceptions into RemoteException and should throw to the Consumer.

Below snippet shows the Implementation class for the above SEI interface.

```
package com.store.service;

import java.rmi.RemoteException;

public class StoreImpl implements Store {

    public float getBookPrice(String isbn) {
        float price = 0.0f;

        if (null != isbn && !"".equals(isbn)) {
            // connect to db
            // query table
            // get data and return
            if ("ISBN1001".equals(isbn)) {
                price = 232.23f;
            } else if ("ISBN1002".equals(isbn)) {
                price = 242.35f;
            }
        }
        return price;
    }
}
```

c) **Generate Binding classes** – After developing the SEI interface and Implementation classes we need to run a tool called wscompile to generated binding classes.

- What tools are available?

The wscompile tool is shipped as part of JWSDP. For developing a JAX-RPC API RI based Web service we need some set of tools/compilers to generate some classes similar to we generate in JAX-B, but where are these tools under? These tools are available to you when you install JWSDP.

Basically we will use two tools for JAX-RPC API RI based web service development.

i) Wscompile – Used for generating binding classes from SEI interface or WSDL

ii) Wsdeploy – Used for generating service related configurations which are required to deploy your service.

Both these tools are shipped as part of JWSDP when you install and are available under the below folder.

```
C:>Sun\JWSDP-2.0
| - jaxrpc
     | - bin
          | - wscompile.bat, wscompile.sh, wsdeploy.bat, wsdeploy.sh
```

- Why to run wscompile?

Now let us try to understand why I need to generate binding classes?

As we know the Consumer will send's the request to the Provider requesting for some data or processing. Here sending the request to the Provider is nothing but invoking a method on the Provider for e.g. in the above code he will call getBookPrice method to get the price of the book.

While calling this method, the consumer needs to send data required for executing the method. Does the consumer need to send the Java type data? He shouldn't rather send XML equivalent of it for example if it is rpc-encoded, the request XML will look as shown below.

```
<getBookPrice>
     <isbn xsi:type="xs:string>ISBN1001</isbn>
</getBookPrice>
```

"RPC" stands for remote procedural call, so the request XML the consumer is sending would exactly represent like a method call with root element equal to method name and parameters of the method as sub-elements.

This XML would be placed inside SOAP XML as shown below.

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
     <soap:Header/>
     <soap:Body>
          <getBookPrice>
               <isbn xsi:type="xs:string>ISBN1001</isbn>
          </getBookPrice>
     </soapy:Body>
</soap:Envelope>
```

And the above SOAP XML will be placed inside the HTTP Message and sends a HttpRequest to Provider.

Header	POST http://localhost:8080/BookStoreWeb/bookStore HTTP/1.1 Host: localhost Content-Type: text/xml; charset=utf-8 Content-Length: length SOAPAction: "http://localhost:8080/BookStoreWeb/bookStore#getBookPrice"
Payload	<?xml version="1.0"?> <soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope" soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding"> <soap:Header/> <soap:Body> <getBookPrice> <isbn xsi:type="xs:string>ISBN1001</isbn> </getBookPrice> </soapy:Body> </soap:Envelope>

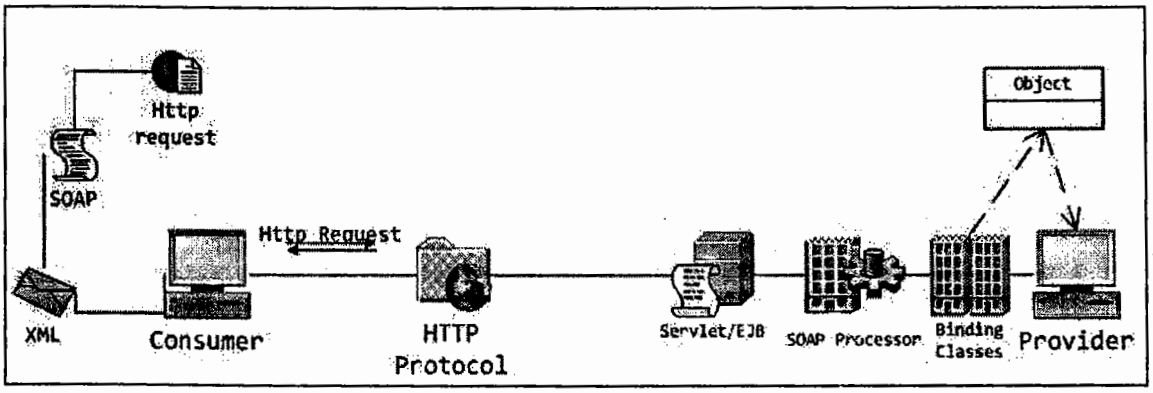
Now the above Http Request message would be sent to the Provider. On the Provider side, they should be some person acting as a listener who can receive the above HTTP Request. This listener at the Provider end typically would be a Servlet as he is the best person to process HTTP requests.

Upon Servlet receiving the request, as it can understand HTTP message, it would crack the message and process HTTP Headers. After processing the HTTP header, now it tries to access the PAYLOAD, now the payload portion of it contains SOAP XML which cannot be understood by Servlet, so it simply passes it down to SOAP Processors.

SOAP Processors are the components who know how to parse a SOAP XML, SOAP Processors will process the SOAP Headers if any and then extracts the Body of the Soap Message. The body of the SOAP XML is business XML. Now the SOAP Processors don't know how to process business XML, so they delegate this XML to Binding classes.

Binding classes are the classes who know how to convert XML to Java Object. Now these classes convert the XML into Java Object from which, runtime takes the data and passes them as parameters in calling the Implementation class methods.

Following figure depicts the above explanation.



Now here comes the question, based on what I need to generate the binding classes, if you analyze carefully the above explanation, the request XML would be converted into Java Object and would be passed as Web service method parameters, which means the XML would be converted to method parameters, this indicates I need to generate binding class for every web service method parameters. So, while running the wscompile tool, I need to pass the input as SEI interface, so that the tool will loads the SEI interface and identifies the methods and their parameters and based on the message exchange format, it will generates the binding classes.

From the above we understood we need to generate binding classes and to generate we need to run wscompile tool by giving input as SEI interface.

- How to run wscompile?

Let us understand how to run wscompile tool:

Wscompile tool is proprietary tool shipped by JWSDP for building JAX-RPC SI. From the above, we understood we need to give input to the wscompile tool as SEI interface, but passing the Interface information differs from Vendor who ships the tool.

In our case, JWSDP provided wscompile tool will of course takes the SEI interface as input, but not directly rather you need to configure this information in a configuration file called config.xml. It is similar to web.xml in which you configure the Servlet information. But unfortunately this config.xml is not common configuration file that every tool uses and it is specific to JAX-WS API RI implementation and is provided by vendor.

Now developer in order to run the wscompile tool, he needs to configure the SEI interface, implementation class, package name, Service Name, targetNameSpace and typeNameSpace in the configuration file and needs to provide this as input.

Sample configuration file is shown below.

config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
    <!--
        Copyright 2004 Sun Microsystems, Inc. All rights reserved. SUN
        PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
    -->
<configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
    <service name="StoreService"
        targetNamespace="http://safaribooks.com/books/sale"
        typeNamespace="http://safaribooks.com/books/types"
        packageName="com.store.service.binding">
        <interface name="com.store.service.Store"
            servantName="com.store.service.StoreImpl" />
    </service>
</configuration>
```

Along with configuration file he needs to specify some more switches while running the command as described below.

"-d"	stands for directory under which you want to generate the source files
"-gen:server"	The binding classes generated by wscompile tool are required by both consumer and provider to exchange data, but the classes generated at Provider side would be different from classes that are generated at Consumer side. The wscompile tool is designed to generate both Provider side as well as Consumer side classes. But while running the tool you need to specify whether you want to run it Server side or client side. Here "gen:server" indicates Provider side classes
"-keep"	wscompile tool will takes the SEI interface and message encoding format, generates the binding classes. After that it will compiles the binding classes and deletes the source files generated. If you want to retain the source files which are generated you need to use "-keep"
"cp"	As we know wscompile tool will takes the SEI interface as input and generates binding classes. So in-order to read the method signatures in the SEI interface, does the wscompile takes the Source of SEI as input or its .class file? Always the .class files are compiled outcomes which

	doesn't have any syntax errors, so tools will always takes the class files as input rather than source. So using "-cp" we will specify the location of class files
"-model"	If config.xml contains a <service> or <wsdl> element, wscompile generates a model file that contains the internal data structures that describe the service. If you've already generated a model file in this manner, then you can reuse it the next time you run wscompile. For example: <modelfile location="mymodel.xml.gz"/>

Below is the piece of code showing how to run the command?

```
wscompile -gen:server -d src -cp build\classes -keep -verbose -model
model-rpc-enc.xml.gz <PATH>\config.xml
```

With the above command, the wscompile reads the contents of the config.xml and identifies the SEI interface and Implementation class and validates against all the rules in implementing them. Once those are valid, it starts reading the methods of the SEI interface and generates the following artifacts

RequestStruct	representing the structure of request xml
ResponseStruct	representing the structure of response xml
RequestStructSOAPSerializer/Builders	These classes knows how to convert request XML into RequestStruct class objects
ResponseStructSOAPSerializer	This knows how to convert ResponseStruct object to response XML
Tie	Who facilitates the processing of request
Model	Contains the internal data structures using which we can recreate the service
WSDL	Describes the information about the service

If you observe the above, the WSDL has been generated taking the input as Java classes and hence this is called Contract Last approach.

With this above we generated all the classes that are required to facilitate the conversion of XML to Java and Java to XML at the Provider side for every method.

d) Write jaxrpc-ri.xml

After generating the necessary binding classes to expose our class as service, we need to configure the endpoint information of our service in a configuration file called jaxrpc-ri.xml

As how you write web.xml to attach for a Servlet an URL to access, for even your Web service class we need to attach an URL to expose it as an service so that Consumers can access the service by using the URL we specified here.

Here we configure the complete endpoint information along with attaching it to an URL to access as shown below.

Note: - The file name should be **jaxrpc-ri.xml** and it is mandatory to use the same file name you should place this file under WEB-INF directory only.

```
<?xml version="1.0" encoding="UTF-8"?>
<webServices
   xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/dd"
   version="1.0"
   targetNamespaceBase="http://safaribooks.com/books/sale"
   typeNamespaceBase="http://safaribooks.com/books/types"
   urlPatternBase="/store">

   <endpoint
      name="Store"
      displayName="Get Book Price"
      description="Used for finding price of book"
      wsdl="/WEB-INF/StoreService.wsdl"
      interface="com.store.service.Store"
      implementation="com.store.service.StoreImpl"
         model="/WEB-INF/wsdl-rpc-enc.xml.gz"/>

   <endpointMapping
      endpointName="Store"
      urlPattern="/store"/>

</webServices>
```

e) Run wsdeploy tool

After writing the jaxrpc-ri.xml file, we need to run wsdeploy tool which generates web service deployment descriptor that carries the information describing the service, which is used for servicing the request from the consumer.

While running this tool we need to give the input as war of our application. So first we need to export the project and then need to run the wsdeploy tool by giving this war as input shown below.

```
Wsdeploy -o target.war BookWeb.war
```

The above command reads the contents of BookWeb.war and identifies the web.xml and jaxrpc-ri.xml. It reads the contents of web.xml and updates it with some more additional configuration. Along with this it uses the jaxrpc-ri.xml as input file and generates web service deployment descriptor jaxrpc-ri-runtime.xml which contains the configuration related to service we are exposing.

These two files will be generated and placed inside the target.war. Now we can either directly deploy the target.war on the Application Server or extract the target.war and copy web.xml and jaxrpc-ri-runtime.xml and can paste our eclipse project and deploy from eclipse as well.

Deploy the war and start your server and you should be able to access the service.

11.1.2 Request Processing Flow

The Consumer sends HTTP request to the Provider using the URL pattern specified in the jaxrpc-ri.xml.

For e.g. http://localhost:8080/BookWeb/store (as per the example above)

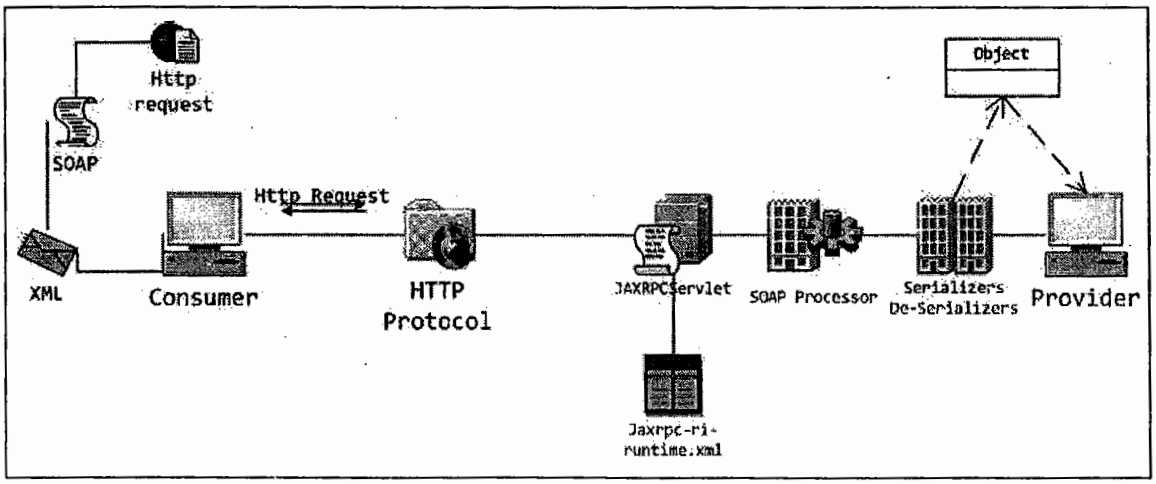
As we know we developed Servlet Endpoint based Web Service, on the Provider side, a Servlet will receive the request to process it. If you look at the web.xml generated, it contains the JAXRPCServlet configured to listen on the URL "/store" (this was configured when we run wsdeploy tool).

web.xml

```
<servlet>
  <servlet-name>Store</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>Store</servlet-name>
  <url-pattern>/store</url-pattern>
</servlet-mapping>
```

Once the request has been received by the JAXPRCServlet, it would process the HTTP Headers and then tries to extract the payload of it. The payload would be SOAP XML, as Servlet cannot understand the SOAP XML; it needs to find the relevant SOAP Parser's to process it.

So, it performs a lookup into jaxrpc-ri-runtime.xml (generated by wsdeploy tool by taking jaxrpc-ri.xml as input) to identify an endpoint whose url would match to the url pattern of the Servlet Path ("/store").



jaxrpc-ri-runtime.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns='http://java.sun.com/xml/ns/jax-rpc/ri/runtime'
version='1.0'>
  <endpoint
    name='Store'
    interface='com.store.service.Store'
    implementation='com.store.service.StoreImpl'
    tie='com.store.service.Store_Tie'
    model='/WEB-INF/wsdl-rpc-enc.xml.gz'
    wsdl='/WEB-INF/StoreService.wsdl'
    service='{http://safaribooks.com/books/sale}StoreService'
    port='{http://safaribooks.com/books/sale}StorePort'
    urlpattern='/store'/>
</endpoints>
```

From the above configuration it picks up the Tie class name matching the Servlet Path and forwards the request to the Tie for further processing.

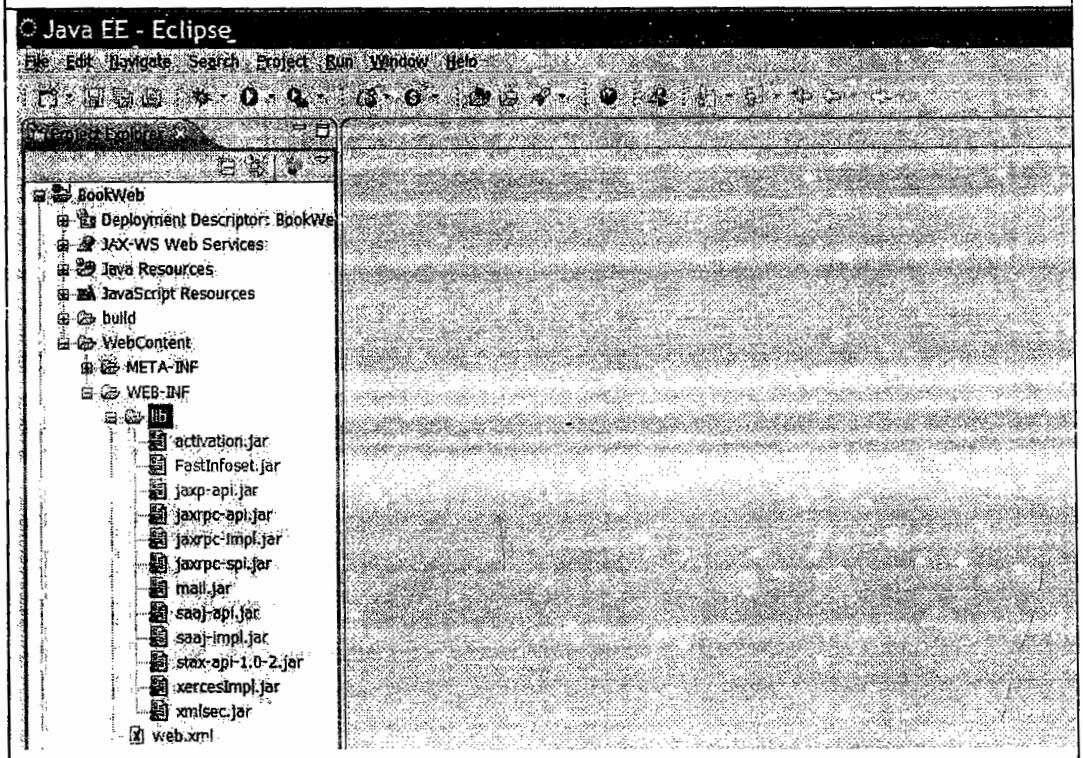
Tie class has the logic for processing the SOAP XML, and extracts the actual XML from the SOAP Body and then passes it to request SOAP serializers/De-Serializers to convert into Java Object. Then passes the Object data as input to the Service method. Once the Service method finishes processing, it returns Java Object data as response to the Tie. Now Tie converts this back into XML and then into SOAP XML and places it into HTTP Response Message and sends back it to the Consumer.

11.1.3 Contract Last – Activity Guide

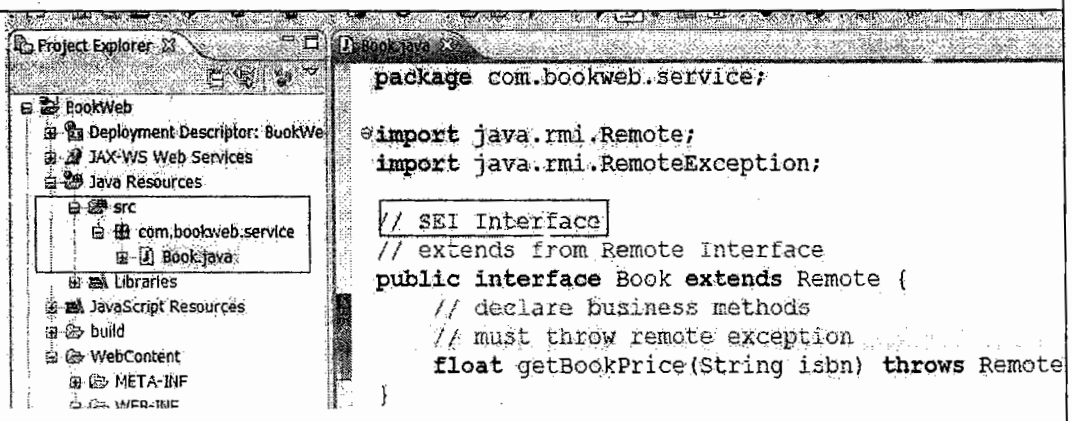
The following section walks you through the detailed steps of creating the Service using the above described steps in Screen shots.

New Dynamic Web Project

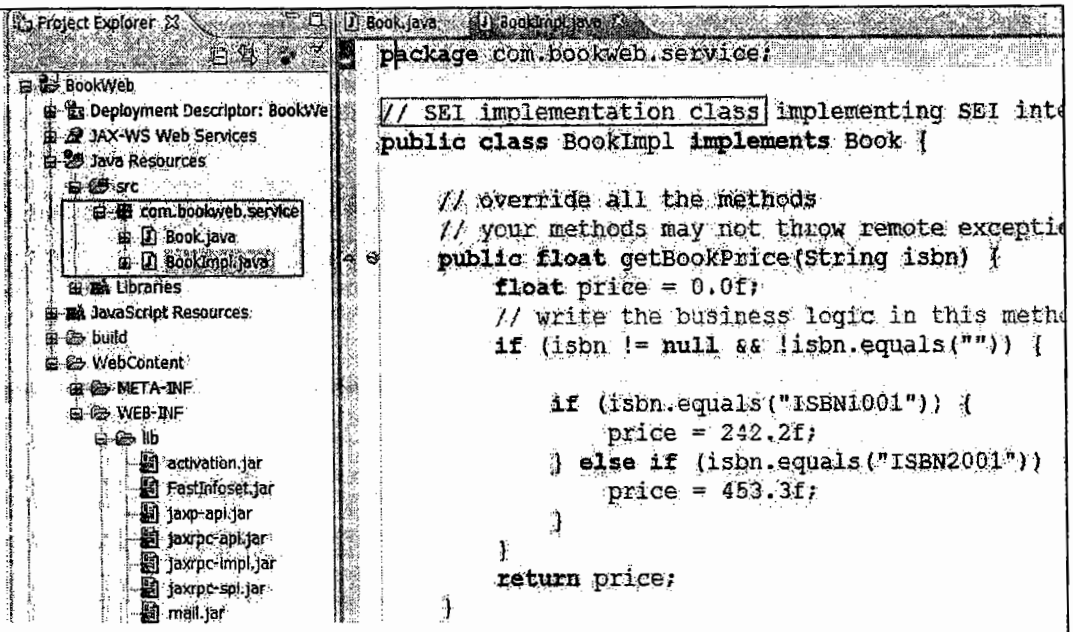
Create a new Dynamic Web Project in eclipse, expand the Project directory and navigate into WebContent/WEB-INF/lib directory. Copy/paste all the jaxrpc jar's that are given in Jar bundles folder to the above lib project. Configure the Build path by right clicking on the project, select configure build path and switch to libraries tab and add all the jar's in the lib to the Project class Path



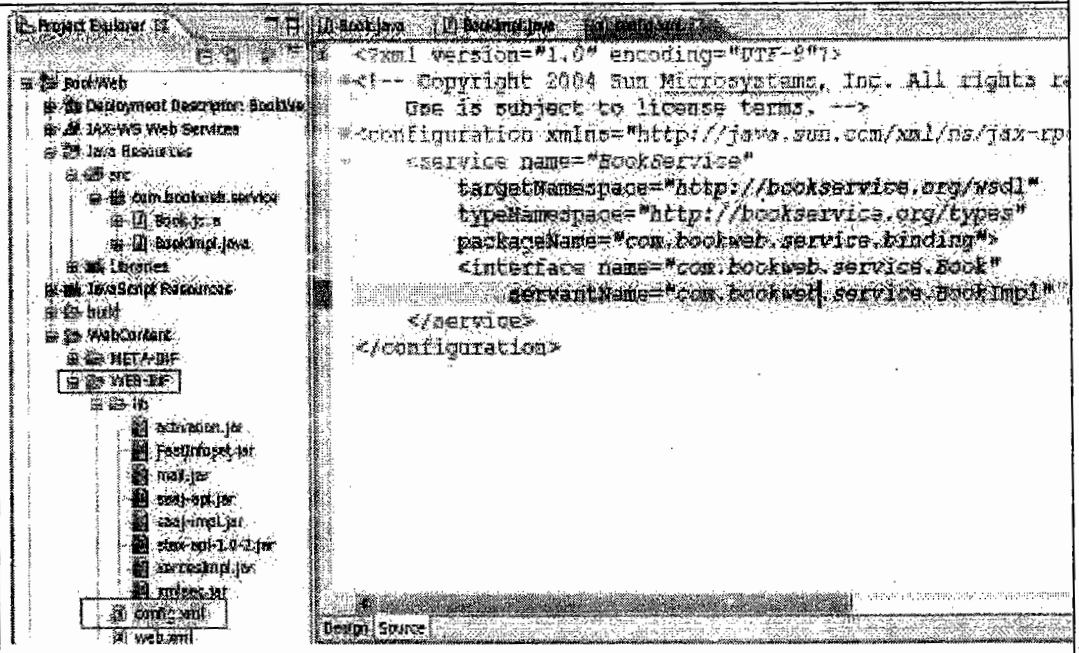
Write SEI Interface



Write SEI Implementation class

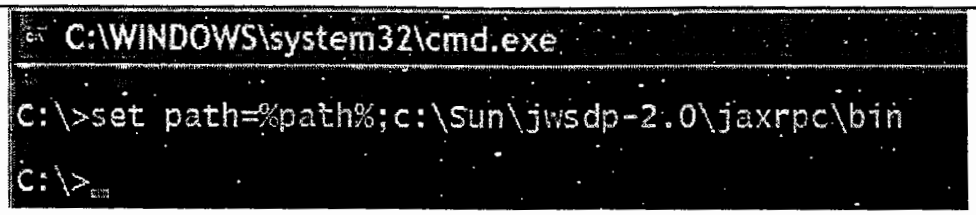


Write config.xml in WEB-INF



Run the wscompile

As explained earlier WSCOMPILE tool will generates the binding classes by taking the config.xml as input. As you have to generate the binding classes into Project→ Src directory, you need to run the tool from Project directory. So you need to set First of all the System Path pointing to the location of WSCOMPILE bat file, and then switch to the Project directory and run WSCOMPILE as shown below.



Switch to Project directory for example

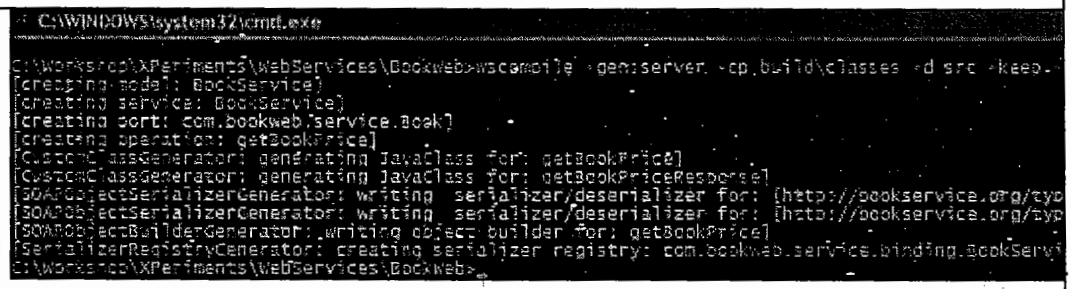
Cd D:\Folder1\Folder2\BookWeb

If you execute DIR command under this directory you must see "src" till then you need to navigate.

Now execute the below command.

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

Output of wscompile

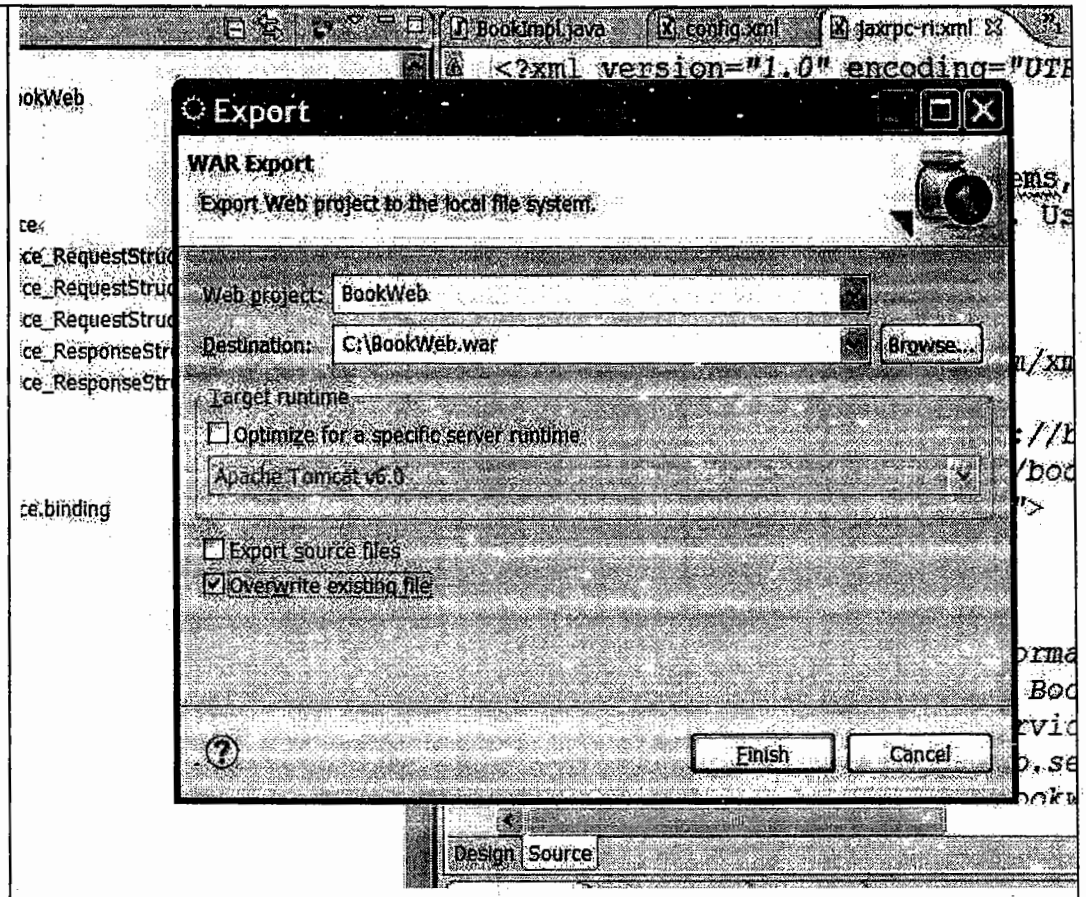


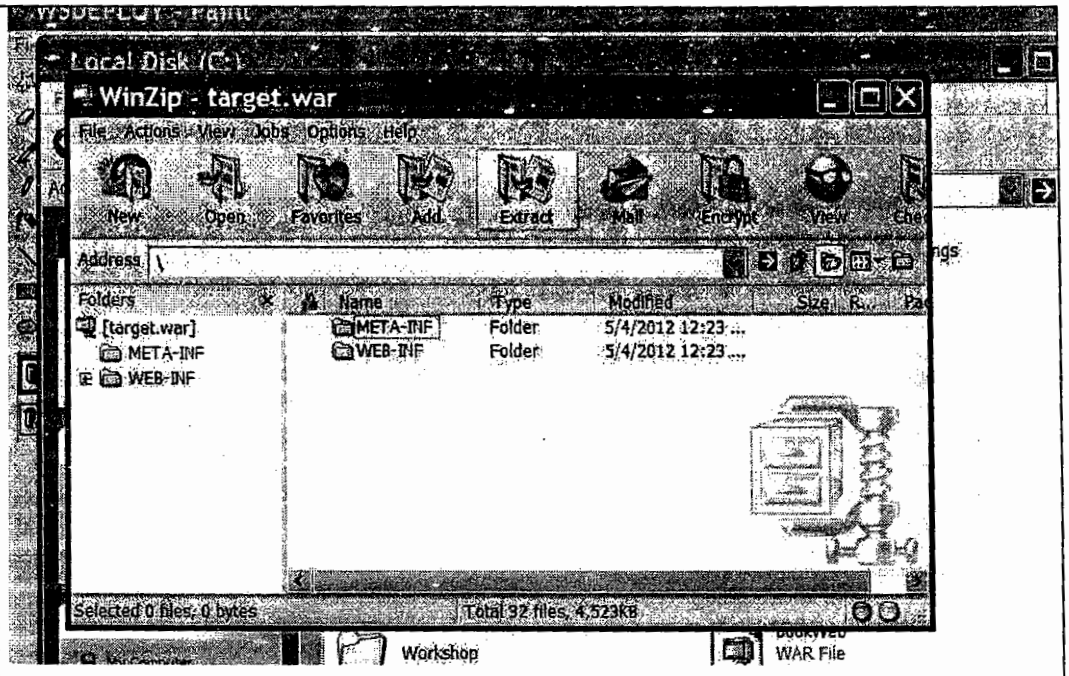
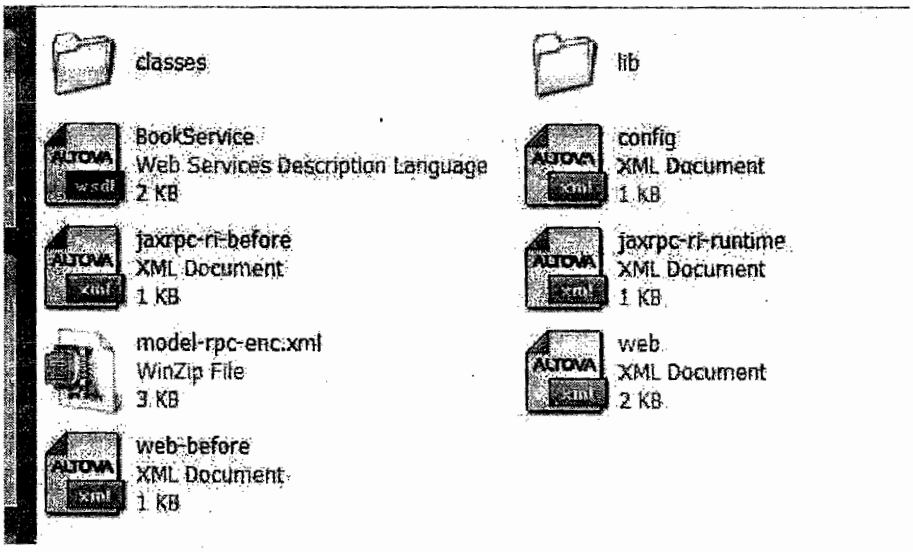
Drag and Drop WSDL and Model files generated into WebContent\WEB-INF directory

Write jaxrpc-ri.xml

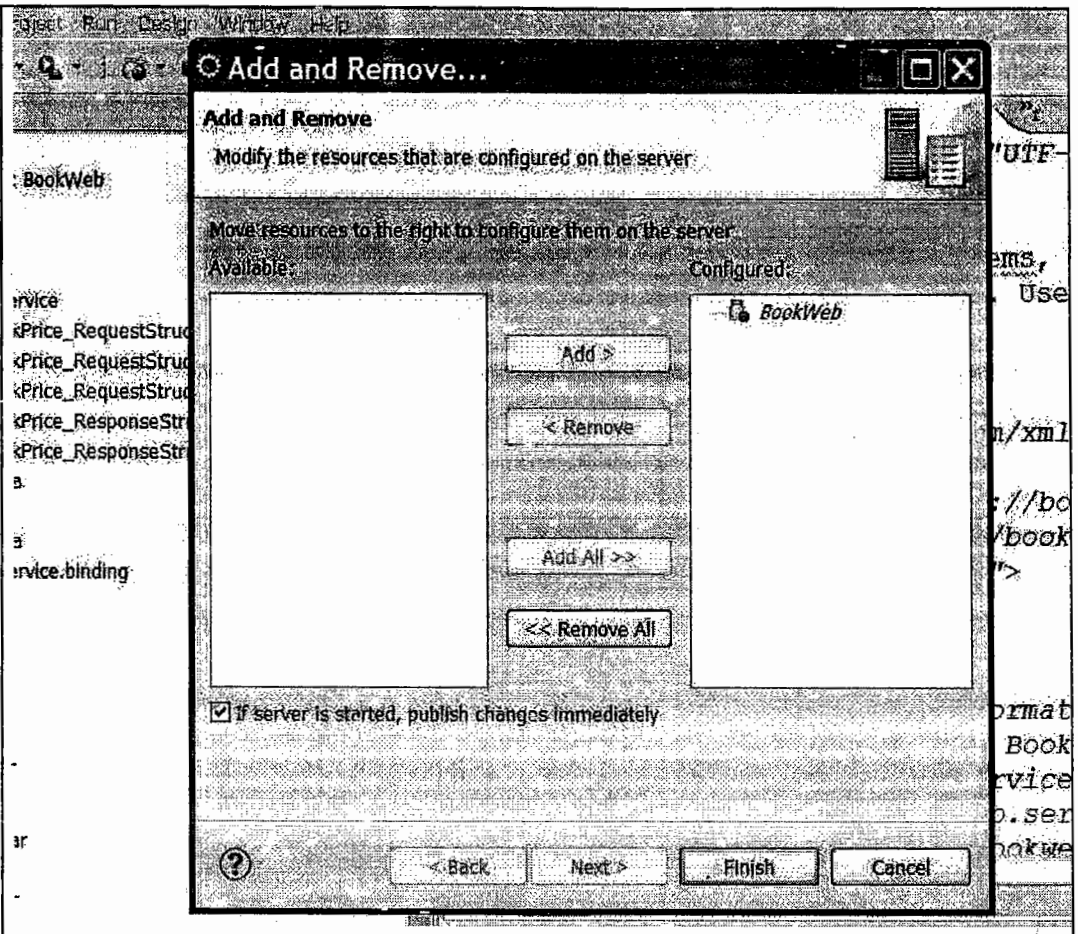
```
<!--
  Copyright 2004 Sun Microsystems, Inc. All rights reserved.
  SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license term
-->

<webServices
      xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/dd"
      version="1.0"
      targetNamespaceBase="http://bookservice.org/wsdl"
      typeNamespaceBase="http://bookservice.org/types"
      urlPatternBase="/getPrice">

      <endpoint
          name="Book"
          displayName="Book Information Service"
          description="A simple Book Service"
          wsdl="/WEB-INF/BookService.wsdl"
          interface="com.bookweb.service.Book"
          implementation="com.bookweb.service.BookImpl"
           model="/WEB-INF/model-rpc-enc.xml.gz"/>

      <endpointMapping
          endpointName="Book"
          urlPattern="/getPrice"/>

</webServices>
```

Export the project as WAR

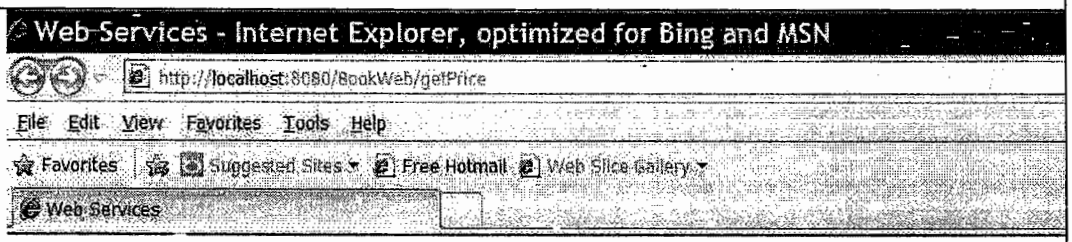


Extract the generated WAR	 
Copy/Paste extracted files	Now the generated web.xml and jaxrpc-ri-runtime.xml generated here should be copy pasted into the <Project>\WebContent\WEB-INF and override.

Deploy the Project and start Server



Access the Dashboard



Web Services

Port Name	Status	Information	
Book	ACTIVE	Address:	http://localhost:8080/BookWeb/getPrice
		WSDL:	http://localhost:8080/BookWeb/getPrice?WSDL
		Port QName:	{http://bookservice.org/wsdl}BookPort
		Remote interface:	com.bookweb.service.Book
		Implementation class:	com.bookweb.service.BookImpl
		Model:	http://localhost:8080/BookWeb/getPrice?model

You can test your service in SOAP UI by importing the WSDL

11.1.4 Contract First (Servlet Endpoint, Sync req/reply with rpc-encoded)

This is no way different from contract last approach, all the request processing flow and other concepts (serializers/de-serializers) will still applicable even you develop the service using contract first approach. Only difference would be the way you are developing the service.

As indicated earlier, you can develop a Web service in two approaches contract first and contract last. We already discussed about contract last approach. Now we are trying to build the Provider using contract first approach.

In web services, the contract between provider and consumer is WSDL, as it is contract first approach, the development will starts from WSDL rather than from Java. Now the developer has to write the WSDL document from which he generates the necessary java classes to expose it as service.

By looking at the WSDL – PortType we can create the SEI interface and based on the operation input and output types we can create method with parameters and return types. We need serializers and de-serializers to map incoming XML data to method parameters and return value of the method to XML back.

Sample WSDL document is provided here for your reference.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:tns="http://ebay.in/sales/wsdl"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
name="PurchaseOrderService"
 xmlns:et="http://ebay.in/sales/types"
targetNamespace="http://ebay.in/sales/wsdl">
 <wsdl:types>
       <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://ebay.in/sales/types"
xmlns:et="http://ebay.in/sales/types"
             elementFormDefault="qualified">
             <xs:complexType name="purchaseOrderType">
                   <xs:sequence>
                         <xs:element name="item" type="xs:string" />
                         <xs:element name="quantity" type="xs:int" />
                   </xs:sequence>
             </xs:complexType>
             <xs:complexType name="orderStatusType">
                   <xs:sequence>
                         <xs:element name="orderId" type="xs:string" />
                         <xs:element name="status" type="xs:string" />
                   </xs:sequence>
             </xs:complexType>
       </xs:schema>
 </wsdl:types>
 <wsdl:message name="Order_placeOrder">
       <wsdl:part name="purchaseOrder" type="et:purchaseOrderType" />
 </wsdl:message>
 <wsdl:message name="Order_placeOrderResponse">
       <wsdl:part name="result" type="et:orderStatusType" />
 </wsdl:message>
 <wsdl:portType name="Order">
       <wsdl:operation name="placeOrder">
             <wsdl:input message="tns:Order_placeOrder" />
             <wsdl:output message="tns:Order_placeOrderResponse" />
       </wsdl:operation>
 </wsdl:portType>
```

```
<wsdl:binding name="OrderSOAPBinding" type="tns:Order">
        <soap:binding style="rpc"
                transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="placeOrder">
                <soap:operation
soapAction="http://ebay.in/sales/wsdl/placeOrder" />
                <wsdl:input>
                        <soap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                                namespace="http://ebay.in/sales/wsdl"
use="encoded" />
                </wsdl:input>
                <wsdl:output>
                        <soap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                                namespace="http://ebay.in/sales/wsdl"
use="encoded" />
                </wsdl:output>
        </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="OrderService">
        <wsdl:port binding="tns:OrderSOAPBinding" name="OrderPort">
                <soap:address location="http://www.example.org/" />
        </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Instead of developer writing the above classes, we can run the wscompile tool which is shipped as part of JWSDP by giving input as WSDL. Now the wscompile will loads the WSDL document and reads various elements of WSDL and generates the required classes to build the Web service. Following are the classes generated when we run wscompile.

SEI Interface	Will be generated based on PortType in WSDL
Input/Output classes	You SEI Interface methods will take parameters and by looking at the PortType operations we will know what are input/output elements. Based on this we can find their relevant XSD types and can generate classes representing input/output parameters of the SEI Methods.
Request/Response Struct, Serializers and Deserializers	These classes are required to convert the input XML to Object and Object back to XML
Service Interface and Impl	Required for building Consumer (we can ignore this class if we are building Provider)

As we know wscompile will not directly takes the WSDL, rather we need to write the config.xml and need to configure WSDL location in it rather than SEI interface and Implementation. Now give this config.xml as input to wscompile so that it generates the classes required to build the Web service.

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:8080/StoreWeb/store?WSDL"
packageName="com.store.service">
  </wsdl>
</configuration>
```

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

As all the classes are available to build the Service, we need to write now the Implementation class implementing the SEI interface. Then we need to write the jaxrpc-ri.xml, which attaches url to the endpoint.

Now export the project as war and run wsdeploy tool by giving war as input

```
wsdeploy -o target.war source.war
```

It will generate the target.war which contains the jaxrpc-ri-runtime.xml and web.xml. Copy these two files into the project WEB-INF directory and deploy the project and start the server.

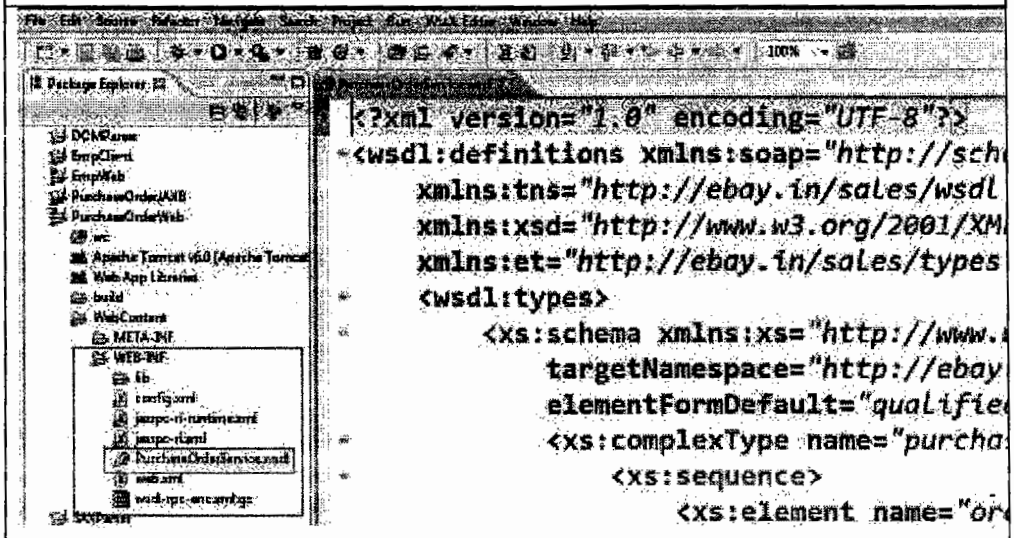
You should be able to access your Provider/Service.

11.1.5 Contract First– Activity Guide

The following section walks you through the detailed steps of how to create JAX-RPC SI based service using Contract First approach.

Write WSDL document

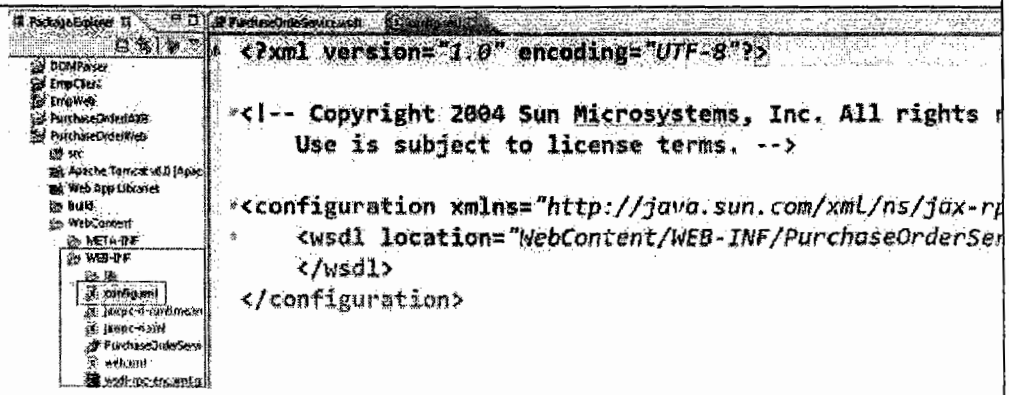
As it is contract first approach, the developer has to write the WSDL document identifying the Service and its operations. Need to place the WSDL document in WebContent/WEB-INF directory



Write config.xml

From the wsdl, we need to generate Java classes required to build the Service. As we know, wscompile can generate the required classes but it will takes the input as config.xml

But as opposed to the earlier approach we don't have Java classes, rather we need to configure the location of WSDL in config.xml to generate Java classes as shown below.



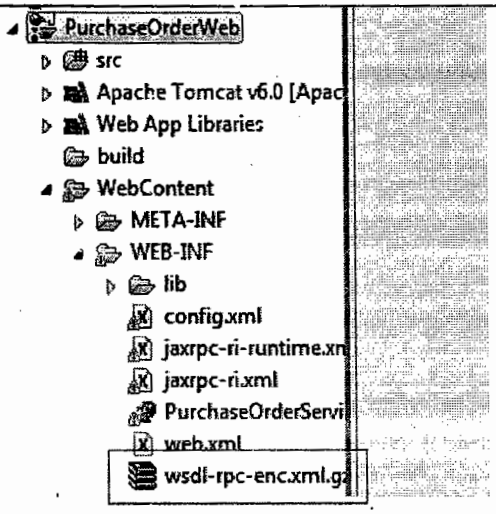
Run wscompile tool

Now we need to run the wscompile tool to generate classes by giving input as config.xml

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

(If you observer -cp is not there as we are not providing any classes as input rather it reads the wsdl)

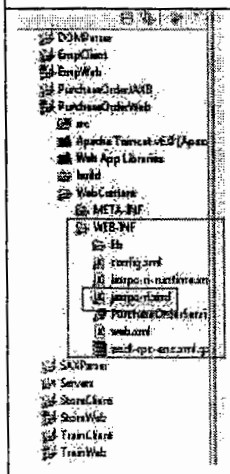
Move model file to WEB-INF



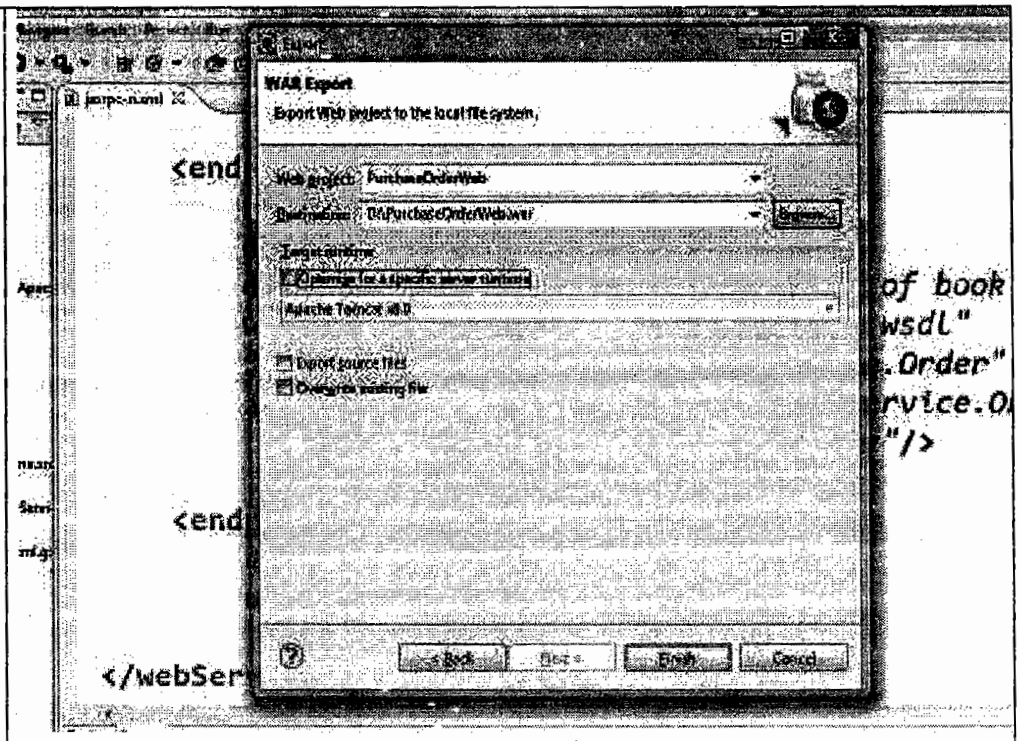
Write jaxrpc-ri.xml

We need to provide the endpoint configuration in jaxrpc-ri.xml. Here we map the endpoint with an url, from which the jaxrpc-ri-runtime.xml and web.xml would be generated by wsdeploy tool.

