* In case of XSD, to represent occurances use minOccurs="" and maxOccurs="".

* Possible values are

minOccurs = 0, 1

maxOccurs = 1, unbounded

Other than root element, attributes every (child) element can contain occurances as 0, unbounded, 1.

Syntax:

```
<xs:element name="" type="____"
    minOccurs="0/1"
    maxOccurs="1/unbounded">
</element>
```

* Default values are minOccurs="1" and maxOccurs="1".

ex: define <BookVersion> with type double and min=1 to max=no limit.

```
<xs:element name="bookVersion" type="xs:double"
    minOccurs="1" maxOccurs="unbounded">
</xs:element>
```

ex 2 Define bookName which is optional and max = no limit with type string

<xs:element name="bookName" type="xs:string"
minOccur="0" maxOccur="unbounded" />

book (bid, bname, amt)
bname (name(1,n), auth)
bname_attribute – version
(double)

<xs:element name="book">
<xs:complexType>
<xs:~~seq~~ element ~~stri~~ ~~type~~ = "name = "bid"
type = "xs:int">

sequence / choice / all under complexType we are going to use any one of above type

i:> sequence : provide all the child of in xml in given order.

ii:> choice: provide any one child in xml from give child. It indicates choose one.

iii:> all : write all child in xml in any order.

for choice

book (amt / cost / price)

XSD code:

```
<xs:element name="book" >
    <xs:complexType>
    <xs:choice>
     <xs:element name="amt" type="xs:int"/>
    <xs:element name="cost" type="xs:int"/>
    <xs:element name="price" type="xs:int"/>
    </xs:complexType>
</xs:element>
```

SOA (Service Oriented Architecture)

* It is a design used by web service (SOAP).
* It contains 3 components and 3 operations

Components are

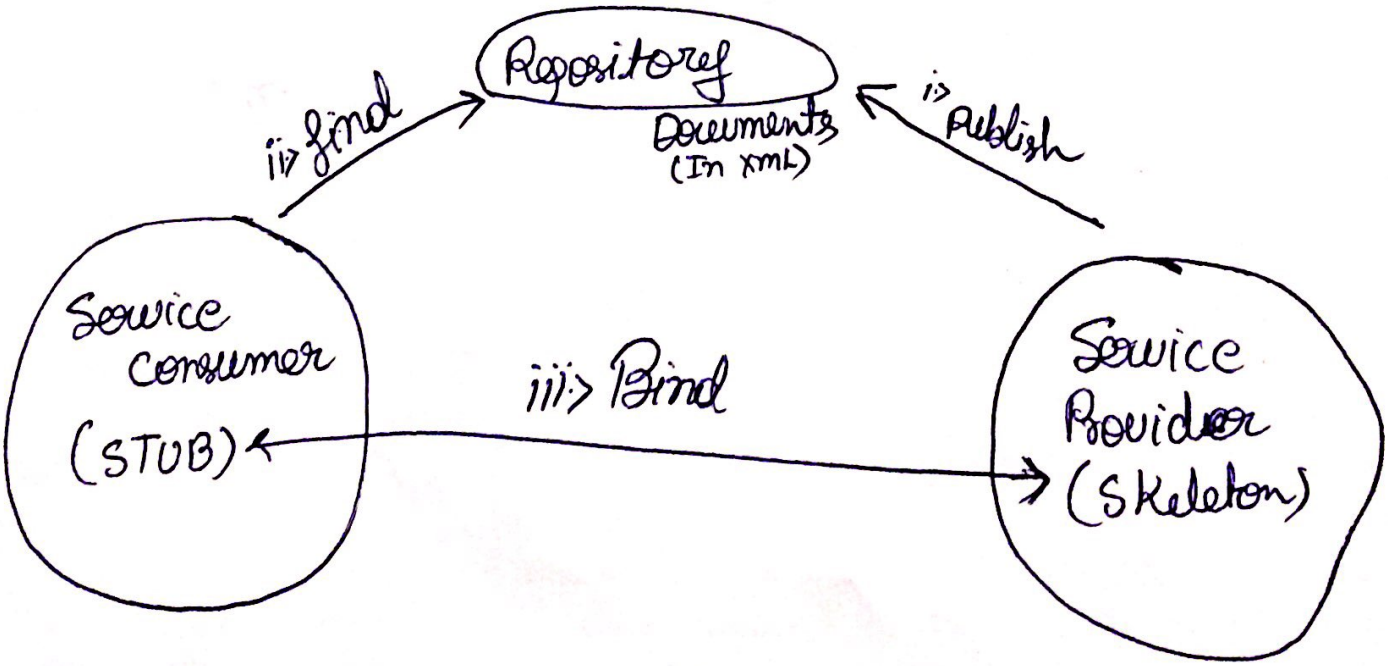
i> service provider
ii> service consumer
iii> repository

Operations are

i> publish
ii> find
iii> bind

SOA Design :

(Level 1 Design)