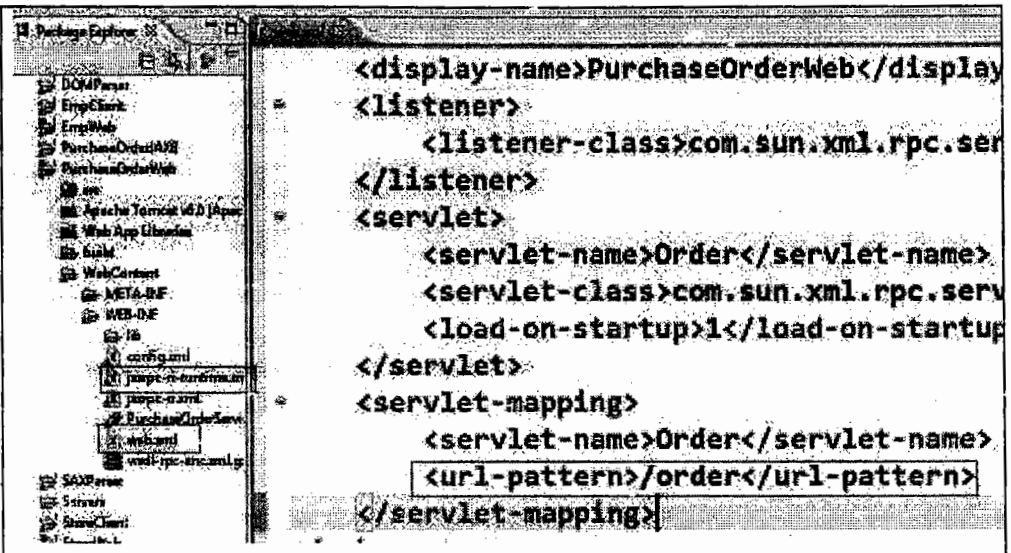
Note: - We need to place this file in WEB-INF directory and the file name should be jaxrpc-ri.xml only.



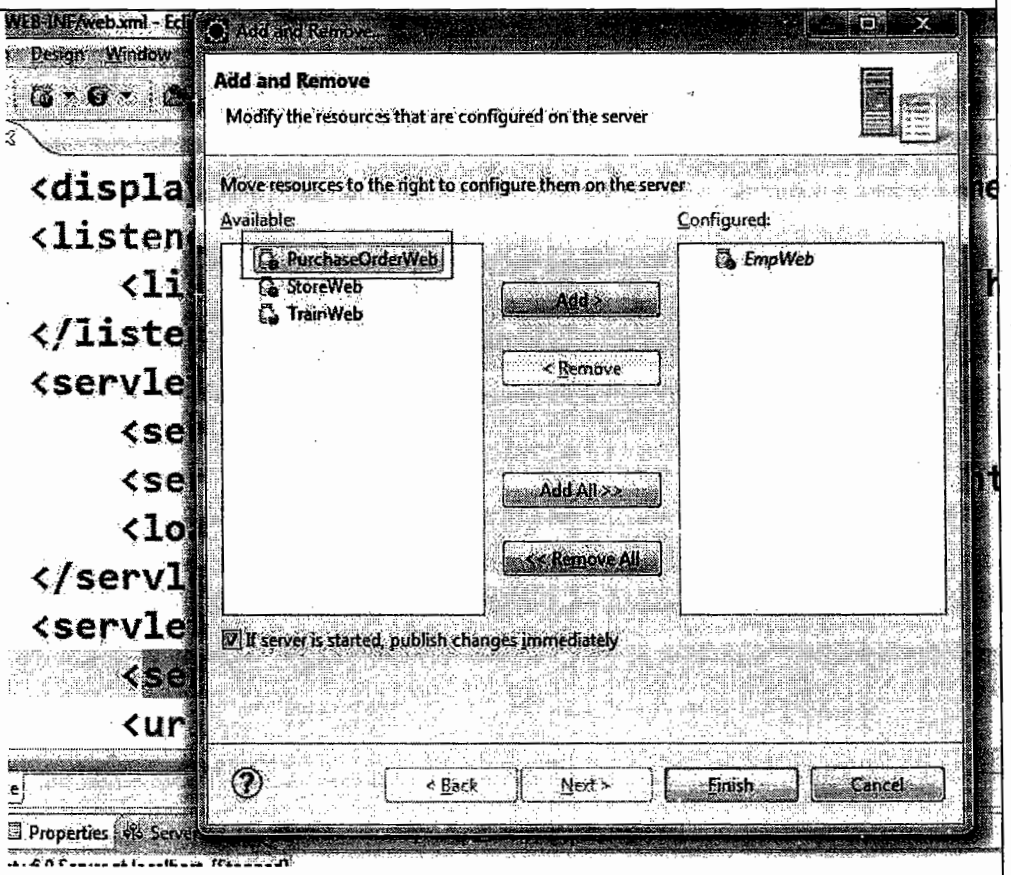
```
    <endpoint
        name="Order"
        displayName="Get Book Price"
        description="Used for finding price of book"
        wsdl="/WEB-INF/PurchaseOrderService.wsdl"
        interface="com.purchaseorder.service.Order"
        implementation="com.purchaseorder.service.Order
         model="/WEB-INF/wsdl-rpc-enc.xml.gz"/>

    <endpointMapping
        endpointName="Order"
        urlPattern="/order"/>

</webServices>
```

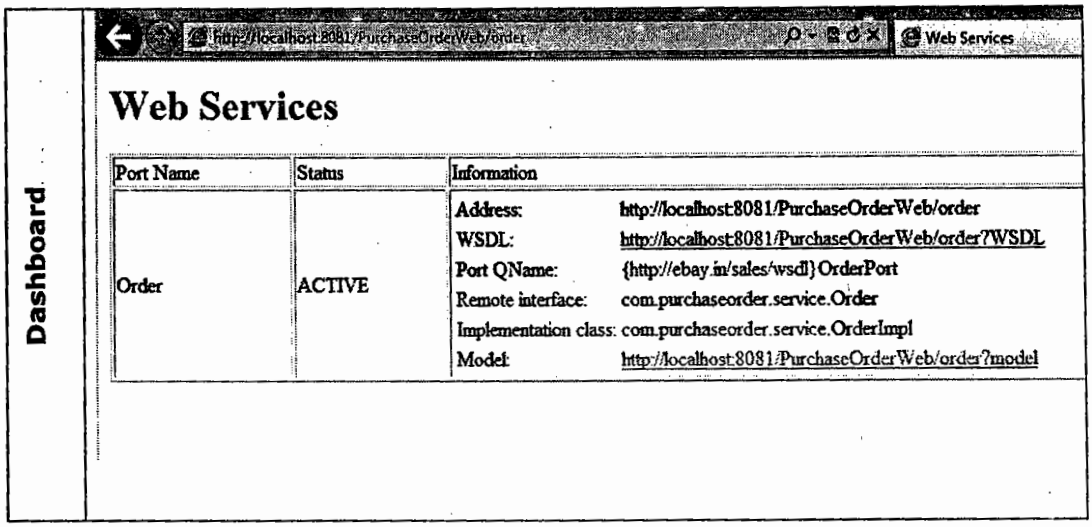
Export the project as WAR	
Run wsdeploy	Now on the generated war, you need to run the wsdeploy tool. This tool will take the input as PurchaseOrderWeb.war and generates the specified war as output. wsdeploy –o target.war PurchaseOrderWeb.war Inside the PurchaseOrderWeb.war, it searches for jaxrpc-ri.xml and based on the configuration provided, it generates the jaxrpc-ri-runtime.xml and web.xml.

Copy jaxrpc-ri-runtime.xml and web.xml to WEB-INF



Deploy and Test





11.2 Building Consumer

As you know JAX-RPC is the API which facilitates in development of both Providers and consumers as well. In the earlier session we understood how to build a JAX-RPC SI based provider using contract first and contract last. In this session we are going to see how to build a JAX-RPC SI based consumer to access the Provider.

JAX-RPC API supports three types of clients.

1) Stub based client
2) Dynamic Proxy
3) Dynamic Invocation Interface

Let's try to understand how to develop each of them in the following section.

11.2.1 Stub based client

It is the most widely used client, and is easy to build. In a typical web service interaction the consumer sends data to the provider, here the consumer could be a Java program so he holds data in a Java Object. But the consumer should not serialize the Java Object to send it to the Provider rather he should convert that Java Object to its equivalent XML, should place it in SOAP XML and then send it over the HTTP request.

So, to convert Object to XML and place it SOAP XML, we need serializers and de-serializers even at the consumer side as well. But how do we know what are the inputs to the Provider and what is the output from it. Because based on the Input/Output and number of operations only we can generate the binding classes.

As the Provider was built, we can access the WSDL of the Provider. If we look at the WSDL document, it's PortType represents the SEI Interface and its operations defines the methods of the SEI interface and Input/Output represents the parameters and return values of each method.

So, WSDL contains the entire information about the service. If we give WSDL as input to the wscompile tool it will generates all the required classes like request/response structs, soap serializers/de-serializers etc. But to access the Provider, we need an implementation for the SEI interface. The implementation of the SEI interface should contain the logic for sending the request and getting the response from the Provider. For this wscompile will generates a Stub, it is nothing but implementation of SEI interface at the consumer side. The presence of this Stub is abstracted from the programmer by giving a factory called Service.

As we know already, wscompile tool is designed to take input as config.xml and it cannot take directly the wsdl document. So, the location of the wsdl document has to be placed in config.xml and should give it as input.

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:8080/StoreWeb/store?WSDL"
packageName="com.store.service">
        </wsdl>
</configuration>
```

Now run the wscompile tool by giving the above config.xml as input. While running as we are developing the consumer, we need to use -gen:client as we need consumer side binding classes. As we don't have any classes right now we don't need to specify -cp. The command reference and set of classes that would be generated are shown below.

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

SEI Interface	Required to know the methods that are available at Provider side
Stub	Implementation of SEI interface at the Consumer side contains the logic for sending the request and extracting the response from the Provider
Service/Impl	Factory, knows how to create the object of Stub
Serializers/De-Serializers	Facilitates in converting Object to XML and wise versa
Request/Response struct	Classes represents the structure of input/output XML

If you open the generated Stub class it clearly indicates as implementing from SEI interface and it overrides the methods in the SEI interface, but those methods will not have business logic, rather they have the logic for converting object data into XML, and send over the HTTP connection to the Provider.

```
package com.store.service;

public class Store_Stub
   extends com.sun.xml.rpc.client.StubBase
   implements com.store.service.Store {

   // skipped code for clarity

   public float getBookPrice(java.lang.String string_1)
      throws java.rmi.RemoteException {
      try {

         StreamingSenderState _state = _start(_handlerChain);

         InternaiSOAPMessage _request = _state.getRequest();
         _request.setOperationCode(getBookPrice_OPCODE);
         com.store.service.Store_getBookPrice_RequestStruct
_myStore_getBookPrice_RequestStruct =
            new com.store.service.Store_getBookPrice_RequestStruct();

         _myStore_getBookPrice_RequestStruct.setString_1(string_1);

         // skipped code for clarity.....

         _send((java.lang.String) _getProperty(ENDPOINT_ADDRESS_PROPERTY),
_state);

         com.store.service.Store_getBookPrice_ResponseStruct
_myStore_getBookPrice_ResponseStruct = null;
         Object _responseObj = _state.getResponse().getBody().getValue();
         if (_responseObj instanceof SOAPDeserializationState) {
            _myStore_getBookPrice_ResponseStruct =

(com.store.service.Store_getBookPrice_ResponseStruct)((SOAPDeserializationStat
e)_responseObj).getInstance();
         } else {
            _myStore_getBookPrice_ResponseStruct =

(com.store.service.Store_getBookPrice_ResponseStruct)_responseObj;
         }

         return _myStore_getBookPrice_ResponseStruct.getResult();
         // skipped code for clarity
```

Implements SEI Interface

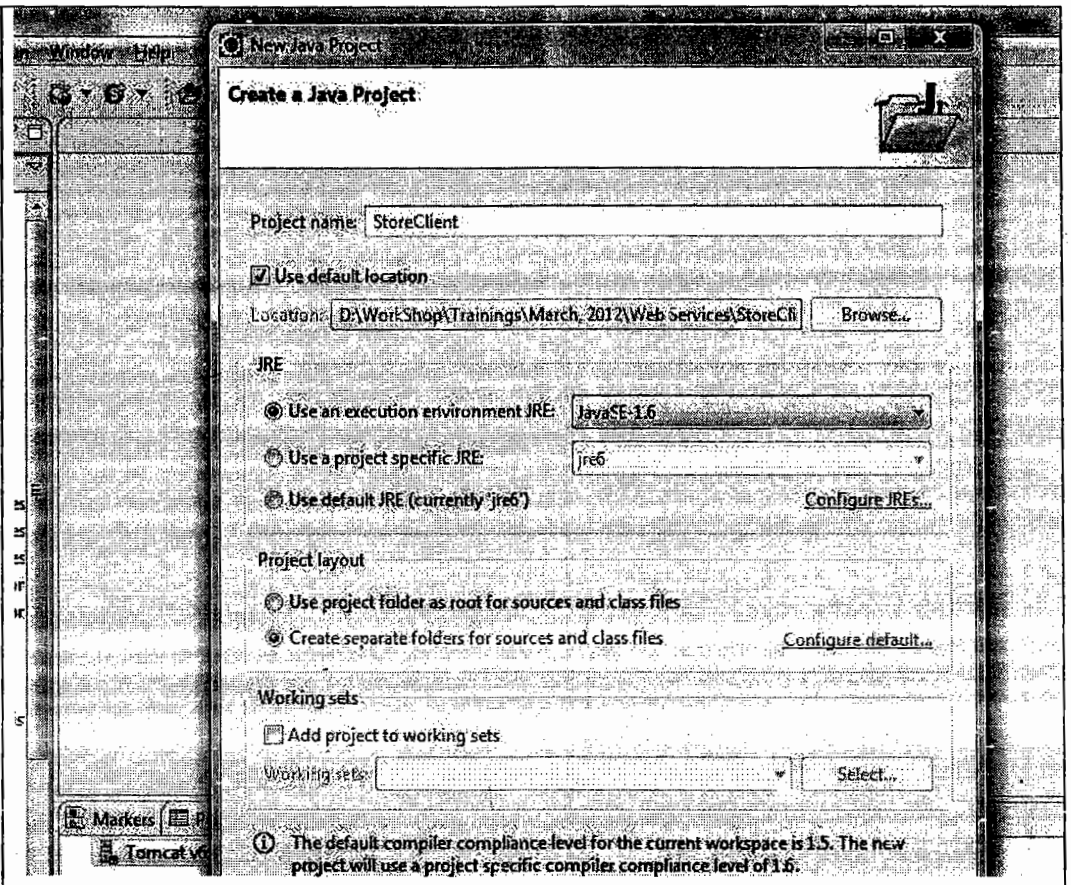
Overrides method

Now write test class in which create the Object of SEI interface, we know the Implementation of SEI interface at the consumer side is Stub. Rather than we directly exposed to the generated Stub, it is being created by Service. Blow is the code snippet shows you how to build the program

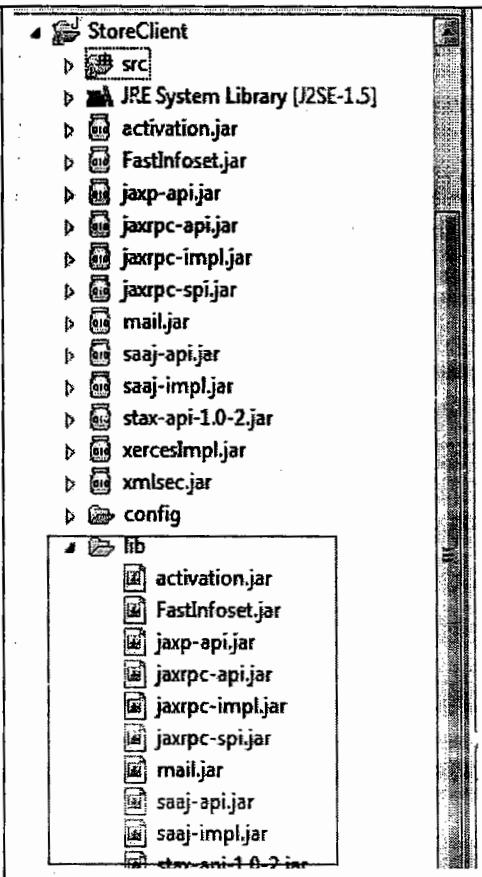
```
package com.store.service;

import java.rmi.RemoteException;

import javax.xml.rpc.ServiceException;

public class AccessStoreTest {
  public static void main(String args[]) throws ServiceException,
                RemoteException {
        StoreService service = new StoreService_Impl();
        Store port = service.getStorePort();
        float price = port.getBookPrice("ISBN1001");
        System.out.println("Price : " + price);
  }
}.
```

11.2.2 Stub based client – Activity Guide

Create a new Java Project

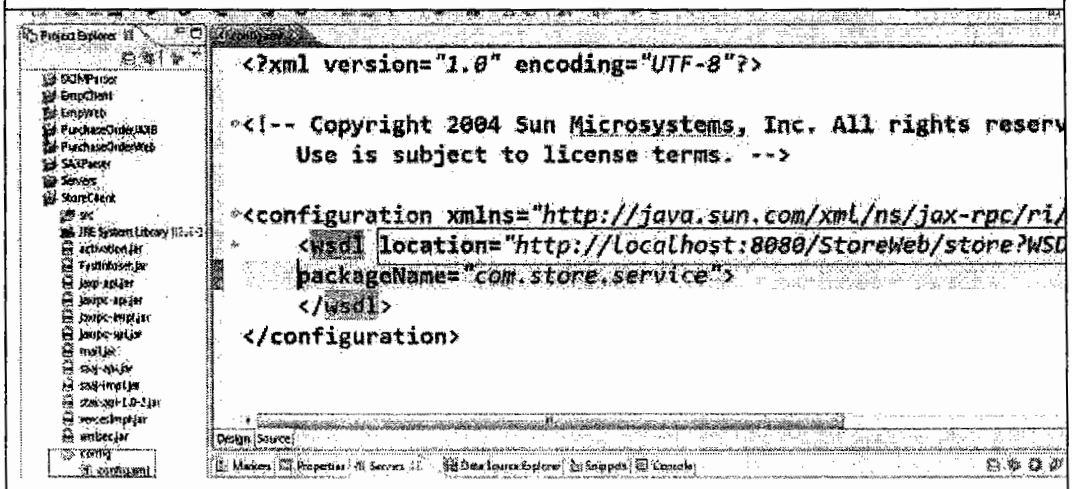


Copy Jar's and set build path



Write config.xml

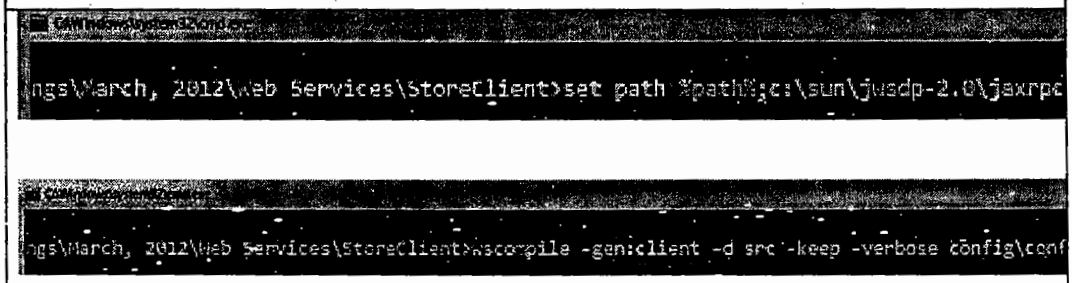
Create a Config folder in the Project directory; create config.xml inside the config folder created. Now configure the wsdl location of the provider for whom you want to generate the binding classes.



Run wscompile

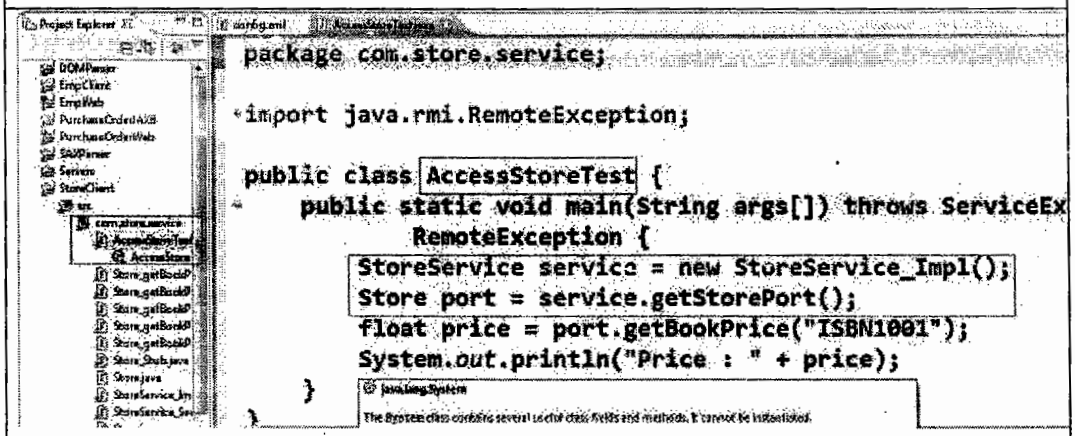
Navigate to the Project directory. Set the Path pointing to the location of wscompile tool. Run the wscompile tool by giving above config.xml as input.

Note: - While running the wscompile tool, the provider should be up and running.



Create Test class

Create Test class, in which create the Object of SEI interface with the help of Service as shown below. And run the client will be able to get the data from the Provider.



11.2.1 Dynamic Proxy

In this type of consumer, we don't need to generate any binding classes or Stub object at the design time. The Stub (implementation of SEI interface at the client side) will get generated on fly at runtime within the JVM.

In order to generate the Stub class at runtime, we need to use the JAX-RPC API specific classes, who know how to generate it.

First the programmer has to browse the WSDL document. In the WSDL document he needs to identify the PortType, operations and input/output messages, from which he needs to build the SEI interface with methods, parameters and return types.

Once the SEI interface has been build, we need to use couple of JAX-RPC API classes to provide the Stub implementation of the SEI interface. The steps for the same are detailed below.

1) Create ServiceFactory – Stub is the implementation of SEI interface which acts as a Port through which we communicate to the Provider. But we don't know how to create the Port Object. As we know Service knows how to create the Port Object, we need to create the javax.xml.rpc.Service object. But Service is an Interface and to Instantiate the Implementation of this Interface, we need to approach ServiceFactory.

   ServiceFactory sfactory = ServiceFactory.newInstance();

2) Create Service from the ServiceFactory – As Service acts as factory for creating the Port, from the ServiceFactory we need to create the Service. While creating the Service, we need to pass two parameters as input. A) WSDL Url. B) qName of the Service. So, while creating the Service Object, it loads the WSDL document and searches for the existence of the Service qName in the WSDL document. If available, with the definitions of that Service (references to the Port) will creates the Service.

   Service service = sfactory.createService(new java.net.URL(wsdlurl), new QName(nmspc, serviceName));

3) Call getPort method – On the service created above, we need to call the getPort method by passing qName of the Port and SEI Interface Type. By reading the Port definition, the getPort method will try to generate a Stub class at runtime by implementing the SEI interface which is passed as reference and returns the Object of it.

   Store port = service.getPort(new QName(nmspc, portName), Store.class);

Once we have the port, we can any methods on the Port which inturn makes a remote call to the Provider. The following code snippet shows how to build a Dynamic Proxy based client.

```
package com.store.service.dp.client;

public class StoreDPClient {
    private static final String WSDL_URL =
"http://localhost:8080/StoreWeb/store?WSDL";
    private static final String SERVICE_NM = "StoreService";
    private static final String PORT_NM = "StorePort";
    private static final String TRG_NMSPC =
"http://safaribooks.com/books/sale";

    public static void main(String[] args) throws ServiceException,
                MalformedURLException, RemoteException {
        ServiceFactory factory = ServiceFactory.newInstance();
        Service storeService = factory.createService(
                    new java.net.URL(WSDL_URL), new
QName(TRG_NMSPC, SERVICE_NM));

        Store port = (Store) storeService.getPort(
                    new QName(TRG_NMSPC, PORT_NM), Store.class);
        System.out.println("Price : " + port.getBookPrice("ISBN1001"));
    }
}
```

11.2.2 Dynamic Invocation Interface

It is similar to Java reflection technic. In reflection we would be able to call a method on a class using the class name and method name dynamically at runtime on fly. Similarly with Dynamic Invocation Interface, a client can call a remote procedure even the signature of the remote procedure or the name of the service is unknown until runtime.

In case of Stub or DP based client, we need to have SEI interface or generated classes to access the Provider; unlike in case of DII we just need couple of things like Service Name, Port Name, Operation Name and Target Endpoint address etc to access the Service.

Steps to build a DII based client

1) Create the ServiceFactory – As we know ServiceFactory is the class who knows how to instantiate the Implementation of Service Interface.

   ServiceFactory sfactory = ServiceFactory.newInstance();

2) Create the Service – We need to create the Service from the ServiceFactory. While create the Service, we need to give the input as qName of the Service from the WSDL Service element.

   Service service = sfactory.createService(new QName(nmspc, servicenm));

3) Create the call – Call is the Object which allows you to call a remote procedure on the Provider. To create the Call object, we need to call createCall method on the Service Object. While creating the call object, to the createCall() method, we need to pass the Port Name from which you want to get the Call object as shown below.

   Call methodCall = service.createCall(new QName(nmspc, portnm));

4) Set the parameters on the Call object – The call just we retrieved is a generic Call object which is not pointing to any specific endpoint. Now we need to set the values like method name, Target Endpoint Address, Input/Ouput paramters etc.

   methodCall.setTargetEndpointAddress("URL");

   methodCall.setOperationName(new QName(nmspc, "getBookPrice");

   methodCall.addParameter("String_1", new QName(nmspc, typename), ParamterMode.IN);

   methodCall.setReturnType(new QName(nmspc, typename));

5) Along with that we need to set the below properties

   SOAPACTION_URI_PROPERTY

   SOAPACTION_USE_PROPERTY

   ENCODING_STYLE_PROPERTY

6) Invoke the method on the Call Object – To invoke the remote procedure on the service, we need to call Call.invoke(new Object[]{}) by passing required inputs as Object[] where it returns the response back to the user.

The below code snippet shows the complete example on the same.

```
package com.store.service.dii.client;

public class StoreDIIClient {
	private static final String ENDPOINT_URL =
"http://localhost:8080/StoreWeb/store";
	private static final String SERVICE_NM = "StoreService";
	private static final String PORT_NM = "StorePort";
	private static final String TRG_NMSPC =
"http://safaribooks.com/books/sale";
	private static final String METHOD_NM = "getBookPrice";
	private static final String TYP_NMSPC =
"http://www.w3.org/2001/XMLSchema";

	public static void main(String[] args) throws ServiceException,
			RemoteException {
		ServiceFactory factory = ServiceFactory.newInstance();
		Service storeService = factory.createService(new
QName(TRG_NMSPC,
				SERVICE_NM));

		Call getBookPriceCall = storeService.createCall(new
QName(TRG_NMSPC,
				PORT_NM), new QName(TRG_NMSPC, METHOD_NM));
		getBookPriceCall.setTargetEndpointAddress(ENDPOINT_URL);
		getBookPriceCall.setReturnType(new QName(TYP_NMSPC, "float"));
		getBookPriceCall.addParameter("String_1",
				new QName(TYP_NMSPC, "string"),
ParameterMode.IN);
		getBookPriceCall.setProperty(Call.SOAPACTION_USE_PROPERTY,
true);
		getBookPriceCall.setProperty(Call.SOAPACTION_URI_PROPERTY,
"");

	getBookPriceCall.setProperty(Call.ENCODINGSTYLE_URI_PROPERTY,
				"http://schemas.xmlsoap.org/soap/encoding/");
		Object price = getBookPriceCall.invoke(new Object[] { "ISBN1001"
});
		System.out.println("Price : " + price);
	}

}
```

WSDL

12 Web Service Description Language (WSDL)

WSDL stands for Web Service Description language. It is the document that describes the information about the Provider. From architectural point of view, WSDL acts as a contract between consumer and provider. WSDL describes the information about the provider in an interoperable manner.

WSDL plays a crucial role even in Web Service development, where service can build from the WSDL document. Only way of consumer knowing the information about the provider is by looking at the WSDL document. As per BP 1.1, WSDL 1.1 is used in describing the information about the service. WSDL 2.0 is also has been finalized, but yet no one started adopted it, and if we uptake WSDL 2.0, interoperability would become a challenge.

WSDL contains total six elements. Let us try to understand various elements of the WSDL document. WSDL is also an XML, every XML will starts with PROLOG and every XML has a root element. If WSDL is also an XML document, it also starts with PROLOG and has root element <definitions>. Including <definitions> there are six elements.

<definitions name="SERVICE_NAME " xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">	This is the root element of the WSDL document. It indicates the rest of the elements in the document provide the definition of the service.
<types>	Types section defines the input/output types of a Web Service method. If your web service method takes any Objects as input or returns any Object as output, those relevant complex type representations are declared under <types> section of the WSDL document. Either the complex types are declared directly or a Schema document is imported in the <types> section as shown below.

For e.g.

I have a Provider shown below who has a method enroll taking the parameter as MemberShipInfo and returning MemberCard

```
public interface PolicyRegistration extends Remote {
   public MemberCard enroll(MemberShipInfo) throws RemoteException;
}
```

As the method takes parameters and return types as objects, the equivalent XSD complex types representing the input/output XML is declared under <types> section.

```
<types>
  <xsd:schema targetNameSpace = "http://ebay.in/sales/types">
    <xsd:complexType name="MemberShipInfo">
            <xsd:sequence>
                  <!-- elements representing attributes of the class -->
            </xsd:sequence>
    </xsd:complexType>
        <xsd:complexType name="MemberCard">
            <xsd:sequence>
                  <!-- elements representing attributes of the class -->
            </xsd:sequence>
    </xsd:complexType>
  </xsd:schema>
</types>
```

Note: - In the above types section, we have an inline XSD document. And all the complex types are declared under target namespace.

<message>	The next element after <types> is <message>. <message> represents the input and output of a Web Service method. Let's say the provider method is taking two parameters, the consumer cannot send those two values independently, rather those has to be wrapped under single message as two parts. A Web Service method takes input as parameters and output as return values, to represents all of its inputs, we need one input <message> and to represents its output, we need one output <message>.

For e.g.

The provider method is taking two parameters as shown below.

```
public interface PolicyRegistration {
        public MemberCard enroll(MemberInfo memberInfo, PolicyInfo policyInfo)
throws RemoteException;
}
```

In the above, the enroll method is taking two parameters, in order to call this method, the consumer shouldn't pass two request to send the inputs to the method rather both the parameters has to be wrapped under one input <message> as two parts, Where each part represents one parameter of the method. And the output of the method is represented with a different <message>

```
<message name="PolicyRegistration_enroll">
        <part name="memberInfo" type="pt:MemberInfo"/>
        <part name="policyInfo" type="pt:PolicyInfo"/>
</message>
<message name="PolicyRegistration_enrollResponse>
        <part name="result" type="pt:MemberCard"/>
</message>
```

A message has name which allows to refer/identify it. Generally it would be named as SEI Interface_<method> – Input message and SEI Interface_<method>Response – Output message.

Note: - In the above message "pt:" represents the prefix namespace under which the complex types are declared under.

<PortType>	<PortType> declares the operations of the Service. PortType exactly represents the SEI interface. Generally an SEI interface doesn't provider any Implementation, rather it providers information like what are the methods and their parameters and return types. Similarly PortType describes about the Service operations and their input/output types. It doesn't provide transport specific information about the service.

For e.g.

For the earlier show SEI interface, the PortType representation is shown below.

```
<PortType name="PolicyRegistration">
        <operation name="enroll">
                <input message="tns:PolicyRegistration_enroll"/>
                <output message="tns:PolicyRegistration_enrollResponse"/>
        </operation>
</PortType>
```

In the above PortType declaration, we declared one operation enroll who has input/output as messages, which are referring to the messages we declared under messages section.

Note:- "tns:" represents the namespace under which the <message> elements are declared. Generally, for a WSDL also we have targetNameSpace declaration and all the message, PortType and other elements of WSDL will be by default binding to that targetNameSpace.

<binding>	As <PortType> is an abstract type, it doesn't talk about transport related aspects of the Service. Binding provides implementation of the PortType where in it describes how is the data being transported between Consumer and Provider. Which protocol is being used for transmission and what is the binding protocol. Along with this it describes how is the messages are exchanged between consumer and provider and in which format as well. In the binding while definition the operations we will specify soap:action for each operation. soapAction is used for resolving an input request to a method on the provider.

SoapAction: - While consumer sending the request to the provider, he will place soapAction (of the method we want to invoke from WSDL) in the HTTP Request header. Once the Provider has received it, he will identify the respective method to invoke based on the soapAction value in the request header.

For e.g.

For the above <PortType>the binding looks like below.

```
<binding name="PolicyRegistrationBinding" type="tns:PolicyRegistration">
      <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
style="rpc"/> <!-- Indicates soap over http -->
   <operation name="enroll">
            <soap:operation soapAction=""/>
            <input>
                  <soap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" use="encoded"
namespace="http://ebay.in/sales/types"/>
            </input>
            <output>
                  <soap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" use="encoded"
namespace="http://ebay.in/sales/types"/>
            </output>
      </operation>
</binding>
```

In the above binding we define the transport type as soap over http. Along with this we specifying the style and use as rpc and encoded respectively.

<service>	Service acts as a Factory for create the <Port> objects. Port attaches an endpoint to the Binding defined above. <service> element wraps several ports under it and is responsible for creating the port objects at the consumer side.

For e.g.

For the above binding, the service declaration is shown below.

```
<service name="PolicyRegistrationService">
   <port name="PolicyRegistrationPort" binding="tns:PolicyRegistrationBinding">
            <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
      </port>
</service>
```

Apache Axis

13 JAX-RPC API (Apache Axis)

Apache Axis is formerly known as "Apache SOAP". There after it's been named as Apache Axis stands for "Apache eXtensible Interaction System". Axis is an implementation of SOAP protocol. It shields you from the details of dealing with SOAP and WSDL. It is one of the open source implementation of JAX-RPC API.

Few points about Axis have been described below.

1) Axis latest stable release Axis 1.4 Final
2) Axis is written completely in Java
3) Even c++ implementation is being planned for writer clients
4) A server which plugs into servlet engine such as Tomcat
5) Extensive support for Web Service Description language
6) It has tools which can generate Java classes from WSDL and WSDL from Java classes
7) A tool for monitoring TCP/IP Packets

Salient Features for Apache Axis

1) Apache axis uses SAX based processing, results in faster parsing.
2) The Axis architecture gives the developer complete freedom to insert extensions into the engine and system management
3) You can easily define reusable handlers to implement common patterns of processing
4) Apache Axis supports various transports; it provided simple abstracts on SOAP over various protocols like SMTP, FTP etc.
5) It supports WSDL 1.1

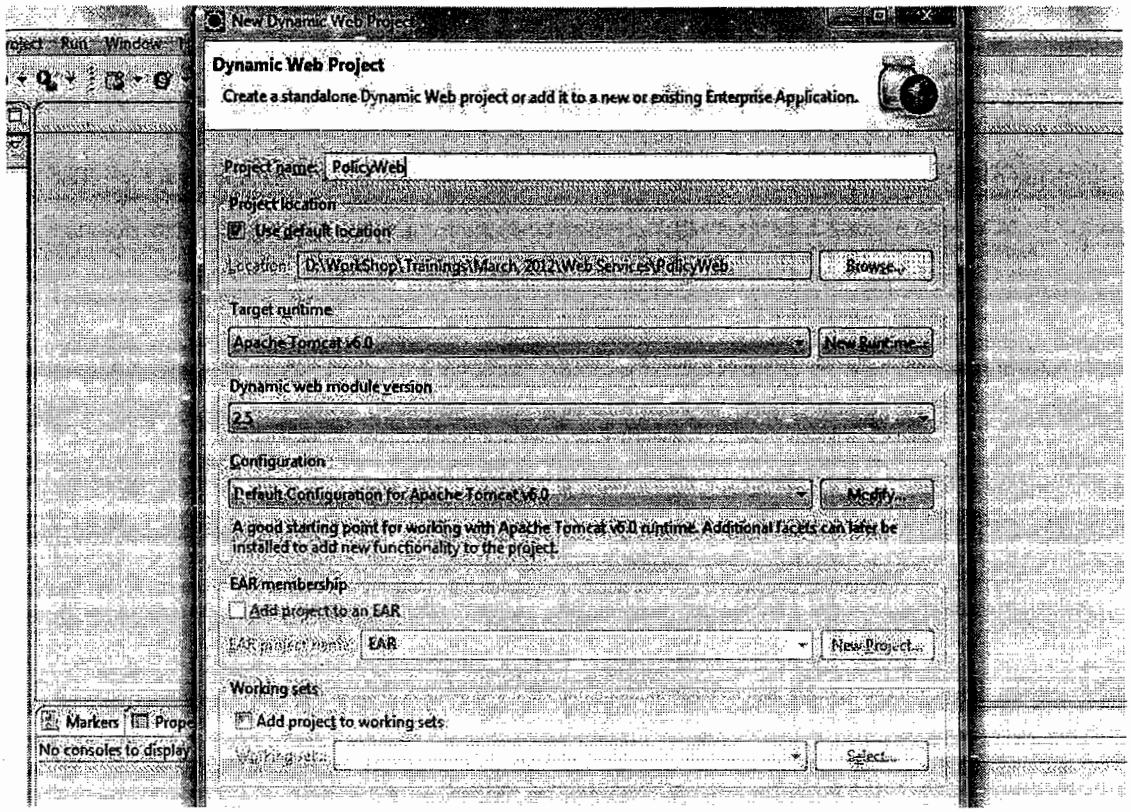
13.1 Configuring Apache Axis

In order to work on Apache Axis, first the developer has to configure the environment as shown below.

1) Download the Apache Axis distribution from the http://www.apache.org/dyn/closer.cgi/ws/axis/1_4 or axis-bin-1_4.zip is available in software CD under "Libraries and Distributes" folder. Extract and copy the contents of the zip C:\ drive. For example "c:\axis-1_4\"

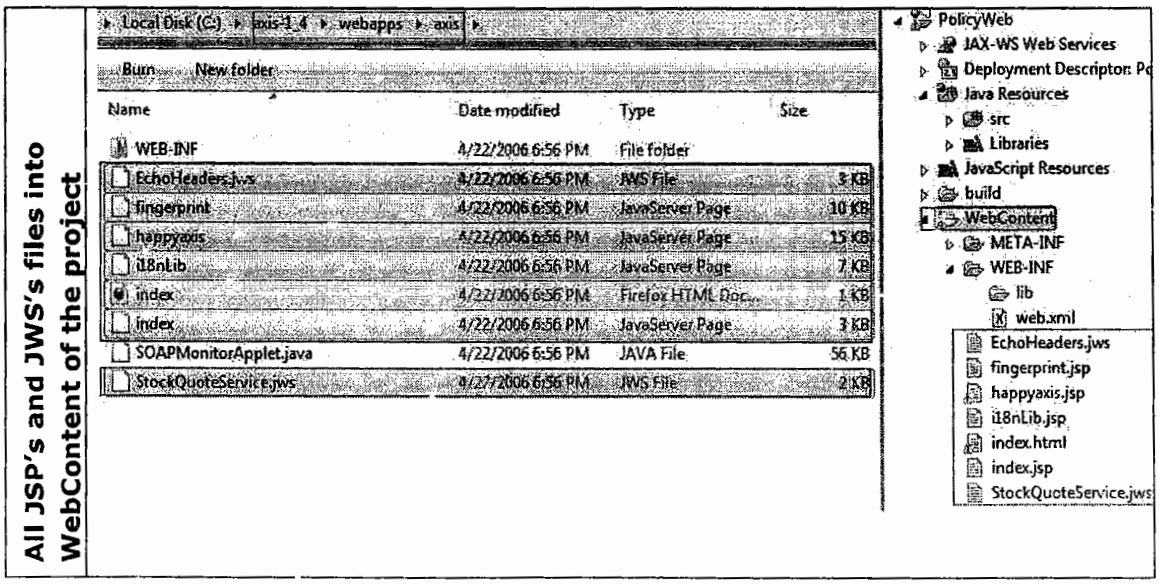
2) Copy mail.jar and activation.jar files from "Jar Bundles"\Apache Axis\ folder to c:\axis-1_4\lib directory.

3) Create a new Eclipse "Dynamic Web Project" as shown below.

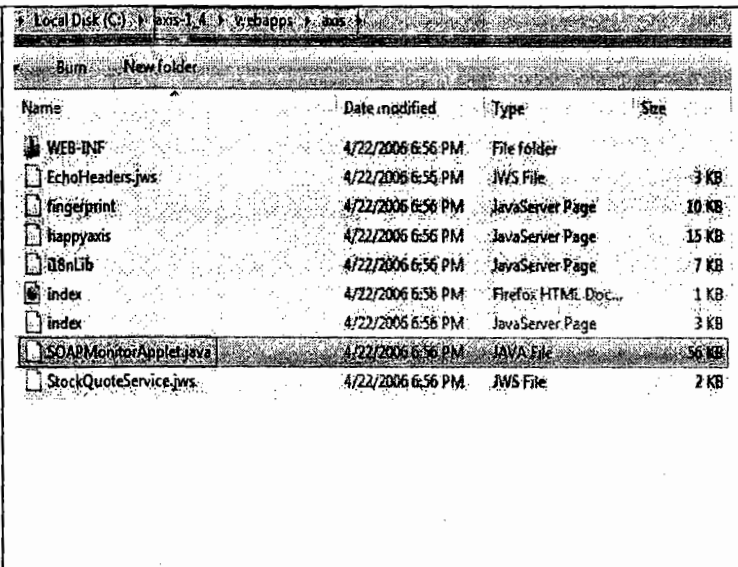


4) Copy all the Jar's from Jar Bundles\Apache Axis folder to project WEB-INF\lib directory

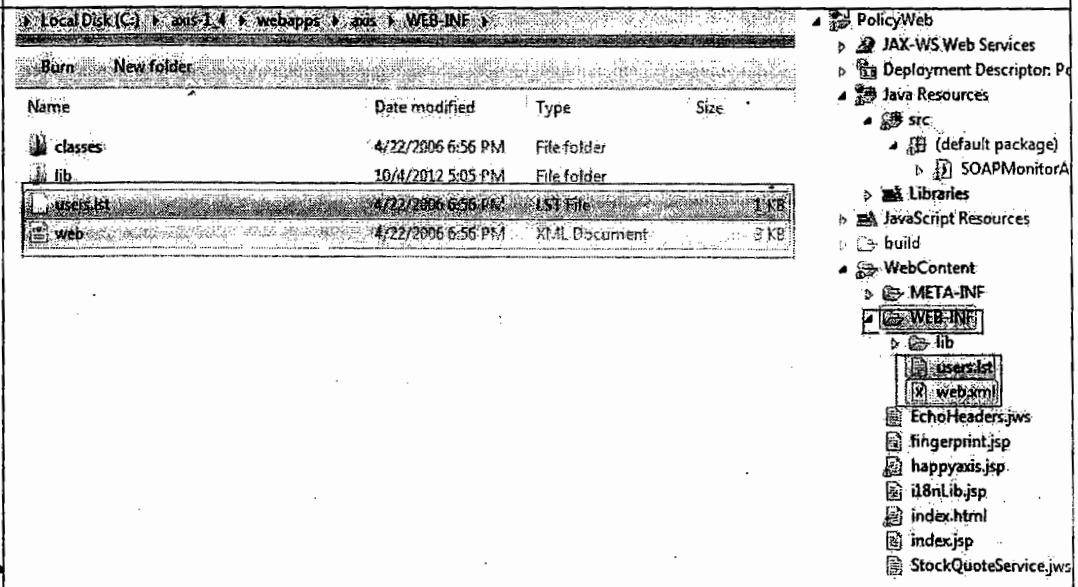
5) Now navigate to C:\axis-1_4\webapps\axis folder and copy the resources to your project as shown below.

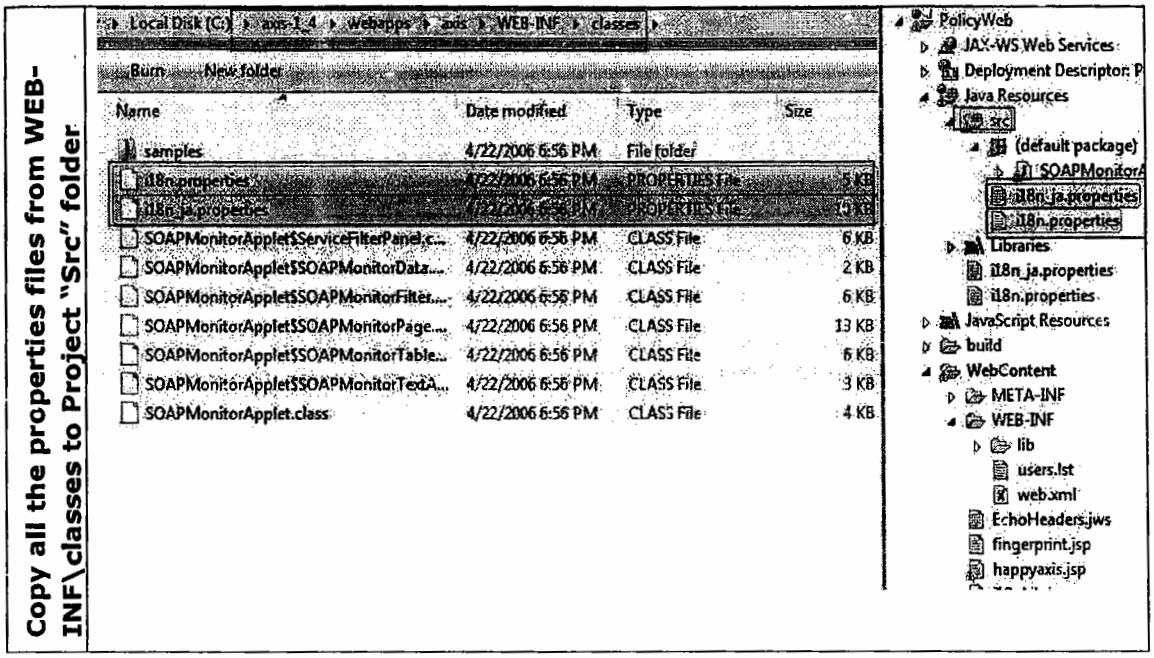


Copy all the .Java files to "src"

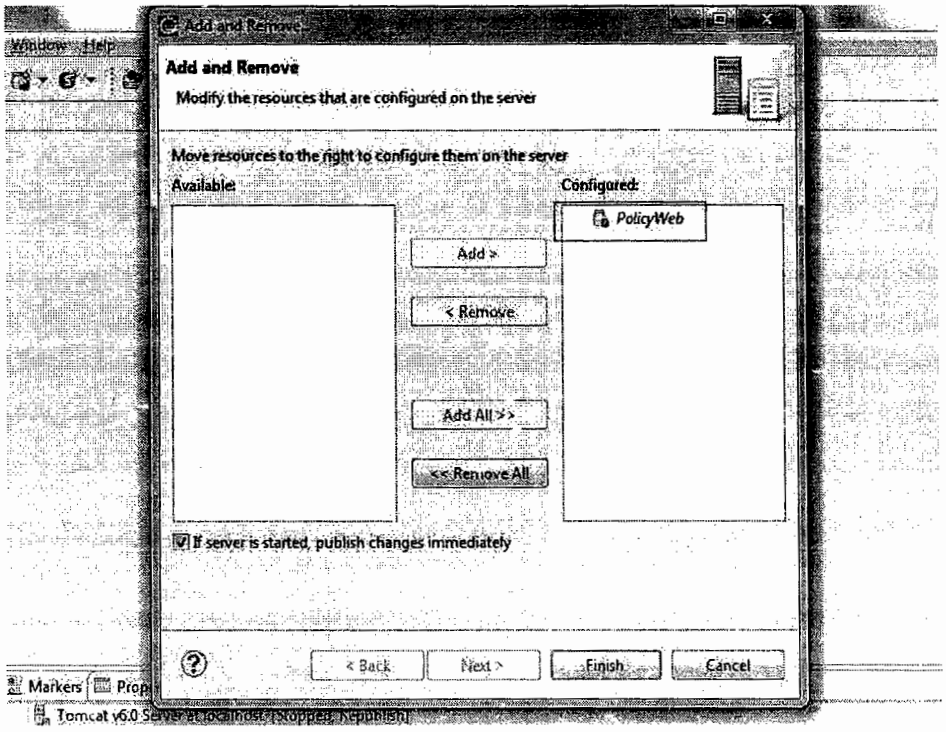


Copy contents of WEB-INF folder to Project WEB-INF





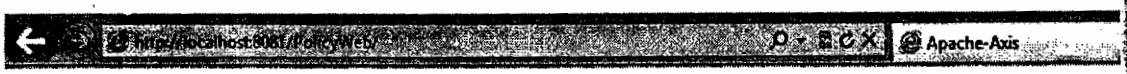
6) With the above we completed configuring the project, now we need to check the environment. To do it first deploy the Project on the Tomcat server configured in IDE as shown below.



7) Start the Server and browse to the context root of the application

http://<adminhost>:<port>/PolicyWeb

This will display the happy axis home page shown below. You can click on validate link will validate and shows whether axis has been configured properly or not.



Apache-AXIS

Hello! Welcome to Apache-Axis.

What do you want to do today?

- Validation - Validate the local installation's configuration
  see below if this does not work.
- List - View the list of deployed Web services
- Call - Call a local endpoint that list's the caller's http headers (or see its WSDL).
- Visit - Visit the Apache-Axis Home Page
- Administer Axis - [disabled by default for security reasons]
- SOAPMonitor - [disabled by default for security reasons]

To enable the disabled features, uncomment the appropriate declarations in WEB-INF/web.xml in the webapplication and restart it.

Validating Axis

13.2 Contract First (JAX-RPC API, Apache Axis, Servlet Endpoint, Sync req/reply)

As you know there are two parts in building a Web Service, Provider and Consumer. Axis support both development of Provider as well as consumer. It allows you to build the provider using contract first as well as contract last approach. Let us build JAX-RPC API based Apache Axis based service using Contract First approach. Steps are described below.

1) As we are building the Provider using contract first approach. We need to start development with the WSDL. I want to build a PolicyRegistration service, it contains a method enroll which takes the parameters as MemberInformation and PolicyInformation and returns MemberCard to the consumer. The Java equivalent of it is shown below.

```
public interface PolicyRegistration extends Remote{
  public MemberCard enroll(MemberInformation mInfo, PolicyInformation pInfo)
throws RemoteException;
}
```

Now we need to write the WSDL document which describes the information for the above service. The input/output types of the methods MemberInformation,

PolicyInformation and MemberCard java class equivalent complex types has to be declared in <types> section.

<message> represents the input and output of the Method. PolicyRegistration SEI interface is being represented by <PortType> section and remain sections will describe the transport specific information.

Create the WSDL document under WEB-INF folder. The WSDL for the above described service is shown below.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
	xmlns:tns="http://licindia.org/policy/wsdl"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
	xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="policyService"
	xmlns:pt="http://licindia.org/policy/types"
targetNamespace="http://licindia.org/policy/wsdl">
	<wsdl:types>
		<xsd:schema targetNamespace="http://licindia.org/policy/types">

			<xsd:complexType name="MemberInformation">
				<xsd:sequence>
					<xsd:element name="ssn" type="xsd:string" />
					<xsd:element name="name" type="xsd:string"
/>
				</xsd:sequence>
			</xsd:complexType>

			<xsd:complexType name="PolicyInformation">
				<xsd:sequence>
					<xsd:element name="policyId"
type="xsd:string" />
					<xsd:element name="tenure" type="xsd:int" />
				</xsd:sequence>
			</xsd:complexType>

			<xsd:complexType name="MemberCard">
				<xsd:sequence>
					<xsd:element name="enrollmentId"
type="xsd:string" />
					<xsd:element name="coverageType"
type="xsd:int" />
				</xsd:sequence>
			</xsd:complexType>

		</xsd:schema>
	</wsdl:types>
```

```
<wsdl:message name="PolicyRegistration_enroll">
        <wsdl:part name="mInfo" type="pt:MemberInformation" />
        <wsdl:part name="pInfo" type="pt:PolicyInformation" />
    </wsdl:message>
    <wsdl:message name="PolicyRegistration_enrollResponse">
        <wsdl:part name="result" type="pt:MemberCard" />
    </wsdl:message>
    <wsdl:portType name="PolicyRegistration">
        <wsdl:operation name="enroll" parameterOrder="mInfo pInfo">
            <wsdl:input message="tns:PolicyRegistration_enroll" />
            <wsdl:output
message="tns:PolicyRegistration_enrollResponse" />
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="PolicyRegistrationSOAPBinding"
type="tns:PolicyRegistration">
        <soap:binding style="rpc"
            transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="enroll">
            <soap:operation
soapAction="http://licindia.org/policy/wsdl/enroll" />
            <wsdl:input>
                <soap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                    namespace="http://licindia.org/policy/wsdl"
use="encoded" />
            </wsdl:input>
            <wsdl:output>
                <soap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                    namespace="http://licindia.org/policy/wsdl"
use="encoded" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="PolicyService">
        <wsdl:port binding="tns:PolicyRegistrationSOAPBinding"
name="PolicyRegistrationPort">
            <soap:address location="http://www.example.org/" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

2) After writing the WSDL, we need to generate Java classes out of It. The classes would be generated based on the PortType, their input/output message and bindings sections. Along with that it will generate binding classes required to facilitate the serialization/de-serialization of objects to XML.

To generate the Java classes from WSDL document, Axis has provided a tool called WSDLToJava, we need to run this by giving WSDL as an Input. Unlike wscompile tool, WSDLToJava is not a ".bat" file rather it's a Java class we need to run by setting the class path to all the jars in c:\axis-1_4\lib.

First we need to prepare the CLASSPATH variable pointing to the entire jar's in the axis distribution lib directory as shown below.

```
set AXISPATH=c:\axis-1_4\lib

set AXISCLASSPATH=%AXISPATH%\activation.jar;%AXISPATH%\axis-
ant.jar;%AXISPATH%\axis.jar;%AXISPATH%\commons-discovery-
0.2.jar;%AXISPATH%\commons-logging-
1.0.4.jar;%AXISPATH%\jaxrpc.jar;%AXISPATH%\log4j-
1.2.8.jar;%AXISPATH%\mail.jar;%AXISPATH%\saaj.jar;%AXISPATH%\wsdl4j
-1.5.1.jar
```

Now we need to run the Java class org.apache.axis.wsdl.WSDL2Java by giving the input as wsdl file. Before executing switch to the project directory and then execute the command using the below switches.

```
java -cp %AXISCLASSPATH% org.apache.axis.wsdl.WSDL2Java --server-side -
-skeletonDeploy false --verbose -o src WebContent\WEB-
INF\policyService.wsdl
```

--server-side	Generate server side classes for building provider
--verbose	Display the output
-o	Generate the output files in specified directory
-skeletonDeploy	Generates the Skeleton class

When we run the above command it will generates the following files

PolicyRegistration	SEI Interface
PolicyRegistrationSOAPBindingImpl	Implementation class with empty logic in methods
PolicyRegistrationSOAPBindingStub	Consumer class (not required)
PolicyService, PolicyServiceLocation	Consumer classes (not required)
MemberCard, MemberInformation, PolicyInformation	Input/Output classes containing serialization logic as well.
Deploy.wsdd, UnDeploy.wsdd	These two files web service deployment descriptors that are used for deploy and undeploy of the service.

3) Drag and Drop Deploy.wsdd and UnDeploy.wsdd to the WEB-INF directory of the project
4) Write business logic in enroll method of PolicyRegistrationSOAPBindingImpl class to expose it as Web Service.
5) Deploy the Project and start the Server, this will deploys the project and gives a warning that server-config.wsdd is not found and generating one default.

   If you refer to the request processing model below, similarly JAX-RPC API JAXRPCServlet which acts as an engine in processing the request. Here the AxisServlet plays a crucial role in handling and processing the request. When a request comes to AxisServlet, it looks up in server-config.wsdd at runtime to identify the respective Implementation class to process the request.

   But this file will not be generated when we run WSDL2Java instead we need to generate this file on fly in the server to deploy the service; this server-config.wsdd file will be generated on the server side based on the information described in deploy.wsdd.

   To do this, we need to run a tool provided by axis called AdminClient. This is also a Java class, to this tool we need to give two inputs.
   a) Url of the AxisServlet – We need to give the –l url of the AxisServlet on to which you want to deploy/generate the server-config.wsdd.
   b) Input as deploy.wsdd – path to the deploy.wsdd

   The AdminClient will reads the contents of the deploy.wsdd and upload it to the AxisServlet where the AxisServlet upon receiving the request will generates the server-config.wsdd containing our service information.

```
Java –cp %AXISCLASSPATH% org.apache.axis.client.AdminClient –
lhttp://<adminhost>:<port>/<contextroot>/servlet/AxisServlet deploy.wsdd
```

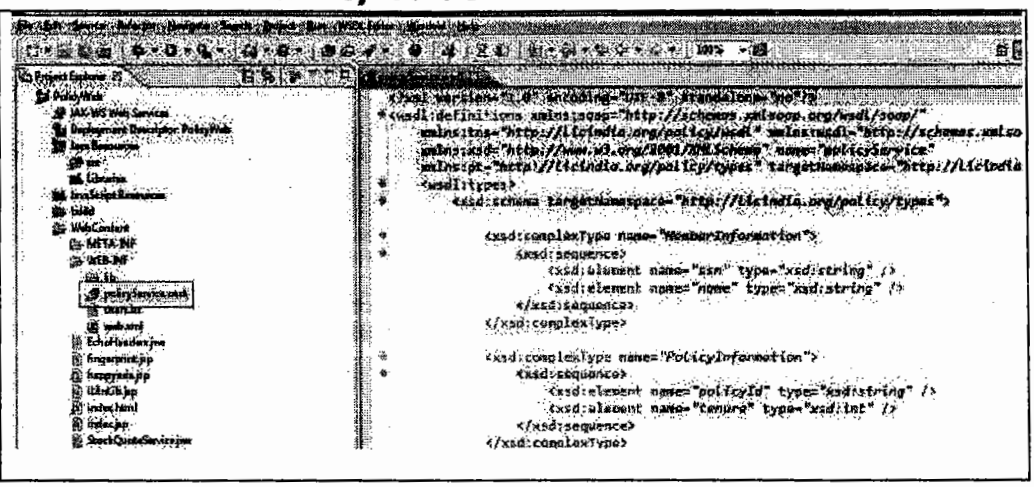
6) Now browse the Happy Axis homepage and click on the list. It will show our service information as well. Click on the WSDL will generate the WSDL document onfly.

13.3 Request Processing Flow

In case of JAX-RPC SI, JAXRPCServlet will handle the incoming request and tries to process it by looking up into jaxrpc-ri-runtime.xml. In case of Apache Axis, AxisServlet will handle the incoming request and looks into server-config.wsdd to find the relevant service configuration and process the request.

13.4 Contract First – Activity Guide

After creating the Project write the WSDL File under WEB-INF directory



Run WSDL2Java

First build the AXISCLASSPATH pointing to the axis distribution lib directory

```
D:\WorkShop\XpertimentL\eclipse\jkbservices\notes\PolicyWeb>set AXISPATH=C:\axis-1_4\lib
```

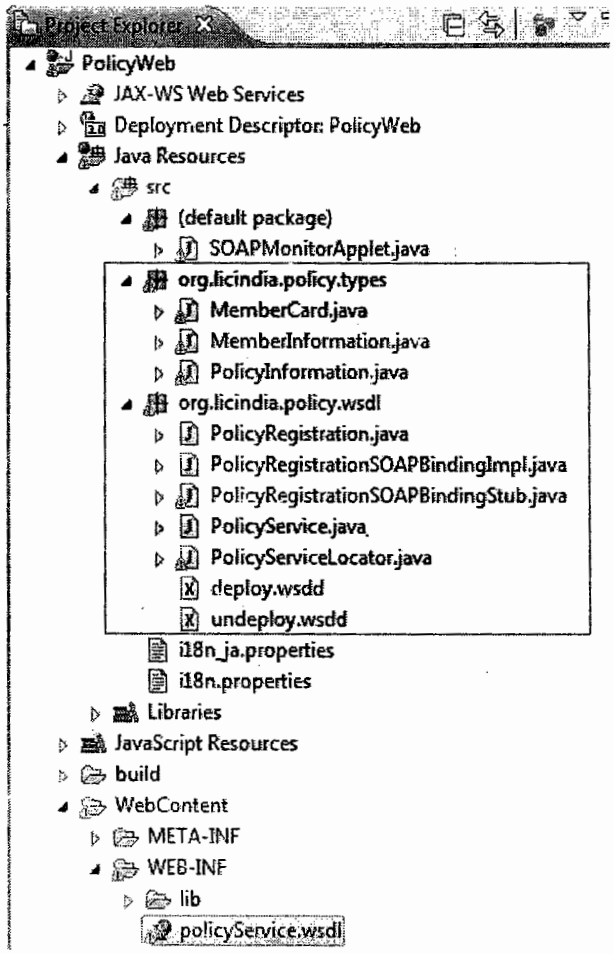
```
D:\WorkShop\XpertimentL\eclipse\jkbservices\notes\PolicyWeb>set AXISCLASSPATH=%AXISPATH%\activation.jar;%AXISPATH%\
xmlj.jar;%AXISPATH%\wsdl4j-1.5.1.jar
```

Note: - For complete command refer to the previous section.

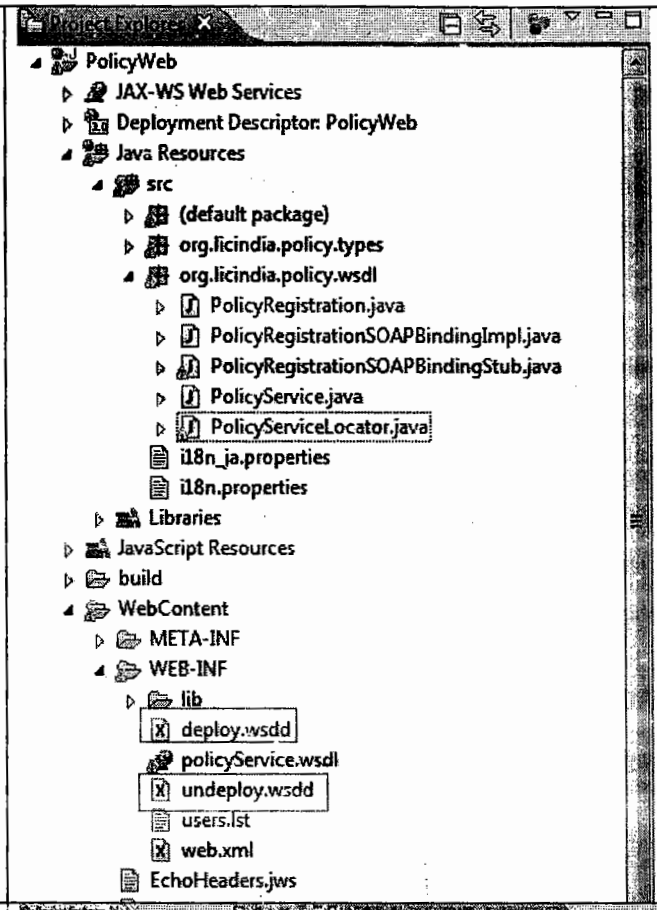
Now switch to the Project directory and run the WSDL2Java command

```
PolicyWeb>java -cp %AXISCLASSPATH% org.apache.axis.wsdl.WSDL2Java --server-side --skeletonDeploy true
--verbose -o src WebContent\WEB-INF\policyService.wsdl
```

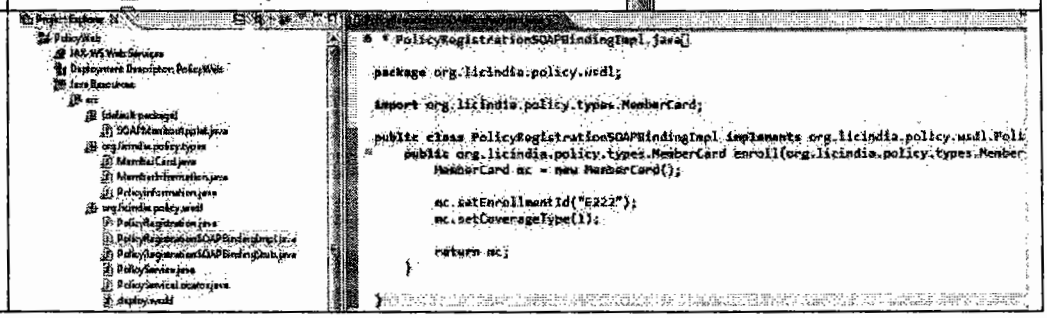
This will generates the classes and deployment descriptors that are required to build the service as shown here



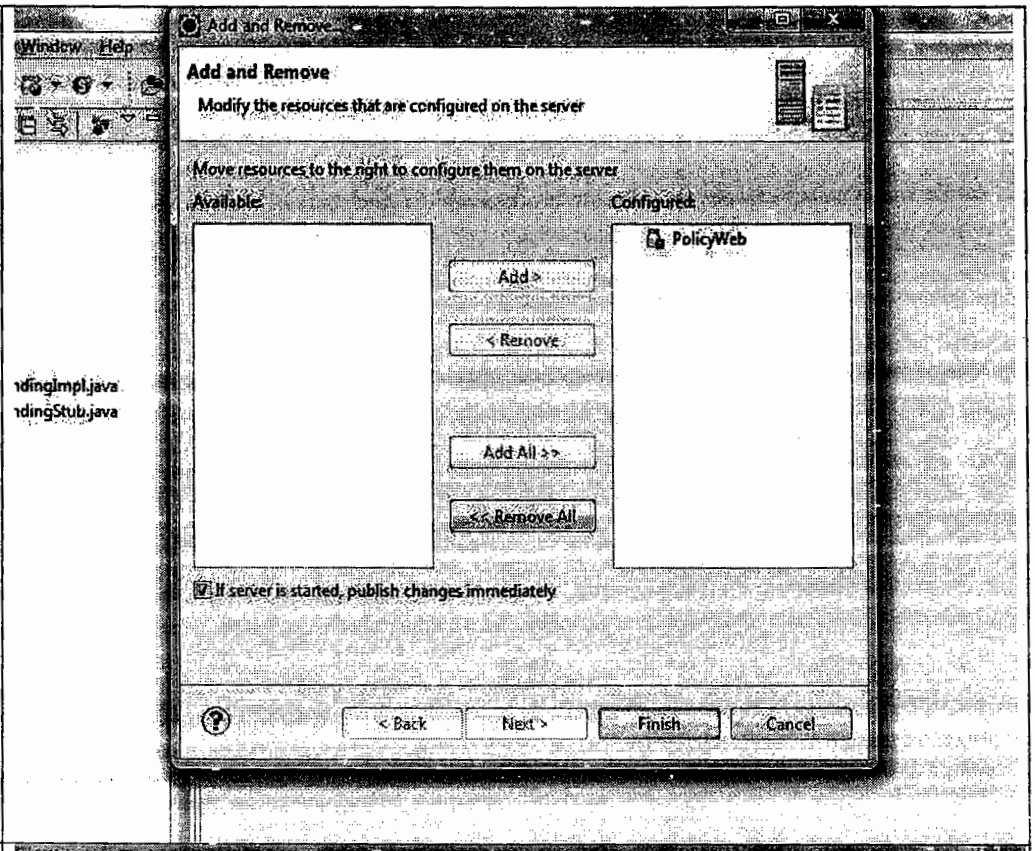
Drag & Drop deploy.wsdd and undeploy.wsdd into WEB-INF folder



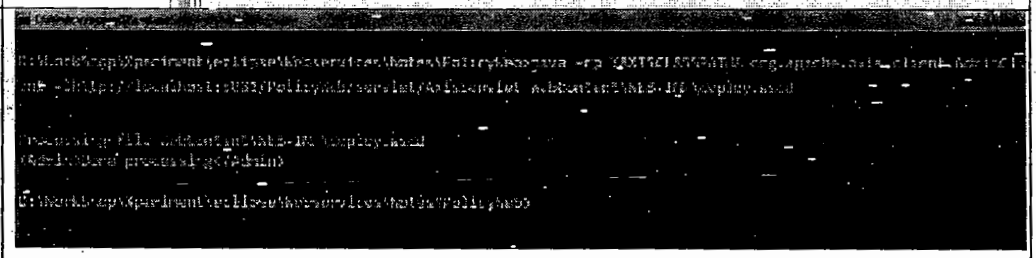
Write Implementation logic



Deploy the Project and Start Server



Now run AdminClient



Now test your service by accessing the happy axis home page and browse the WSDL.

JAX-WS

14 JAX-WS API (Sun Reference Implementation)

JAX-WS stands for Java API for XML Web Services; it is the successor of JAX-RPC API. JAX-WS API adheres to the rules defined in WS-I BP1.1 specification, which means we can build BP 1.1 complaint services using JAX-WS API.

After BP1.1, WS-I has released BP 1.2 and BP 2.0 specifications as well. JAX-WS API adopts even the BP 1.2 and BP 2.0 as well.

BP 1.2 – Added support to WS-Addressing and MTOM
BP 2.0 – Added support to WS-Addressing, MTOM and WSDL 2.0

Before discussing development of JAX-WS API, let us first understand to understand what are the differences between JAX-RPC API and JAX-WS API? or when to use JAX-RPC API and JAX-WS API.

14.1 Difference between JAX-RPC and JAX-WS API

Web Services has been around a while, it was introduced into the market late 90's. First there was SOAP, but SOAP only described what the messages should look like. Then came WSDL, But WSDL doesn't talks about how to write web services in Java or any language.

Then came Java added its support to web services by releasing JAX-RPC 1.0 API. After few months of its use, JCP (Java community folks) felt few modifications are required for the API and has released JAX-RPC 1.1. After few years of its use, JCP folks wanted to build a better version: JAX-RPC 2.0

A primary goal was to align with the industry standards, but the industry is doing message oriented web services (async req/reply). But Java supports JAX-RPC where RPC represents Remote Procedural calls which mean sync req/reply. So, Sun has added its support to Message oriented web services by renaming "RPC" to "WS" which results in JAX-WS API.

Let's talk about what are common between JAX-RPC and JAX-WS API.

1) JAX-RPC API uses SOAP 1.1 over HTTP 1.1 and JAX-API still supports SOAP 1.1 over HTTP 1.1. This means the same message can still flow across the wire. So interoperability is not affected.

2) JAX-RPC supports WSDL 1.1 and JAX-WS along with WSDL 1.1 it even supports WSDL 2.0

Now let us talk about the differences between JAX-RPC and JAX-WS API's

a) SOAP: - JAX-RPC supports SOAP 1.1. JAX-WS also supports SOAP 1.1 along with that it supports SOAP 1.2 and SOAP 2.0.

b) XML/HTTP: - WSDL 1.1 specification has support for HTTP Binding. It means you can send XML messages over the HTTP without SOAP. JAX-RPC ignored the HTTP binding, whereas JAX-WS added support for Http binding.

c) WS-I's Basic Profiles: - JAX-RPC API supports WS-I Basic Profile 1.0. JAX-WS API supports WS-I Basic Profile 1.1.

d) New Java features: - Minimum JDK that is required to work with JAX-RPC API is Jdk 1.4. But JAX-WS relies on many of the features of Java 5. Java 5 adding support to JAX-WS, it even retains support for JAX-RPC API.

e) The data mapping model: - JAX-RPC has its own data mapping model, which covers about 90 percent of all schema types. JAX-WS's data mapping model is JAX-B.

f) The interface mapping model: - JAX-WS API makes use of Java 5 and introduced new interfaces to support Asynchronous programming.

g) MTOM (Message Transmission Optimization Mechanism): - JAX-WS via JAXB adds support for MTOM the new attachment specification. JAX-RPC uses only SAAJ (Soap with attachments API).

h) The handler model: - JAX-RPC handlers use SAAJ 1.2 API to work with SOAP XML whereas JAX-WS handlers rely on SAAJ 1.3.

By the above we understood the differences between JAX-RPC API and JAX-WS API now let us understand how to build a JAX-WS API based SI implementation service using contract last approach.

14.2 Contract Last (JAX-WS API (RI), Servlet Endpoint, Sync req/reply)

In this section let us try to understand how to develop a JAX-WS API Sun RI based web service/provider using Contract Last approach. As you know there are always two approaches in building a Provider, now we are choosing Contract Last approach, where the development starts with Java and generates WSDL out of it.

The following section shows the detailed steps of creating a JAX-WS API RI based service using contract last approach. We will compare various steps in comparison with JAX-RPC SI Implementation to better understand them.

a) Write SEI Interface – As you know from the Programming point of view SEI interface acts as a contract between Provider and Consumer. As it is contract last approach, the development starts with SEI Interface. We need to follow certain rules in writing SEI Interface described below.

JAX-RPC API SI	JAX-WS API RI
a) SEI Interface must extend from Remote Interface, to indicate the methods are accessible remotely b) Declare all the methods you want to expose as web service methods c) Methods should take parameters and return values as serializable d) Methods must throw RemoteException (to indicate the exception is caused by remote commnucation)	a) Write SEI Interface, no need to extend it from Remote interface. But to indicate it as Web Service interface, annotate with @WebService. Along with that we can mark some attributes like name, targetNameSpace etc to indicate endpoint name and namespace of the wsdl b) Declare the methods, here you can choose which methods you want to expose as web service methods. By default all the methods are exposed as Web service methods. Let's say out of 3 methods you want to expose only 1 method as web service method then annotate that method with @WebMethod. You can add attribute to the annotation describing the operationName and soapAction value as well c) Parameters and return types are

	not necessarily serializable, but as per java standards recommended those objects to be serializable. You can mark parameters and return values as @WebParam and @WebResult annotations indicating the partName and targetNameSpace of the XSD Type. d) SEI Interface methods may or may not throw any exception. In JAX-WS API Sun came up with its own Web Service exception hierarchy, where all the Web Service related exceptions should throw WebServiceException.

Now following the above rules let's write an SEI interface shown below.

```
package com.tw.service;

@WebService(name = "reservation", targetNamespace =
"http://irctc.org/reservation/wsdl")
public interface Reservation {
    @WebMethod(operationName = "doBooking", action =
"http://irctc.org/reservation/wsdl#doBooking")
    @WebResult(name = "return", targetNamespace =
"http://irctc.org/reservation/types")
    Ticket doBooking(
            @WebParam(name = "pInfo", targetNamespace =
"http://irctc.org/reservation/types") PassengerInfo pInfo,
            @WebParam(name = "jInfo", targetNamespace =
"http://irctc.org/reservation/types") JourneyInfo jInfo);
}
```

In the above code we have written an SEI interface name reservation and we indicated it as SEI interface with @WebService annotation. Along with that, we marked it name attribute indicating the name of the endpoint and targetNamespace which is the WSDL Namespace of that service.

In addition we have a method doBooking and indicated as Web Service method by annotation it with @WebMethod in which we gave two attributes operationName and soapAction pointing to operationName of PortType and soapAction to map the incoming request to the method.

You can declare typeNamespace for the Input and Output of a service method using @WebParam and @WebResult as well.

b) **Write SEI Interface Implementation class –** SEI Implementation should not implement from SEI interface, but we will denote by using the annotation @WebService(endPointInterface="Qualified name of Interface").

Implementation class methods need not throw WebServiceException. If there is any logic throwing exception, all such exceptions should be wrapped into WebServiceException or any of the sub classes of WebServiceException and should throw to the user.

WebServiceException is an un-checked exception, so program no need to handle it.

```
package com.tw.service;

import javax.jws.WebService;

@WebService(endpointInterface = "com.tw.service.Reservation")
public class ReservationImpl {
    public Ticket doBooking(PassengerInfo pInfo, JourneyInfo jInfo) {
        Ticket ticket = new Ticket();
        ticket.setPnr("PNR24242");
        ticket.setStatus("CONFIRMED");

        return ticket;
    }
}
```

c) **SEI Interface is optional –** In JAX-WS, the SEI interface is not mandatory, it is optional. In case if you haven't written any SEI interface, all the annotations on Interface like WebMethod, WebParam and WebResult has to be declared directly in Implementation class itself. If SEI interface is presented you must declare them at the Interface level, only in the absence of Interface then only you should declare them in the class.

At the consumer side to access the Provider, JAX-WS API automatically generates an SEI interface based on the WSDL to access it, even we don't provide an SEI interface the Provider side.

In absence of SEI interface – Implementation class

```
package com.tw.service;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;

import com.tw.business.JourneyInfo;
import com.tw.business.PassengerInfo;
import com.tw.business.Ticket;

@WebService
public class ReservationImpl {
    @WebMethod(operationName = "doBooking", action =
"http://irctc.org/reservation/wsdl#doBooking")
    @WebResult(name = "return", targetNamespace =
"http://irctc.org/reservation/types")
    Ticket doBooking(
                @WebParam(name = "pInfo", targetNamespace =
"http://irctc.org/reservation/types") PassengerInfo pInfo,
                @WebParam(name = "jInfo", targetNamespace =
"http://irctc.org/reservation/types") JourneyInfo jInfo) {
            Ticket ticket = new Ticket();
            ticket.setPnr("PNR24242");
            ticket.setStatus("CONFIRMED");

            return ticket;
    }
}
```

d) Generate Binding classes - It is similar to the generation of binding classes in JAX-RPC. Always the consumer sends XML embedded in SOAP over HTTP request to the Provider. The incoming XML has to be mapped to method parameters and outgoing return type has to be converted back to XML. This means for the Inputs & Outputs of the methods we need to generate binding classes.

Rather than we writing the binding classes to perform this we use the tool that is provided by JWSDP, nothing but wsgen. Wsgen will takes the input as the annotated classes and generates binding classes to perform marshaling and un-marshalling.

a. What tools are available?

When we install JWSDP it has provided two tools to work on JAX-WS API RI based web services those are

i. Wsgen – It is used for generating binding classes by taking input as Java class

ii. Wsimport – Will takes WSDL as input and generates binding classes for both consumer and provider.

These tools are shipped as part of the below directory.

```
C:\Sun\Jwsdp-2.0
|-Jaxws
    |-lib
    |-bin – wsgen, wsimport
```

Even when we install JDK 1.6 also these tools will be shipped as part of JDK under bin directory.

b. Why to run these tools?

To generate binding classes from Java classes we need to use wsgen tool. Wsgen will take the annotated Implementation class as input and generates binding classes as output. Wsgen tool will directly takes the Implementation class (as SEI interface is optional) as input and generates binding class, we don't need separate configuration file, because we provided configuration information about the class using annotations directly written at class level.

As discussed earlier, JAX-WS will uses JAX-B as the XML Binding technology to perform marshaling and un-marshaling. So if we look at any method we have one input and one output, so we need to JAX-B binding classes who know how to convert XML to Object and wise versa.

c. How to run the wsgen?

Running wsgen is pretty straight, just we need to give annotated implementation class as input, so that it will generates JAX-B binding classes as output. While running the tool we don't need to specify –gen:server or –gen:client, because when it comes to JAX-B the same binding class can be used to convert xml to object and object to xml, so we don't need two different binding classes for client and server.

```
Wsgen –d src –keep –cp build\classes –verbose
com.tw.service.RegistrationImpl
```

When we run the above command, it just generates two JAXB binding classes as output, one for converting input xml to object and other for converting object data back to xml.

e) **Write sun-jaxws.xml –**We need to provide the endpoint configuration, which is known as deployment descriptor. To map our service with an URL, we need to write file called sun-jaxws.xml and should place in WEB-INF directory as shown below.

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

<endpoints xmlns='http://java.sun.com/xml/ns/jax-ws/ri/runtime'
version='2.0'>
   <endpoint
      name='Reservation'
      implementation='com.tw.service.ReservationImpl'
      url-pattern='/bookTicket'/>
</endpoints>
```

If we look at the contents of the above file, it is straight forward and contains Implementation class name and url-pattern with which you want to access it.

f) **Write web.xml –** As you know we are developing an Servlet Endpoint based service, we need to configure a Servlet to handle our incoming request and to process it to return the response. In case of JAX-WS the servlet is WSServlet. Here there is no tool like wsdeploy to generate the sun-jaxws.xml and web.xml the programmer has to manual configure them in these files.

While writing the WSServlet configuration in the web.xml, we need to ensure the url pattern of the servlet and url-pattern of the endpoint in sun-jaxws.xml should be matching so that the servlet once received the request would be able to identify the implementation class to service the request.

Along with the servlet, we need to configure an Listener in the web.xml, the listener will be fired when the application context has been initialized and checks for the file sun-jaxws.xml under WEB-INF. As the file is mandatory, if that file misses in WEB-INF, it will not deploys the application rather throws error while starting the application itself.

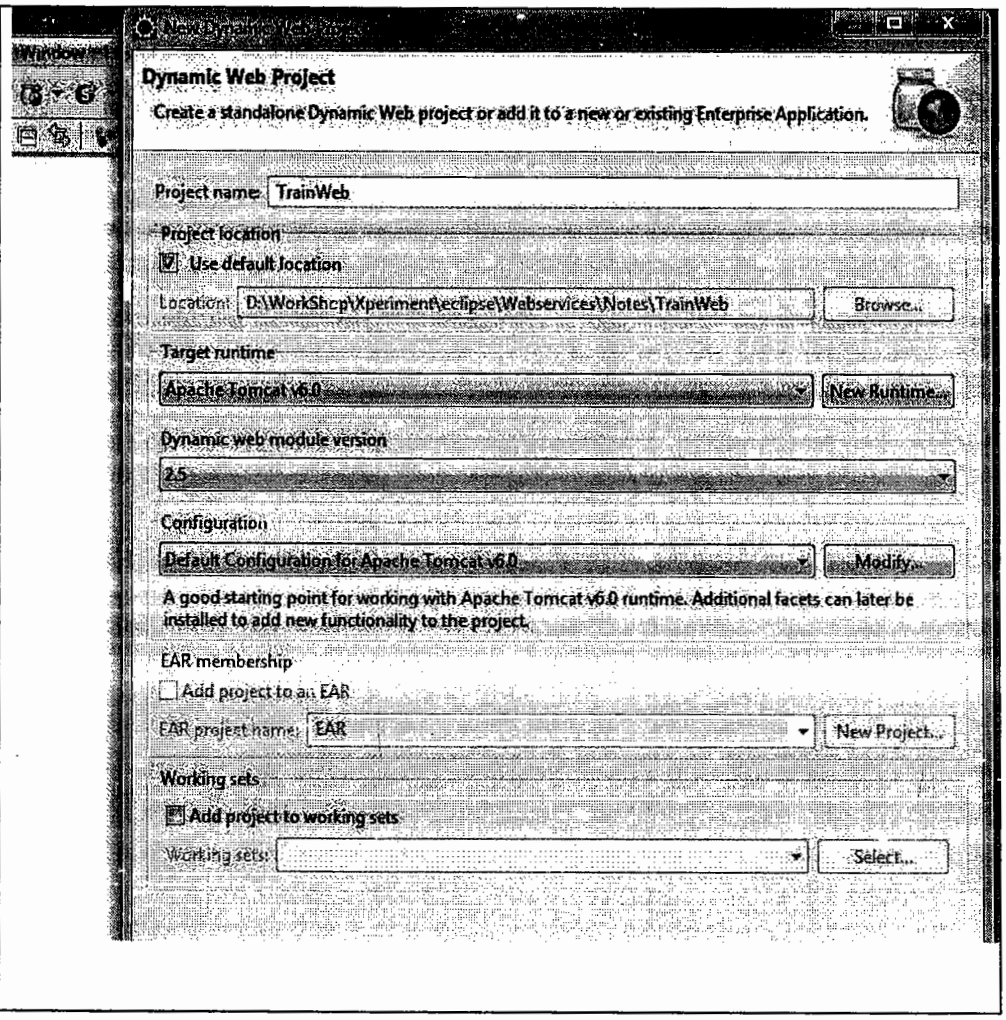
web.xml

```
<?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>TrainWeb</display-name>
    <listener>
        <listener-
class>com.sun.xml.ws.transport.http.servlet.WSServletContextListener</listen
er-class>
    </listener>
    <servlet>
        <servlet-name>WSServlet</servlet-name>
        <servlet-
class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>WSServlet</servlet-name>
        <url-pattern>/bookTicket</url-pattern>
    </servlet-mapping>
</web-app>
```

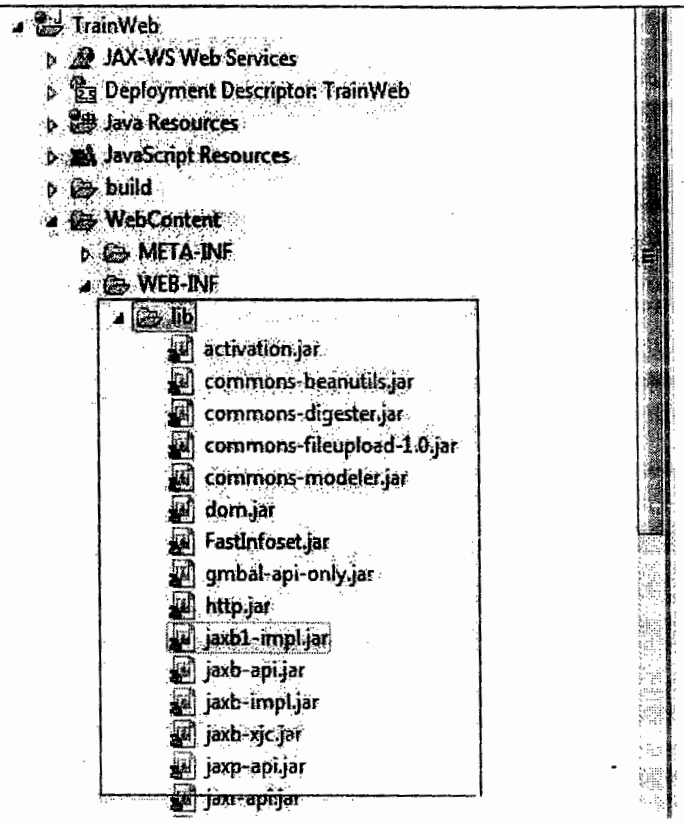
14.3 Contract Last– Activity Guide

The following steps guides you in developing a JAX-WS API RI based consumer using contract last approach.

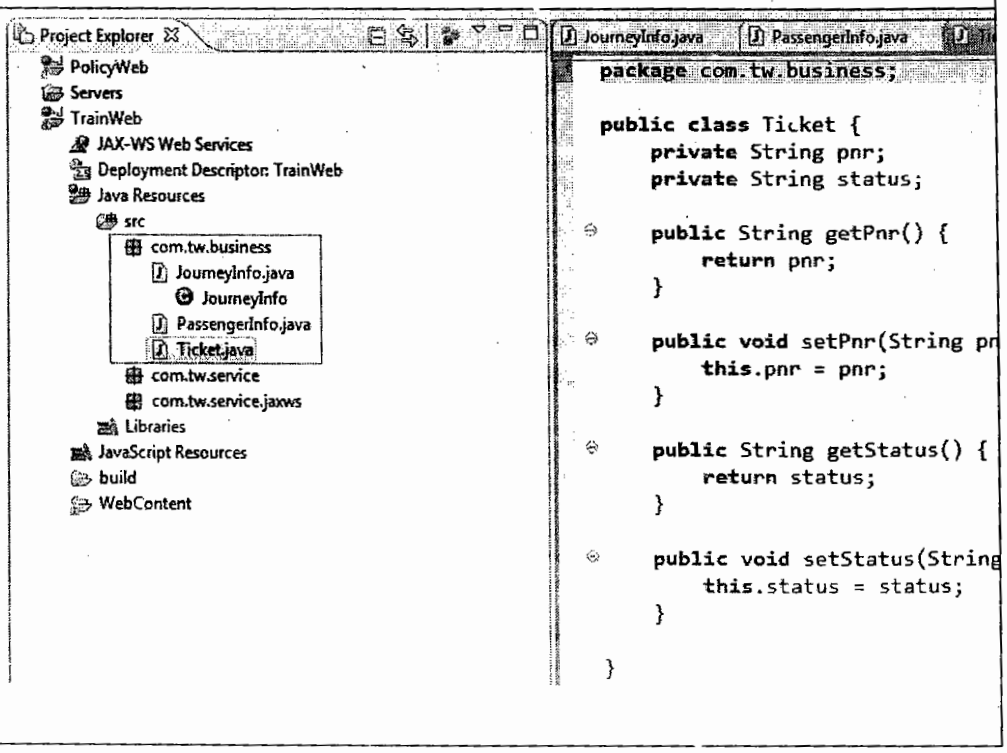
Create a Web project



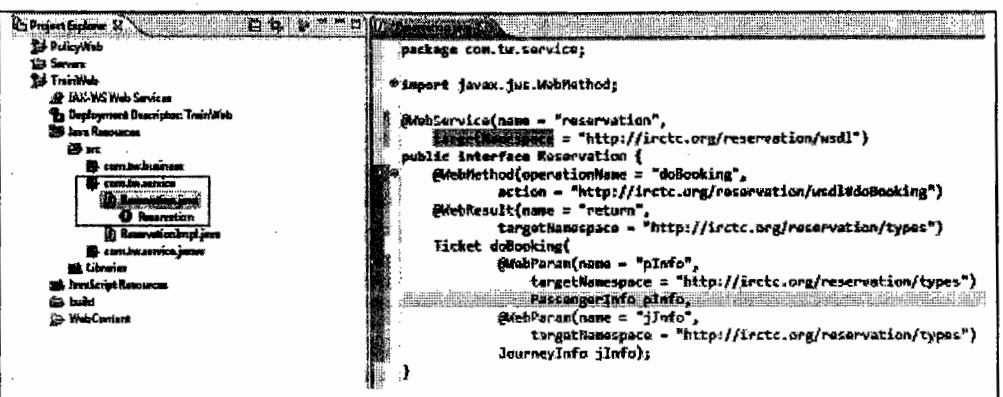
Copy all the Jars into lib



Write Input/output classes of the Service



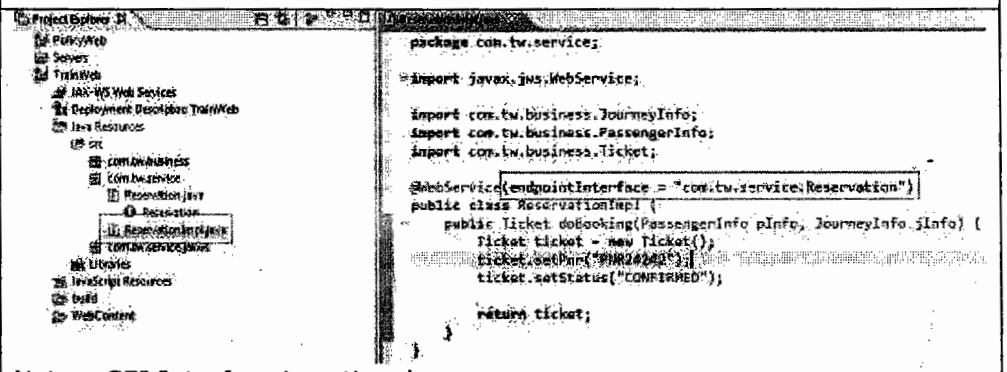
Write SEI Interface



```
package com.tw.service;

import javax.jws.WebMethod;

@WebService(name = "reservation",
    targetNamespace = "http://irctc.org/reservation/wsdl")
public interface Reservation {
    @WebMethod(operationName = "doBooking",
            action = "http://irctc.org/reservation/wsdl#doBooking")
    @WebResult(name = "return",
            targetNamespace = "http://irctc.org/reservation/types")
    Ticket doBooking(
            @WebParam(name = "pInfo",
                targetNamespace = "http://irctc.org/reservation/types")
                PassengerInfo pInfo,
            @WebParam(name = "jInfo",
                targetNamespace = "http://irctc.org/reservation/types")
            JourneyInfo jInfo);
}
```

To indicate as SEI Interface we need to annotate it with @WebService. For a method we can optionally annotate it as @WebMethod. If mark at least one method as @WebMethod, the other methods will not be exposed as Web Service methods.

Write Implementation class



```
package com.tw.service;

import javax.jws.WebService;

import com.tw.business.JourneyInfo;
import com.tw.business.PassengerInfo;
import com.tw.business.Ticket;

@WebService(endpointInterface = "com.tw.service.Reservation")
public class ReservationImpl {
    public Ticket doBooking(PassengerInfo pInfo, JourneyInfo jInfo) {
        Ticket ticket = new Ticket();
        ticket.setPnr("PNR2400");
        ticket.setStatus("CONFIRMED");

        return ticket;
    }
}
```

Note:- SEI Interface is optional

In case if you write an SEI Interface, in implementation class we need to mark it with @WebService(endPointInterface="qualified name of Interface").

In the absence of SEI interface, all the annotations like @WebMethod, @WebParam and @WebResult should be marked at the class level itself.

Generate Binding classes

To generate Binding classes we need to run the tool wsgen. We don't need to write any config.xml rather it takes the Implementation class as input, because already the classes are annotated with configuration information.

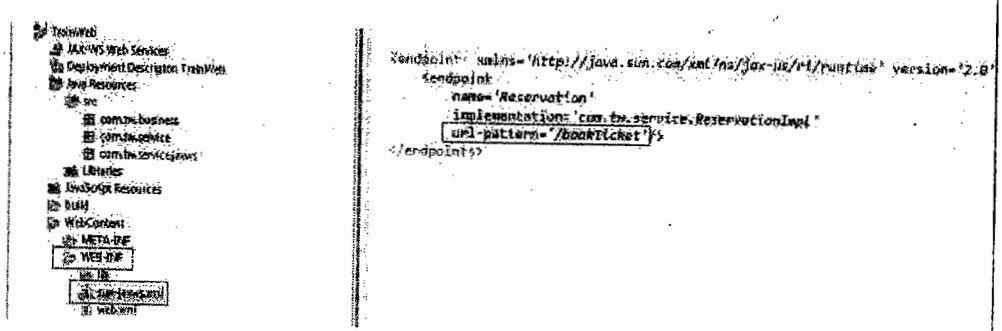
```
>set -path=%path%;c:\sun\jwsdp-2.0\jaxws\bin
```

```
>wsgen -d src -cp build\classes -keep -verbose com.tw.service.ReservationImpl
Note:   ap round: 1
[ProcessedMethods Class: com.tw.service.ReservationImpl]
[should process method: doBooking hasWebMethods: true ]
[endpointReferencesInterface: false]
[declaring class has WebSevice:-true]
[returning: true]
[WrapperGen - method: doBooking(com.tw.business.PassengerInfo,com.tw.business.JourneyInfo)]
[method.getDeclaringType(): com.tw.service.ReservationImpl]
[requestWrapper: com.tw.service.jaxws.DoBooking]
[ProcessedMethods Class: java.lang.Object]
com\tw\service\jaxws\DoBooking.java
com\tw\service\jaxws\DoBookingResponse.java
Note:   ap round: 2
```

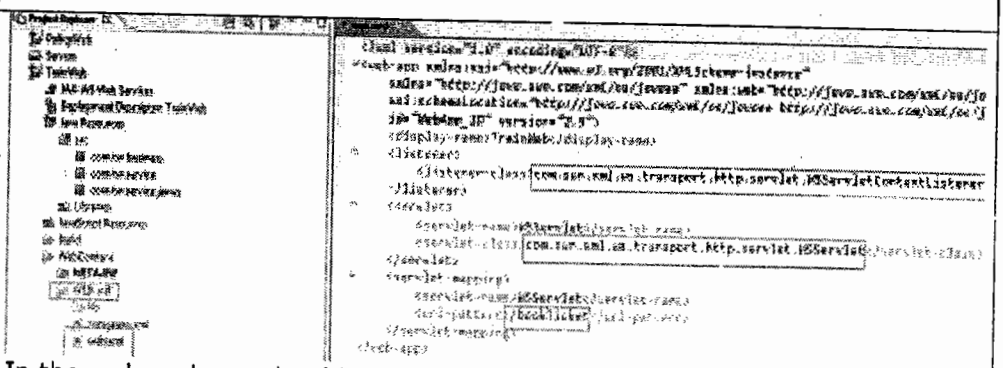
The above tool will generates the JAXB binding classes

Write sun-jaxws.xml

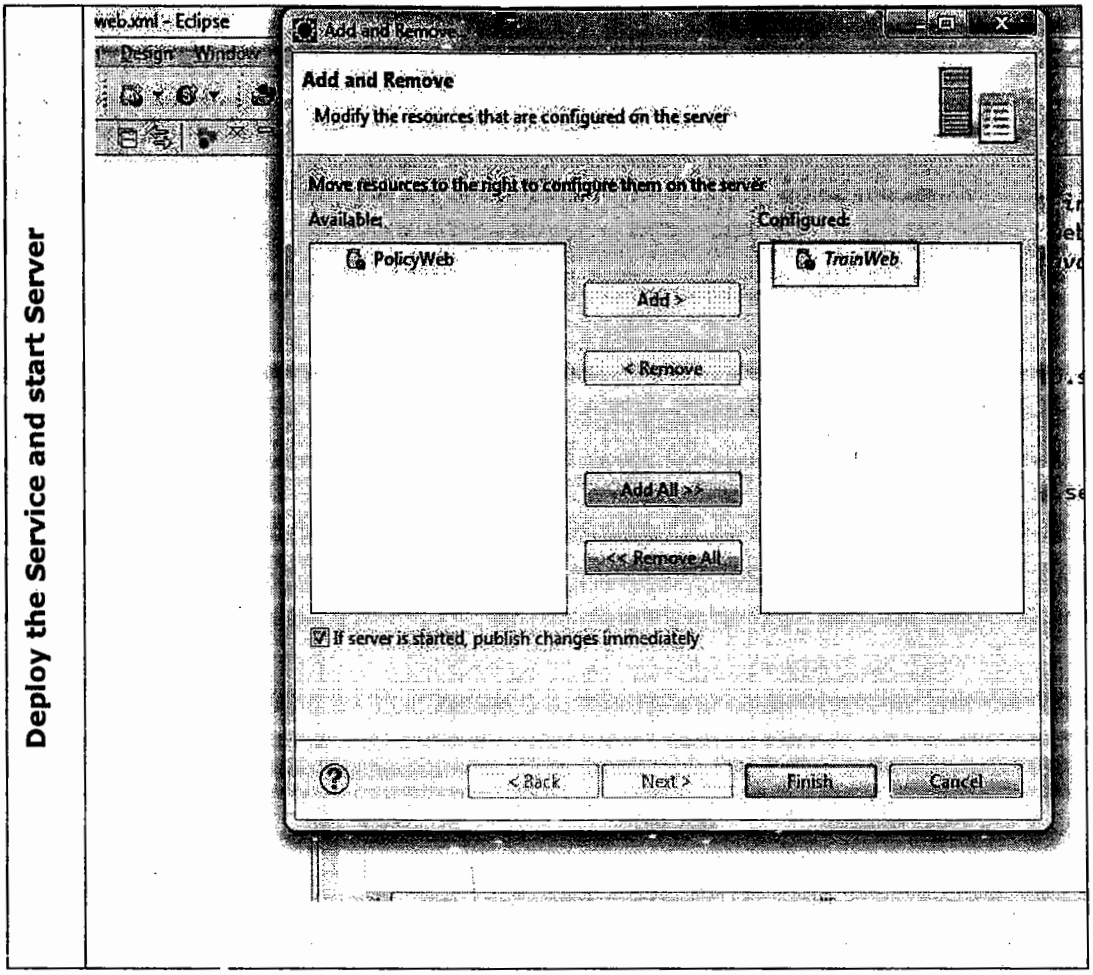
In this file we will provide the endpoint configuration, means attaches URL to the Service.



Write web.xml



In the web.xml, we should configure the WSServlet and WSServletContextListener.



With the above we should be able to access the Dashboard and WSDL of the Service as well. You can test it using SOAP UI.

14.4 Contract First (JAX-WS API (RI), Servlet Endpoint, Sync req/reply)

Contract First based development is fairly simple, as it is contract first, the development will starts with WSDL document. WSDL describes the entire information about the service including the Input/output complex types, PortType, Binding etc.

Using this WSDL document we can generate the classes that are required to create the Service.

a) **Write WSDL document –** WSDL defines the service by declaring the operations, input and output types, binding protocol and transport protocol as well.

Note: - If you are developing a JAX-WS based service, the default encoding format will be document/literal. If it is document literal, you XML elements itself will become the input of the web service method and output of the web service method. So in the types section rather than declaring complex types, you need to declare elements. Messages refer to elements rather than types as shown below.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://icicibank.org/service/wsdl"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="bank"
    xmlns:bt="http://icicibank.org/service/types"
targetNamespace="http://icicibank.org/service/wsdl">
    <wsdl:types>
        <xsd:schema
targetNamespace="http://icicibank.org/service/types">
            <xsd:element name="WithdrawlInfo">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="accNo"
type="xsd:string" />
                        <xsd:element name="type"
type="xsd:string" />
                        <xsd:element name="amount"
type="xsd:float" />
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
```

```
                <xsd:element name="TransactionDetails">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="balance"
type="xsd:float" />
                            <xsd:element name="transactionCode"
type="xsd:string" />
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
            </xsd:schema>
        </wsdl:types>
        <wsdl:message name="withdraw">
            <wsdl:part element="bt:WithdrawlInfo" name="withdrawInfo" />
        </wsdl:message>
        <wsdl:message name="withdrawResponse">
            <wsdl:part element="bt:TransactionDetails"
name="transactionDetails" />
        </wsdl:message>
        <wsdl:portType name="Bank">
            <wsdl:operation name="withdraw">
                <wsdl:input message="tns:withdraw" />
                <wsdl:output message="tns;withdrawResponse" />
            </wsdl:operation>
        </wsdl:portType>
        <wsdl:binding name="BankBinding" type="tns:Bank">
            <soap:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/http" />
            <wsdl:operation name="withdraw">
                <soap:operation
soapAction="http://icicibank.org/service/wsdl/withdraw" />
                <wsdl:input>
                    <soap:body use="literal" />
                </wsdl:input>
                <wsdl:output>
                    <soap:body use="literal" />
                </wsdl:output>
            </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="BankService">
            <wsdl:port binding="tns:BankBinding" name="BankPort">
                <soap:address location="http://www.example.org/" />
            </wsdl:port>
        </wsdl:service>
    </wsdl:definitions>
```

b) Generate Java and Binding classes from WSDL – We need to give WSDL as input and should generate required Java classes and binding classes as output. As we know JWSDP has provided two tools wsgen and wsimport, we need to use wsimport to generate java classes from WSDL.

It takes directly the WSDL as input and generates the classes supporting to build both Provider as well as Consumer. As we need to build provider, we can use them to build provider.

```
wsimport -d src -keep -verbose WebContent\WEB-INF\bank.wsdl
```

It generates SEI interface, Binding classes, Service class as output.

c) Write Implementation class for SEI interface – Follow all the rules and write the Implementation class declaring the endpointInterface as the generated interface as shown below.

```
package org.icicibank.service.wsdl;

import javax.jws.WebService;

import org.icicibank.service.types.TransactionDetails;
import org.icicibank.service.types.WithdrawlInfo;

@WebService(endpointInterface = "org.icicibank.service.wsdl.Bank")
public class BankImpl {

    public TransactionDetails withdraw(WithdrawlInfo withdrawInfo) {
        TransactionDetails td = new TransactionDetails();
        td.setBalance(352.3f);
        td.setTransactionCode("T3424");
        return td;
    }

}
```

d) Write sun-jaxws.xml and web.xml – Configure the endpoint configuration in sun-jaxws.xml and write WSServlet and WSServletListener in web.xml. The Servlet url pattern should be same as url-pattern of the endpoint specified in sun-jaxws.xml

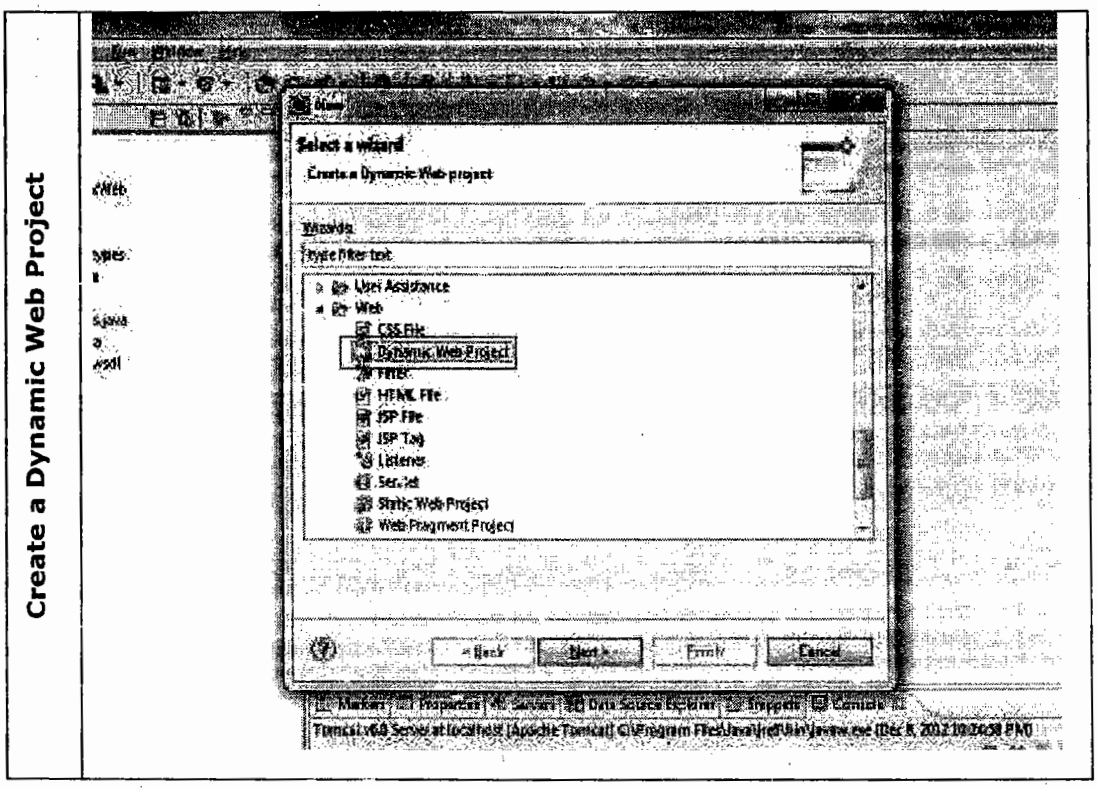
```
<?xml version="1.0" encoding="UTF-8"?>

<endpoints xmlns='http://java.sun.com/xml/ns/jax-ws/ri/runtime'
version='2.0'>
   <endpoint
      name='Bank'
      implementation='org.icicibank.service.wsdl.BankImpl'
      url-pattern='/withdraw'/>
</endpoints>
```

web.xml

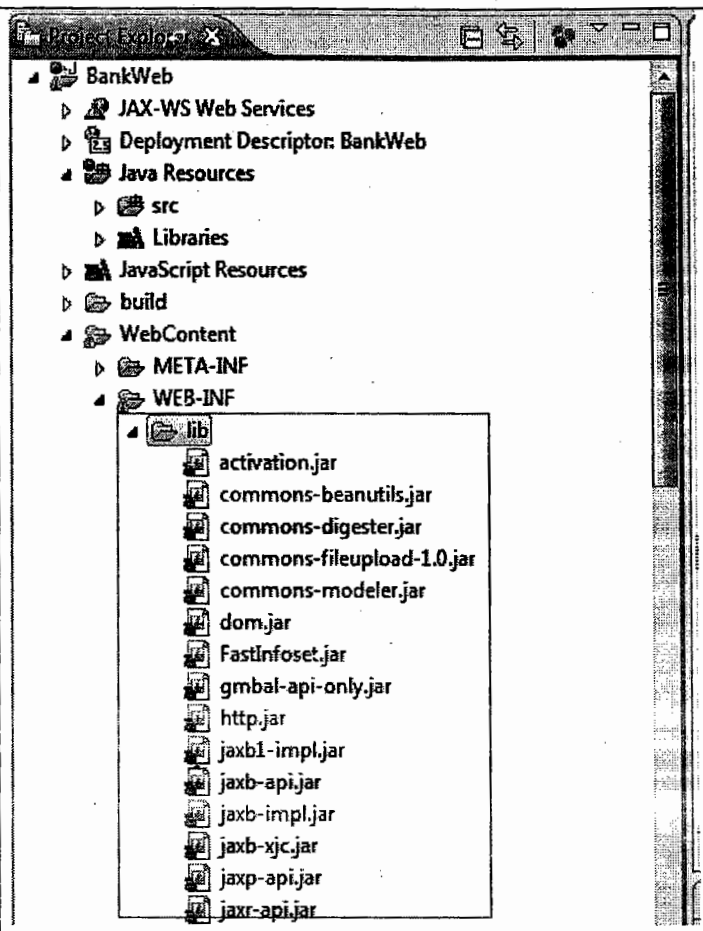
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="WebApp_ID" version="2.5">
    <display-name>TrainWeb</display-name>
    <listener>
        <listener-
class>com.sun.xml.ws.transport.http.servlet.WSServletContextListener</listen
er-class>
    </listener>
    <servlet>
        <servlet-name>WSServlet</servlet-name>
        <servlet-
class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>WSServlet</servlet-name>
        <url-pattern>/withdraw</url-pattern>
    </servlet-mapping>
</web-app>
```

14.5 Contract First– Activity Guide

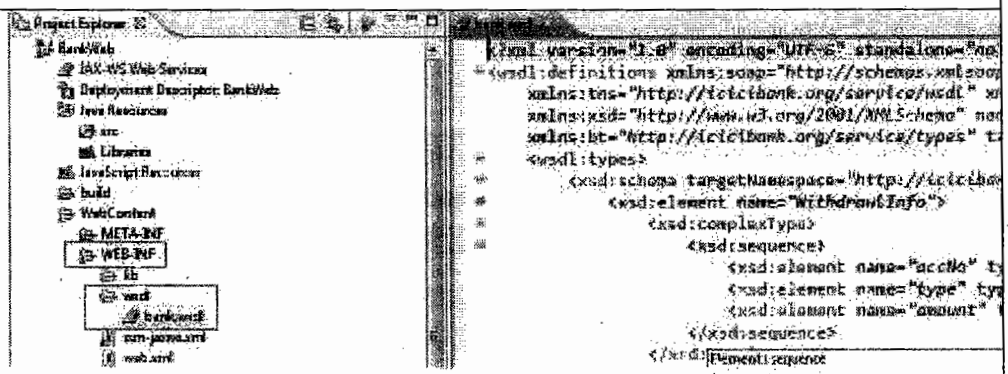
Following sections shows you the step by step approach of create a JAX-WS API RI based web service using contract first approach.



Copy all the Jars to lib



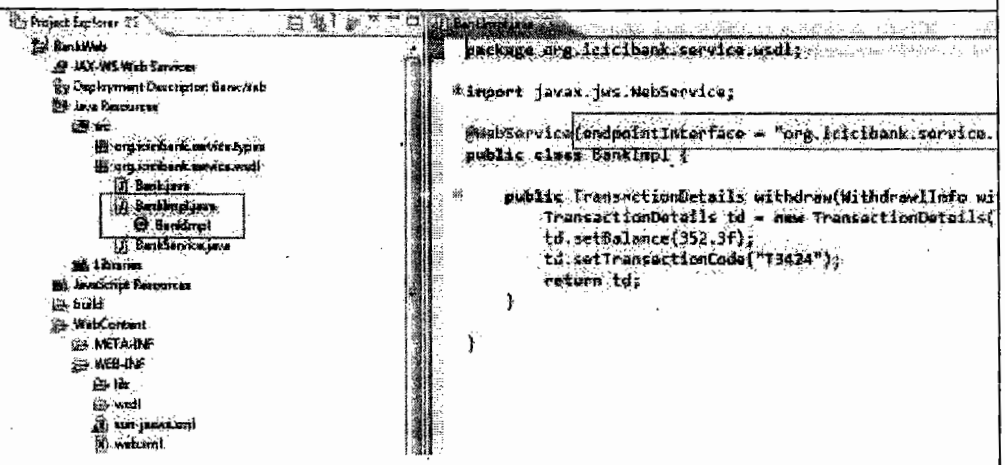
Write the WSDL in the WEB-INF/wsdl

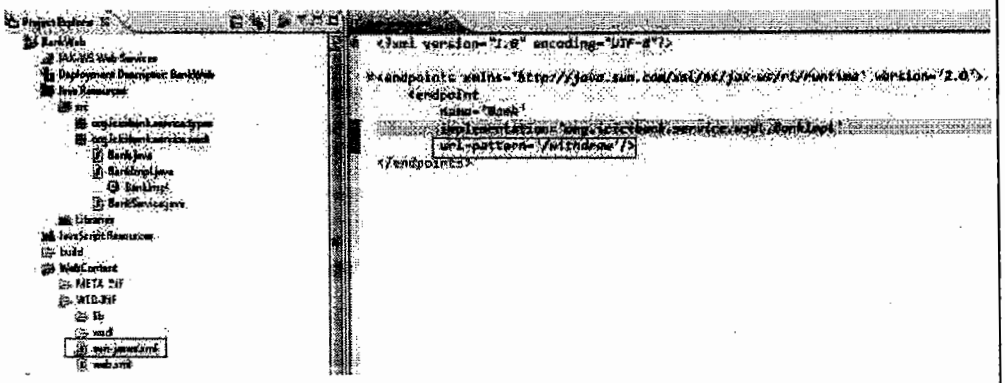
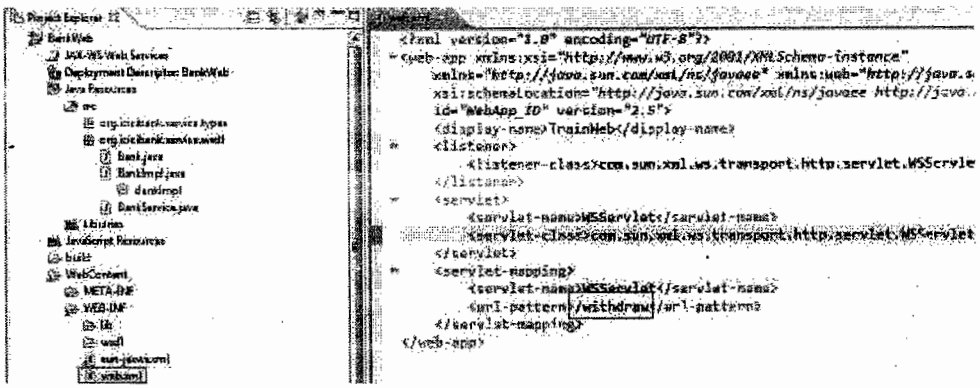


Your input and output's must be elements not types as document-literal means takes the XSD elements as inputs and returns them directly as output.

Run wsimport tool

Write Implementation class



Write sun-jaxws.xml	Write endpoint configuration in sun-jaxws.xml. It should be placed under WEB-INF. 
Write web.xml	The url of the WSServlet should match with the url pattern of the endpoint configured in sun-jaxws.xml 

Deploy your project, start the server and access the Dashboard and browse the WSDL. Use soap ui tool to test it.

14.6 Building Consumer

After developing the provider, in order to access it, we need to build the Consumer. JAX-WS API supports both the developments of Provider as well as Consumer.

Unlike JAX-RPC API, JAX-WS API supports two types of consumer

a) Stub based

b) Dispatch API

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

a) **Run wsimport tool by giving wsdl as input –** In order for a consumer to know the information about the provider, we need the WSDL. WSDL describes the entire information about the provider including the input/output, bindings and address location of the provider.

Using the WSDL, we can generate the classes supporting for development of consumer. So, run the wsimport tool by giving WSDL as input, it will generate classes for building both consumer and provider.

As we are building consumer, we need to consider the classes for Consumer.

SEI Interface	Contains the operations of the Provider
Binding classes	JAXB binding classes used for marshaling/un-marshaling
Input/Output classes	Represents the parameters and returntypes of the service methods
Service	Factory who knows how to create the Stub to communicate with the Provider

```
wsimport -d src -keep -verbose
http://locahost:8081/BankWeb/withdraw?wsdl
```

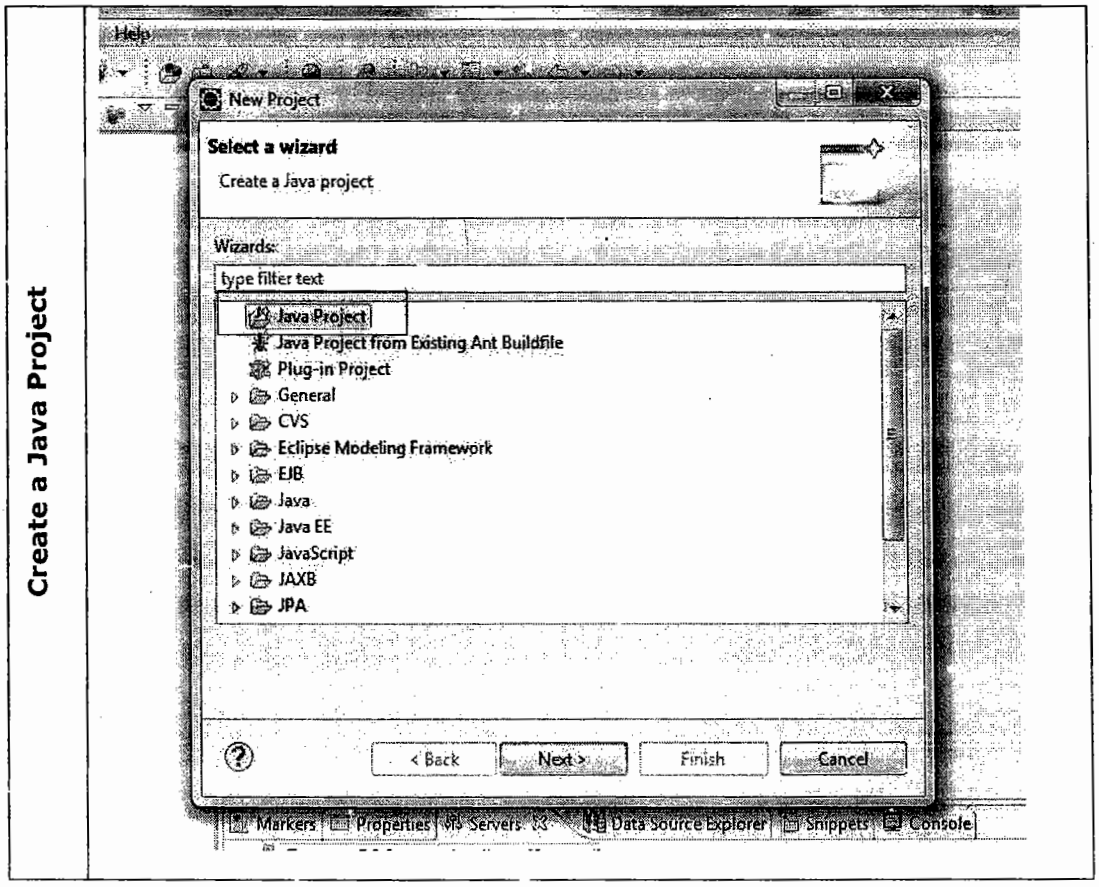
b) **Create the consumer class, which sends the request to provider with inputs –** Once you generate all the classes, we need to build a Consumer class which will creates the Object of Service and call getPort() method to get the Stub object.

Once the Stub object is there we can call methods on the Stub object to send the request to the provider.

```
package com.bank.client;

public class BankClient {
    public static void main(String args[]) {
        BankService service = new BankService();
        Bank port = service.getBankPort();
        WithdrawlInfo withdrawInfo = new WithdrawlInfo();
        withdrawInfo.setAccNo("AC2542");
        withdrawInfo.setAmount(35.342f);
        withdrawInfo.setType("SAVINGS");

        TransactionDetails td = port.withdraw(withdrawInfo);
        System.out.println("Balance : " + td.getBalance());
    }
}
```

14.7 Stub based consumer – Activity Guide

Following steps shows the detailed steps in create a Stub based consumer.



Run wsimport

Run wsimport by giving dynamic wsdl url as input. It will generates classes required for both Provider as well as client.

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

D:>wsimport -d src -keep -verbose http://localhost:8081/BankWeb/withc
parsing WSDL...


generating code...

org\icicibank\service\wsdl\Bank.java
org\icicibank\service\wsdl\BankService.java
org\icicibank\service\types\ObjectFactory.java
org\icicibank\service\types\TransactionDetails.java
org\icicibank\service\types\WithdrawlInfo.java
org\icicibank\service\types\package-info.java

compiling code...

javac -d D:\WorkShop\Xperiment\eclipse\Webservices\Notes\Bank
eriment\eclipse\Webservices\Notes\BankWeb\src\org\icicibank\s
kShop\Xperiment\eclipse\Webservices\Notes\BankWeb\src\org\ici
\package-info.java
```

Create a Test class

Test class is nothing but the class in which you want to call the service.

Create generated Service class Object and then call getPort() method on it which will generates the Stub object. As you know Stub is the implementation of SEI interface, you can assign it to SEI interface reference variable. Now call the methods, indeed calls the methods on generated Stub class.

```
package com.bank.client;

import org.icicibank.service.types.TransactionDetails;

public class BankClient {
    public static void main(String args[]) {
        BankService service = new BankService();
        Bank port = service.getBankPort();

        WithdrawlInfo withdrawInfo = new WithdrawlInfo();
        withdrawInfo.setAccNo("AC2542");
        WithdrawlInfo withdrawInfo - com.bank.client.BankClient.main(String[])
                                                    Press 'F2' for focus
        System.out.println("Balance : " + td.getBalance());
    }
}
```

14.8 Dispatch API Client

As we know using JAX-WS API we can build provider as well as consumer. JAX-WS allows us to develop two types of consumer

a) Dynamic Proxy

b) Dispatch API

We already saw how to work on Dynamic Proxy based consumer, now we are trying to understand how to work with Dispatch API Client.

The dispatch API is a low level API that requires clients to construct messages or message payloads using XML based technology to send request to provider.

The Dispatch supports two usage modes as shown below.

javax.xml.ws.Service.Mode.PAYLOAD – This assumes we are sending plain xml without SOAP

javax.xml.ws.Service.Mode.MESSAGE - Here the message is being transmitted as part of SOAP Message.

For invoking non-soap based services like JAX-WS XML Binding provider or REST we use dispatch API.

The below code provides you an in-sight of how to work with Dispatch API client.

TransactOnlineDispatchClient.java

```
package com.bw.service.dispatch.client;
public class TransactOnlineDispatchClient {
  private static final String TARG_NMSPC = "http://idbi.com/transaction/wsdl";
  private static final String SERVICE_NM = "TransactOnlineService";
  private static final String PORT_NM = "TransactOnlineSoapPort";
  private static final String TYP_NMSPC = "http://idbi.com/transaction/types";
  private static final String TRG_ENDPOINT_URL =
"http://localhost:8080/BankWeb/withdrawl";

  public static void main(String[] args) throws SOAPException, IOException {
        Service transactOnlineService = Service.create(new
QName(TARG_NMSPC,
                    SERVICE_NM));
        transactOnlineService.addPort(new QName(TARG_NMSPC, PORT_NM),
                    SOAPBinding.SOAP11HTTP_BINDING, TRG_ENDPOINT_URL);
        Dispatch<SOAPMessage> dispatch =
transactOnlineService.createDispatch(
                    new QName(TARG_NMSPC, PORT_NM), SOAPMessage.class,
                    Service.Mode.MESSAGE);
        MessageFactory mf = MessageFactory.newInstance();
        SOAPMessage request = mf.createMessage();
        SOAPPart part = request.getSOAPPart();
        SOAPEnvelope envelope = part.getEnvelope();
        SOAPBody body = envelope.getBody();

        SOAPElement operationElem = body.addChildElement("withdraw", "wsdl",
                    TARG_NMSPC);
        SOAPElement aInfoElem = operationElem.addChildElement("accountInfo",
                    "typ", TYP_NMSPC);
        SOAPElement accNoElem = aInfoElem.addChildElement("accNo");
        accNoElem.addTextNode("3532");
        SOAPElement amtElem = aInfoElem.addChildElement("amount");
        amtElem.addTextNode("242");
        // request.writeTo(System.out);

        SOAPMessage response = dispatch.invoke(request);
        response.writeTo(System.out);
  }

}
```

14.9 Difference between various Message Exchanging formats

WSDL stands for Web Service Description language. It describes various elements of a Web Service such as Binding Protocol and message exchanging pattern. SOAP binding talks about whether the message is being transported as "RPC" or "document" style. SOAP binding also specifies encoded use or literal use. With these we have four combinations

- RPC/encoded
- RPC/literal
- Document/encoded
- Document/literal

In addition to the above we have one more pattern commonly used is Document/wrapped. Hence we have total five messaging formats.

We have messaging models or programming models, which talks about how the messages are being exchanged between consumer and provider, those are "RPC" and message oriented programming models. There is no relation between programming models and your exchanging styles. You can use any style with any programming models.

Let us try to understand how a message typically looks like when we use different messaging styles and their advantages and dis-advantages.

Sample SEI Interface

```
public interface MemberRegistration extends Remote {
        MemberCard enroll(MemberInfo mInfo, PolicyInfo pInfo) throws
                                                                RemoteException;
}
```

14.9.1 RPC/Encoded

For the above SEI Interface here is the WSDL PortType and soap message format.

RPC/encoded WSDL

```
<types>
        <xsd:complexType name="MemberInfo">
                ....
        </xsd:complexType>
        <xsd:complexType name="PolicyInfo">
                ....
        </xsd:complexType>
        <xsd:complexType name="MemberCard">
                ....
        </xsd:complexType>
</types>
<message name="MemberRegistration_enroll">
        <part name="mInfo" type="typ:MemberInfo"/>
        <part name="pInfo" type="typ:PolicyInfo"/>
</message>
<message name="MemberRegistration_enrollResponse">
        <part name="result" type="typ:MemberCard"/>
</message>
<portType name="MemberRegistration">
        <operation name="enroll">
                <input message="tns:MemberRegistration_enroll"/>
                <output message="tns:MemberRegistation_enrollResponse"/>
        </operation>
</portType>
```

RPC/encoded SOAP Message

```
<soap:envelope>
        <soap:header/>
        <soap:body>
                <enroll>
                        <mInfo>
                                <ssn xsi:type="xsd:string>SSN22</ssn>
                                <age xsi:type="xsd:int">43</age>
                        </mInfo>
                        <pInfo>
                                ...
                        </pInfo>
                </enroll>
        </soap:body>
</soap:envelope>
```

14.9.1.1 Advantage

a) The WSDL is self-explanatory

b) The operation name appears in the message, so that the receiver has easy time in dispatching this message to the implementation of the operation.

14.9.1.2 Dis-Advantage

a) The xsi:type encoding specified at the SOAP message is as extra overhead which degrades the throughput performance

b) You cannot easily validate this message since the <ssn> and <age> are the elements coming from <schema> and rest of the elements are derived from WSDL definition.

c) Although it is legal WSDL, it is not WS-I complaint.

14.9.2 RPC/Literal

The RPC/Literal WSDL looks almost same as RPC/Encoded WSDL except the binding section will declares the style as RPC and use as literal.

SOAP Message for RPC/Literal

```
<soap:envelope>
      <soap:header/>
      <soap:body>
            <enroll>
                  <mInfo>
                        <ssn>SSN22</ssn>
                        <age>43</age>
                  </mInfo>
                  <pInfo>
                        ...
                  </pInfo>
            </enroll>
      </soap:body>
</soap:envelope>
```

14.9.2.1 Advantage

a) The WSDL is still straight forward and self-explanatory

b) The operation name still appears in the message

c) The type info encoding is eliminated

d) RPC/Literal is WS-I complaint

14.9.2.2 Dis-Advantage

a) You still cannot easily validate this message, since it has only <ssn> and <age> coming from <schema>. Rest of the elements are derived from WSDL.

Is the document styles overcomes the dis-advantage with RPC style. Let us try to understand.

14.9.3 Document/Encoded

Nobody follows this style. It is not WS-I complaint

14.9.4 Document/Literal

The WSDL for Document/Literal would be more or less looks similar to RPC, but there are some small differences like under <types> section we need to define elements rather than complexTypes. Messages takes the part's as elements not types as shown below.

```
<types>
        <xsd:element name="MemberInfo">
                <xsd:complexType>
                        ....
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="PolicyInfo">
                <xsd:complexType>
                        ....
                </xsd:complexType>
        </xsd:complexType>
        <xsd:element name="MemberCard">
                <xsd:complexType>
                        ....
                </xsd:complexType>
        </xsd:complexType>
</types>
<message name="MemberRegistration_enroll">
        <part name="mInfo" element="typ:MemberInfo"/>
        <part name="pInfo" element="typ:PolicyInfo"/>
</message>
<message name="MemberRegistration_enrollResponse">
        <part name="result" element="typ:MemberCard"/>
</message>
<portType name="MemberRegistration">
        <operation name="enroll">
                <input message="tns:MemberRegistration_enroll"/>
                <output message="tns:MemberRegistation_enrollResponse"/>
        </operation>
</portType>
```

Document/Literal SOAP Message

```
<soap:envelope>
        <soap:header/>
        <soap:body>
                <memberInfo>
                        <ssn>SSN22</ssn>
                        <age>43</age>
                </memberInfo>
                <policyInfo>
                        ...
                </policyInfo>
        </soap:body>
</soap:envelope>
```

14.9.4.1 Advantage

a) There is no type encoding

b) You can finally validate the message with any XML validator. Everything with in soap:body is defined in the <schema>

c) Document/literal is WS-I complaint, but with restrictions

14.9.4.2 Dis-Advantage

a) The WSDL is getting bit complicated. This is a minor weakness.

b) The operation name in the SOAP message is lost. Without the name, dispatching can be difficult and sometimes impossible.

c) WS-I only allows one child of the soap:body in the SOAP message.

14.9.5 Document/Wrapped

The document literal seems to be merely fix all the dis-advantages of rpc/literal model. We can now validate the soap message. But we lost the operation name. Is there anything you can do to improve upon this? Yes, it is called Document/Wrapped.

Document/Wrapped WSDL document

```
<types>
      <xsd:element name="enroll>
            <xsd:complexType>
                  <xsd:element name="mInfo" type="typ:MemberInfo"/>
                  <xsd:element name="pInfo" type="typ:PolicyInfo"/>
            </xsd:complexType>
      <xsd:element>
      <xsd:element name="enrollResponse>
            <xsd:complexType>
                  <xsd:element name="result" type="typ:MemberCard"/>
            </xsd:complexType>
      <xsd:element>
      <xsd:complexType name="MemberInfo">
            ....
      </xsd:complexType>
      <xsd:complexType name="PolicyInfo">
            ....
      </xsd:complexType>
      <xsd:complexType name="MemberCard">
            ....
      </xsd:complexType>
</types>
<message name="enroll">
      <part name="enroll" element="typ: enroll"/>
</message>
<message name="enrollResponse">
      <part name="result" element="typ: enrollResponse"/>
</message>
<portType name="MemberRegistration">
      <operation name="enroll">
            <input message="tns:enroll"/>
            <output message="tns:enrollResponse"/>
      </operation>
</portType>
```

Document/Wrapped SOAP Message

```
<soap:envelope>
      <soap:header/>
      <soap:body>
            <enroll>
                  <memberInfo>
                        <ssn>SSN22</ssn>
                        <age>43</age>
                  </memberInfo>
                  <policyInfo>
                        ...
                  </policyInfo>
            </enroll>
      </soap:body>
</soap:envelope>
```

There are few notable characteristics of the document/wrapped pattern:

a) The input message has a single part
b) The part is an element
c) The element has the same name as operation name
d) The elements complex type has no attributes.

Here are the advantages and dis-advantages of document/wrapped.

14.9.5.1 Advantage

a) There is no type encoding
b) Everything that defines under the soap:body is defined by the schema. You can validate the soap message with any of the XML validator
c) Once again you have method name in the soap message
d) Document/literal is WS-I complaint, wrapped pattern meets the WS-I restrictions where soap:body should contain only one element as child element.

14.9.5.2 Dis-Advantage

a) The WSDL became even more complicated.

Note: - If you want to build document/literal message format service, on the SEI interface or at the implementation class, you need to mark with an annotation @SOAPBinding (parameterStyle=SOAPBinding.ParameterStyle.BARE).

If you want to build document/wrapped service you need to use the ParameterStyle.WRAPPED in the above annotation.

SOAP

15 SOAP

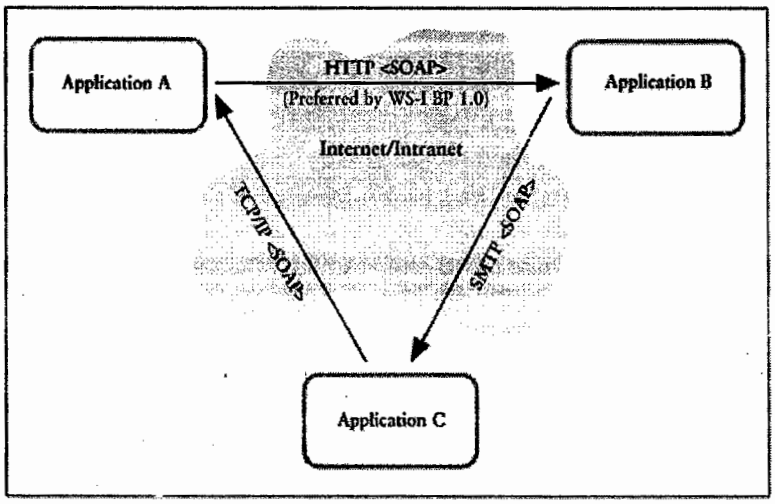
15.1 Introduction

SOAP 1.2 is the standard messaging protocol used by J2EE Web Services, and is the de facto standard for Web services in general.

SOAP is just another XML markup language accompanied by rules that dictate its use. SOAP has a clear purpose: exchanging data over networks. SOAP is a network application protocol

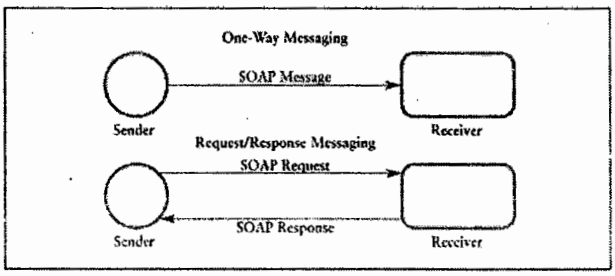
A SOAP XML document instance, which is called a SOAP message, is usually carried as the payload of some other network protocol.

The SOAP XML document is also called the SOAP envelope.



A lightweight protocol for exchange of information in a decentralised, distributed environment.

Web services can use One-Way messaging or Request/Response messaging.



SOAP defines how messages can be structured and processed by software in a way that is independent of any programming language or platform, and thus

facilitates interoperability between applications written in different programming languages and running on different operating systems.

15.2 The SOAP Message

SOAP message is a kind of XML document. SOAP has its own XML schema, namespaces, and processing rules.

A SOAP message is encoded as an XML document, consisting of an <Envelope> element, which contains an optional <Header> element, and a mandatory <Body> element. The <Fault> element, contained within the <Body>, is used for reporting errors.

The Header element contains information about the message, in the form of one or more distinct XML elements, each of which describes some aspect or quality of service associated with the message.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Header>
      <!-- Header blocks go here -->
   </soap:Header>
   <soap:Body>
      <!-- Application data goes here -->
   </soap:Body>
</soap:Envelope>
```

15.3 The SOAP Namespaces

XML namespaces play an important role in SOAP messages. A SOAP message may include several different XML elements in the Header and Body elements, and to avoid name collisions each of these elements should be identified by a unique namespace.

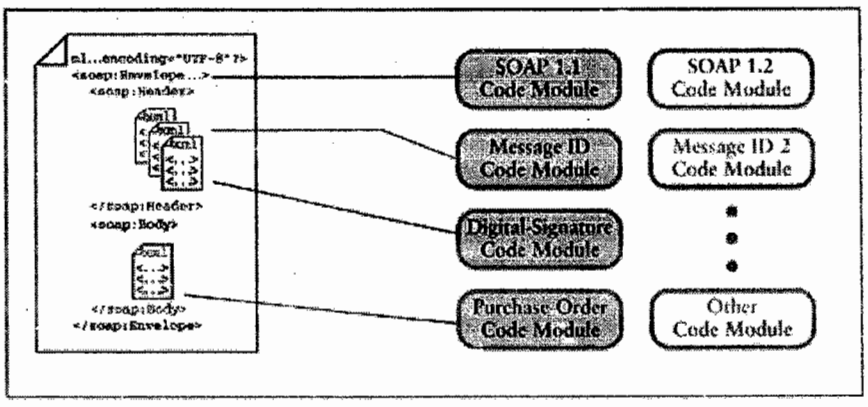
SOAP message that contains the purchaseOrder element as well as message-id and XML digital-signature header blocks would include no fewer than six different namespaces

```
<?xml version="1.0" encoding="UTF-8"?><soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:sec="http://schemas.xmlsoap.org/soap/security/2000-12"
xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id">
<soap:Header>
    <mi:message-id>11d1def534ea:b1c5fa:f3bf8000</mi:message-id>
    <sec:Signature>     ...     </sec:Signature>
</soap:Header>
<soap:Body sec:id="Body">
  <po:purchaseOrder orderDate="2003-09-22"
          xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <!—business xml →
  </po:purchaseOrder>
</soap:Body>
</soap:Envelope>
```

15.4 Code Modules with SOAP Namespaces

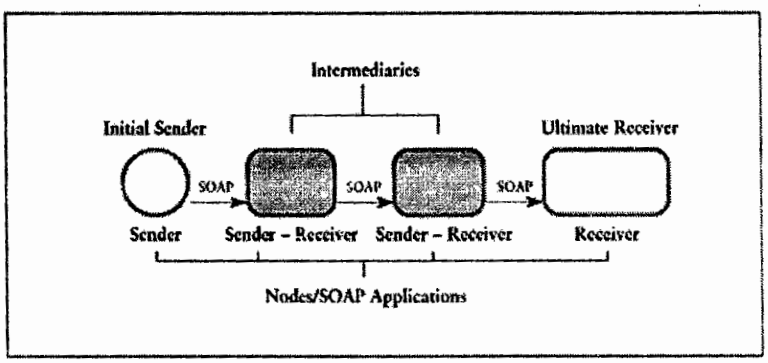
The modularity of SOAP messaging permits the code that processes the SOAP messages to be modular as well. The code that processes the element Envelope is independent of the code that processes the header blocks, which is independent of the code that processes application-specific data in the SOAP Body element.

Modularity enables you to use different code libraries to process different parts of a SOAP message.



15.5 The SOAP Message Path

A SOAP message travels along the message path from a sender to a receiver. All SOAP messages start with the initial sender, which creates the SOAP message, and end with the ultimate receiver. The term client is sometimes associated with the initial sender of a request message, and the term Web service with the ultimate receiver of a request message.




```
<?xml version="1.0" encoding="UTF-8"?><soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id"
xmlns:proc="http://www.Monson-Haefel.com/jwsbook/processed-by">
aaxzzz<soap:Header>
   <mi:message-id>11d1def534ea:cd7:-8000</mi:message-id>
   <proc:processed-by>
   <node>
     <time-in-millis>1013694680000</time-in-millis>
     <identity>http://www.customer.com</identity>
   </node>
   <node>
     <time-in-millis>1013694680010</time-in-millis>
     <identity>http://www.Monson-Haefel.com/sales</identity>
    </node>
    </proc:processed-by>
</soap:Header>
<soap:Body>
      <!-- Application-specific data goes here -->
</soap:Body>
</soap:Envelope>
```

15.6 The SOAP Body

Although the Header element is optional, all SOAP messages must contain exactly one Body element.BP The Body element contains either the application-specific data or a fault message. Application-specific data is the information that we want to exchange with a Web service.

SOAP Messaging Modes

Except in the case of fault messages, SOAP does not specify the contents of the Body element (although it does specify the general structure of RPC-type messages). As long as the Body contains well-formed XML, the application-specific data can be anything. The Body element may contain any XML element or it can be empty.

Although SOAP supports four modes of messaging (RPC/Literal, Document/ Literal, RPC/Encoded, and Document/Encoded) the BP permits the use of RPC/ Literal or Document/Literal only

15.7 SOAP Faults

SOAP fault messages are the mechanism by which SOAP applications report errors "upstream," to nodes earlier in the message path. It's the mission of this section to provide a full and detailed explanation of SOAP fauits so that you can handle them appropriately in your own Web services.

SOAP faults are generated by receivers, either an intermediary or the ultimate receiver of a message. The receiver is required to send a SOAP fault back to the sender only if the Request/Response messaging mode is used. In One-Way mode, the receiver should generate a fault and may store it somewhere, but it must not attempt to transmit it to the sender.

A SOAP message that contains a Fault element in the Body is called a fault message.

When a fault message is generated, the Body of the SOAP message must contain only a single Fault element and nothing else. The Fault element itself must contain a faultcode element and a faultstring element, and optionally faultactor and detail elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote" >
    <soap:Body>
        <soap:Fault>
          <faultcode>soap:Client</faultcode>
          <faultstring>the ISBN value contains invalid characters
          </faultstring>
          <faultactor>http://www.xyzcorp.com</faultactor>
          <detail>
             <mh:InvalidIsbnFaultDetail>
                 <offending-value>19318224-D</offending-value>
                 <conformance-rules>          The first nine characters must be
digits. The last          character may be a digit or the letter 'X'. Case is
not important.
                 </conformance-rules>
             </mh:InvalidIsbnFaultDetail>
          </detail>
        </soap:Fault>
    </soap:Body>
</soap:Envelope>
```

15.7.1 Faultcode Element

The faultcode element may use any of four standard SOAP fault codes to identify an error.

- SOAP Standard
- Fault Codes
- ClientServerVersionMismatch
- MustUnderstand

Although you're allowed to use arbitrary fault codes, you should use only the four standard codes listed.BP

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
    <soap:Body>
      <soap:Fault>
        <faultcode>soap:Client</faultcode>
        <faultstring>The ISBN contains invalid characters</faultstring>
        <detail/>
     </soap:Fault>
    </soap:Body>
</soap:Envelope>
```

15.7.2 Detailed Element

The detail element of a fault message must be included if the fault was caused by the contents of the Body element, but it must not be included if the error occurred while processing a header block

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote" >
   <soap:Body>
      <soap:Fault>
         <faultcode>soap:Client</faultcode>
         <faultstring>      The ISBN value contains invalid characters
</faultstring>
         <detail>
            <mh:InvalidIsbnFaultDetail>
               <offending-value>19318224-D</offending-value>
               <conformance-rules>The first nine characters must be digits.
The last      character may be a digit or the letter 'X'. Case is      not important.
</conformance-rules>
            </mh:InvalidIsbnFaultDetail>
         </detail>
      </soap:Fault>
   </soap:Body>
</soap:Envelope>
```

15.8 SOAP over HTTP

The vast majority of all Internet traffic today is data transferred using HTTP (HyperText Transfer Protocol).

SOAP messages sent over HTTP are placed in the payload of an HTTP request or response, an area that is normally occupied by form data and HTML

Transmitting SOAP with HTTP POST Messages

```
POST /jwsbook/BookQuote HTTP/1.1
Host: www.Monson-Haefel.com
Content-Type: text/xml; charset="utf-8"
Content-Length: 295
SOAPAction=""
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
  <soap:Body>
    <mh:getBookPrice>
    <isbn>0321146182</isbn>
    </mh:getBookPrice>
  </soap:Body>
</soap:Envelope>
```

Apache Axis 2

16 JAX-WS API (Apache Axis 2 - Implementation)

The Apache Axis 2 is the third generate Web service engine released by Apache Foundation which is more powerful than its predecessor Apache Axis. The latest stable release of Apache Axis 2 1.6.2.

The Apache Axis 2 project is Java-based implementation of both the client and server sides of the Web Service.

Axis2 enables you to easily perform the following tasks:
a) Send SOAP messages
b) Receive and process SOAP Messages
c) Create a Web Service out of a Plain Java class
d) Create Implementation class for both client and server using WSDL
e) Easily retrieve WSDL for a Service
f) Send and receive SOAP with attachments
g) Create or utilize a REST-based web service
h) Create or utilize services that take the advantage of the WS-Security, WS-ReliableMessaging, WS-Addressing, WS-Coordination and WS-AtomicTransaction recommendations.
i) Use Axis's 2 modular structure to easily add support for new recommendations as they emerge

16.1 Understanding Axis Distribution

If you want to develop Axis2 based web services or consume web service based on Apache Axis2, you need to download the Apache Axis2 Binary distribution. It is a zip after extracting contains various Jars' and scripts that facilitates the development of Web Services.

The Binary Distribution directory structure is shown below.

```
Axis2
 | - bin – axis2.bat, axis2Server.bat, java2wsdl.bat, wsdl2java.bat and equivalent
                                                                          .sh files
 | - lib – Contains all Jar's
 | - repository
        | - modules – modules.list, addressing-1.1.mar
        | - services – services.list, version.aar
 | - samples
 | - webapp
 | - config
        | - axis2.xml
```

16.2 Axis2.war Distribution

axis2.war is the war (Web Archive) distribution. It is the J2EE application that is shipped by Apache acts as Server side runtime in which your applications are deployed into it as services and exposed.

Apache axis 2 comes in two distributions.

a) Binary Distribution – Which contains Jar's and tools that facilitate the development of Web Service
b) War Distribution – Acts as server-side runtime engine in which your services are deployed and exposed.

The typical structure of the axis2-web war is as shown below.

```
axis2-web
| - META-INF
| - WEB-INF
        | - classes
        | - conf – axis2.xml
        | - lib – activation.jar, xmlschema.jar ....
        | - modules – modules.lst, addressing.mar, soapmonitor.mar
        | - services – services.lst, aservice.aar, version.aar....
        | - web.xml
```

Axis2.war archive is a collection of Jsp's that make up the Axis2 Administration application. You can perform the operations like adding a services or deploying a module or undeploy a module. If you want to deploy your service in the Axis 2 runtime, you need to archive your server as .aar and should place under services directory. So, that Axis engine will be automatically able to deploy that service.

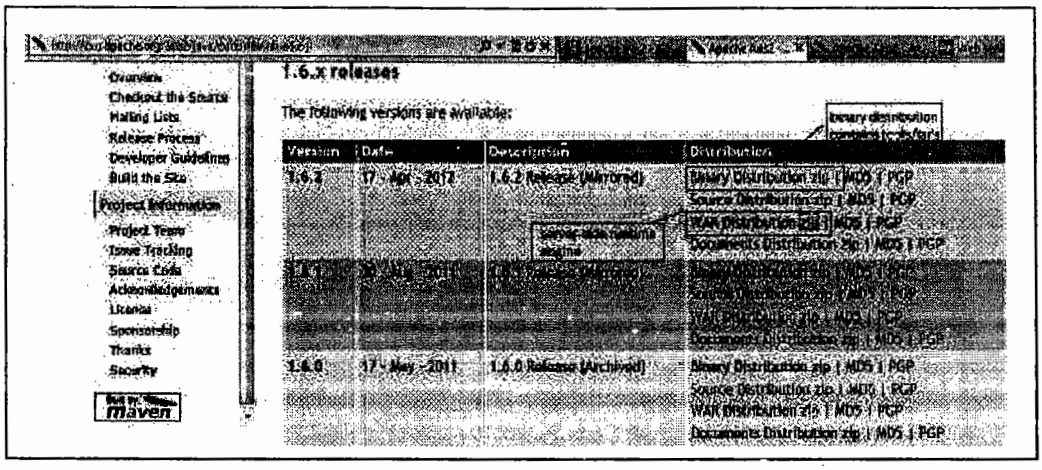
The main file in the above is axis2.xml which contains the entire configuration related to message senders and receiver and their transport receivers etc. It would also contain the information like which modules are active etc.

16.3 Configuring axis 2 environment

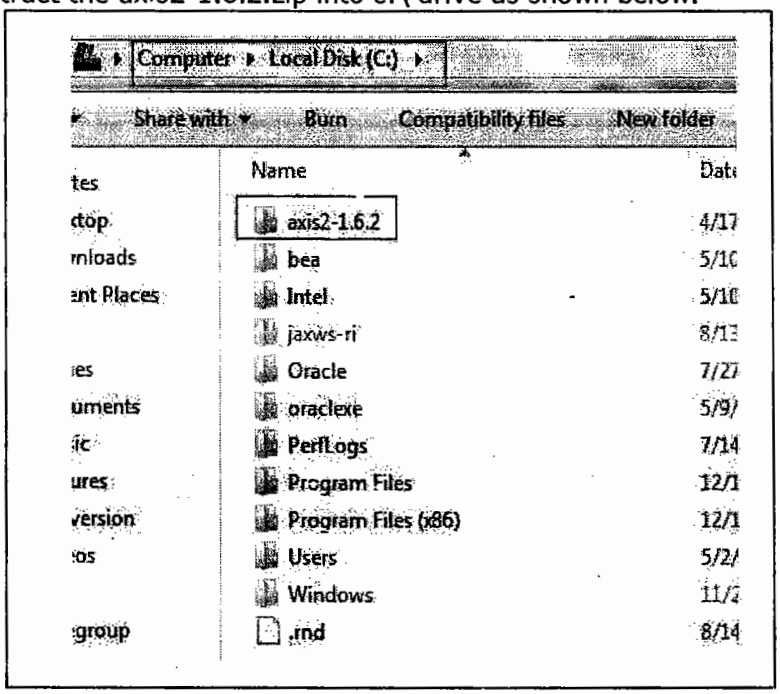
In order to develop the service in Apache Axis2 we need binary distribution. In order to expose them as services, we need server-side runtime engine, so download axis binary and war distributions to build and expose our services in the runtime.

Below are the steps you need to follow in configuring the Apache axis 2.

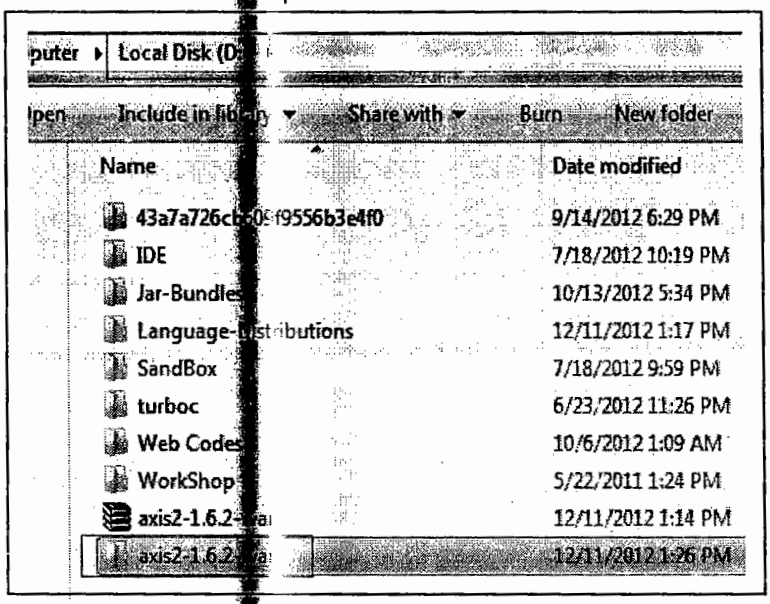
a) Download or copy the axis binary and axis2.war distribution into your local drive – You need to copy the axis2-1.6.1.zip and axis2.war that is being distributed by apache axis2 into your local system - http://apache.techartifact.com/mirror//axis/axis2/java/core/1.6.2/axis2-1.6.2-war.zip



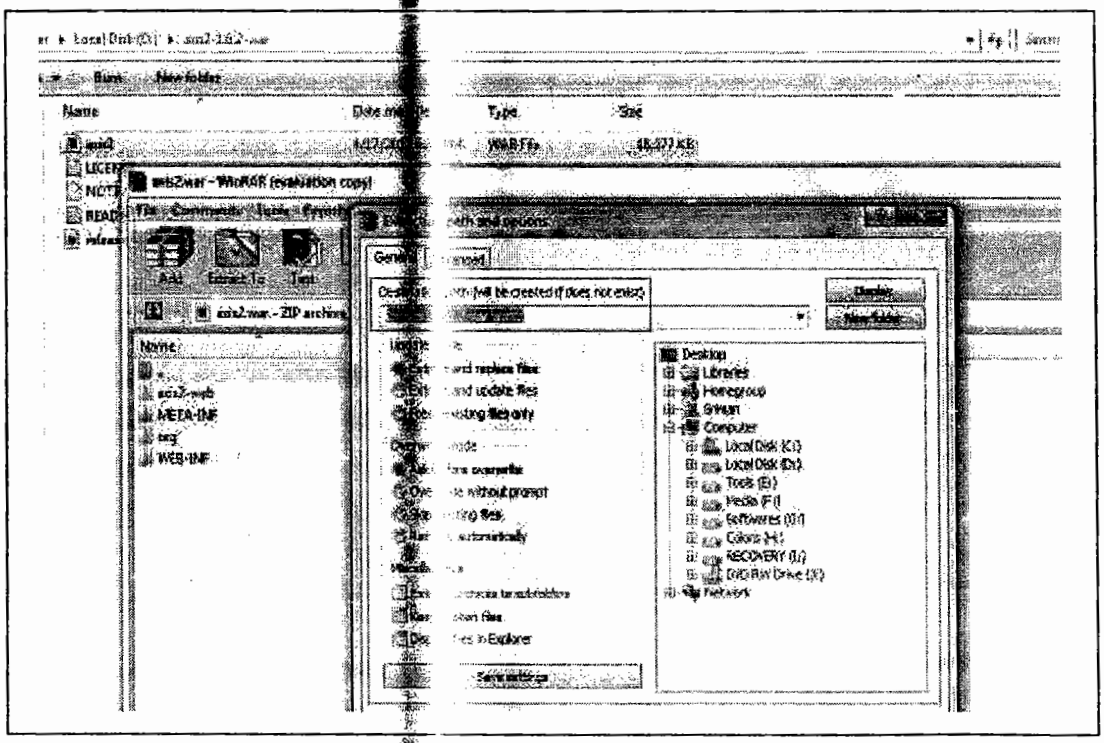
b) Extract the axis2-1.6.2.zip into c:\ drive as shown below.



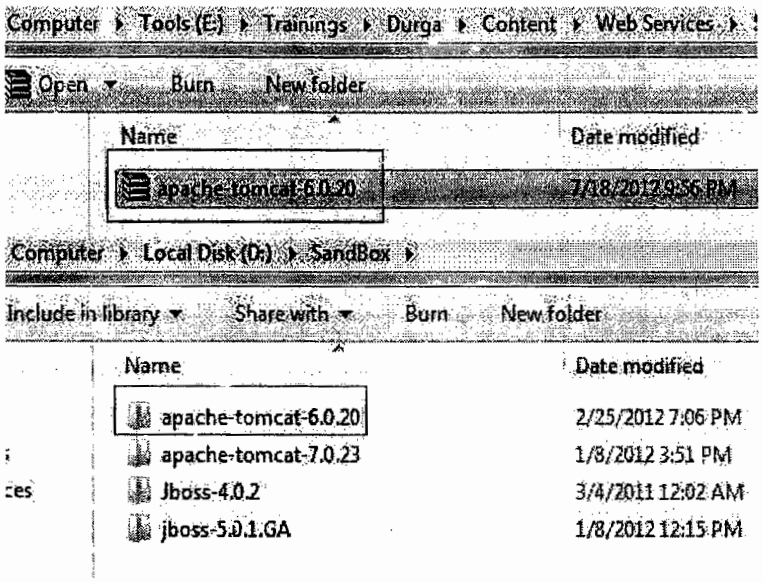
c) Extract axis2-1.6.2-wa .zip into some local drive.



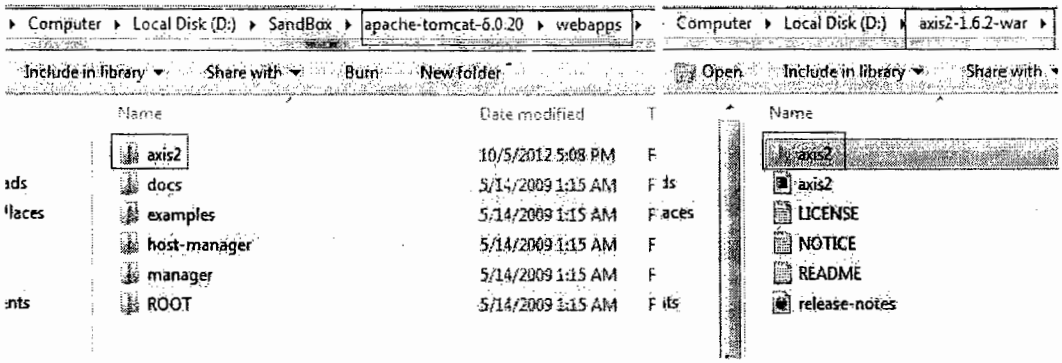
d) Now navigate into the extracted folder shown above, you will find axis2.war, now extract the axis2.war into the axis2 folder as shown below.



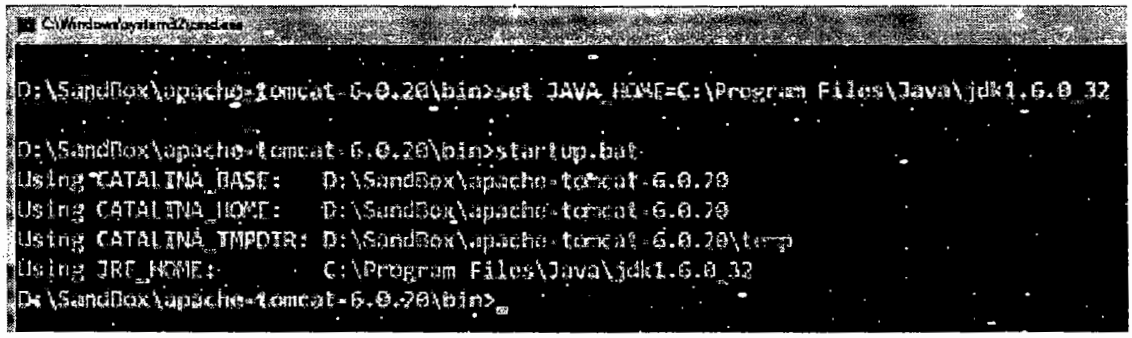
e) As part of the software CD we have given the apache tomcat as a binary distribution. Extract that and paste under some local directory as shown below.



f) Now copy the axis2 folder (which we extracted from axis2-war.war) into the above tomcat webapps directory



g) Now go to the command prompt navigate to the Tomcat directory bin and start the tomcat server by running startup.bat as shown below.

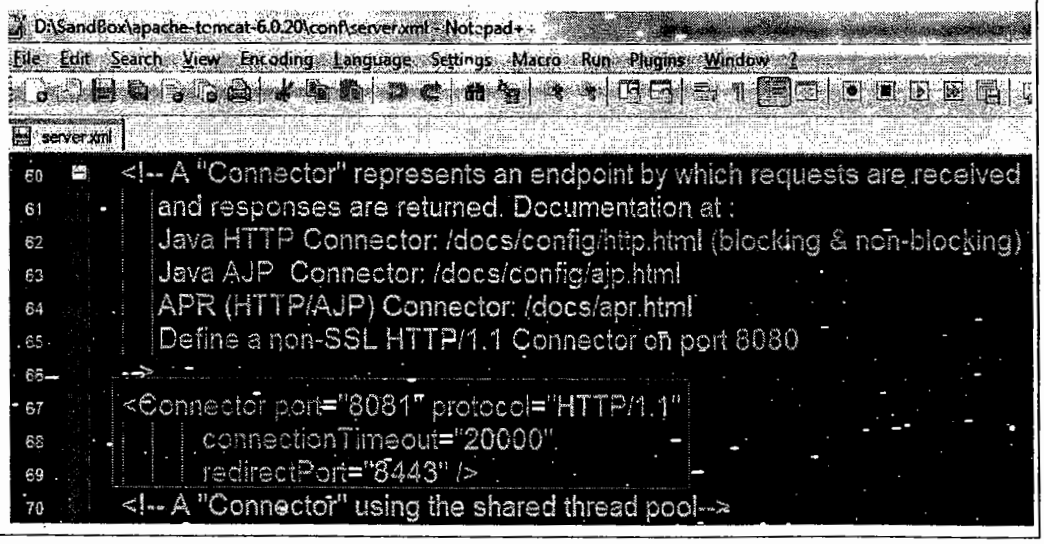


This will start the tomcat server in the new window after typing the startup.bat.

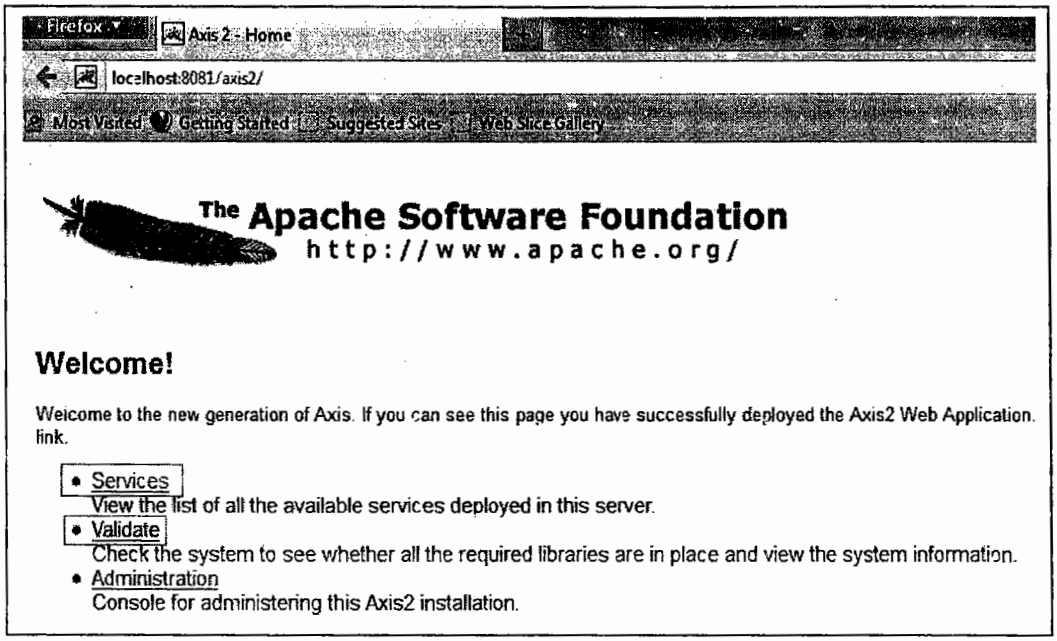
Note: - You need to set the JAVA_HOME environment variable before starting the tomcat server.

Port Issues while starting tomcat server: - In few systems you might have already been using the 8080 port. In such a case if you try to start the tomcat server, it gives you an error 8080 port is already under use. In that case you need to change the port of the tomcat server before starting the server as shown below.

Go to Tomcat Directory/conf and locate the server.xml file. Open the server.xml file and search for <connector> element whose protocol is "HTTP/1.1". If you look at the value there it is 8080, now change it to 8081, save the file and restart the server. Now your tomcat runs under 8081 port.



h) Now open the browser and access the axis home page with http://<adminhost>:<adminport>/axis2 will open the home page of the Apache axis2. You can verify the correctness of your configuration in this page.



This completes the configuration of Apache Axis2 and allows you to proceed for development of web service under this environment.

16.4 Deployment Model

One of the significant improvements from Apache Axis to Apache Axis2 is its deployment model. In case of Apache Axis it is even said as a dynamic deployment model most of the developers felt clumpsy in understanding the deployment process.

Rather Apache axis2 has come up with more consiged and easy deployment model which is called ".aar" apache application archive. Once you finished development of you Java classes, now you need to archive your application in to .aar archive. The typical structure of this archive is as shown below.

```
<service-name>
 | - <package> - containing classes of your service
 | - lib – any jars that your code is dependent on. Axis2 related jars are not
         required as the runtime already contains.
 | - META-INF – it contains wsdl and services.xml
                  (services.xml is web service deployment descriptor)
```

Now you need to copy this archive into server-side runtime folder in tomcat which is under tomcat\webapps\axis2\WEB-INF\services directory.

16.5 Building Provider

Let us understand how to build a Provider in Apache Axis2. As you know there are two ways in building the provider. Now we are trying to understand how to build the provider using contract first approach. The following section walks you through the steps involved in building the provider using contract first.

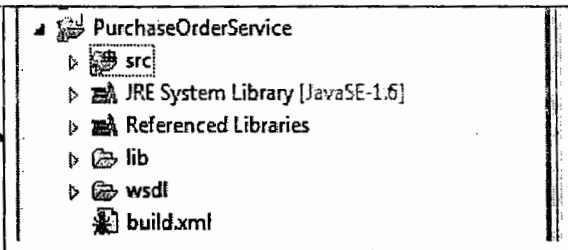
a) Write the WSDL document – As it is a contract first approach, always the development will starts with WSDL document. Now the developer has to write the WSDL document using document/literal encoding.

Let us consider the below PuchaseOrder.wsdl document for developing the Service.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://ebay.in/sales/wsdl"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="po"
    xmlns:et="http://ebay.in/sales/types"
    targetNamespace="http://ebay.in/sales/wsdl">
    <wsdl:types>
        <xsd:schema targetNamespace="http://ebay.in/sales/types">
            <xsd:element name="purchaseOrder">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="itemCode"
type="xsd:string" />
                        <xsd:element name="quantity"
type="xsd:int" />
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="orderConfirmation">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="orderId"
type="xsd:string" />
                        <xsd:element name="status"
type="xsd:string" />
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:schema>
    </wsdl:types>
    <wsdl:message name="order">
        <wsdl:part element="et:purchaseOrder" name="purchaseOrder" />
    </wsdl:message>
```

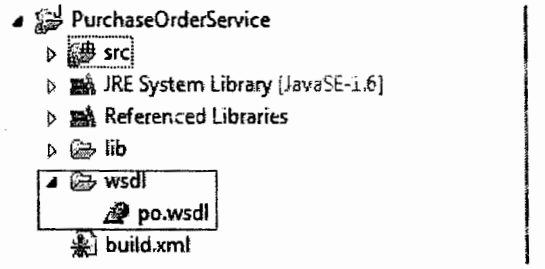
```
    <wsdl:message name="orderResponse">
        <wsdl:part element="et:orderConfirmation" name="return" />
    </wsdl:message>
    <wsdl:portType name="Buy">
        <wsdl:operation name="order">
            <wsdl:input message="tns:order" />
            <wsdl:output message="tns:orderResponse" />
        </wsdl:operation>
    </wsdl:portType>



<wsdl:binding name="BuyBinding" type="tns:Buy">
        <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="order">
            <soap:operation
soapAction="http://ebay.in/sales/wsdl/order" />
            <wsdl:input>
                <soap:body use="literal" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="BuyService">
        <wsdl:port binding="tns:BuyBinding" name="BuyPort">
            <soap:address location="http://www.example.org/" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Create a Java Project

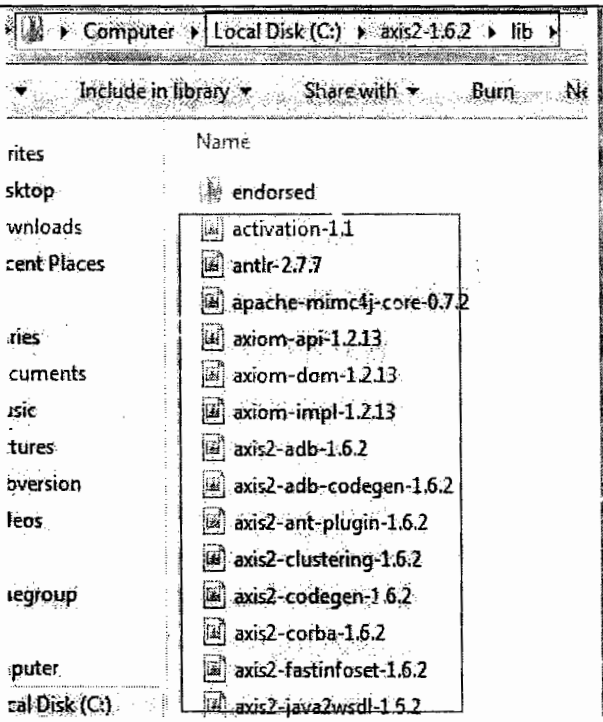


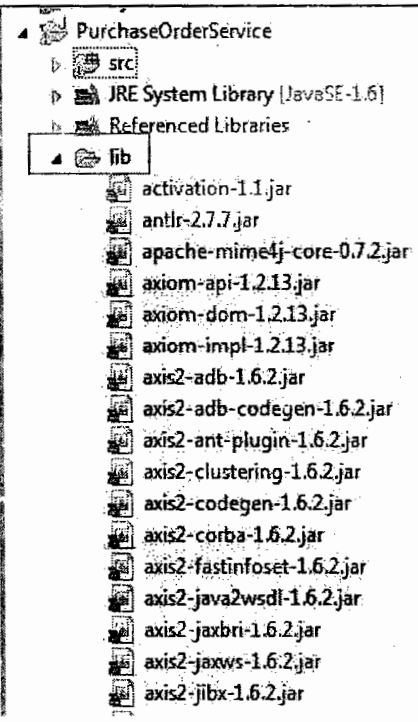
Create a folder WSDL

In the WSDL folder create the po.wsdl with the above wsdl document.



Copy all the axis2 jars into the lib directory of the Project





Copy all the jars from c:\axis2-1.6.2\lib directory to the Project Lib directory and configure the build path.

Run wsdl2java

As it is contract first approach, we need to give wsdl as input so that it will generate the necessary classes for developing the service.

So we need to set the path pointing to c:\axis2-1.6.2\bin, then we can run wsdl2java.bat from any directory.

Now switch to the project directory and run the wsdl2java.bat compiler which will generates all the required classes for developing the service as shown here.

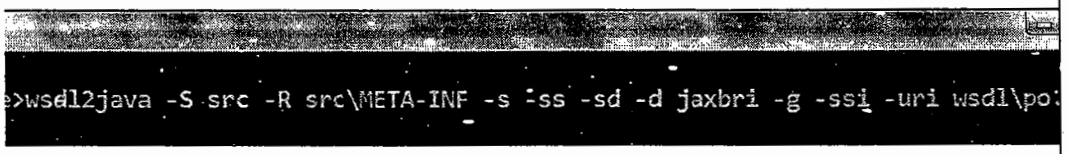
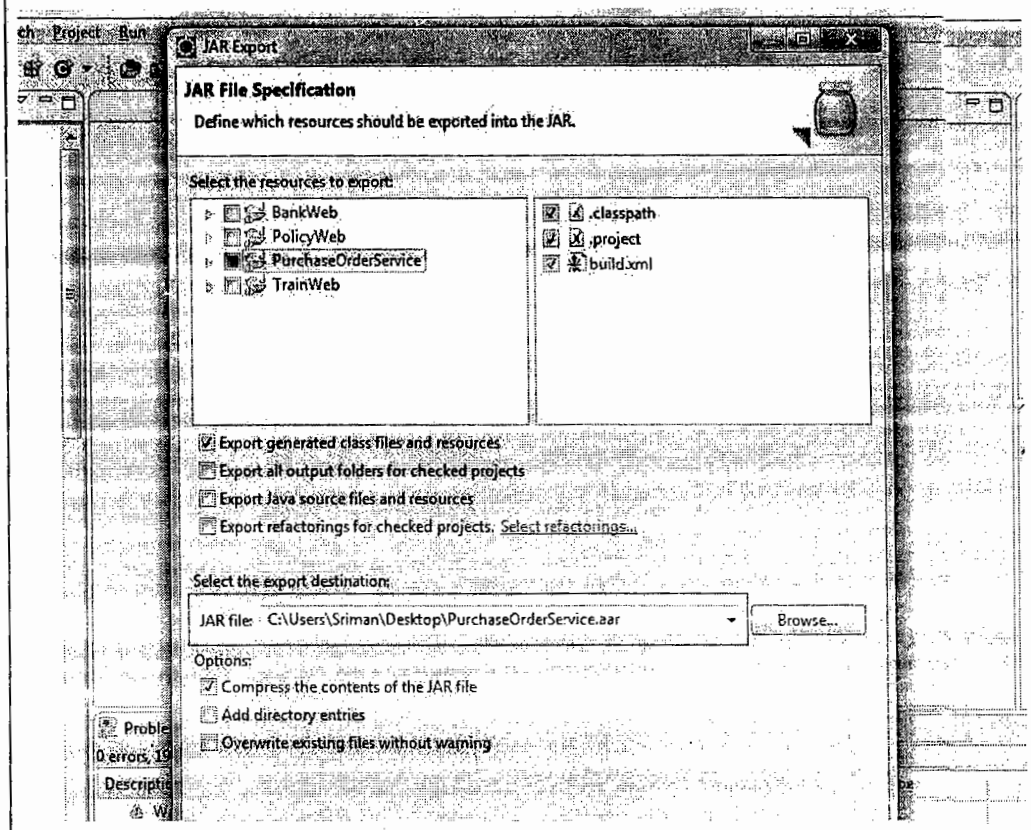


Table describing the option used in wsdl2java

-S	Output directory into which you want to generate Java classes
-R	Where to generate the resource files like services.xml and wsdl. Generally these files are placed under META-INF so the path has been specified as –R src\META-INF
-s	Generate sync style service
-ss	Generate Server side code
-sd	Generate Service Description file which is services.xml contains the information about the service (Web service deployment descriptor)
-d	Stands for databinding. The possible values could be adb, xmlbeans, jibx and jaxbri
-g	Generate all the classes, it generates class for building both client and server
-ssi	Generate SEI interface

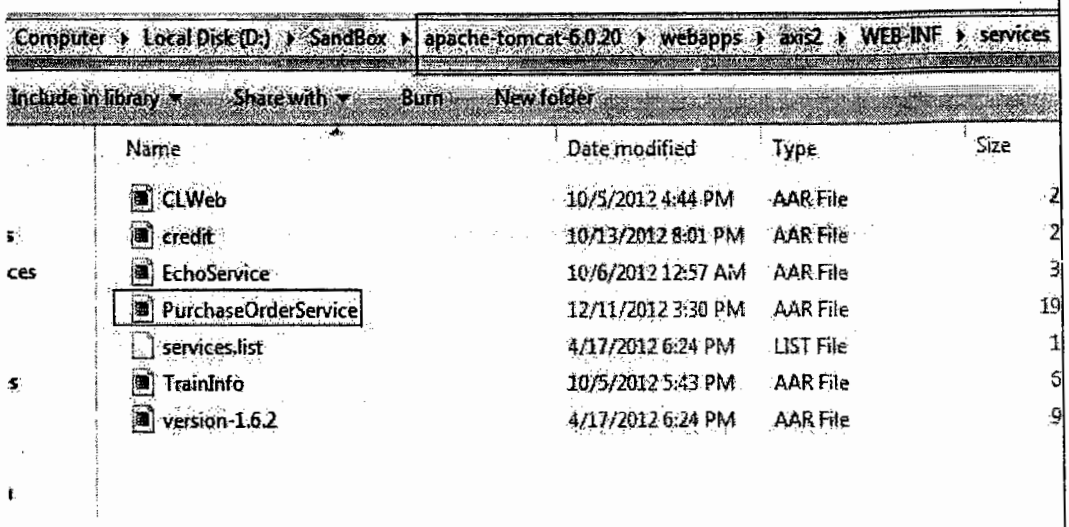
Export the project as jar

Now while exporting the project as Jar, you need to enter the archive name as ".aar" extension. ".aar" stands for apache application archive, where axis2 reads this special packing structure to deploy your application as service.



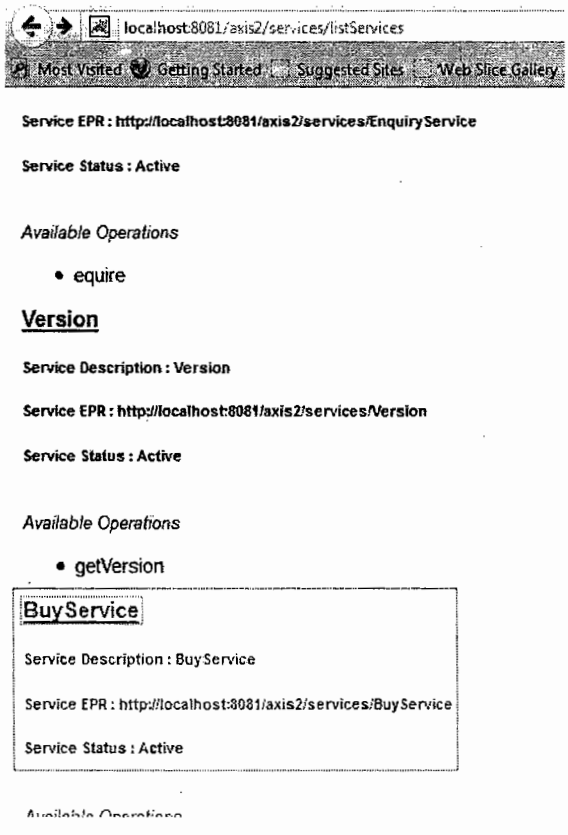
Copy the .aar exported file to Tomcat axis2 directory (services)

Now copy the exported ".aar" archive to tomcat/webapps/axis2/services directory. Once you paste the aar file under services Apache Axis2 runtime engine would be able to automatically detect and deploys it as a service.



Browse to Axis2 home page

Browse to Axis2 Homepage and click on services link you should see your service being hosted.



This completes the development steps of building Apache Axis2 based web services using contract first approach.

17 Securing JAX-WS (Apache Axis2)

17.1 Securing a Web Services

In this we are trying to understand how to secure a web service which we build in the earlier step.

All we need to do is add policy binding in services.xml which is generated by wsdl2java tool as shown below.

```
<wsp:Policy wsu:Id="UsernameToken"
        xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-wssecurity-utility-1.0.xsd"
        xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
        <wsp:ExactlyOne>
                <wsp:All>

                        <sp:SupportingTokens

 xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
                                <wsp:Policy>
                                        <sp:UsernameToken

 sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/Inclu
deToken/AlwaysToRecipient" />
                                </wsp:Policy>
                        </sp:SupportingTokens>

                        <ramp:RampartConfig
xmlns:ramp="http://ws.apache.org/rampart/policy">
                                <!-- <ramp:user>alice</ramp:user> -->

 <ramp:passwordCallbackClass>org.amazon.sales.wsdl.sec.handler.PasswordCall
backHandler</ramp:passwordCallbackClass>
                        </ramp:RampartConfig>
                </wsp:All>
        </wsp:ExactlyOne>
</wsp:Policy>
```

This should be added with in the service tag of services.xml indicating for which service you want to apply the UserToken Security policy.

Along with that we need to write one more class PasswordCallbackHandler which is declared in the above xml in <ramp:RampartConfig> tag.

In this class we need to write the logic for validation the user/password credentials that client has passed as part of Soap header.

PasswordCallbackHandler.java

```
package org.amazon.sales.wsdl.sec.handler;

import java.io.IOException;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;

import org.apache.ws.security.WSPasswordCallback;

public class PasswordCallbackHandler implements CallbackHandler {

   public void handle(Callback[] callbacks) throws IOException,
                  UnsupportedCallbackException {
         for (int i = 0; i < callbacks.length; i++) {
                  WSPasswordCallback pwcb = (WSPasswordCallback) callbacks[i];
                  if (pwcb.getUsage() ==
WSPasswordCallback.USERNAME_TOKEN_UNKNOWN) {
                        if (pwcb.getIdentifier().equals("john")
                                    && pwcb.getPassword().equals("welcome1")) {
                              return;
                        } else {
                              throw new UnsupportedCallbackException(callbacks[i],
                                          "Invalid Username/password");
                        }
                  }

         }

   }

}
```

Oracle Weblogic Server Webservices

18 JAX-WS API (Oracle JDeveloper IDE)

Tili now we saw how to develop a JAX-RPC or JAX-WS API based various implementations' services using command line tools. Now let us try to understand how we can build a Web service using Integrated Development Environment (IDE) rather than command line tool.

As we know for JAX-RPC or JAX-WS API we have various implementations like JAX-RPC SI, Apache Axis, JAX-WS RI and Apache Axis2 etc. Along with that Oracle also provided its own implementations for developing a JAX-RPC or JAX-WS API based web services using Weblogic Server platform. This means, when you install Weblogic Application Server, it will be shipped with all the Jar Implementations for JAX-RPC and JAX-WS API, in addition it provides tools for supporting the development of Web Service in Oracle Weblogic Server.

Instead of developer working directly with the Weblogic server shipped tools, Oracle has provided a convineint integrated development environment Oracle JDeveloper which allows you to build web services sophisticated.

Oracle JDeveloper along with supporting various application development types like Java, Web applications, Distributed EJB applications, SOA, it provides an GUI Wizard's which allows you to navigate between wizards by clicking on next to create/generate Webservices. Oracle JDeveloper not only supports development of Provider, it even provides GUI Wizard for developing Consumer's as well.

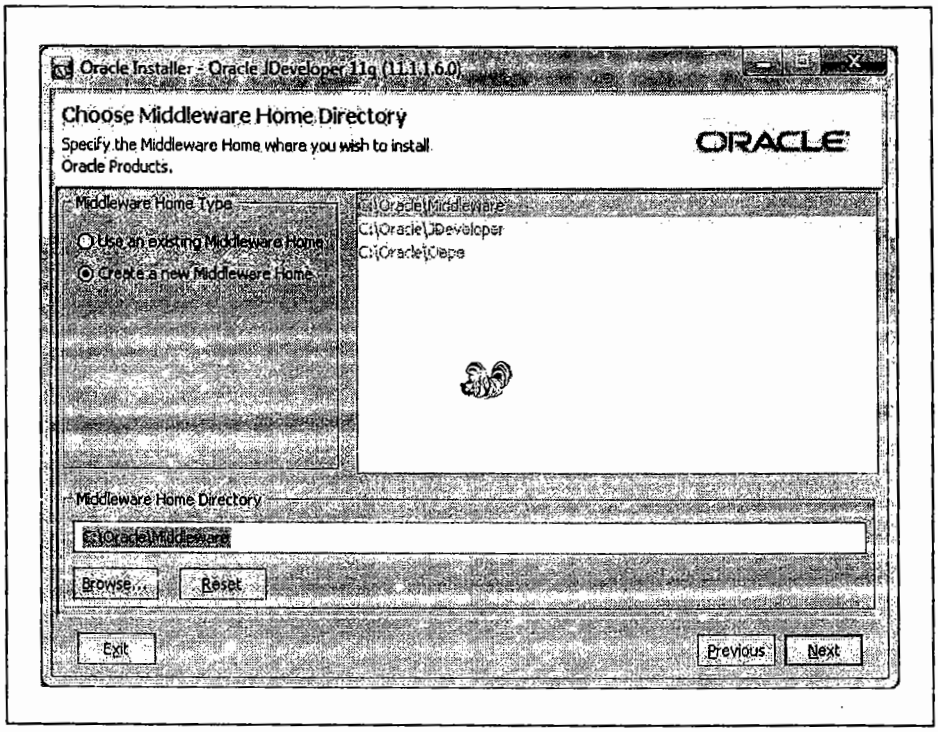
Before proceeding further let us try to understand how to setup the environment.

18.1 Installation & Configuration

In order to build the Webservices, you need to install Oracle JDeveloper IDE. The latest version of Oracle JDeveloper is 11.1.1.6.0. You can download the Oracle JDeveloper from the Oracle OTN site. Rather as part of the software CD, we have provided you the downloaded software. You need to install the Oracle JDeveloper by running the Jar version provided.

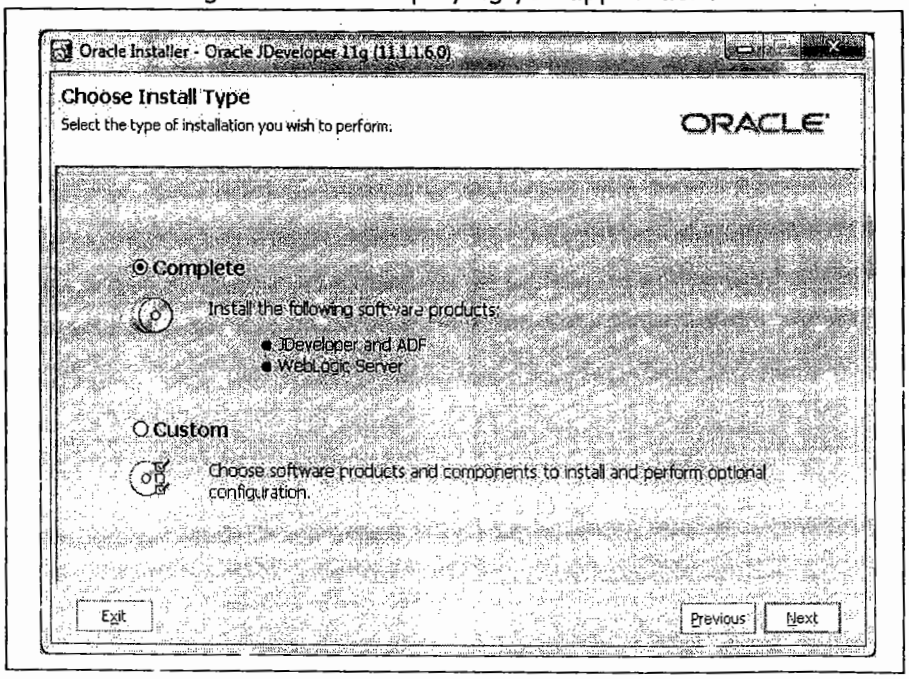
It is the Universal Installer which can be installed on any platform. To install and run the Jdeveloper we need minimum JDK 1.6. One you double click on the Jar, it takes couple of minutes to open the Installation wizard. You just need to walkthrough the wizards by clicking on next.

In the second screen it will ask you to choose the Middleware Home directory.



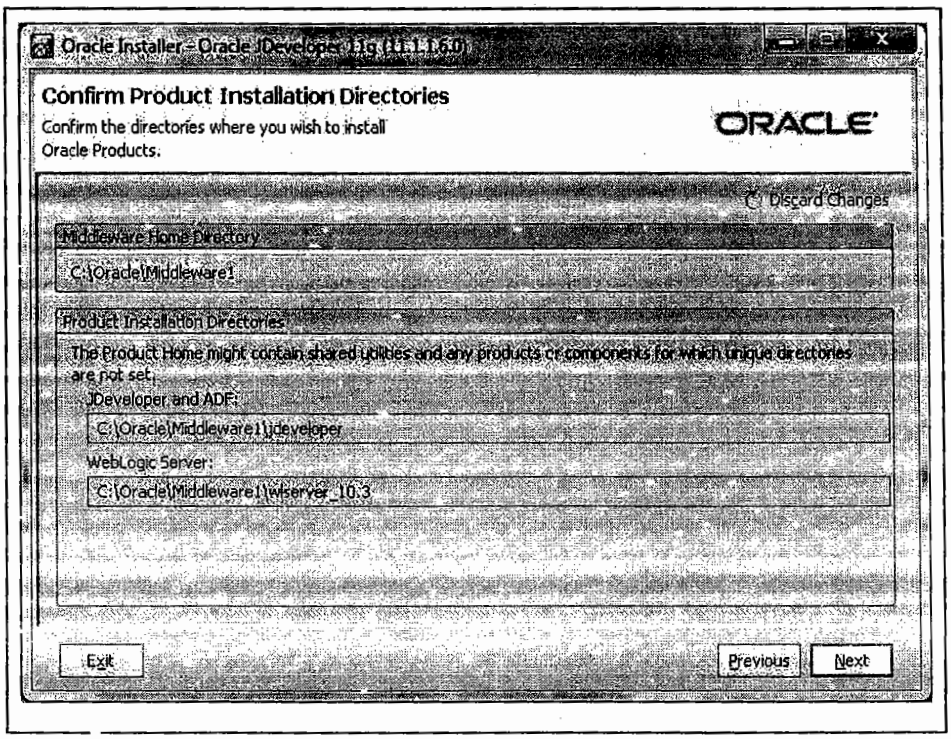
Select the create a new Middleware Home and leave it to the default path and continue the installation by clicking on next.

In the Screen# - 3 it will ask for type of installation here, you need to select as complete. By selecting complete, along with installing Oracle JDeveloper, it even installs Oracle Weblogic server for deploying you applications.



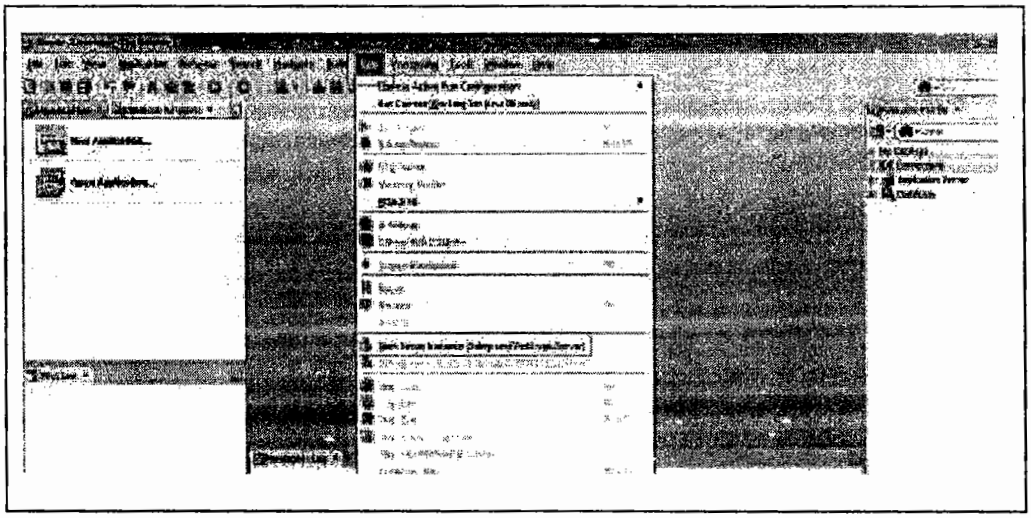
In the Screen# - 4 it will ask for the JDK path, provide the directory path under which your Java has been installed, and click on next.

In the Screen# - 5 it will display the directories under which JDeveloper and Weblogic server is being installed.



Finally click on next and next to complete installation. Now open the Oracle Jdeveloper by clicking on the Start menu.

This will pulls up the Oracle JDeveloper on the desktop, which contains the Integrated Weblogic Server in it. If it is the first time we are running the Oracle Weblogic Server, when we click on start Server Instance from Run menu it will ask you the Port, Username and password with which you want to configure the default domain on the Weblogic server as shown below.



Now when you click on the Start Server Instance, it will popup a dialog window and ask you to enter the port, username and password to configure the domain. From the next run onwards, it will not prompt again.

This finishes the setting up of environment to develop Oracle Weblogic implementation Web Services.

18.2 Building Provider (Contract Last)

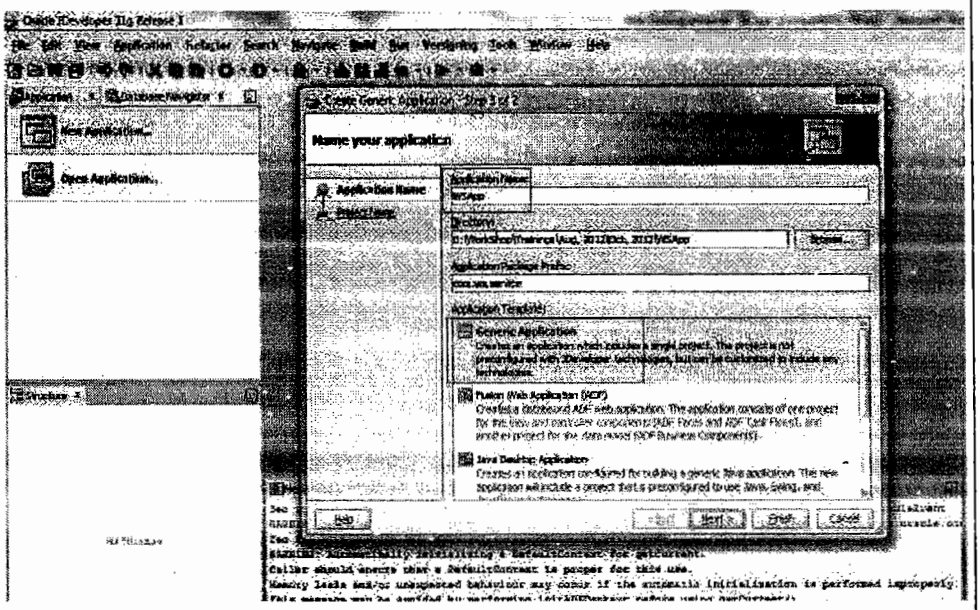
JDeveloper allows you to build Webservice provider using both contract first and contract last approach. All it differs is choosing the menu items to start with.

Let us try to understand how to build a JAX-WS API Weblogic Server Implementation based web service using contract last approach.

Create a New Application in Oracle JDeveloper

Application in JDeveloper is similar to the concept of Workspace in Eclipse. A workspace will contain more than one projects combined to create it as EAR. Similarly JDeveloper Application also comprises of multiple projects combined together to farm an EAR.

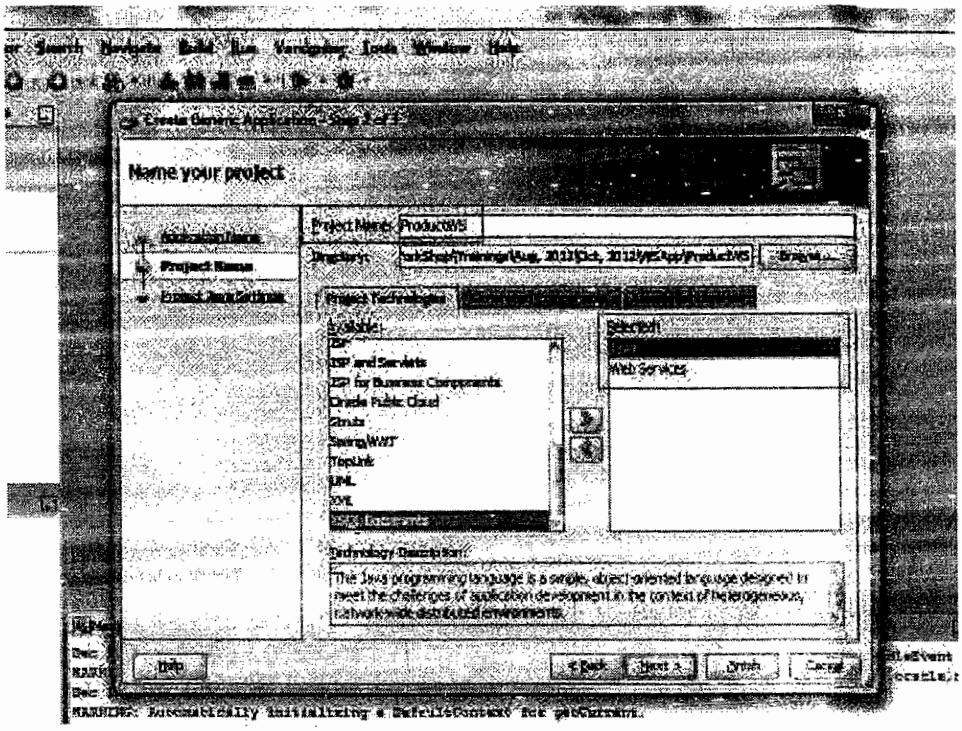
Click on File menu, select New will opens a dialog window. Select here the Application type as generic application with a name and click on next as shown below.



Create the Project

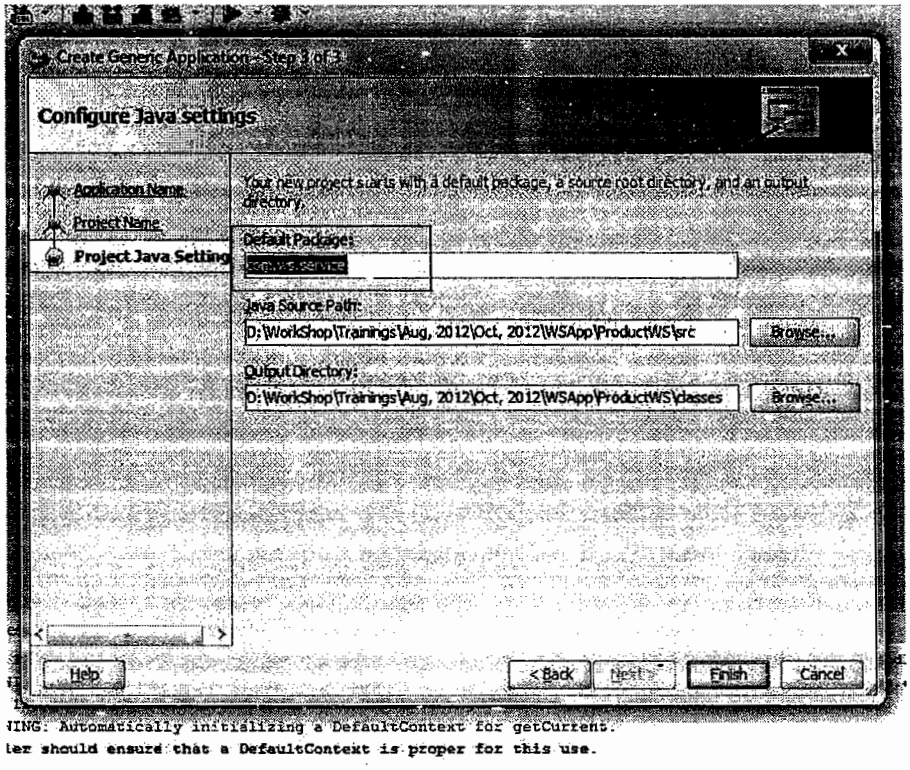
After clicking on the Next, it will show a Project Dialog wizard. Here you need to give the Project Name and select the type of technologies you want to use in building the project and click on Next.

You need to select the technologies as Webservice and automatically Java will also be added to the stream.



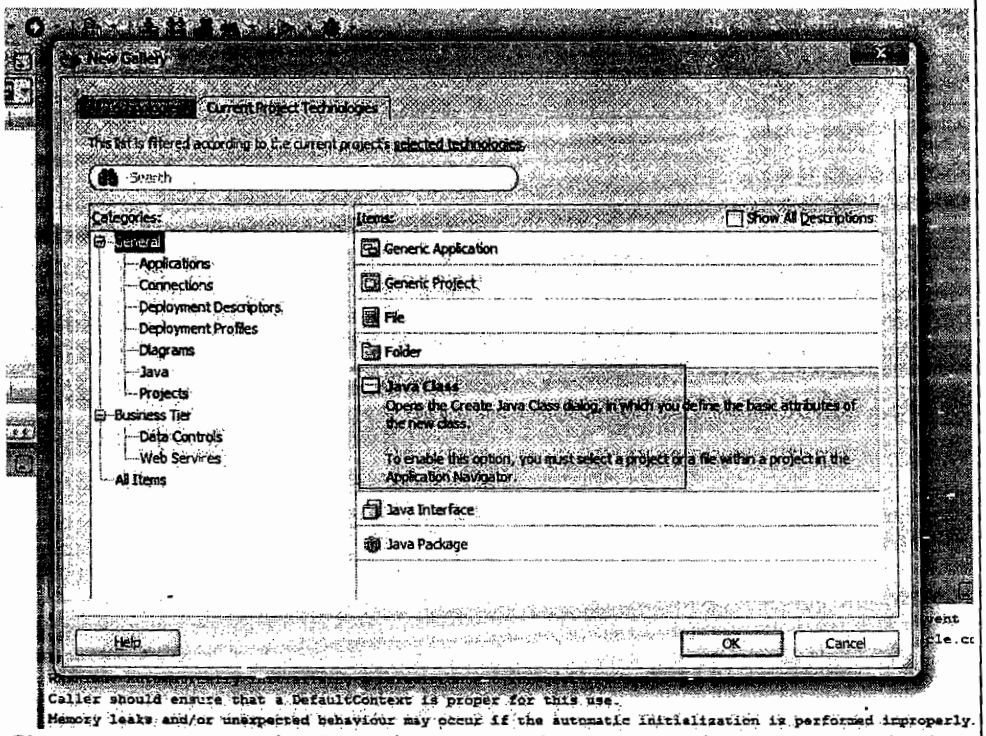
Default Package

In this you need enter the Default Package you want to use as part of your project and click on Finish.

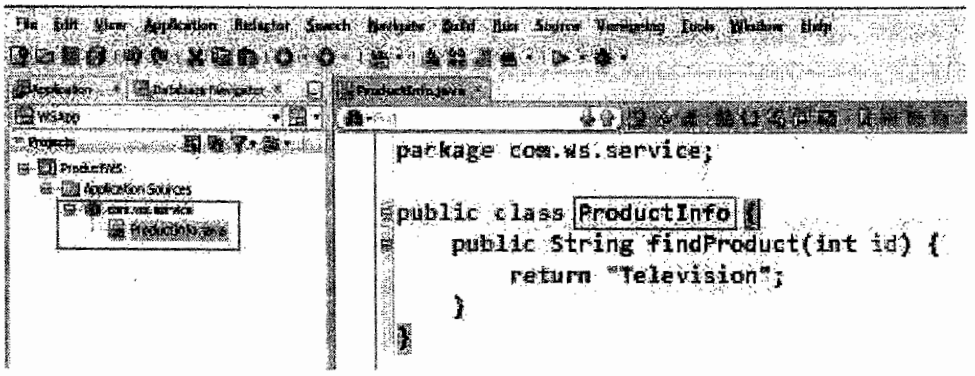


Create a Pojo class

After creating the Application & Project, you need to right click on the Project and select new Java class. Write a class with some methods which you want to expose as Web Service. You don't need to mark with any annotations and let it be absolutely a Pojo class as shown below.

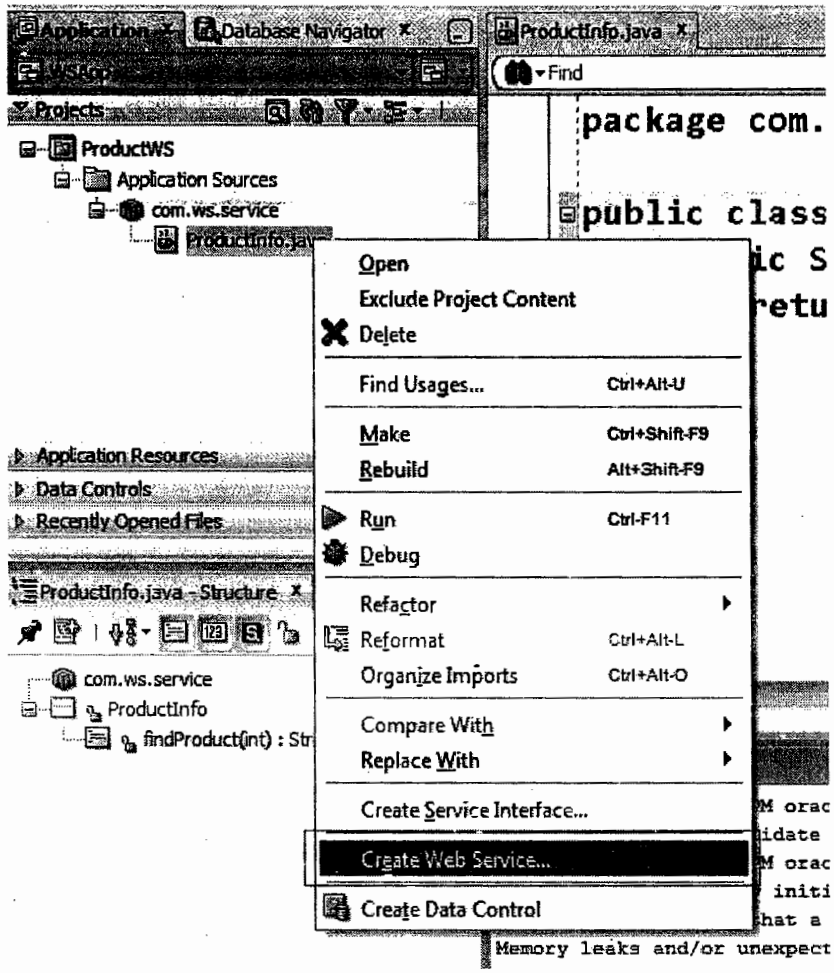


Give some name to the Java class as ProductInfo. Declare some methods you want to expose as web service methods as shown below.



Expose as Webservice

Now right click on the ProductInfo.java file shown in the Projects window left. Click on the menu item create WebService.

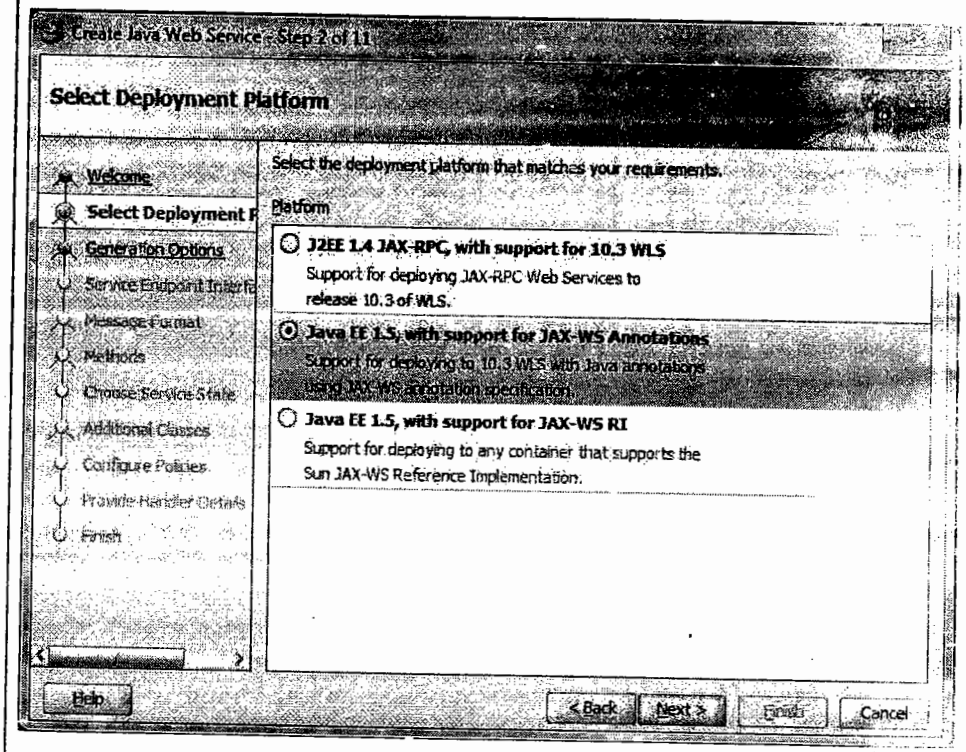


Navigate thorough wizard to create service

After click on Create Web Service option, it opens a wizard dialogue where it will take you through set of screens in creating the Web Service

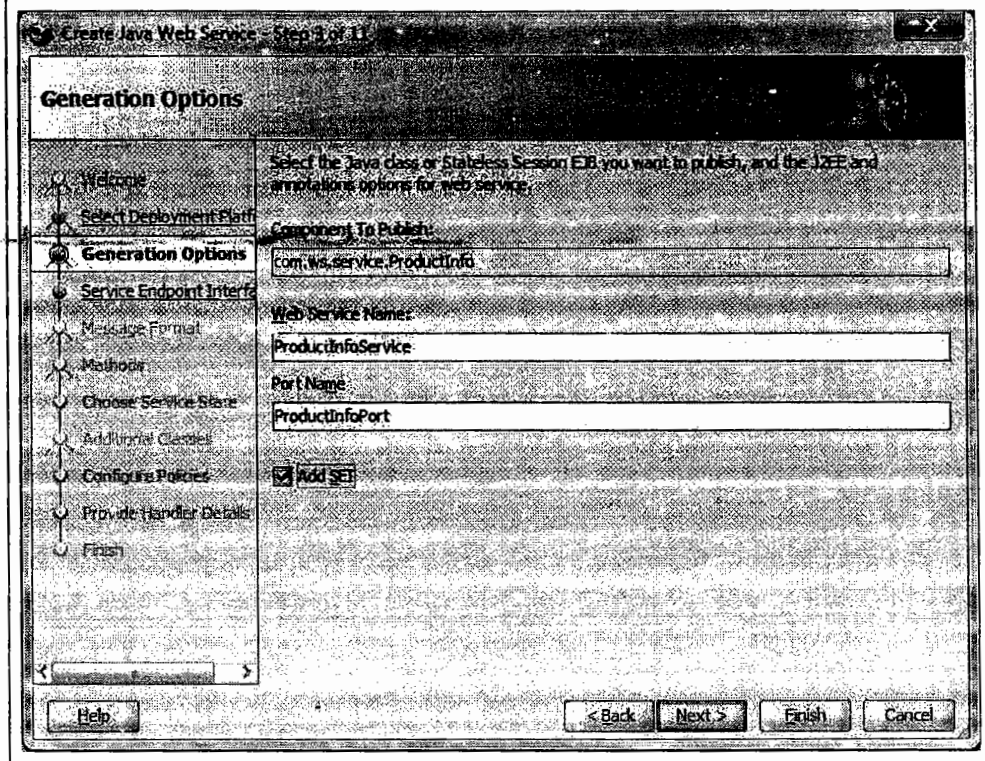
Dialogue# - 1

You need to select which type of service you want to build JAX-RPC or JAX-WS RI or WLS based JAX-WS service.



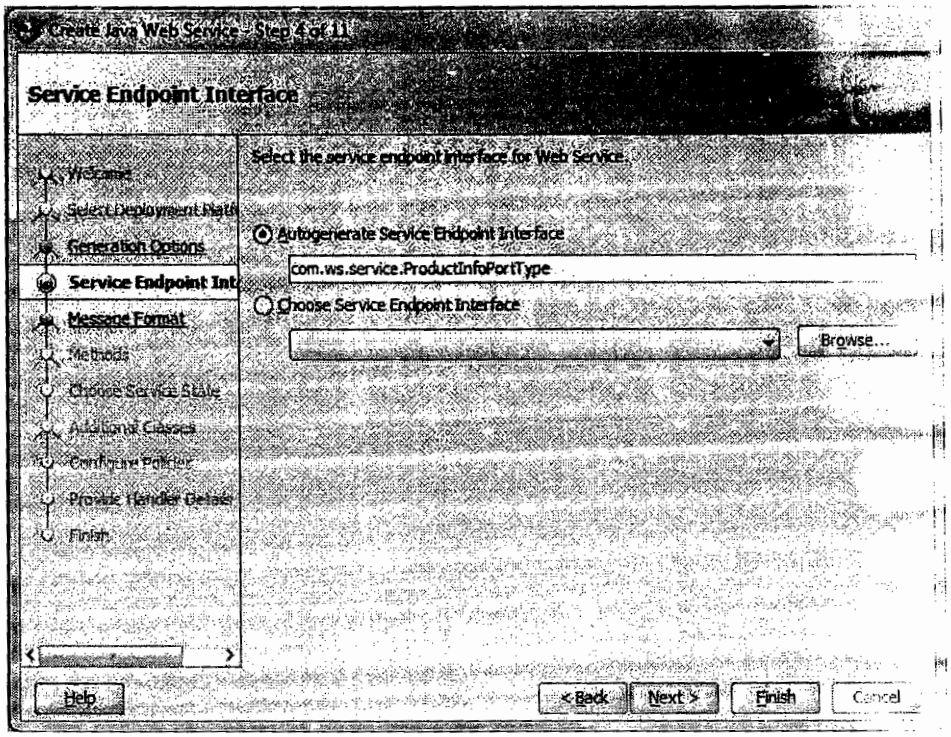
Dialogue# - 2

You can add an SEI interface and configure the Service Name etc.



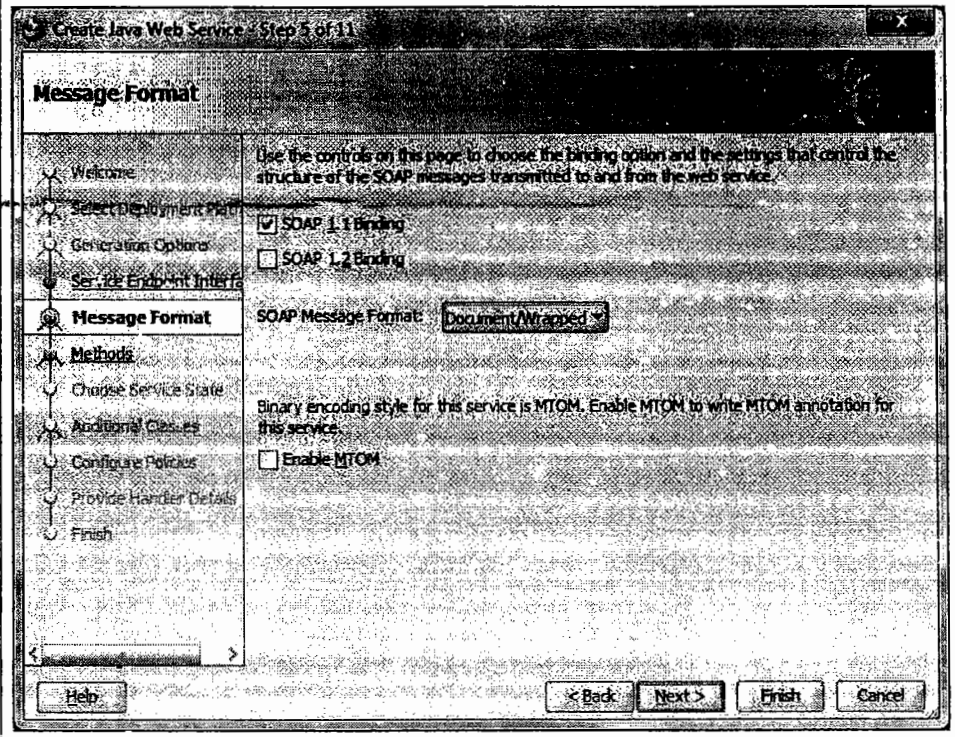
Dialogue# - 3

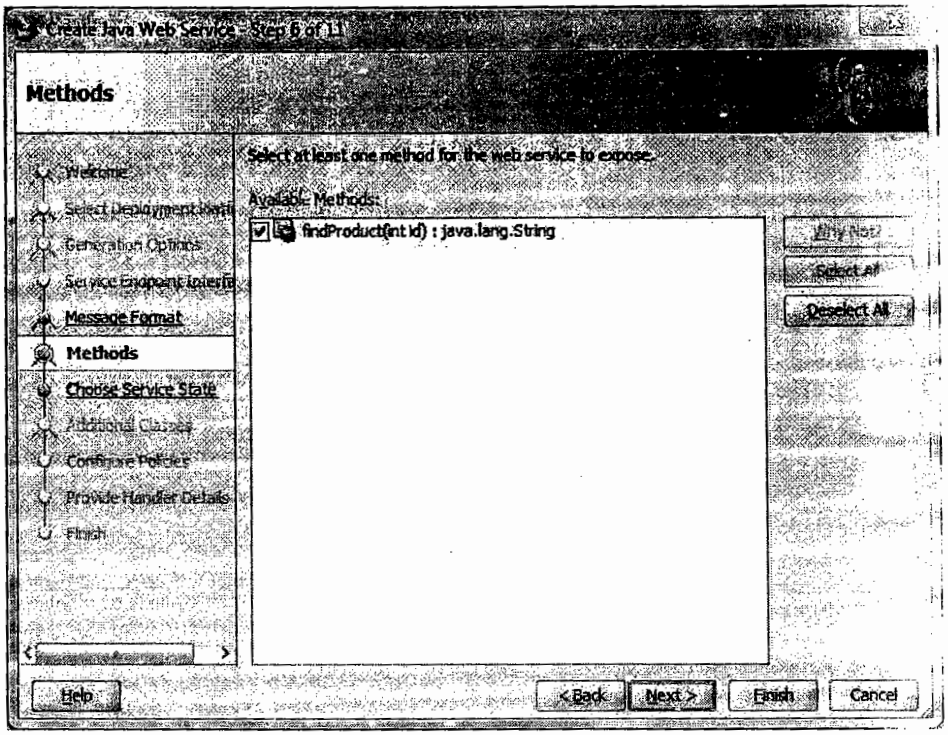
You can select the existing interface as SEI interface or can decide to generate one by default.



Dialogue# - 4

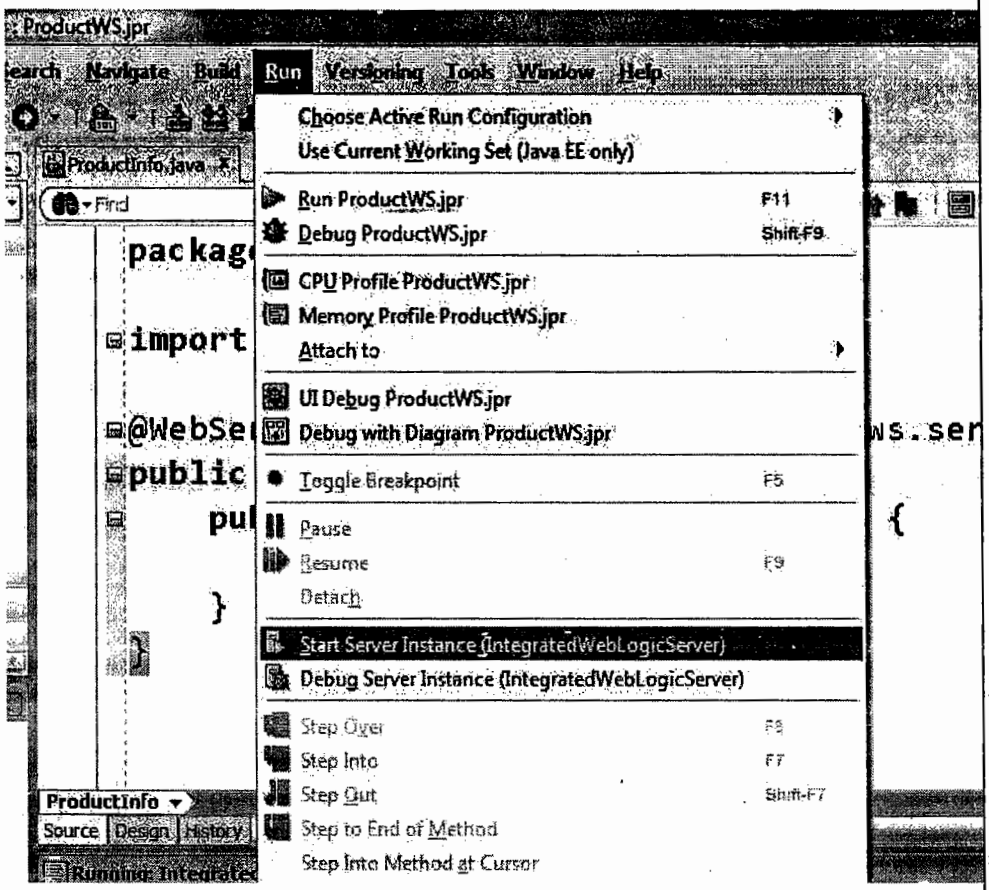
Here you need to choose the Message Format whether it is SOAP 1.1 or SOAP 1.2 based service



Dialogue # - 5	You need to select the methods you want to expose as Web service methods in your class. 
Dialogue # 6,7 and 8	Dialogue# - 6:- Let you choose the classes for whom you want to generate binding classes. Dialogue# - 7:- Ask you to configure the security policies. We will leave it to default by selecting No Policies. Dialogue# - 8: - If you have handlers you need to provide those handler classes here for which the handler configuration would be generated. Finally you need to click on Finish to generate supported classes for exposing it as Webservice.

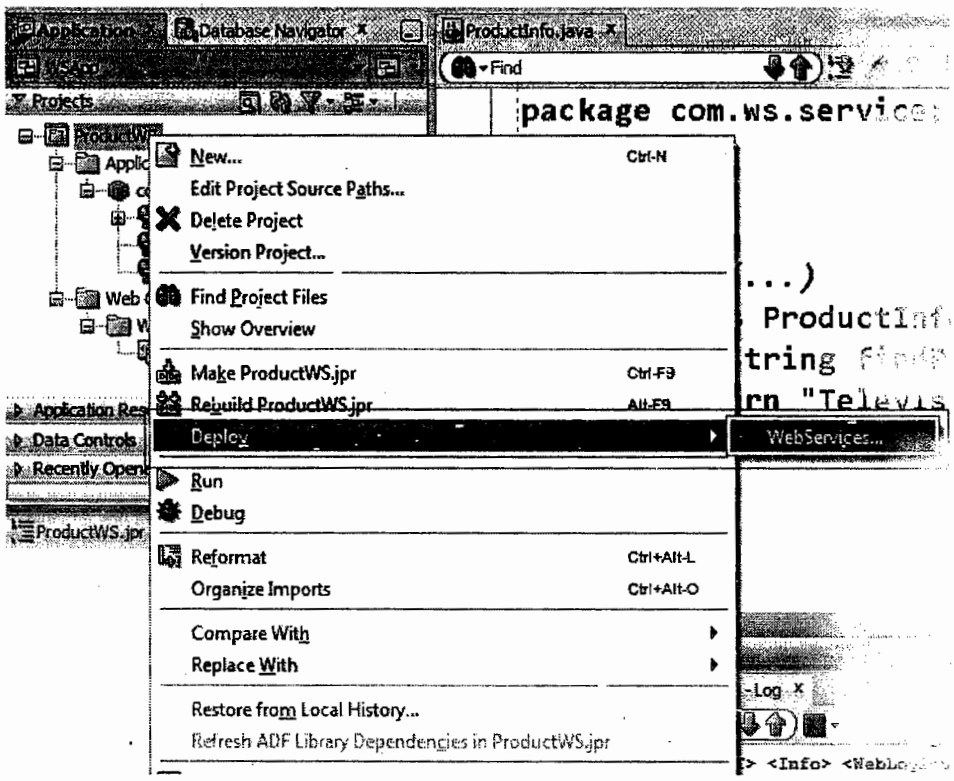
Start the ServerInstance

Now goto run menu and select start server instance. It will start the Weblogic Server. Wait until begins running.



Deploy the Project

Once the Server Instance is running. You need to right click on the Project from the left projects pane. Select the deploy and deploy the project onto Integrated Weblogic Server Instance.

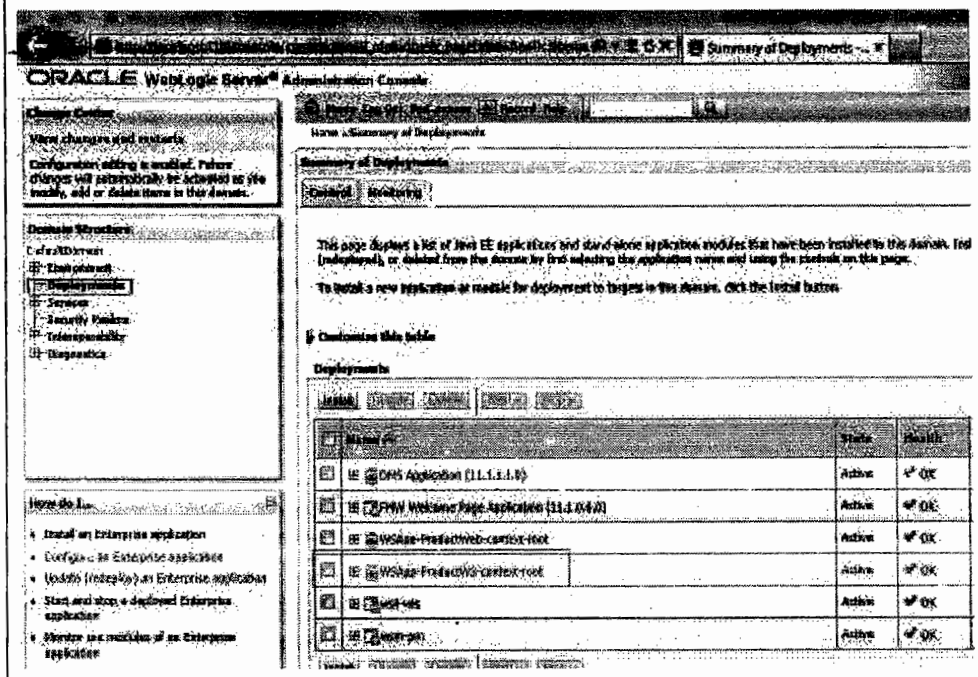


Access the Admin console

Now open the Web browser and access the Admin console of Weblogic Server

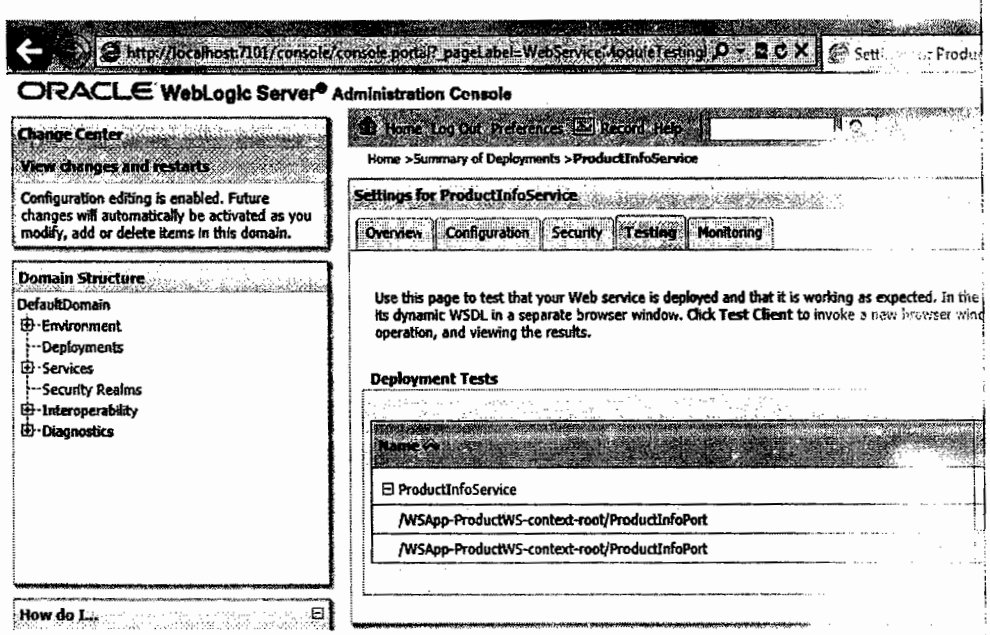
http://localhost:7101/console (the default user/pwd is weblogic/weblogic1)

Now goto deployments, you should find your service deployed.



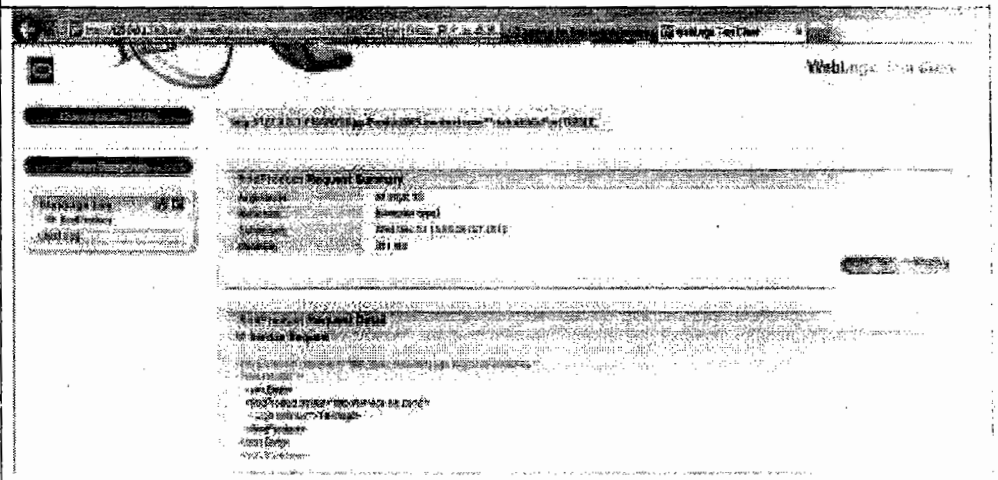
Expand the application and select test client

Expand the (+) sign on the application and click on ProductInfoService will opens a dialogue window. Click on Testing tab will open the below dialogue window.



Now click on Test client link will open a test dialogue window.

Test Client



This completes the development and testing of Oracle Weblogic Server based Web services through Oracle JDeveloper development IDE.

Apache CXF

19 JAX-WS API (Apache CXF Implementation)

Apache CXF is an open source web service framework. CXF helps you build and develop web services using frontend programming APIs, like JAX-WS and JAX-RS.

It can work on various protocols like SOAP/HTTP, XML/HTTP, RESTful/HTTP, or CORBA and work with variety of protocols like HTTP, JMS and JBI.

Some of the features/advantages of Apache CXF:

a) It supports various frontends like JAX-WS and JAX-RS

b) Ease of use, strong tooling support is there to work with contract-first or contract-last approach.

c) Little amount of code is being generated when compared with other frameworks

d) Eventually going to replace Apache Axis2

e) Easy to integrate with Spring framework (makes spring as a first-class citizen)

f) CXF supports variety of Web Service Standards including SOAP, the WS-I Basic profile, WSDL, WS-Addressing, WS-Security (WSSE), WS-Policy, WS-ReliableMessaging, WS-SecurityPolicy, WS-SecureConversation, WS-Trust etc.

g) It can be deployable on any Web Application Container (Container Independent implementation)

19.1 Understanding Apache CXF distribution

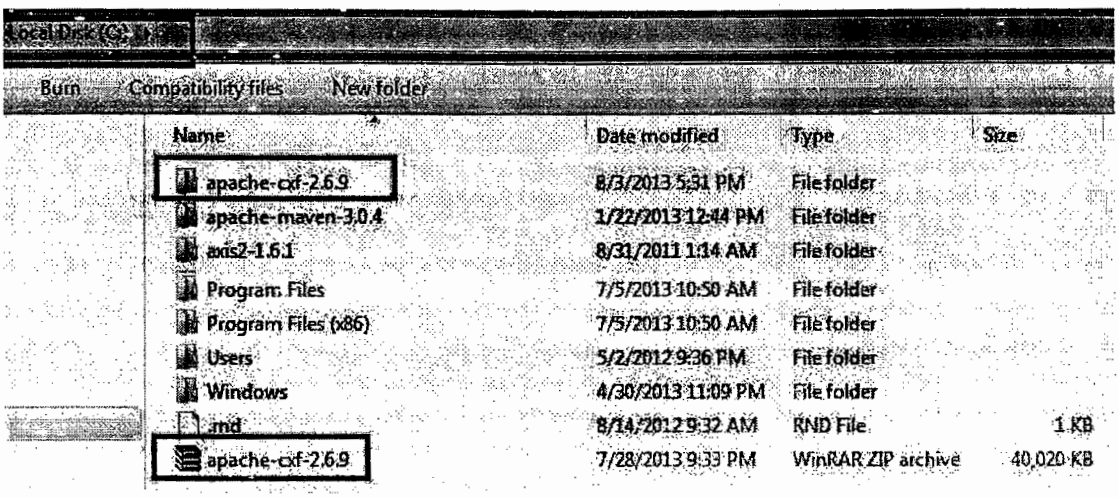
If you want to develop Apache CXF based web service or consumer, you need to download the Apache CXF Binary Distribution. It is a zip containing Jar's and Tool's that are required to develop Apache CXF based provider or consumer.

Here is the link to download Apache CXF distribution.

→ http://cxf.apache.org/download.html

As we mentioned the current release is 2.7.6. We are using in this examples the 2.6.9 version.

Once you download the binary zip distribution apache-cxf-2.6.9.zip, copy it to the c:\drive and extract it.



The binary distribution will contain the following directory structure.

```
apache-cxf-2.6.9
| - bin        - Contains all the tools wsdl2java.bat, java2wsdl.bat,
wsdlvalidator.bat etc
| - docs       – java docs
| - etc
| - lib        - All the jars
| - modules
| - samples – all the samples of apache cxf
```

You have completed setting up the tools/environment; to build apache cxf based web services.

19.2 Building Provider

In building a provider always there are two approaches. Contract First and Contract Last. Let's try to build a contract first based developer using Apache CXF.

Now we are trying to develop JAX-WS API, Apache CXF implementation based provider using Servlet endpoint, using Contract First approach, using Sync Req-Reply based provider using document-literal based web service.

a) **Write WSDL Document** – As it is contract first based approach always the development will starts with WSDL document. Now we need to write the WSDL document with document-literal as the message exchanging pattern.

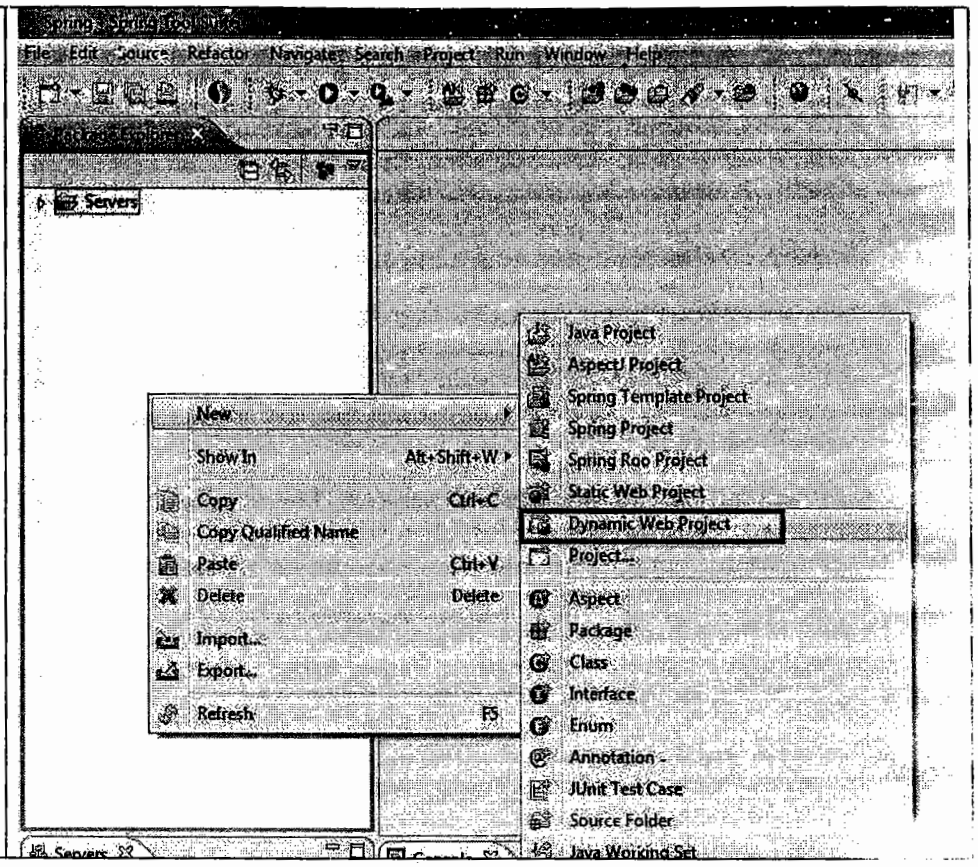
   Let us consider the below traininfo.wsdl document for developing the service.

traininfo.wsdl

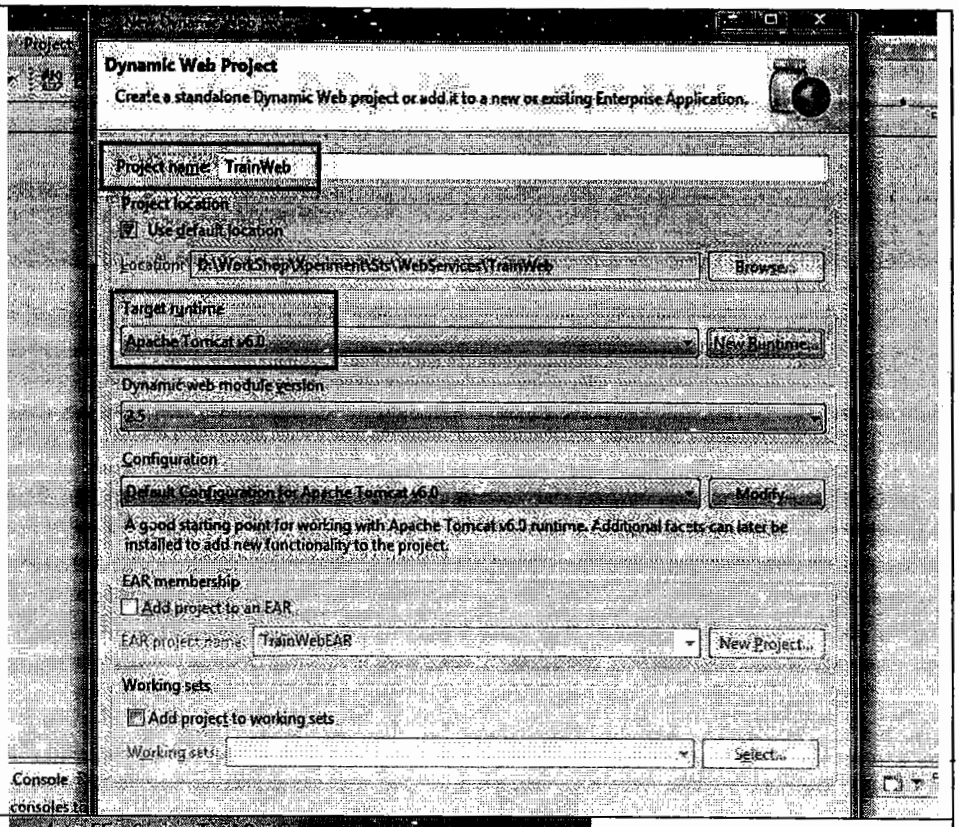
```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://irctc.co.in/reservation/wsdl"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="traininfo"
    targetNamespace="http://irctc.co.in/reservation/wsdl"
xmlns:it="http://irctc.co.in/reservation/types">
    <wsdl:types>
        <xsd:schema
targetNamespace="http://irctc.co.in/reservation/types">
            <xsd:element name="ReservationInfo">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="ssn"
type="xsd:string" />
                        <xsd:element name="name"
type="xsd:string" />
                        <xsd:element name="trainNo"
type="xsd:string" />
                        <xsd:element name="src"
type="xsd:string" />
                        <xsd:element name="dest"
type="xsd:string" />
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="Ticket">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="pnr"
type="xsd:string" />
                        <xsd:element name="coach"
type="xsd:string" />
                        <xsd:element name="berth"
type="xsd:string" />
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:schema>
    </wsdl:types>
```

```
<wsdl:message name="reserve">
        <wsdl:part element="it:ReservationInfo" name="in" />
    </wsdl:message>
    <wsdl:message name="reserveResponse">
        <wsdl:part element="it:Ticket" name="out" />
    </wsdl:message>
    <wsdl:portType name="TrainInfo">
        <wsdl:operation name="reserve">
            <wsdl:input message="tns:reserve" />
            <wsdl:output message="tns:reserveResponse" />
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="TrainInfoSOAPBinding" type="tns:TrainInfo">
        <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="reserve">
            <soap:operation
soapAction="http://irctc.co.in/reservation/wsdl#reserve" />
            <wsdl:input>
                <soap:body use="literal" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="TrainInfoService">
        <wsdl:port binding="tns:TrainInfoSOAPBinding"
name="TrainInfoSOAPPort">
            <soap:address location="http://www.example.org/" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

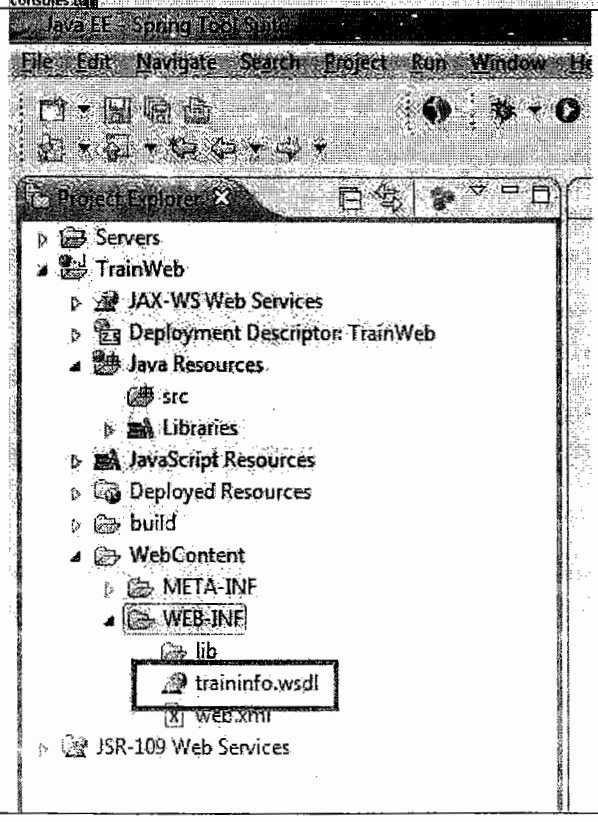
Create a Dynamic Web Project

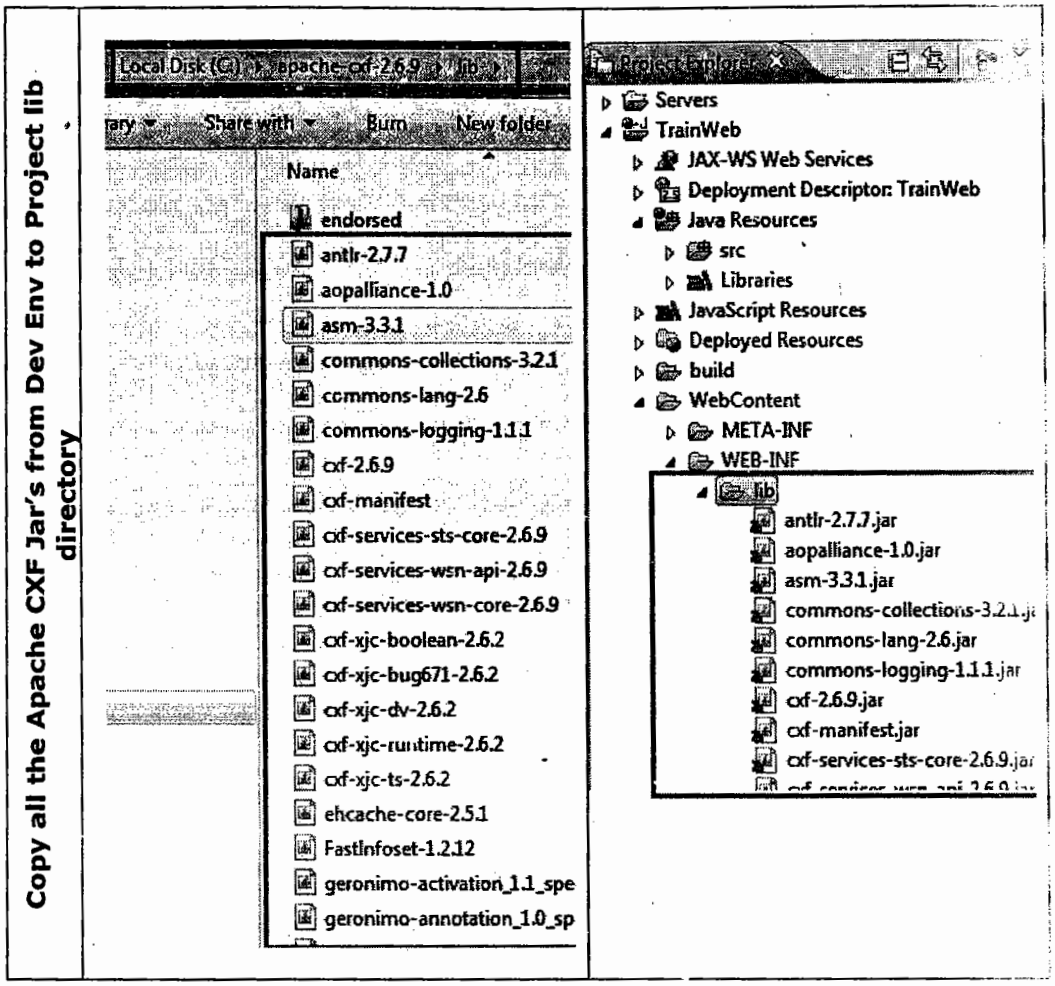


Configure Target runtime (while creating project)



Create WSDL document under WEB-INF directory




Copy all the Apache CXF Jar's from Dev Env to Project lib directory
Local Disk (Q:) ▸ apache-cxf-2.6.9 ▸ lib ▸
Share with
Burn
New folder
Name
endorsed
antlr-2.7.7
aopalliance-1.0
asm-3.3.1
commons-collections-3.2.1
commons-lang-2.6
commons-logging-1.1.1
cxf-2.6.9
cxf-manifest
cxf-services-sts-core-2.6.9
cxf-services-wsn-api-2.6.9
cxf-services-wsn-core-2.6.9
cxf-xjc-boolean-2.6.2
cxf-xjc-bug671-2.6.2
cxf-xjc-dv-2.6.2
cxf-xjc-runtime-2.6.2
cxf-xjc-ts-2.6.2
ehcache-core-2.5.1
FastInfoset-1.2.12
geronimo-activation_1.1_spe
geronimo-annotation_1.0_sp
Project Explorer
Servers
TrainWeb
JAX-WS Web Services
Deployment Descriptor: TrainWeb
Java Resources
src
Libraries
JavaScript Resources
Deployed Resources
build
WebContent
META-INF
WEB-INF
lib
antlr-2.7.7.jar
aopalliance-1.0.jar
asm-3.3.1.jar
commons-collections-3.2.1.ja
commons-lang-2.6.jar
commons-logging-1.1.1.jar
cxf-2.6.9.jar
cxf-manifest.jar
cxf-services-sts-core-2.6.9.jar

Run WSDL2Java

As it is contract first appraoch, once we are done with writing the WSDL document, we need to generate binding classes, to generate binding classes from WSDL, we need to run WSDL2Java tool that is provided by Apache CXF.

This tool is provided to us as part of Binary Distribution which we configured as development environment in c:\ drive.
Now setup the path pointing to the directory c:\apache-cxf-2.6.9\bin

```
c:\>set path=%path%;c:\apache-cxf-2.6.9\bin
```

Verify whether you are able to run the tool or not

```
C:\>wsdl2java -help
wsdl2java -fe|-frontend <front-end-name> -db|-databinding <data-bind
y> -impl -server -client -clientjar <jar-file-name> -all -autoNameRe
a)> -validate<[=all|basic|none]> -keep -wsdlLocation <wsdlLocation>
nSuper <exceptionSuper> -mark-generated -h|-?|-help -version|-v -ver

Options:

   -fe|-frontend <front-end-name>
                 Specifies the front end. (defaults to JAXWS)

   -db|-databinding <data-binding-name>
                 Specifies the data binding. (defaults to JAXB)
```

Switches

-d src	Specify the directory, where you want to generate those files under
-impl	A dummy impl class is generated
-server	Server side code is generated
-frontend jaxws21	By default the api we use is jax-ws2.1. If you want to use jax-ws2.2 then you need to copy the endorsed jars as well. To skip the Jax-ws 2.2 specific methods use this switch

Change to the Project directory and run the tool

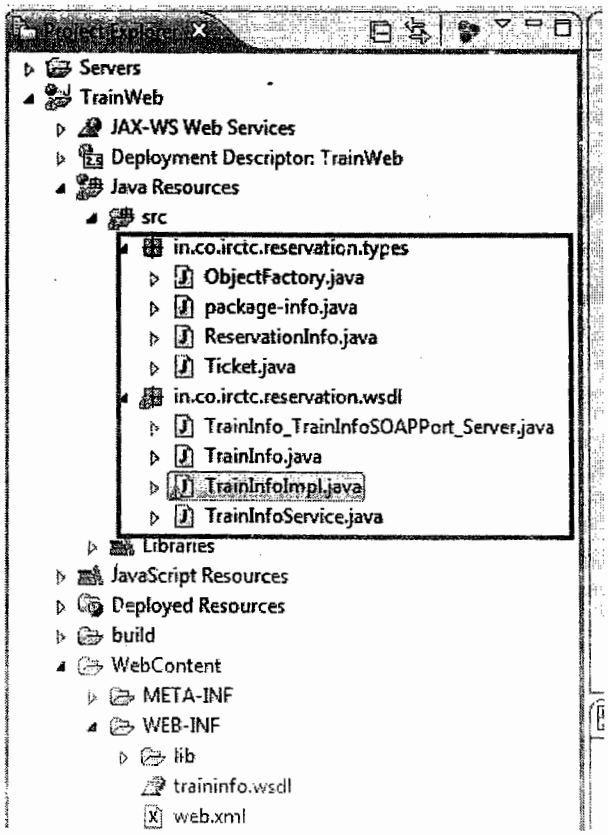
```
C:\>cd D:\WorkShop\Xperiment\Sts\WebServices\TrainWeb
```

Command: wsdl2java –d src –server –impl –verbose –frontend jaxws21 WebContent\WEB-INF\traininfo.wsdl

```
D:\WorkShop\Xperiment\Sts\WebServices\TrainWeb>wsdl2java -d src -se
Loading FrontEnd jaxws ...
Loading DataBinding jaxb ...
wsdl2java -d src -server -impl -verbose WebContent\WEB-INF\traini
wsdl2java - Apache CXF 2.6.9

D:\WorkShop\Xperiment\Sts\WebServices\TrainWeb>
```

Generates SEI Interface, Impl and Input/output classes

It generates Input/Ouput classes, SEI Interface, Dummy Implementation class, xxx_Server.java to publish the service and one Service class used at client side.

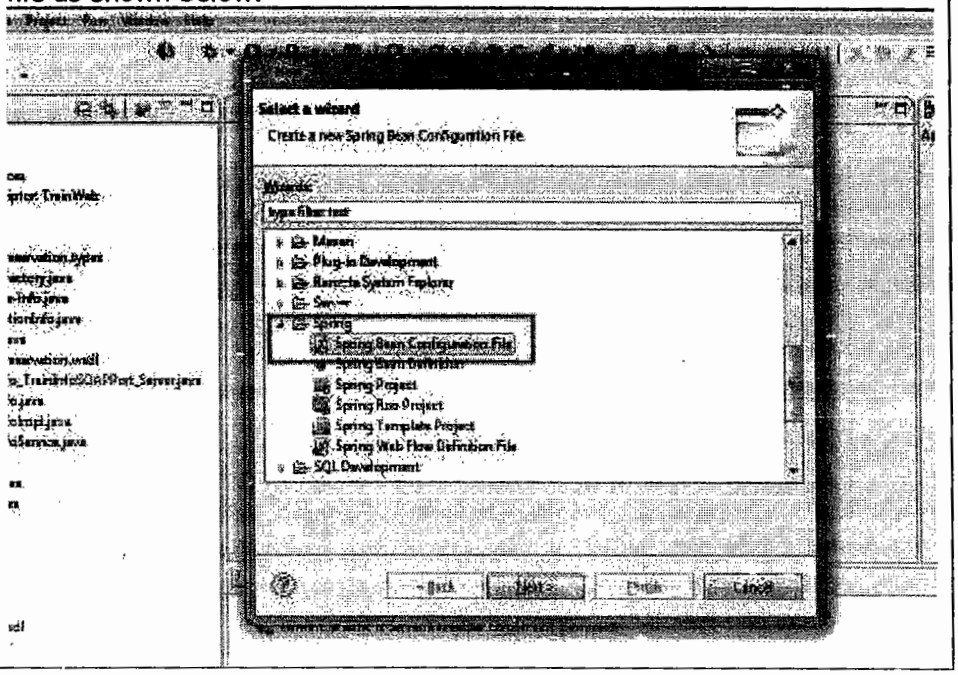


Provide the Implementation in Impl class

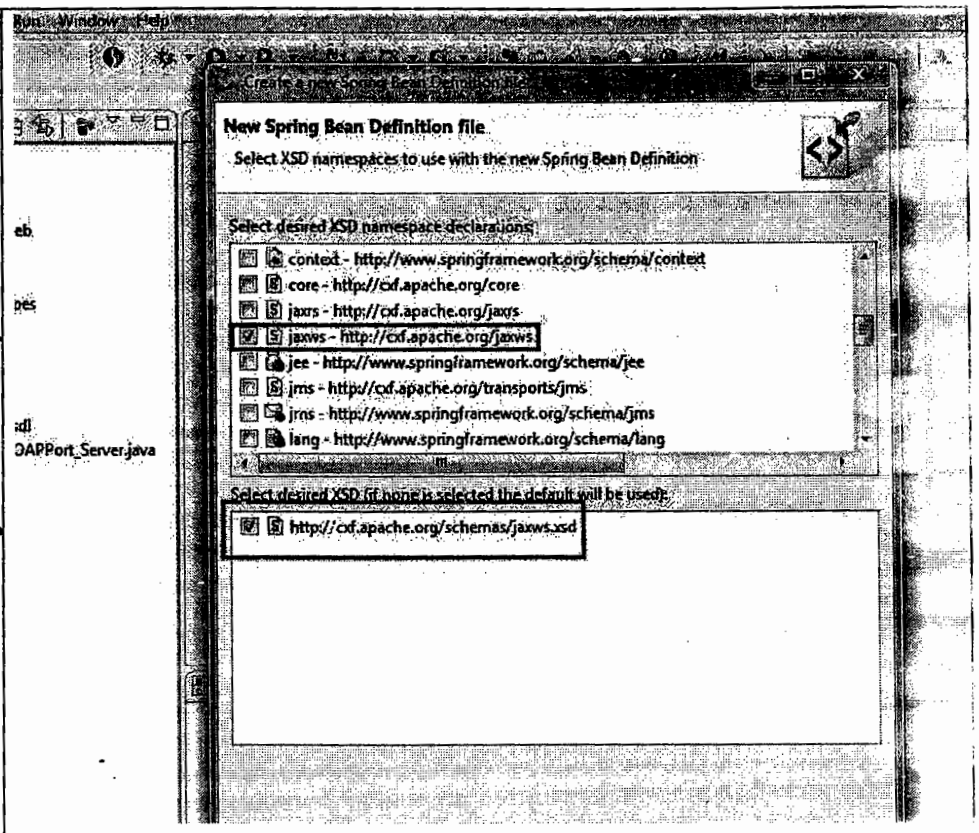
```
/*
 * (non-Javadoc)
 *
 * @see
 * in.co.irctc.reservation.wsdl.TrainInfo#reserve(in.co.irctc.reservation
 * .types.ReservationInfo in )*
 */
public in.co.irctc.reservation.types.Ticket reserve(
        in.co.irctc.reservation.types.ReservationInfo in) {
    LOG.info("Executing operation reserve");
    try {
        in.co.irctc.reservation.types.Ticket _return = new Ticket();
        _return.setPnr("Pnr1131");
        _return.setBerth("S1");
        _return.setCoach("24");
        return _return;
    } catch (java.lang.Exception ex) {
        ex.printStackTrace();
        throw new RuntimeException(ex);
    }
}
```

Create spring beans configuration file

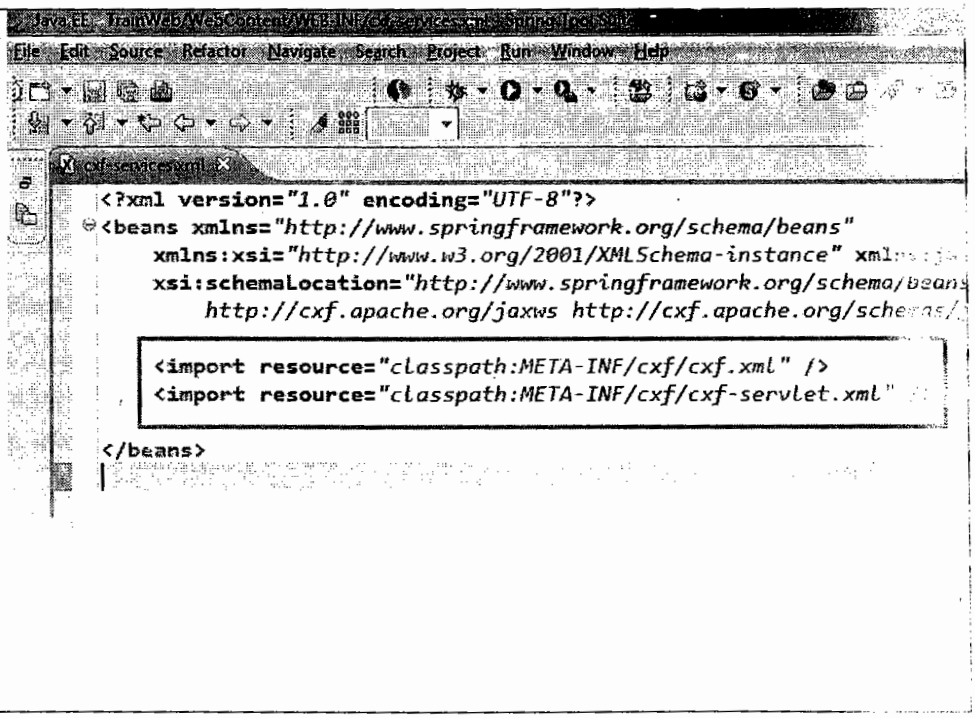
Right click on WEB-INF directory and create spring beans configuration file as shown below.



Give the name of the file as cxf-services.xml and import the jaxws namespace



Import two additional xml files provided by cxf



Configure the implemntaion as jaxws:endpoint bean

Configure the Implementation class as jaxws:endpoint bean to expose it as service as show below.

TrainInfoImpl.java

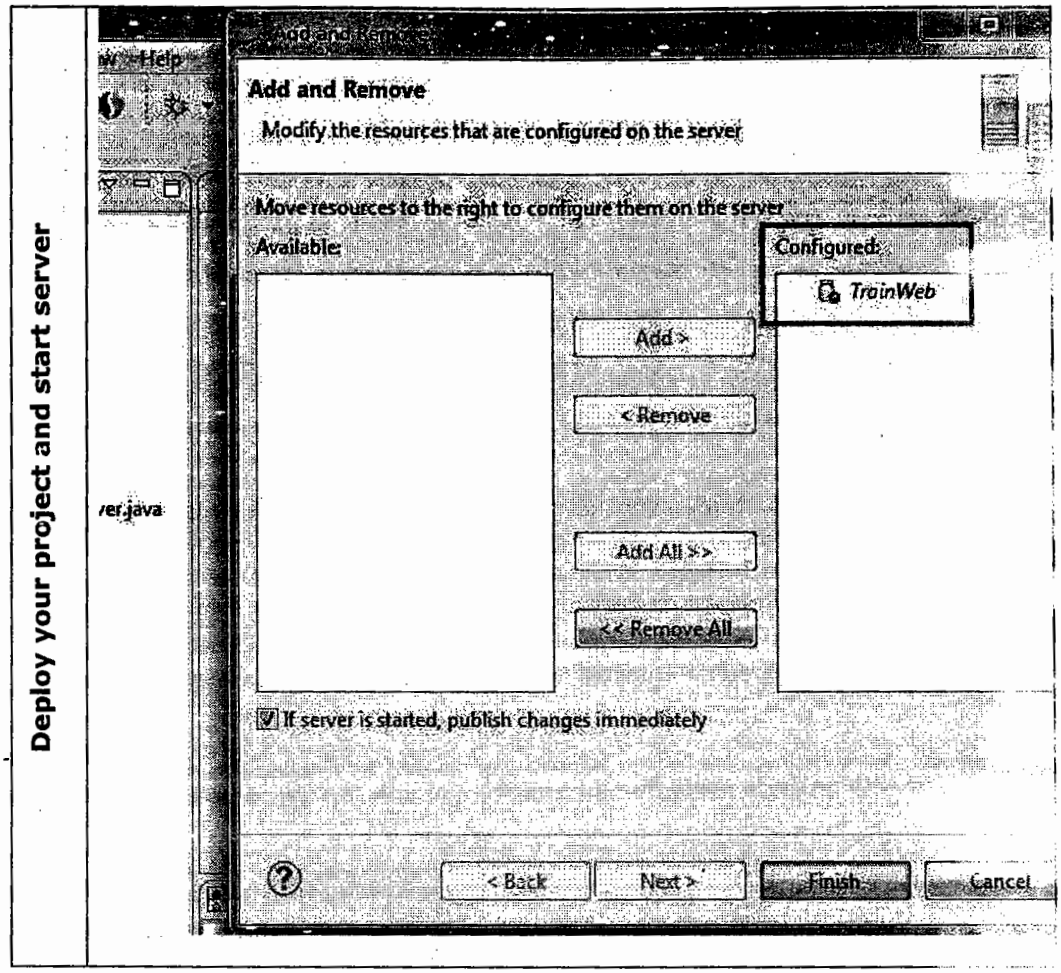
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:jaxws="http:
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://v
        http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd"

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

    <jaxws:endpoint id="trainInfoSOAPPort"
        implementor="in.co.irctc.reservation.wsdl.TrainInfoImpl" address="/t

</beans>
```

Configure Web.xml

Configure web.xml

1) Configure ContextLoaderListener which will takes the cxf-services.xml as input to expose your jaxws endpoint as bean
2) Configure CXFServlet which will acts as an servlet endpoint in processing the request and dispatching the response as shown below.

```
<?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-
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web
    id="WebApp_ID" version="2.5">
    <display-name>TrainWeb</display-name>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/cxf-services.xml</param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <servlet>
        <servlet-name>CXFServlet</servlet-name>
        <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>CXFServlet</servlet-name>
        <url-pattern>/services/*</url-pattern>
    </servlet-mapping>
</web-app>
```

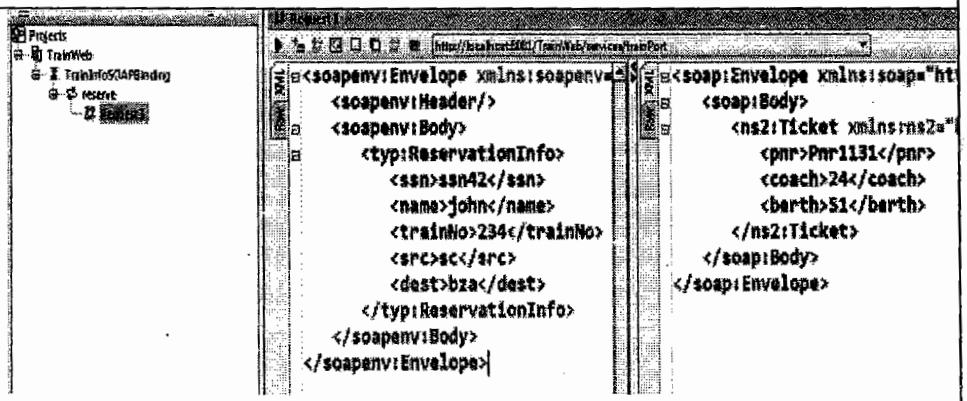

Deploy your project and start server
Help
Add and Remove
Modify the resources that are configured on the server
Move resources to the right to configure them on the server
Available:
Configured:
TrainWeb
Add >
< Remove
Add All >>
<< Remove All
If server is started, publish changes immediately
< Back
Next >
Finish
Cancel

Test by accessing the wsdl url

localhost

```
<?xml version="1.0" encoding="UTF-8"?>
- <wsdl:definitions targetNamespace="http://irctc.co.in/reservation/wsdl" name="traininfo" xmlns:it=
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://irctc.co.in/reservation/
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  - <wsdl:types>
    - <xsd:schema targetNamespace="http://irctc.co.in/reservation/types" xmlns:it="http://irctc
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://irctc.co.in/reser
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      - <xsd:element name="ReservationInfo">
        - <xsd:complexType>
          - <xsd:sequence>
              <xsd:element name="ssn" type="xsd:string"/>
              <xsd:element name="name" type="xsd:string"/>
              <xsd:element name="trainNo" type="xsd:string"/>
              <xsd:element name="src" type="xsd:string"/>
              <xsd:element name="dest" type="xsd:string"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      - <xsd:element name="Ticket">
        - <xsd:complexType>
          - <xsd:sequence>
              <xsd:element name="pnr" type="xsd:string"/>
              <xsd:element name="coach" type="xsd:string"/>
              <xsd:element name="berth" type="xsd:string"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:schema>
```

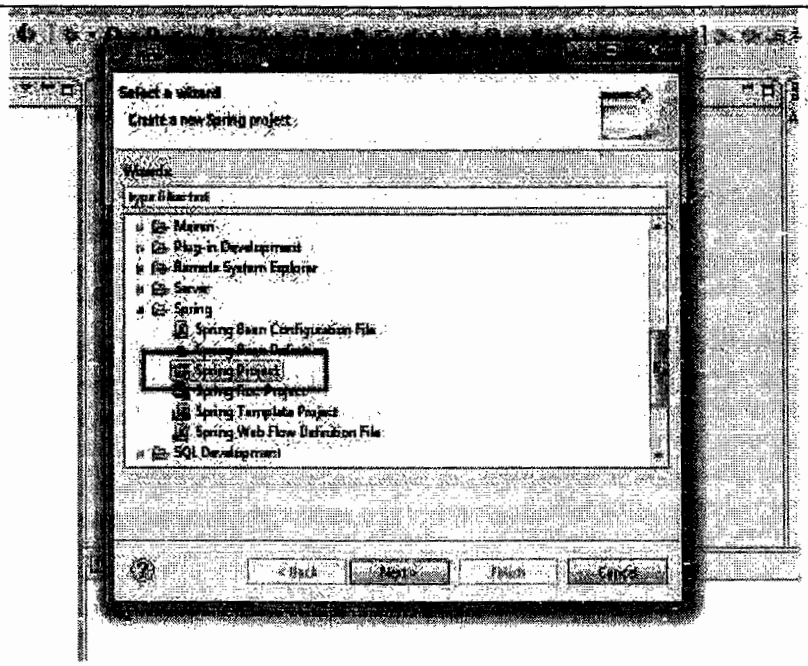
Test service using SOAP UI



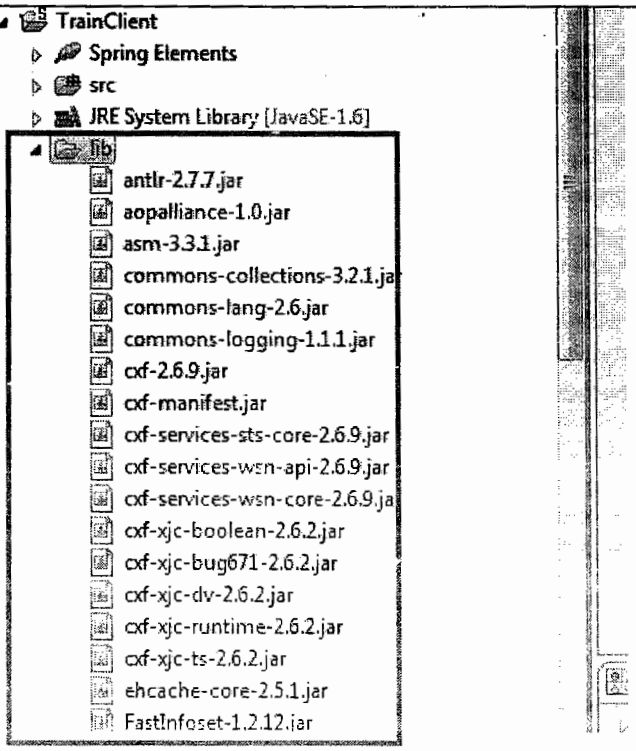
19.3 Building Consumer

In order to access a provider always the input for the consumer is WSDL. Check whether the WSDL is available. Follow the steps in accessing the provider.

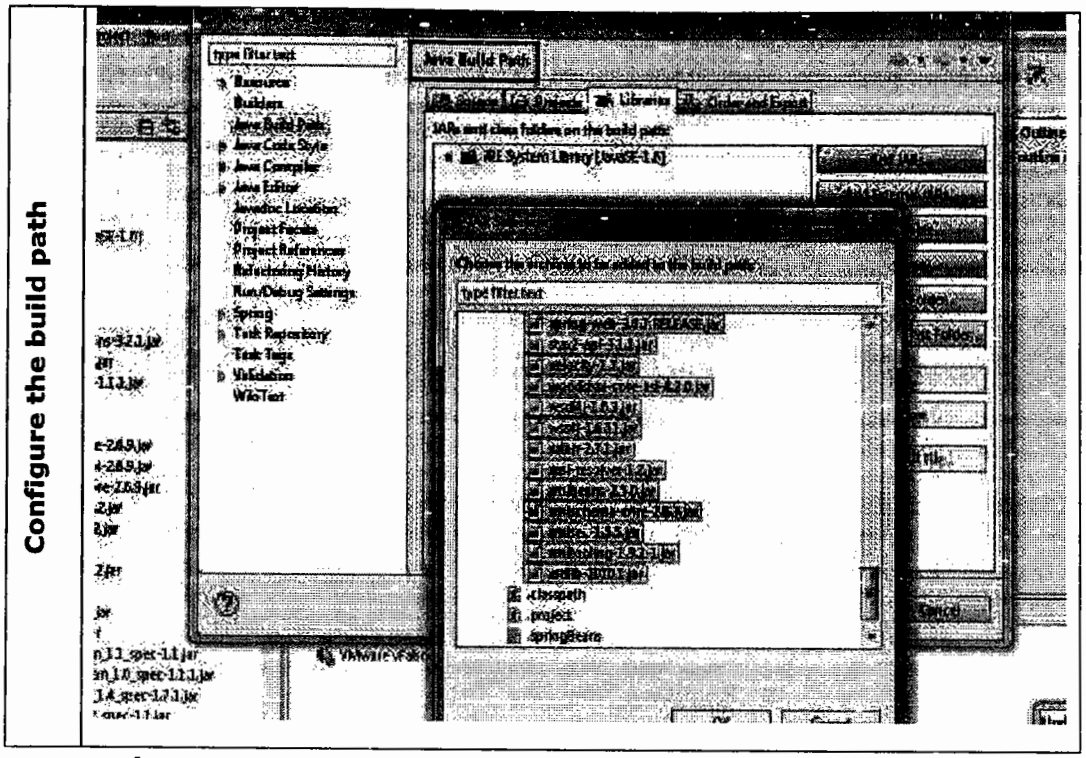
Create a new Spring Project



Create a lib directory and copy paste all the apache cxf jars in to the lib directory

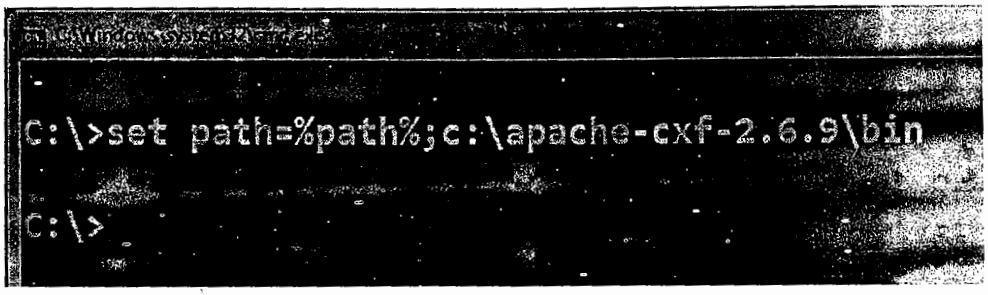


Configure the build path

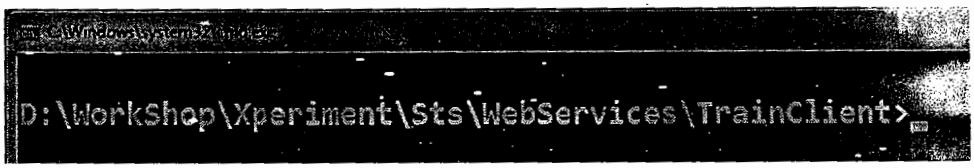


Run the wsdl2java tool to generate the client side classes

Open the command prompt and configure the path pointing to the directory where wsdl2java tool is there.

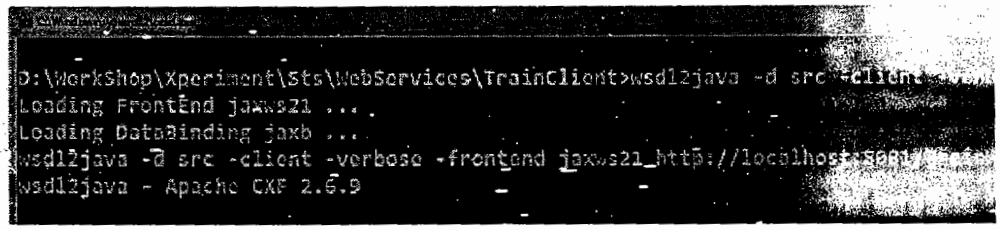


Switch to the project directory



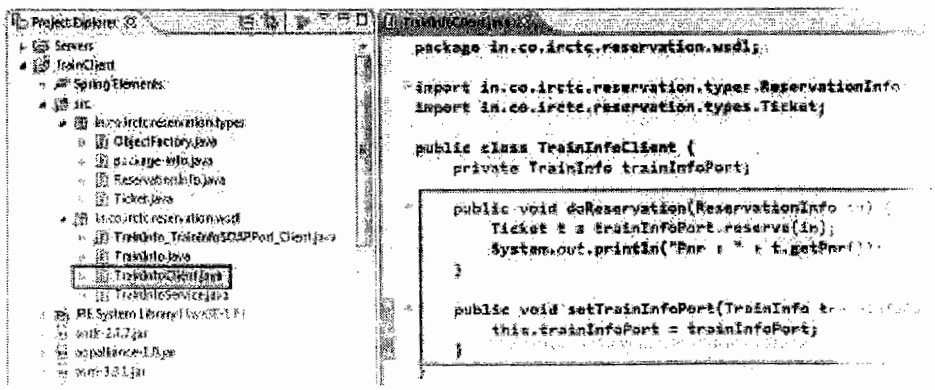
Run the tool with the following switches

Command: wsdl2java –d src –client –verbose –frontend jaxws21 wsdlurl



Create a Client class in the project

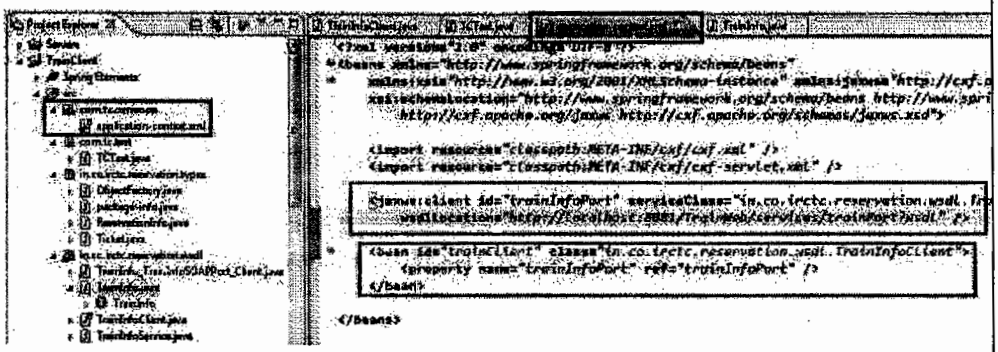
Refresh the project and create one more class TrainInfoClient bean, which you need to inject TrainInfo as port.



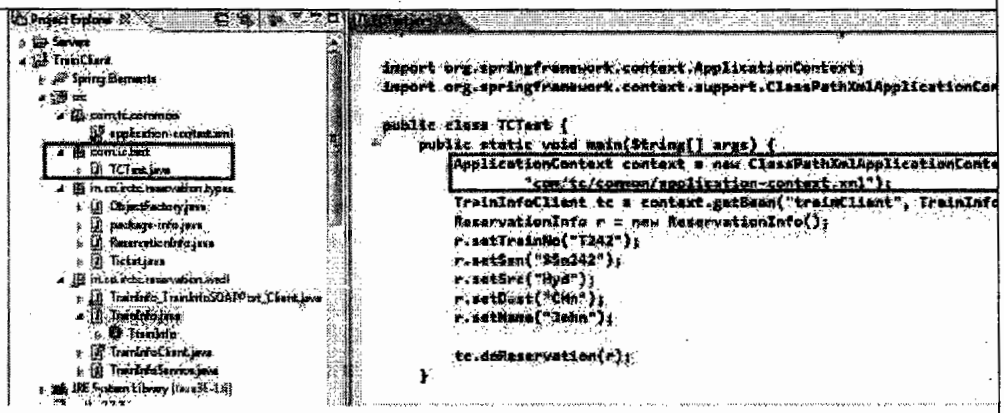
Configure classes as beans

Configure TrainInfo SEI Interface as JAXWS:consumer endpoint

Configure TrainInfoClient also as bean and inject TrainInfo consumer endpoint as dependent via setter injection.



Create a Test class and run



JAX-RS

20 JAX-RS API (Restful Service)

REST stands for representation state transfer. It is a new Architectural style that defines set of rules that can be applied on distributed system, which will lead to interoperable distributed applications.

REST architecture has been higly influenced from web architecture, Roy Fielding one of the principal authors of HTTP specification and co-founder of the Apache HTTP Server project, as part of his doctrial thesis has defined several architectural principles that defines the REST.

Following are the principles of REST

20.1 Principles

1) **Addressable resource**: Every resource has to have a unique directly addressable URI so that we can reach directly. To make integration efforts much easy it is very important to have a direct reachable address. If you see in an Jee most of the distributed applications will not have direct addressable URI, like Ejb or Web Services, due to which other applications has to understand the specific access techniques to integrate.

2) **Uniform, Constrained Interfaces**: It is about exposing your service over a finite set of operations of the application protocols. In case of a Web service if you have to access an operation on a web service, we need to send an extra parameter called "action" or "soapaction". But if you rely on application protocols methods you don't need to use an extra "action" or "soapaction", in case of HTTP it has fixed set of methods with predefined meanings as follows.

    a. **GET**: GET is a readonly operation. It is used for query the server for some information. It is safe and idempotent method.

    b. **PUT**: PUT is modeled as an insert or update. It is also idempotent, as client will always send the identity of the resource he is trying to create/modify on the server.

    c. **POST**: POST is modeled as insert or update. It is non-idempotent and non-safe as client will not send the identity of the resource, sending a POST multiple times will creates a new resource every time.

    d. **HEAD**: It is exactly like GET except that instead of returning the response body.

    e. **DELETE**: It is used for removing the resource on the server.

    f. **OPTIONS**: Options is used to request information about communication options of the resource you are interested in.

Constrained interfaces has many more advantages like below.

Familiarity: If you have an URI to access the resource, you know which methods are exactly there on that resource. You don't need an IDL or WSDL to know the information about the service.

Interoperability: HTTP is universal protocol. Most of the programming languages have support for HTTP protocol; this indicates if your service is built on HTTP it will be interoperable by just sending the data in the format requested by the service. We don't need any additional standards like SOAP or WS-* standards.

Scalability: As the operations are fixed, we can predict the behavior of those methods, let's say a GET always returns the information of the resource, it never modifies the resource. This indicates if we access the resource and for the next access to the resource if there is not PUT or POST or DELETE requests inbetween we don't need to send the request to the server rather we can cache and return the information from client.

3) **Representation Oriented**: Different clients wants different representation of the resource, let's say a Javascript client want the resource to present in JSON format. A typical java application may want the resource to be presented in XML. In which representation we requested the server will present the resource in that specific representation.

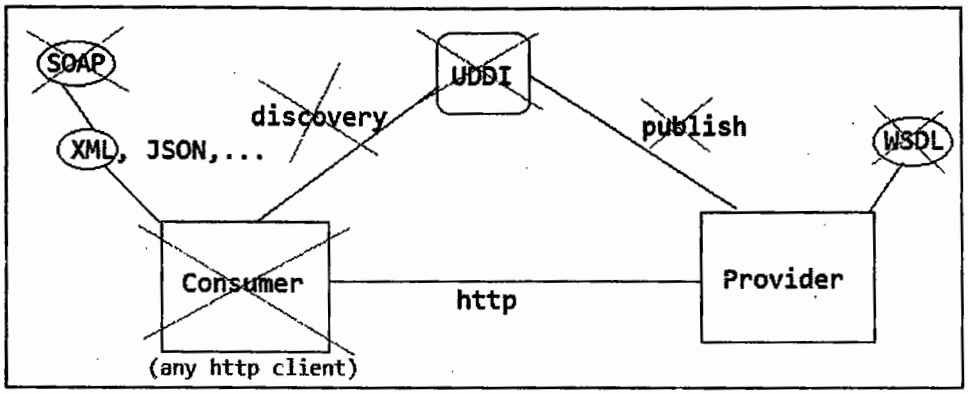
4) **Communicate Statelessly**: Nothing will be stored on the server to remember the client state. Everything will be stored at the client level only. This makes application highly scalable

5) **HATEOAS**: Hypermedia as the engine of Application State. The resource as part of response returned to the client will even send some hyperlinks indicate, what is the next course of action you can do with the resource. This elimates the need of WSDL or IDL describing the resource, which means with no prior knowledge of the resource we should be able to interact with the service.

If your service is being built on above principles then it is called RESTful Resource.

20.2 Architecture

Restful resource architecture is different from a normal web service architecture. Below is the diagram depicting the same.



JAX-RS API has been designed completely based on http protocol. So we cannot built REST Resources on any other protocol apart from Http. To exchange the data between consumer and provider in case of Webservice we need to use only XML, in case of REST in addition to XML we can use any other data formats as well like JSON or TEXT or HTML. Most people use JSON as alternate format to XML to easily parse the data. SOAP is completely eliminated as it is an extra overhead over the XML. To describe the information about provider in case of web service we have WSDL, but in case of REST there is no service description language, as there is not description, there is no registry and publish and suscribe. In case of REST the consumer can be any consumer who can send Http request with data format that is requested by the provider. He need not be a special program built on standards like web service.

As you have seen lot of things are changed between web service and REST, that is the reason it is also called as new architectural style of developing web services is called REST.

20.3 Java support for REST

Java has provided JAX-RS API to develop RESTful services. Initialy it has released jaxrs 1.1 api. Later on it revised and released one more api jaxrs2.0. For which there are several implementations like Jersey (Sun/Oracle) and RestEasy (Jboss).

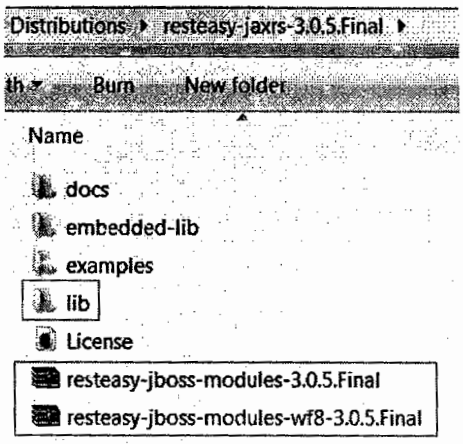
As part of this we are going to see examples on how to work with Jersey as well as RestEasy.

20.4 Setting up RestEasy (Jboss)

To work with Jboss RestEasy implementation we need to do some basic configuration. First we need to download RestEasy distribution it will be shipped as an zip.

RestEasy implementation closely works with JBoss AS 7.1 application server or Wildfly 8.0. So download JBoss AS 7.1.

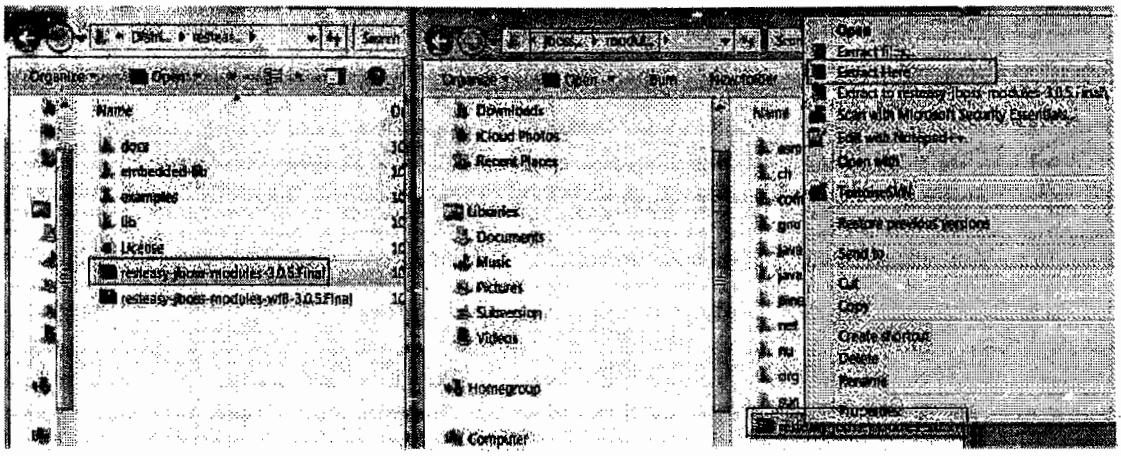
Now extract the RestEasy binary distribution zip downloaded. It contains a lib and two more zip files as shown below.



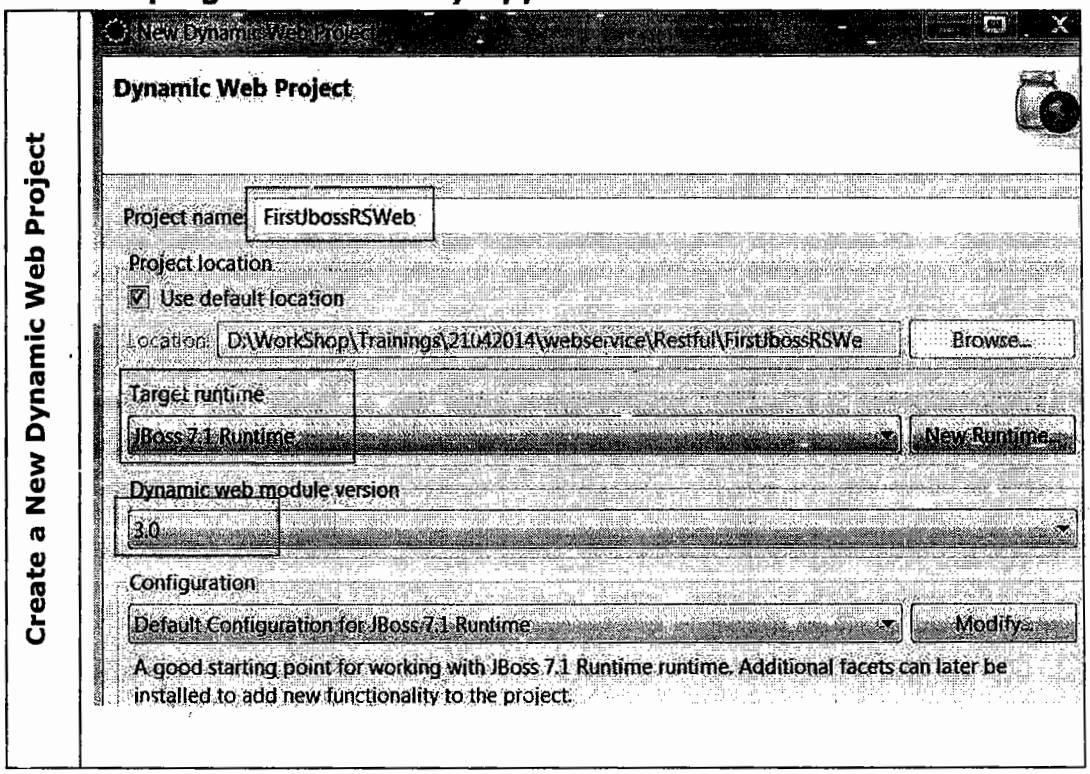
The lib directory contains generic jar's that should be used to develop RestEasy services that can be deployed on any app server. But the other two zip's contains jars one for Jboss and other for Wildfly servers resp.

As we are deploying on Jboss we need to use Jboss related jars. Now copy the zip and goto JBoss AS 7.1 modules directory. For e.g.. <JBOSS_HOME_DIR>\modules and page the zip.

Right click on the zip and extract to here. It will ask you to replace some of the existing jar's just say "yes" will completes copying all the RestEasy jars into the JBoss application server.



20.5 Developing First RestEasy Application



Create a new Java class

GreetResource.java

```
package com.fjrs.resource;

import javax.ws.rs.GET;

@Path("/greet")
public class GreetResource {

    @Produces("text/plain")
    @GET
    public String greet(@QueryParam("person") String person) {
        return "Good Morning " + person + " !";
    }
```

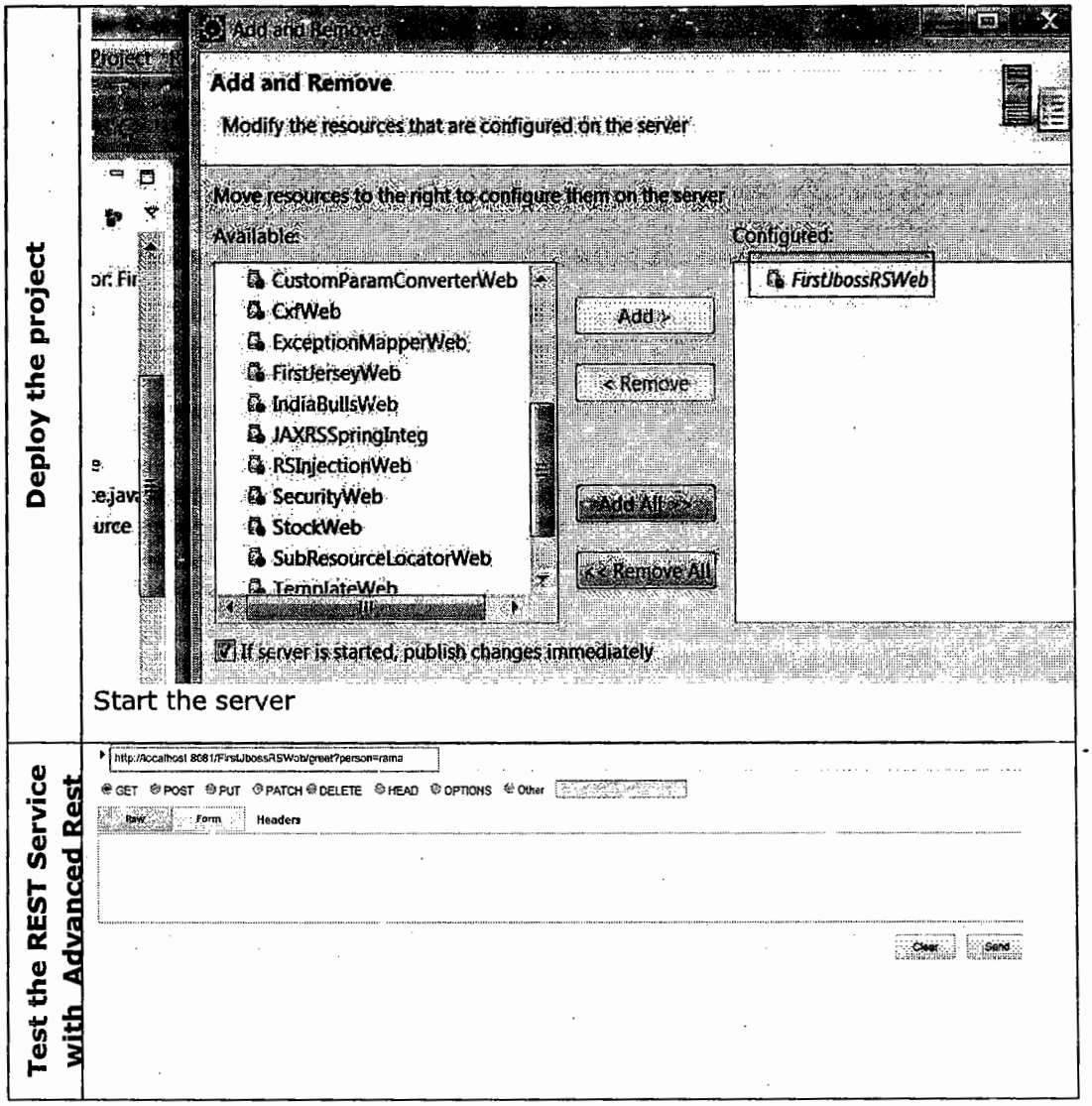
Annotate the class with JAX-RS annotation to make this as Resource class.

Configure web.xml

web.xml

```
    <servlet>
        <servlet-name>resteasy</servlet-name>
        <servlet-class>org.jboss.resteasy.plugins.server.servl
        <init-param>
            <param-name>resteasy.scan</param-name>
            <param-value>true</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>resteasy</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
```

Configure HttpServletDispatcher with url-pattern as /*



20.6 Developing First Jersey Application

Developing the Jersey REST resource is same, only place the change will be configuration in web.xml and we need to add Jerysey specific jar's to the WEB-INF\lib directory of the project. For Jersey we need to configure Jersey specific Servlet configuration as shown below.

```
<?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"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
 id="WebApp_ID" version="3.0">
 <display-name>FirstJerseyWeb</display-name>
 <servlet>
        <servlet-name>jersey</servlet-name>
        <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-
class>
        <init-param>
                <param-name>jersey.config.server.provider.packages</param-
name>
                <param-value>com.fj.resource</param-value>
        </init-param>
        <init-param>
                <param-
name>jersey.config.server.provider.scanning.recursive</param-name>
                <param-value>true</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet-mapping>
        <servlet-name>jersey</servlet-name>
        <url-pattern>/rest/*</url-pattern>
 </servlet-mapping>
</web-app>
```

web.xml

20.7 Bootstrap options

20.7.1 RestEasy

If you observe in case of RestEasy we wrote the url-pattern as /*, but /* is not recommended url-pattern. To customize this we need to write one more init-parameter as shown below

```
<servlet>
	<servlet-name>resteasy</servlet-name>
	<servlet-
class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-
class>
	<init-param>
		<param-name>resteasy.scan</param-name>
		<param-value>true</param-value>
	</init-param>
	<init-param>
		<param-name>resteasy.servlet.mapping.prefix</param-name>
		<param-value>/rest</param-value>
	</init-param>
	<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
	<servlet-name>resteasy</servlet-name>
	<url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

20.7.2 Jersey

In case of jersey, we configured one init-param, instead of "jersey.config.server.provider.packages" we can give classes also as input as shown below.

```
<servlet>
	<servlet-name>jersey</servlet-name>
	<servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-
class>
	<init-param>
		<param-name>jersey.config.server.provider.classnames</param-
name>
		<param-value>com.fj.resource.PersonResource</param-value>
	</init-param>
	<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
	<servlet-name>jersey</servlet-name>
	<url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

20.7.3 Common bootstrap option

Instead of using vendor specific bootstrap options there is another way of bootstapping. We can write a class extending from javax.ws.rs.core.Application class. We need to override methods either getSingletons or getClasses and should return Resource Objects or Class references resp.

If we return Resource object then the scope of the resource will become singleton. Only one object of that resource will be used to process any number of requests. If we return class reference in getClasses() method then for every request new object of the resource will be created.

To make this class deployable we need to annotate this class with @ApplicationPath annotation as shown below.

```
package com.fj.app;

import java.util.HashSet;
import java.util.Set;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

import com.fj.resource.PersonResource;

@ApplicationPath("/rest")
public class FirstApplication extends Application {
        private Set<Object> singletons;

        public FirstJerseyApplication() {
                singletons = new HashSet<Object>();
                singletons.add(new PersonResource());
        }

        @Override
        public Set<Object> getSingletons() {
                return singletons;
        }

}
```

20.8 Working with Http Method

Below example depicts how to work with various http request methods in Jax-Rs

```
package com.bhm.dto;
import java.io.Serializable;
public class ParcelInfo implements Serializable {
  private int awbNo;
  private String src;
  private String dest;
  private String descr;
  private String status;

// setters and getters

}
```

```
package com.bhm.resource;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.atomic.AtomicInteger;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.StreamingOutput;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

import com.bhm.dto.ParcelInfo;

@Path("/bluedart")
public class BluedartCourierResource {
	private Map<Integer, ParcelInfo> parcelDataMap;
	private AtomicInteger index;

	public BluedartCourierResource() {
		parcelDataMap = new HashMap<Integer, ParcelInfo>();
		index = new AtomicInteger(0);
	}

	@POST
	@Consumes(MediaType.APPLICATION_XML)
	@Produces(MediaType.TEXT_PLAIN)
	public int createParcel(InputStream is) {
		ParcelInfo parcelInfo = null;
		int awbNo = 0;
```

```
		try {
			parcelInfo = buildParcelInfo(is);
			awbNo = index.incrementAndGet();
			parcelInfo.setAwbNo(awbNo);
			parcelInfo.setStatus("New");
			parcelDataMap.put(awbNo, parcelInfo);

		} catch (Exception e) {
			e.printStackTrace();
		}
		return awbNo;
	}

	@GET
	@Produces(MediaType.APPLICATION_XML)
	public StreamingOutput getParcelInfo(@QueryParam("awbNo") int awbNo)
{
		ParcelInfo parcelInfo = null;
		String xml = null;

		parcelInfo = parcelDataMap.get(awbNo);
		if (parcelInfo != null) {
			xml = convert(parcelInfo);
		}
		return new ParcelInfoWriter(xml);
	}

	private final class ParcelInfoWriter implements StreamingOutput {
		private String xml;

		public ParcelInfoWriter(String xml) {
			this.xml = xml;
		}

		@Override
		public void write(OutputStream out) throws IOException,
				WebApplicationException {
			PrintWriter writer = null;

			writer = new PrintWriter(out);
			writer.print(xml);
			writer.close();
		}

	}
```

```
        private ParcelInfo buildParcelInfo(InputStream is)
                        throws ParserConfigurationException, SAXException,
    IOException {
                DocumentBuilderFactory factory = null;
                DocumentBuilder builder = null;
                ParcelInfo parcelInfo = null;
                Document doc = null;
                Node root = null;
                NodeList children = null;
                Node child = null;

                factory = DocumentBuilderFactory.newInstance();
                builder = factory.newDocumentBuilder();
                doc = builder.parse(is);

                if (doc != null) {
                        parcelInfo = new ParcelInfo();
                        root = doc.getFirstChild();
                        children = root.getChildNodes();

                        for (int i = 0; i < children.getLength(); i++) {
                                child = children.item(i);
                                if (child.getNodeName().equals("awbNo")) {
                                        parcelInfo

        .setAwbNo(Integer.parseInt(child.getTextContent()));
                                } else if (child.getNodeName().equals("src")) {
                                        parcelInfo.setSrc(child.getTextContent());
                                } else if (child.getNodeName().equals("dest")) {
                                        parcelInfo.setDest(child.getTextContent());
                                } else if (child.getNodeName().equals("descr")) {
                                        parcelInfo.setDescr(child.getTextContent());
                                } else if (child.getNodeName().equals("status")) {
                                        parcelInfo.setStatus(child.getTextContent());
                                }
                        }
                }
                return parcelInfo;
        }
```

Now you can register the resource either using Servlet approach or using Application class to test it.

```
private String convert(ParcelInfo parcelInfo) {
        StringBuffer buffer = null;

        buffer = new StringBuffer();
        buffer.append("<?xml version='1.0' encoding='utf-8'?>")
                .append("<parcelInfo>").append("<awbNo>")

    .append(parcelInfo.getAwbNo()).append("</awbNo>");

    buffer.append("<src>").append(parcelInfo.getSrc()).append("</src>");

    buffer.append("<dest>").append(parcelInfo.getDest()).append("</dest>"
);
        buffer.append("<descr>").append(parcelInfo.getDescr())
                .append("</descr>");
        buffer.append("<status>").append(parcelInfo.getStatus())
                .append("</status>").append("</parcelInfo>");

        return buffer.toString();
    }

}
```

20.9 JAX-RS Injection

20.9.1 Path Parameters

```
package com.rsi.resource;

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

@Path("/passport/{passportNo}")
public class PassportResource {
 /** You can use @PathParam at the attribute level **/
 /* @PathParam("passportNo") */
 /* private String passportNo; */

 /*
  * You can use @PathParam at constructor level
  * public PassportResource(@PathParam("passportNo") String passportNo) {
  * this.passportNo = passportNo; }
  */

 /** You can use at any method level **/
 @GET
 @Produces(MediaType.TEXT_PLAIN)
 @Path("/getPassportNo/{firstname}-{lastname}")
 public String getPassportNo(@PathParam("firstname") String firstname,
                @PathParam("lastname") String lastname) {
        return lastname + " - " + firstname;
 }

 /** You can programmatically access it using PathSegment interface
**/
 @GET
 @Produces(MediaType.TEXT_PLAIN)
 public String getStatus(
                @PathParam("passportNo") PathSegment passportNoPathSeg) {
        return "Passport No : " + passportNoPathSeg.getPath()
                        + " status : approved";
 }
}
```

20.9.2 Matrix Parameters

Matrix parameters are defined at a specific path of the Uri. There scope will be only to the path parameter to which you defined. These parameters generally appear at any place in the Uri. Those will not participate in resolving a request to a resource. These parameters are optional.

```
package com.rsi.matrix.resource;
import javax.ws.rs.DefaultValue;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.core.PathSegment;

@Path("/car")
public class CarResource {

    /** Programatically accessing Matrix parameter **/
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    @Path("{make}/{model}")
    public String search(@PathParam("make") PathSegment makeSeg,
            @PathParam("model") PathSegment modelSeg) {
        StringBuffer buffer = null;
        MultivaluedMap<String, String> matrixParamMap = null;

        buffer = new StringBuffer();
        buffer.append("make : ").append(makeSeg.getPath())
                .append("matrix parameters :");

        matrixParamMap = makeSeg.getMatrixParameters();
        for (String key : matrixParamMap.keySet()) {
            buffer.append(key);
            for (String val : matrixParamMap.get(key)) {
                buffer.append("," + val);
            }

        }

        buffer.append("model : ").append(modelSeg.getPath())
                .append("matrix parameters : ");

        matrixParamMap = modelSeg.getMatrixParameters();
        for (String key : matrixParamMap.keySet()) {
            buffer.append(key);
            for (String val : matrixParamMap.get(key)) {
                buffer.append("," + val);
            }
```

```
            }

            return buffer.toString();
        }

        /** Using annotation to retrieve matrix parameters **/
        @GET
        @Produces(MediaType.TEXT_PLAIN)
        @Path("/price/{make}-{model}")
        public float getCarPrice(@PathParam("make") String make,
                    @PathParam("model") String model,
                    @MatrixParam("year") int year) {
            System.out.println("model : " + model + " make : " + make + "
year : "
                        + year);
            return year + 10000;
        }

}
```

20.9.3 Header Parameter

```
package com.rsi.headerparam.resource;

import java.io.BufferedInputStream;
import java.io.IOException;
import java.io.InputStream;

import javax.ws.rs.Consumes;
import javax.ws.rs.CookieParam;
import javax.ws.rs.GET;
import javax.ws.rs.HeaderParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;

@Path("/courier")
public class CourierResource {


        @GET
        @Produces(MediaType.TEXT_PLAIN)
        public String getAgentName(@HeaderParam("agentId") String agentId) {
            return "Agent id : " + agentId + " name : apparao";
        }
```

```
	@POST
	@Consumes(MediaType.TEXT_PLAIN)
	@Produces(MediaType.TEXT_PLAIN)
	public String send(InputStream is,
			@HeaderParam("courier-agent-id") String courierAgentId) {
		StringBuffer buffer = null;
		int c = 0;

		buffer = new StringBuffer();
		try (BufferedInputStream bis = new BufferedInputStream(is)) {
			while ((c = bis.read()) != -1) {
				buffer.append((char) c);
			}
		} catch (IOException e) {
			e.printStackTrace();
		}
		if (courierAgentId != null && !courierAgentId.equals("")) {
			return "Agent : " + courierAgentId
					+ " request accepted... with data : " +
buffer.toString();
		}
		return "Awb no: 1313";
	}
}
```

20.9.4 Cookie Parameter

```
package com.rsi.cookieparam.resource;

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

@Path("/courier")
public class CourierResource {


	@GET
	@Path("/status")
	@Produces(MediaType.TEXT_PLAIN)
	public String getCourierStatus(@CookieParam("agentId") String agentId,
			@QueryParam("trackingId") String trackingId) {
		return "Agent : " + agentId + " trackingId : " + trackingId
				+ " is in progress...";
	}
}
```

20.9.5 Form Parameter

```
package com.rsi.formparam.resource;

import javax.ws.rs.FormParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/emp")
public class EmpResource {

	@POST
	@Produces(MediaType.TEXT_PLAIN)
	public String insert(@FormParam("id") int id, @FormParam("name")
String name) {
		return "Inserted id : " + id + " name : " + name + "
successfully...";
	}
}
```

. **Insert.jsp** (to test the above resource)

```
<%@ 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>Insert Emp</title>
	</head>
	<body>
		<form action="${pageContext.request.contextPath}/rest/emp"
method="POST">
		empid:<input type="text" name="id"/><br/>
		name:<input type="text" name="name"/><br/>
		<input type="submit" value="insert"/>
		</form>
	</body>
</html>
```

20.9.6 Accessing Programatic Headers and Cookies

```
package com.rsi.progheaders.resource;

import java.util.List;
import java.util.Map;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.Cookie;
import javax.ws.rs.core.HttpHeaders;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;

@Path("/header")
public class HeaderResource {

	@GET
	@Produces(MediaType.TEXT_PLAIN)
	public String getAllHeaders(@Context HttpHeaders headers) {
		StringBuffer buffer = null;
		MultivaluedMap<String, String> headerMap = null;
		Map<String, Cookie> cookieMap = null;

		buffer = new StringBuffer();
		headerMap = headers.getRequestHeaders();
		for (String paramName : headerMap.keySet()) {
			List<String> paramValues = headerMap.get(paramName);
			buffer.append("paramName : ").append(paramName);
			for (String paramValue : paramValues) {
				buffer.append(",").append(paramValue);
			}
			buffer.append(";");
		}
		buffer.append("********Cokiees********");
		cookieMap = headers.getCookies();
		for (String key : cookieMap.keySet()) {
			Cookie c = cookieMap.get(key);
			buffer.append("Name : ").append(c.getName()).append(" Value : ")
					.append(c.getValue());
		}

		return buffer.toString();
	}
}
```

20.9.7 Working with UriInfo

```
package com.rsi.uri.resource;

import java.util.List;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.core.UriInfo;

@Path("/uri")
public class URIResource {
	@GET
	@Produces(MediaType.TEXT_PLAIN)
	public String getUriInformation(@Context UriInfo uri) {
		StringBuffer buffer = null;
		MultivaluedMap<String, String> queryMap = null;

		buffer = new StringBuffer();
		queryMap = uri.getQueryParameters();

		for (String paramName : queryMap.keySet()) {
			buffer.append("param : " + paramName);
			List<String> values = queryMap.get(paramName);
			for (String value : values) {
				buffer.append(",").append(value);
			}
			buffer.append(";");
		}
		return buffer.toString();
	}
}
```

20.9.8 Automatic Parameter conversion

In most of the cases when we inject values into the attributes or parameters those need not be strings, you can take any java primitive types as parameters or attributes into which jax-rs runtime can perform the injection. There are some predefined supported types into which jax-rs can perform injection

1) Any java primitive types can be taken as candidates for resource injection
2) It can be any class type as well, but the class should have the following
   a. One single string argument constructor
   b. A valueOf method which takes Object type and convert to string
3) It can be a List or Set or SortedSet as a parameter where whose generic types shoud fall under the above two categories.

```
package com.rsi.autoparamconv.resource;

import java.util.List;

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

@Path("/address")
public class AddressResource {

	@Path("/zip/{zip}")
	@Produces(MediaType.TEXT_PLAIN)
	@GET
	public String getAreaCode(@PathParam("zip") int zip) {
		return "zip : " + zip;
	}

	@GET
	@Path("/area/{areaCode}")
	@Produces(MediaType.TEXT_PLAIN)
	public String findZipCodes(@PathParam("areaCode") SearchCriteria criteria) {
		return "Zip for areaCode : " + criteria.getAreaCode();
	}

	@GET
	@Path("/blacklist")
	@Produces(MediaType.TEXT_PLAIN)
	public String addToBlackList(@QueryParam("zip") List<Integer> zipCodes) {
		StringBuffer buffer = null;

		buffer = new StringBuffer();
		for (int z : zipCodes) {
			buffer.append(",").append(z);
		}
		return buffer.toString();
	}

}
```

```
package com.rsi.autoparamconv.resource;

public class SearchCriteria {
	private String areaCode;

	public SearchCriteria(String areaCode) {
		this.areaCode = areaCode;
	}

	public String getAreaCode() {
		return areaCode;
	}

	public void setAreaCode(String areaCode) {
		this.areaCode = areaCode;
	}

	public String valueOf(SearchCriteria criteria) {
		return String.valueOf(areaCode);
	}
}
```

20.9.9 Bean Parameter

```
package com.rsi.beanparam.resource;

import javax.ws.rs.HeaderParam;
import javax.ws.rs.QueryParam;

public class Criteria {
	@QueryParam("itemCode")
	private String itemCode;
	@HeaderParam("category")
	private String category;
	@QueryParam("manufacturer")
	private String manufacturer;
	@QueryParam("price")
	private float price;

	// setters and getters
}
```

```
package com.rsi.beanparam.resource;

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

@Path("/store")
public class StoreResource {

	@GET
	@Produces(MediaType.TEXT_PLAIN)
	public String search(@BeanParam Criteria criteria) {
		return "itemCode : " + criteria.getItemCode() + ", category : "
				+ criteria.getCategory() + ", manufacturer : "
				+ criteria.getManufacturer() + ", price : "
				+ criteria.getPrice();
	}
}
```

20.9.10 Custom Parameter Converter

Sometimes we may have parameters to our methods which will not fall under the above said rules. In such case we can write our own parameter converters, that takes care of converting the values into respective types as shown below.

```
package com.cpc.dto;

public class OrderInfo {
	private int size;

	public int getSize() {
		return size;
	}

	public void setSize(int size) {
		this.size = size;
	}

}
```

```
package com.cpc.resource;

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

import com.cpc.dto.OrderInfo;

@Path("/pizza")
public class PizzaStore {

	@GET
	@Produces(MediaType.TEXT_PLAIN)
	public String placeOrder(
			@QueryParam("size") @DefaultValue("SMALL") OrderInfo
orderInfo) {
		return "You order for " + orderInfo.getSize()
				+ " inch pizza is under processing...";
	}
}
```

```
package com.cpc.converters;

import javax.ws.rs.ext.ParamConverter;

import com.cpc.dto.OrderInfo;

public class OrderInfoParamConverter implements ParamConverter<OrderInfo> {

	@Override
	public OrderInfo fromString(String val) {
		OrderInfo orderInfo = null;

		if (val != null && !val.equals("")) {
			orderInfo = new OrderInfo();
			if (val.toUpperCase().equals("SMALL")) {
				orderInfo.setSize(8);
			} else if (val.toUpperCase().equals("MEDIUM")) {
				orderInfo.setSize(12);
			} else if (val.toUpperCase().equals("LARGE")) {
				orderInfo.setSize(16);
			}
		}
		return orderInfo;
	}
```

```
	@Override
	public String toString(OrderInfo orderInfo) {
		String val = null;

		if (orderInfo != null) {
			if (orderInfo.getSize() == 8) {
				val = "SMALL";
			} else if (orderInfo.getSize() == 12) {
				val = "MEDIUM";
			} else if (orderInfo.getSize() == 16) {
				val = "LARGE";
			}
		}
		return val;
	}

}
```

```
package com.cpc.converters;

import java.lang.annotation.Annotation;
import java.lang.reflect.Type;

import javax.ws.rs.ext.ParamConverter;
import javax.ws.rs.ext.ParamConverterProvider;
import javax.ws.rs.ext.Provider;

import com.cpc.dto.OrderInfo;

@Provider
public class PizzaStoreParamConverterProvider implements ParamConverterProvider {

	@Override
	public <T> ParamConverter<T> getConverter(Class<T> genericType,
			Type rawType, Annotation[] annotations) {
		System.out.println("Generic Type : " + genericType.getName());
		System.out.println("rawType : " + rawType.getClass().getName());
		System.out.println("annotations : ");
		for (Annotation annon : annotations) {
			System.out.println(annon.toString());
		}
		if (genericType.isAssignableFrom(OrderInfo.class)) {
			return (ParamConverter<T>) new OrderInfoParamConverter();
		}
		return null;
	}

}
```

20.10 Sub-Resource locator

20.10.1 Static Dispatch

```
package com.srl.resource;

import javax.inject.Inject;
import javax.ws.rs.Path;

@Path("/courier")
public class CourierResource {

	/**
	 * Sub-Resource locator method
	 * @return
	 */
	@Path("/customerservice")
	// sub-resource method
	public CustomerServiceResource track() {
		return new CustomerServiceResource();
	}

}
```

```
package com.srl.resource;

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

public class CustomerServiceResource {

	@GET
	@Produces(MediaType.TEXT_PLAIN)
	public String getCourierStatus(@QueryParam("awbNo") String awbNo) {
		return "Awb no : " + awbNo + " is ready for delivery";
	}
}
```

20.10.1 Dynamic Dispatch

```
package com.srl.resource;

import javax.ws.rs.Path;
import javax.ws.rs.PathParam;

@Path("/payment")
public class PaymentResource {

	/**
	 * Dynamic Dispatching
	 * @param type
	 * @return
	 */
	@Path("/{type}/{gateWay}")
	public Object process(@PathParam("type") String type) {
		Object subResource = null;

		if (type.equals("creditcard")) {
			subResource = new CreditCardPaymentResource();
		} else if (type.equals("debitcard")) {
			subResource = new DebitCardPaymentResource();
		}
		return subResource;
	}
}
```

```
package com.srl.resource;

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

public class CreditCardPaymentResource {

	@GET
	@Produces(MediaType.TEXT_PLAIN)
	public String processPayment(@QueryParam("cardNo") String cardNo,
			@PathParam("gateWay") String gateWay) {
		return "Processed payment with cardNo : " + cardNo + " gateWay "
				+ gateWay;
	}
}
```

```
package com.srl.resource;

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

public class DebitCardPaymentResource {

	@GET
	@Produces(MediaType.TEXT_PLAIN)
	public String processDebit(@QueryParam("cardNo") String cardNo,
			@QueryParam("pin") String pin, @PathParam("gateWay") String
gateWay) {
		return "Processed payment with cardNo : " + cardNo + " pin : " + pin
				+ " gateWay : " + gateWay;
	}
}
```

20.11 Content Handlers

Always a Resource will not get the data through Query, Path, Matrix, Header or cookie parameters. Even a resource as part of Http request it may get the data as part of request body. How many ways we can read the data that is sent as part of request body or how many ways we can dispatch data as part of response body will be shown here.

```
package com.ch.resource;

import java.io.BufferedInputStream;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.Reader;

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

@Path("/aadhar")
public class AadharRegistration {

	@POST
	@Consumes(MediaType.TEXT_PLAIN)
```

```
@Produces(MediaType.TEXT_PLAIN)
@Path("/inputstream")
public String register(InputStream inputstream) throws IOException {
	int c = 0;
	StringBuffer buffer = null;
	BufferedInputStream bis = null;

	buffer = new StringBuffer();
	bis = new BufferedInputStream(inputstream);
	try {
		while ((c = bis.read()) != -1) {
			buffer.append((char) c);
		}
	} catch (IOException e) {
		e.printStackTrace();
	} finally {
		bis.close();
		inputstream.close();
	}
	return buffer.toString();
}

@POST
@Consumes(MediaType.TEXT_PLAIN)
@Produces(MediaType.TEXT_PLAIN)
@Path("/reader")
public String register(Reader reader) throws IOException {
	String line = null;
	BufferedReader br = null;
	StringBuffer buffer = null;

	buffer = new StringBuffer();
	br = new BufferedReader(reader);
	try {
		while ((line = br.readLine()) != null) {
			buffer.append(line);
		}
	} catch (IOException e) {
		e.printStackTrace();
	} finally {
		br.close();
		reader.close();
	}
	return buffer.toString();
}

@Path("/byte")
@POST
@Produces(MediaType.TEXT_PLAIN)
@Consumes(MediaType.TEXT_PLAIN)
public String register(byte[] body) {
	StringBuffer buffer = null;
```

```
		buffer = new StringBuffer();
		if (body != null && body.length > 0) {
			for (byte b : body) {
				buffer.append((char) b);
			}
		}
		return buffer.toString();
	}
	@GET
	@Produces(MediaType.APPLICATION_XML)
	@Path("/phrase")
	public byte[] getDailyPhrase() {
		String responseBody = null;

		responseBody = "<phrase><msg>Drive safely!</msg></phrase>";
		return responseBody.getBytes();
	}
	@POST
	@Consumes(MediaType.MULTIPART_FORM_DATA)
	@Produces(MediaType.TEXT_PLAIN)
	@Path("/upload")
	public String uploadAadharData(File file) throws IOException {
		int c = 0;
		StringBuffer buffer = null;
		FileInputStream fis = null;

		buffer = new StringBuffer();
		if (file != null) {
			fis = new FileInputStream(file);
			while ((c = fis.read()) != -1) {
				buffer.append((char) c);
			}
		}
		return buffer.toString();
	}
	@POST
	@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
	@Produces(MediaType.TEXT_PLAIN)
	@Path("/form")
	public String register(MultivaluedMap<String, String> formData) {
		StringBuffer buffer = null;

		buffer = new StringBuffer();
		for (String key : formData.keySet()) {
			buffer.append("key : ").append(key).append(" value : ")
					.append(formData.getFirst(key));
		}
		return buffer.toString();
	}

}
```

To send the file as input to the above resource we need to use the below jsp
Uploadfile.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>Upload file</title>
</head>
<body>
    <form action="${pageContext.request.contextPath}/rest/aadhar/upload"
        method="post" enctype="multipart/form-data">
        File:<input type="file" name="aadhardata" /> <input type="submit"
            value="upload" />
    </form>
</body>
</html>
```

20.12 Custom Content Handlers

When client send a data as part of request body, we can collect the information using any of the predefined data types like String, byte [], File, InputStream, Reader etc. But in all these ways we can access the raw data only, but we cannot build the business logic around raw data. If data format changes again we need to modify the business logic. Instead we need to convert the data into Object so that it will be easy to build the logic around object so that our business logic will be data format agnostic. But we need to write the logic for converting raw data into object in each and every Resource of the application.

The problem is not only the code duplication across all the Resources, even the Resources will be coupled with data format. Instead we can use Custom content handler which will convert the raw data into a specific data format.

So, we need to write One content handler which converts input data into Object (MessageBodyReader) and we need to write one more content handler which converts Object back to raw data (MessageBodyWriter).

Here is the example depicts the same.

```
package com.csvch.common;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface CSVType {

}
```

```
package com.csvch.dto;

import com.csvch.common.CSVType;

@CSVType
public class Fare {
        private String billNo;
        private String distance;
        //setter & getters
}
```

```
package com.csvch.dto;

import com.csvch.common.CSVType;

@CSVType
public class TripInfo {
        private String src;
        private String dest;
        // setters & getters
}
```

```
package com.csvch.readers;

import java.io.BufferedInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Type;
import java.util.HashMap;
import java.util.Map;
import java.util.StringTokenizer;

import javax.ws.rs.Consumes;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.ext.MessageBodyReader;
import javax.ws.rs.ext.Provider;

import com.csvch.common.CSVType;

@Provider
@Consumes(MediaType.TEXT_PLAIN)
public class CSVMessageBodyReader implements MessageBodyReader<Object> {
	private final static String DELIMETER = ",";

	@Override
	public boolean isReadable(Class<?> genericType, Type rawType,
			Annotation[] annotations, MediaType mediaType) {
		if (genericType.isAnnotationPresent(CSVType.class)) {
			return true;
		}
		return false;
	}

	@Override
	public Object readFrom(Class<Object> genericType, Type rawType,
			Annotation[] annotations, MediaType mediaType,
			MultivaluedMap<String, String> requestHeaderMap,
			InputStream inputStream) throws IOException,
			WebApplicationException {
		Object obj = null;
		String data = null;
		Map<String, String> dataMap = null;
		try {
			data = extractData(inputStream);
			dataMap = convertToMap(data);
			obj = bind(genericType, dataMap);
		} catch (Exception e) {
			e.printStackTrace();
```

```
                throw new WebApplicationException(e);
            }

            return obj;
        }

        private String extractData(InputStream inputStream) throws IOException {
            int c = 0;
            StringBuffer buffer = null;
            BufferedInputStream bis = null;

            try {
                buffer = new StringBuffer();
                bis = new BufferedInputStream(inputStream);
                while ((c = bis.read()) != -1) {
                    buffer.append((char) c);
                }
            } finally {
                bis.close();
                inputStream.close();
            }
            return buffer.toString();
        }

        private Map<String, String> convertToMap(String data) {
            Map<String, String> dataMap = null;
            String token = null;
            String[] splits = null;
            StringTokenizer tokenizer = null;

            tokenizer = new StringTokenizer(data, DELIMETER);
            dataMap = new HashMap<String, String>();
            while (tokenizer.hasMoreTokens()) {
                token = tokenizer.nextToken();
                splits = token.split("=");
                dataMap.put(splits[0], splits[1]);
            }
            return dataMap;
        }

        private Object bind(Class<Object> classType, Map<String, String> dataMap)
                throws InstantiationException, IllegalAccessException,
                NoSuchFieldException, SecurityException,
IllegalArgumentException,
                InvocationTargetException {
            Field field = null;
            Object obj = null;
            String attrVal = null;

            obj = classType.newInstance();
            for (String attrNm : dataMap.keySet()) {
                attrVal = dataMap.get(attrNm);
```

```
                setValue(obj, classType, attrNm, attrVal);
            }

            return obj;
        }

        private void setValue(Object obj, Class<Object> classType, String attrNm,
                String attrVal) throws IllegalAccessException,
                IllegalArgumentException, InvocationTargetException {
            Method[] methods = null;

            methods = classType.getDeclaredMethods();
            for (Method method : methods) {
                if (method.getName().equalsIgnoreCase("set" + attrNm)) {
                    method.invoke(obj, attrVal);
                    break;
                }
            }
        }
}
```

```
package com.csvch.resource;

import java.util.UUID;

import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import com.csvch.dto.Fare;
import com.csvch.dto.TripInfo;

@Path("/taxi")
public class TaxiService {

        @POST
        @Produces(MediaType.TEXT_PLAIN)
        @Consumes(MediaType.TEXT_PLAIN)
        public Fare hire(TripInfo tripInfo) {
            Fare fare = null;

            fare = new Fare();
            fare.setBillNo(tripInfo.getSrc() + "-" + tripInfo.getDest()
                    + UUID.randomUUID().toString());
            fare.setDistance("10 KM");

            return fare;
        }
}
```

```
package com.csvch.writers;

import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.lang.annotation.Annotation;
import java.lang.reflect.Field;
import java.lang.reflect.Type;

import javax.ws.rs.Produces;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.ext.MessageBodyWriter;
import javax.ws.rs.ext.Provider;

import com.csvch.common.CSVType;

@Provider
@Produces(MediaType.TEXT_PLAIN)
public class CSVMessageBodyWriter implements MessageBodyWriter<Object> {

	@Override
	public long getSize(Object object, Class<?> genericType, Type rawType,
			Annotation[] annotations, MediaType mediaType) {
		return -1;
	}

	@Override
	public boolean isWriteable(Class<?> genericType, Type rawType,
			Annotation[] annotations, MediaType mediaType) {
		if (genericType.isAnnotationPresent(CSVType.class)) {
			return true;
		}
		return false;
	}

	@Override
	public void writeTo(Object object, Class<?> genericType, Type rawType,
			Annotation[] annotations, MediaType mediaType,
			MultivaluedMap<String, Object> responseHeaderMap,
			OutputStream outputstream) throws IOException,
			WebApplicationException {
		PrintWriter writer = null;
		String csv = null;

		try {
			csv = convertToCSV(genericType, object);
			writer = new PrintWriter(outputstream);
			writer.print(csv);
		} catch (Exception e) {
```

```
                e.printStackTrace();
                throw new WebApplicationException(e);
            } finally {
                writer.close();
            }

        }

        private String convertToCSV(Class<?> classType, Object object)
                throws IllegalArgumentException, IllegalAccessException {
            StringBuffer buffer = null;
            Field[] fields = null;
            boolean isFirst = true;

            buffer = new StringBuffer();
            fields = classType.getDeclaredFields();
            for (Field f : fields) {
                f.setAccessible(true);
                if (isFirst) {

        buffer.append(f.getName()).append("=").append(f.get(object));
                } else {
                    buffer.append(",").append(f.getName()).append("=")
                            .append(f.get(object));
                }
            }
            return buffer.toString();
        }

}
```

20.13 Complex Responses

As part of dispatching the response back to the client, we may not only send data rather we may need to send response headers or cookies or status code back to the client. If we take return type as acceptable types we only end up return data with default status codes. But we want to customize the response. In such case we have to use Jax-rs api provided to return responses as shown below.

```
package com.cr.dto;

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

@XmlType
@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "cardUpgradeInfo")
public class CardUpgradeInfo {
        private String cardNo;
        private int month;
        private int year;
        private String name;

        // setters & getters
}
```

```
package com.cr.dto;

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

@XmlType
@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "personelInfo")
public class PersonelInfo {
        private String name;
        private int age;
        private int ssn;

        // setters & getters
}
```

```
package com.cr.dto;

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

@XmlType
@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "reqStatus")
public class ServiceRequestStatus {
	private String cardNo;
	private String msg;
	private String status;

	// setters & getters
}
```

```
package com.cr.resource;

import java.net.URI;
import java.net.URISyntaxException;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.NewCookie;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.ResponseBuilder;
import javax.ws.rs.core.Response.Status;

import com.cr.dto.CardUpgradeInfo;
import com.cr.dto.PersonelInfo;
import com.cr.dto.ServiceRequestStatus;

@Path("/bank")
public class BankResource {

	@GET
	@Path("/bal/{accno}")
	@Produces(MediaType.TEXT_PLAIN)
	public Response getBalance(@PathParam("accno") String accno) {
		Response response = null;
		ResponseBuilder builder = null;

		builder = Response.ok("Balance for accno : " + accno + " is 0.0");
		response = builder.build();
```

```
		return response;
	}

	@POST
	@Path("/checkbookreq/{accno}")
	public Response requestCheckbook(@PathParam("accno") String accno) {
		Response response = null;
		ResponseBuilder builder = null;

		System.out.println("accno : " + accno);
		builder = Response.status(Status.NO_CONTENT);
		response = builder.build();
		return response;
	}

	@PUT
	@Consumes(MediaType.APPLICATION_XML)
	@Produces(MediaType.APPLICATION_XML)
	public Response upgradeCard(CardUpgradeInfo cardUpgradeInfo) {
		Response response = null;
		ResponseBuilder builder = null;
		ServiceRequestStatus status = null;

		status = new ServiceRequestStatus();
		status.setCardNo(cardUpgradeInfo.getCardNo());
		status.setMsg("Upgrade request accepted");
		status.setStatus("Initiated");

		builder = Response.ok();
		builder = builder.header("serviceRequestNumber", "SR31414");
		builder = builder.entity(status);
		response = builder.build();
		return response;
	}

	@POST
	@Path("/apply")
	@Consumes(MediaType.APPLICATION_XML)
	public Response applyCreditCard(PersonelInfo pInfo)
			throws URISyntaxException {
		NewCookie cookie = null;

		cookie = new NewCookie("interaction-id", "2424");
		return Response.created(new URI("/rest/bank/applyCredit_Step2"))
				.cookie(cookie).build();
	}

}
```

20.14 Exception Handling

When we send a request to the resource, the resource always may not endup in performing the operation successfully. Sometimes it may run into problems which lead to exception. In case if the resource throws the exception, Jax-rs runtime will convert that exception into an response message with status code as error response code.

But in case if we want to return a custom error response rather than a default error response message we need to handle the exception and should return as a response shown below.

```
package com.em.dto;

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

@XmlType
@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "customer")
public class Customer {
	private int id;
	private String name;

	// setters & getters
}
```

```
package com.em.exception;

public class CustomerNotFoundException extends Exception {
	public CustomerNotFoundException() {
		super();
	}
	public CustomerNotFoundException(String arg0, Throwable arg1) {
		super(arg0, arg1);
	}
	public CustomerNotFoundException(String arg0) {
		super(arg0);
	}
}
```

```
package com.em.exception.mapper;

import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.Status;
import javax.ws.rs.ext.ExceptionMapper;
import javax.ws.rs.ext.Provider;

import com.em.exception.CustomerNotFoundException;

@Provider
public class CustomerNotFoundExceptionMapper implements
        ExceptionMapper<CustomerNotFoundException> {

    @Override
    public Response toResponse(CustomerNotFoundException ex) {
        return Response.status(Status.BAD_REQUEST).entity(ex.getMessage())
                .build();
    }
}
```

```
package com.em.resource;

import java.net.URI;
import java.net.URISyntaxException;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicInteger;

import javax.annotation.PostConstruct;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.Status;

import com.em.dto.Customer;
import com.em.exception.CustomerNotFoundException;

@Path("/customer")
public class CustomerResource {
    private Map<Integer, Customer> dbMap;
    private AtomicInteger customerindex;

    public CustomerResource() {
        dbMap = new ConcurrentHashMap<Integer, Customer>();
        customerindex = new AtomicInteger(0);
```

```
	}

	@POST
	@Consumes(MediaType.APPLICATION_XML)
	public Response createCustomer(Customer customer) throws
URISyntaxException {
		String baseUri = null;
		int id = 0;

		baseUri = "/customer/";
		id = customerindex.incrementAndGet();
		baseUri += id;
		customer.setId(id);
		dbMap.put(id, customer);
		return Response.created(new URI(baseUri)).build();
	}

	@PUT
	@Consumes(MediaType.APPLICATION_XML)
	@Produces(MediaType.TEXT_PLAIN)
	public Response updateCustomer(Customer customer)
			throws CustomerNotFoundException {

		if (dbMap.containsKey(customer.getId()) == false) {
			throw new CustomerNotFoundException(
					"Customer is not found to update...");
		}
		dbMap.put(customer.getId(), customer);
		return Response.ok("Customer updated").build();
	}

	@GET
	@Path("/{id}")
	@Produces(MediaType.APPLICATION_XML)
	public Response getCustomer(@PathParam("id") int customerId)
			throws CustomerNotFoundException {
		if (dbMap.containsKey(customerId) == false) {
			throw new CustomerNotFoundException("Customer Id is invalid");
		}
		return Response.ok().entity(dbMap.get(customerId)).build();
	}
}
```

20.15 JAX-RS Client

In case of JAX-RS API 1.x it has not provided any classes for accessing the Resource. That is the biggest draw back with 1.x api. In JAX-RS API 2.0 it has provided convienient classes for accessing the Resource on the server.

Examples are here.

```
package com.firstrs.resource.client;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;

public class FirstRSClient {
	public static void main(String[] args) {
		ClientBuilder builder = null;
		WebTarget target = null;
		Client client = null;

		builder = ClientBuilder.newBuilder();
		builder.property("connection.timeout", 1000 * 60);

		client = builder.build();

		target = client.target("http://localhost:8081/FirstJbossRSWeb/greet");
		target = target.queryParam("person", "john");

		Response response = target.request().get();
		if (response.getStatus() == 200) {
			String data = response.readEntity(String.class);
			System.out.println(data);
		}

	}
}
```

```
package com.rsinjection.resource.pathparam.client;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;

public class CarResourceClient {
	private final static String RESOURCE_URI =
"http://localhost:8081/RSInjectionWeb/rest/car";

	public static void main(String[] args) {
		Client client = null;
		WebTarget target = null;
```

```
		client = ClientBuilder.newClient();
		target = client.target(RESOURCE_URI).path("/{make}");
		target = target.resolveTemplate("make", "Maruthi").matrixParam("year",
				"2014");
		target = target.path("/{model}").resolveTemplate("model", "Swift")
				.matrixParam("color", "red");

		System.out.println(target.getUri().getPath());
		String data = target.request().get(String.class);
		System.out.println(data);
	}
}
```

```
package com.rsinjection.resource.pathparam.client;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.NewCookie;
import javax.ws.rs.core.Response;

public class CourierResourceClient {
	private static final String RESOURCE_URI =
"http://localhost:8081/RSInjectionWeb/rest/courier";

	public static void main(String[] args) {
		Client client = null;
		WebTarget target = null;

		try {
			client = ClientBuilder.newClient();
			target = client.target(RESOURCE_URI);

			String data = getCourierStatus(target);
			System.out.println(data);
		} finally {
			client.close();
		}

	}

	public static String getAgentName(WebTarget target) {
		String data = target.request().header("agentId", "007")
				.get(String.class);
		return data;
	}

	public static String sendCourier(WebTarget target) {
		Response response = target.request().header("courier-agent-id", "c1")
				.post(Entity.text("Internal Courier"));
```

```
            String data = response.readEntity(String.class);
            return data;
        }

        public static String getCourierStatus(WebTarget target) {
            NewCookie cookie = null;

            cookie = new NewCookie("agentId", "007");

            return target.path("/status").queryParam("trackingId", "t24242")
                        .request().cookie(cookie).get(String.class);
        }
}
```

```
package com.rsinjection.resource.pathparam.client;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.Status;

public class PassportResourceClient {
        private final static String RESOURCE_URI =
"http://localhost:8081/RSInjectionWeb/rest/passport/{passportNo}";

        public static void main(String[] args) {
            Client client = null;
            WebTarget target = null;

            client = ClientBuilder.newClient();
            client.property("connection.timeout", 1000 * 60);
            target = client.target(RESOURCE_URI);
            target = target.path("/getPassportNo/{firstname}-{lastname}");

            target = target.resolveTemplate("passportNo", "p1");
            target = target.resolveTemplate("firstname", "john");
            target = target.resolveTemplate("lastname", "blake");

            Response response = target.request().get();
            if (response.getStatus() == Status.OK.getStatusCode()) {
                response.bufferEntity();
                String body = response.readEntity(String.class);
                response.readEntity(String.class);
                System.out.println(body);
            }
        }
}
```

```
package com.cch.resource.client;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.Status;

public class TaxiResourceClient {
	private final static String RESOURCE_URI =
"http://localhost:8081/CSVContentHandlerWeb/rest/taxi";

	public static void main(String[] args) {
		Client client = null;
		WebTarget target = null;

		client = ClientBuilder.newClient();
		client.register(CSVMessageBodyReader.class);
		client.register(CSVMessageBodyWriter.class);

		target = client.target(RESOURCE_URI);

		Fare fare = hire(target);
		System.out.println(fare.getBillNo());
	}

	public static Fare hire(WebTarget target) {
		TripInfo trip = null;
		Fare fare = null;

		trip = new TripInfo();
		trip.setSrc("ameerpet");
		trip.setDest("secunderabad");

		Response response = target.request().post(
				Entity.entity(trip, MediaType.TEXT_PLAIN));
		if (response.getStatus() == Status.OK.getStatusCode()) {
			fare = response.readEntity(Fare.class);
			return fare;
		}
		return null;
	}
}
```

20.16 Working with Abstract classes and Interfaces

```
package com.wia.resource;

import javax.ws.rs.GET;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;

abstract public class AbstractOrder {
	@GET
	@Produces("text/plain")
	abstract String getOrderStatus(@QueryParam("orderNo") int orderNo);
}
```

```
package com.wia.resource;

import javax.ws.rs.Path;

@Path("/absorder")
public class AbstractOrderImpl extends AbstractOrder {
	@Override
	String getOrderStatus(int orderNo) {
		return "Status : pending ... for order : " + orderNo;
	}

}
```

```
package com.wia.resource;

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

@Path("/order")
public interface Order {
	@GET
	@Produces(MediaType.TEXT_PLAIN)
	String getOrderStatus(@QueryParam("orderNo") int orderNo);
}
```

```
package com.wia.resource;

public class OrderImpl implements Order {

	@Override
	public String getOrderStatus(int orderNo) {
		return "Order No : " + orderNo + " is in processing...";
	}

}
```

20.17 Async Server and Client

```
package com.aw.resource;
@Path("/stock")
public class StockQuoteResource {
	@GET
	@Produces(MediaType.TEXT_PLAIN)
	public void getQuotePrice(@QueryParam("stocknm") String stockNm,
			@Suspended AsyncResponse asyncResponse) {
		asyncResponse.resume(Response.ok(
				"stocknm : " + stockNm + " price : 2242.234f").build());
	}
}
```

```
<?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"
	xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
	id="WebApp_ID" version="3.0">
	<display-name>AsyncWeb</display-name>
	<servlet>
		<servlet-name>Resteasy</servlet-name>
		<servlet-
class>org.jboss.resteasy.plugins.server.servlet.HttpServlet30Dispatcher</servlet-
class>
		<init-param>
			<param-name>resteasy.scan</param-name>
			<param-value>true</param-value>
		</init-param>
		<init-param>
			<param-name>resteasy.servlet.mapping.prefix</param-name>
			<param-value>/rest</param-value>
		</init-param>
		<load-on-startup>1</load-on-startup>
		<async-supported>true</async-supported>
	</servlet>
	<servlet-mapping>
		<servlet-name>Resteasy</servlet-name>
		<url-pattern>/rest/*</url-pattern>
	</servlet-mapping>
</web-app>
```

```
package com.async.resource.client;

import java.util.concurrent.ExecutionException;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.InvocationCallback;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;

public class StockQuoteClient {
	private final static String RESOURCE_URI =
"http://localhost:8081/AsyncWeb/rest/stock";

	public static void main(String[] args) throws InterruptedException,
			ExecutionException {
		Client client = null;
		WebTarget target = null;

		client = ClientBuilder.newClient();
		target = client.target(RESOURCE_URI);
		target = target.queryParam("stocknm", "icicbank");
		target.request().async().get(new ResponseCallback());
		/*
		 * System.out.println("control released");
		 *
		 * Response response = future.get();
		 * System.out.println("response collected"); if (response.getStatus() ==
		 * 200) { String data = response.readEntity(String.class);
		 * System.out.println(data); }
		 */
		System.out.println("main method ended...");
	}

	private static final class ResponseCallback implements
InvocationCallback<Response> {

		@Override
		public void completed(Response response) {
			String data = response.readEntity(String.class);
			System.out.println(data);
		}

		@Override
		public void failed(Throwable ex) {

		}

	}
}
```

20.18 Caching

```
package com.cw.resource;

import java.util.Calendar;

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

@Path("/pnr")
public class PNRStatusResource {

	@GET
	@Path("/{pnr}")
	@Produces(MediaType.TEXT_PLAIN)
	public Response getPnrStatus(@PathParam("pnr") String pnr) {
		Response response = null;
		Calendar timestamp = null;

		timestamp = Calendar.getInstance();
		timestamp.set(2014, 7, 18, 9, 0, 0);

		response = Response.ok("pnr : " + pnr + " status : WL23")
				.expires(timestamp.getTime()).build();

		return response;
	}

}
```

```
package com.cw.resource;

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.CacheControl;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

@Path("/stock")
public class StockQuoteResource {

	@GET
	@Produces(MediaType.TEXT_PLAIN)
	@Path("/{stocknm}")
	public Response getStockPrice(@PathParam("stocknm") String stocknm) {
		Response response = null;
```

```
            CacheControl cc = null;

            cc = new CacheControl();
            cc.setPrivate(true);
            cc.setMaxAge(20009200);

            System.out.println("Request is processing...");
            response = Response.ok("stock nm : " + stocknm + " price : 2422")
                        .cacheControl(cc).build();

            return response;
        }

}
```

```
package com.cw.dto;

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

@XmlType
@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "stockInfo")
public class StockInfo {
        private String name;
        private double price;
        private String org;

        // setters & getters
        // hascode && equals
}
```

```
package com.cw.resource;

import java.net.URI;
import java.net.URISyntaxException;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.NotFoundException;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.CacheControl;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.EntityTag;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.ResponseBuilder;

import com.cw.dto.StockInfo;

@Path("/stockinfo")
public class StockResource {
	private Map<String, StockInfo> dbMap;

	public StockResource() {
		dbMap = new ConcurrentHashMap<String, StockInfo>();
	}

	@POST
	@Consumes(MediaType.APPLICATION_XML)
	public Response newStock(StockInfo si) throws URISyntaxException {
		dbMap.put(si.getName(), si);
		return Response.created(new URI("/stockinfo/" + si.getName())).build();
	}

	@GET
	@Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
	@Path("/{stocknm}")
	public Response getStock(@PathParam("stocknm") String stocknm,
			@Context Request request) {
		ResponseBuilder builder = null;
		StockInfo si = null;
		CacheControl cc = null;
		EntityTag eTag = null;

		if (dbMap.containsKey(stocknm) == false) {
			throw new NotFoundException("Stock doesn't exists");
```

```
            }

            // sever object
            si = dbMap.get(stocknm);
            eTag = new EntityTag(String.valueOf(si.hashCode()));

            builder = request.evaluatePreconditions(eTag);
            if (builder != null) {// pick the data from cache
                  // send redirect request
                  return builder.build();
            }

            cc = new CacheControl();
            cc.setMustRevalidate(true);
            cc.setMaxAge(1000 * 60 * 60);
            cc.setPrivate(true);

            return Response.ok(si).cacheControl(cc).tag(eTag).build();
      }

      @PUT
      @Consumes(MediaType.APPLICATION_XML)
      public Response updateStock(StockInfo si, @Context Request request) {
            ResponseBuilder builder = null;
            EntityTag etag = null;
            StockInfo ssi = null;

            if (dbMap.containsKey(si.getName()) == false) {
                  throw new NotFoundException("Stock doesn't exists for updating");
            }
            ssi = dbMap.get(si.getName());
            etag = new EntityTag(String.valueOf(ssi.hashCode()));
            builder = request.evaluatePreconditions(etag);
            if (builder != null) {
                  // pre-conditions failed
                  return builder.build();
            }

            dbMap.put(si.getName(), si);
            return Response.accepted().build();
      }

}
```

20.19 Security

```
package com.sw.dto;

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

@XmlType
@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "statusInfo")
public class StatusInfo {
	private String pancard;
	private String status;

	// setters & getters
}
```

```
package com.sw.resource;

import javax.annotation.security.RolesAllowed;
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.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.SecurityContext;

import com.sw.dto.StatusInfo;

@Path("/it")
public class ITResource {

	@Context
	private SecurityContext securityContext;

	@GET
	@Produces(MediaType.TEXT_PLAIN)
	@Path("/{pancard}")
	public String getITRVStatus(@PathParam("pancard") String pancard) {
		if (securityContext.isUserInRole("ADMIN")) {
			return "Internal Approval in progress";
		} else if (securityContext.isUserInRole("USER")) {
			return "In progress";

		}
		return null;
	}
```

```
	@GET
	@Path("/{pancard}")
	@Produces(MediaType.APPLICATION_XML)
	@RolesAllowed("ADMIN")
	public StatusInfo getITRVSubmission(@PathParam("pancard") String pancard) {
		StatusInfo si = new StatusInfo();

		si.setPancard(pancard);
		si.setStatus("Approved");

		return si;
	}
}
```

```
<?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"
	xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
	id="WebApp_ID" version="3.0">
	<display-name>CustomParamConverterWeb</display-name>
	<context-param>
		<param-name>resteasy.role.based.security</param-name>
		<param-value>true</param-value>
	</context-param>
	<servlet>
		<servlet-name>resteasy</servlet-name>
		<servlet-
class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-class>
		<init-param>
			<param-name>resteasy.scan</param-name>
			<param-value>true</param-value>
		</init-param>
		<init-param>
			<param-name>resteasy.servlet.mapping.prefix</param-name>
			<param-value>/rest</param-value>
		</init-param>
		<load-on-startup>1</load-on-startup>
	</servlet>
	<servlet-mapping>
		<servlet-name>resteasy</servlet-name>
		<url-pattern>/rest/*</url-pattern>
	</servlet-mapping>
	<security-role>
		<role-name>ADMIN</role-name>
	</security-role>
	<security-role>
		<role-name>USER</role-name>
	</security-role>
	<security-constraint>
```

```
            <web-resource-collection>
                <web-resource-name>ITResource</web-resource-name>
                <url-pattern>/rest/it/*</url-pattern>
                <http-method>GET</http-method>
            </web-resource-collection>
            <auth-constraint>
                <role-name>ADMIN</role-name>
                <role-name>USER</role-name>
            </auth-constraint>
        </security-constraint>
        <login-config>
            <auth-method>BASIC</auth-method>
        </login-config>
</web-app>
```

20.20 Jax-RS with Spring Integration

```
package com.jrs.app;

import java.util.HashSet;
import java.util.Set;

import javax.servlet.ServletContext;
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;
import javax.ws.rs.core.Context;

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

@ApplicationPath("/rest")
public class JAXRSSpringIntegApplication extends Application {
        private Set<Object> singletons;

        public JAXRSSpringIntegApplication(@Context ServletContext context) {
                String resourceConfigPath = null;
                singletons = new HashSet<Object>();

                resourceConfigPath = context.getInitParameter("resourceConfigPath");

                ApplicationContext appContext = new ClassPathXmlApplicationContext(
                                resourceConfigPath);
                singletons.add(appContext.getBean("flightResource"));
        }

        @Override
        public Set<Object> getSingletons() {
                return singletons;
        }

}
```

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

	<bean id="flightResource" class="com.jrs.resource.FlightResource">
		<property name="flightService" ref="flightService" />
	</bean>
	<bean id="flightService" class="com.jrs.business.FlightService" />
</beans>
```

```
package com.jrs.business;

public class FlightService {
	public float getAirFare(String flightNo) {
		return 4222.23f;
	}
}
```

```
package com.jrs.resource;

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

import com.jrs.business.FlightService;

@Path("/flight")
public class FlightResource {
	private FlightService flightService;

	@GET
	@Produces(MediaType.TEXT_PLAIN)
	@Path("/{flightno}")
	public float getPrice(@PathParam("flightno") String flightNo) {
		return flightService.getAirFare(flightNo);
	}

	public void setFlightService(FlightService flightService) {
		this.flightService = flightService;
	}

}
```

```
<?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"
	xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
	id="WebApp_ID" version="3.0">
	<display-name>FirstJbossRSWeb</display-name>
	<context-param>
		<param-name>resourceConfigPath</param-name>
		<param-value>com/jrs/app/resource-beans.xml</param-value>
	</context-param>
</web-app>
```

XML VS JSON

21 XML VS JSON

Abbreviation:

- XML Stands for Extensible Markup language
- Json stands for Javascrip object notation

Meaning:

- XML is a markup language that defines set of rules for encoding documents in the format that is both human readable and machine readable

Json is a text-based open standard designed for human-readable data interchange. It is derived from JavaScript scripting language for representing simple data structures and associated arrays, called objects. Json format is often used for serializing and transmitting structured data over the network connection. Primarily to transmit data between server and a web application serving an alternative to XML.

Type of format:

- XML - Markup languange
- Json - Data interchange

Extended from:

- XML - SGML
- Json - JavaScript.

Developed By:

- XML - World Wide Web Consortium
- The Json format was originally specified by Douglas Crockford for using it at State Software.

Data types:

- XML – Does not provide any notation of data types [illegible] XML schema for adding type information.
- Json – Provides scalar data types and [illegible] data through arrays and objects.

Support for Arrays:

- XML – Arrays have to be expressed by conventions, for example through the use of outer placeholder element that models the arrays content as inner elements
- Json – Native array support

Support for Objects:

- XML - Objects has to be represented through conventions, often through a mixed use of attributes and elements
- Json - Native Objects support

Null support:

- XML - Requires the use of xsi:nil on elements in an XML instance document plus the import of the appropriate namespace
- Json - Natively recognized the null

Comments:

- XML – Native support and [illegible]
- Json – Not supported

Namespaces:
- XML – Supports namespaces which elements the risk of name collisions when combining the documents
- Json – No concept of namespaces. Naming collisions are avoided by nesting objects or using prefix in an object member

Formatting decisions:
- XML – Requires a greater effort to decide how to map application types to XML elements and attributes
- Json – Simple provides a much more direct mapping of application data

Size:
- XML – Documents tends to be lengthy in size
- Json – Syntax is very terse and yields formatted text where most of the space is consumed

Parsing in Javascript:
- XML – Requires XML DOM implementation and additional application code to map text back to Javascript objects
- Json – No additional code is required apart from using aval() function

Learning curve:
- XML – Generally tends to use of several technologies in concert: Xpath, XML Schema, XSLT, DOM, XML Namespaces etc.
- Json – Already familiar with developers through Javascript

Tools:
- XML – Enjoy a mature set of tools widely available
- Json – Rich tool support - scarce

22SOAP VS REST

22.1 REST

- It is not a protocol rather architectural style
- Completely stateless
- Provide good caching mechanism over Http protocol (can be used to scale-up when the data is not frequently modified on the resource)
- It is Web Style (SOAP is distributed technology)
- There is no standard definition language to expose the interface of resource to the client. They need to have common understanding of content need to be passed (depends on response content container hypermedia)
- It is suitable for few applications such as Mobile platform or PDA's (as SOAP is eliminated)
- These are easy to integrate with existing Web applications, they don't need any changes in existing architecture
- It doesn't enforce the message format, it can be XML or JSON

22.2 SOAP

- SOAP is a XML-based messaging protocol
- It has a specification, but REST has none
- SOAP has specification for stateful implementation
- No Caching support
- It has a standard description language WSDL using which consumer and provider can exchange the data over standard interface
- SOAP requires less plumbing code as transactions, security, addressing etc has been provided.
- SOAP is XML based protocol and will not supports any other formats like JSON