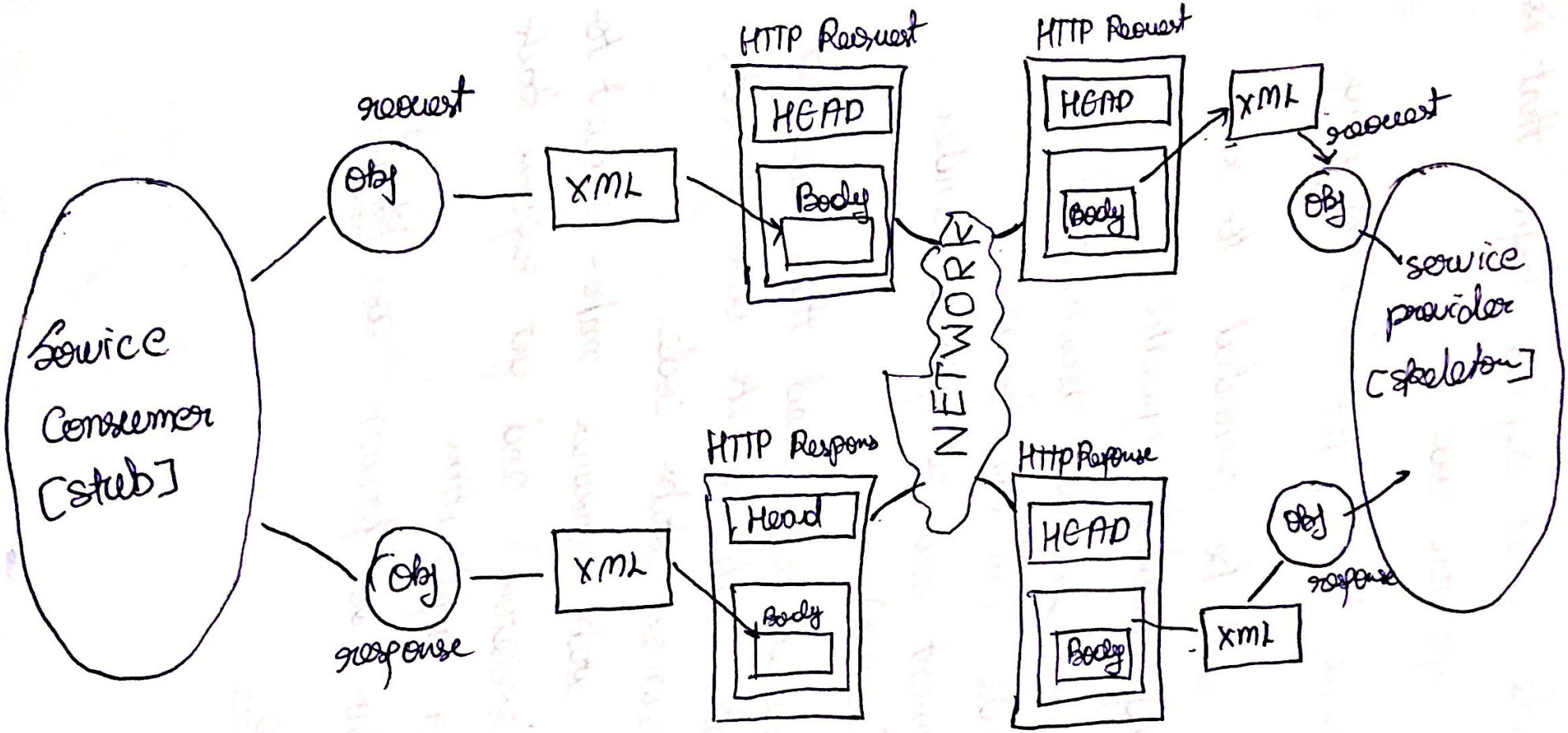


Sowice Poovider : It defines logic that is known as skeleton code

* This code will be in providers language ex: Java
* This code will be converted to xml format using publish operation
* publish will generate document from skeleton code (in xml format).
* This document will be placed under repository (server).
* Sowice Consumer will read this document and generates code in its own format, that is known as stubs code.
* By using stub consumer makes request to provider (skeleton) and get response back to consumer application.

This operation is known as Bind.

* Bind flow

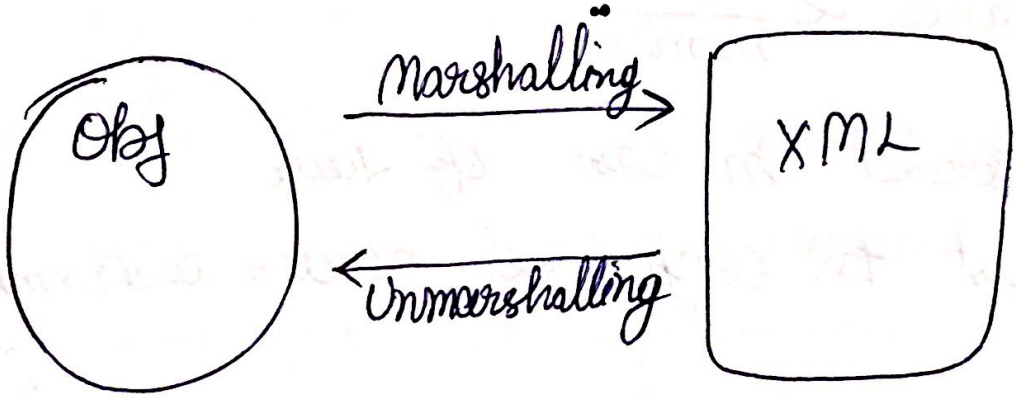


Marshalling : It is a process of converting Object to XML format.

Unmarshalling : It is a process of converting XML to object format.

JAXB (Java API for XML Binding) is used for marshalling and unmarshalling process.

It is given by Sun Microsystem.



- Here XML is used as medium to do communication between provider and Consumer.
- XML can not be sent over network, so it will be placed under HTTP Body.
- Placing XML into Http is known as Wrapping and reverse process is known as UnWrapping.

* Web services provides new programming called as REST which follows a design pattern that is "client-server with FC"

* FC (front controller)

This design pattern is used for increasing performance by reducing memory of application development.

$$\text{Performance} \propto \frac{1}{\text{memory}}$$

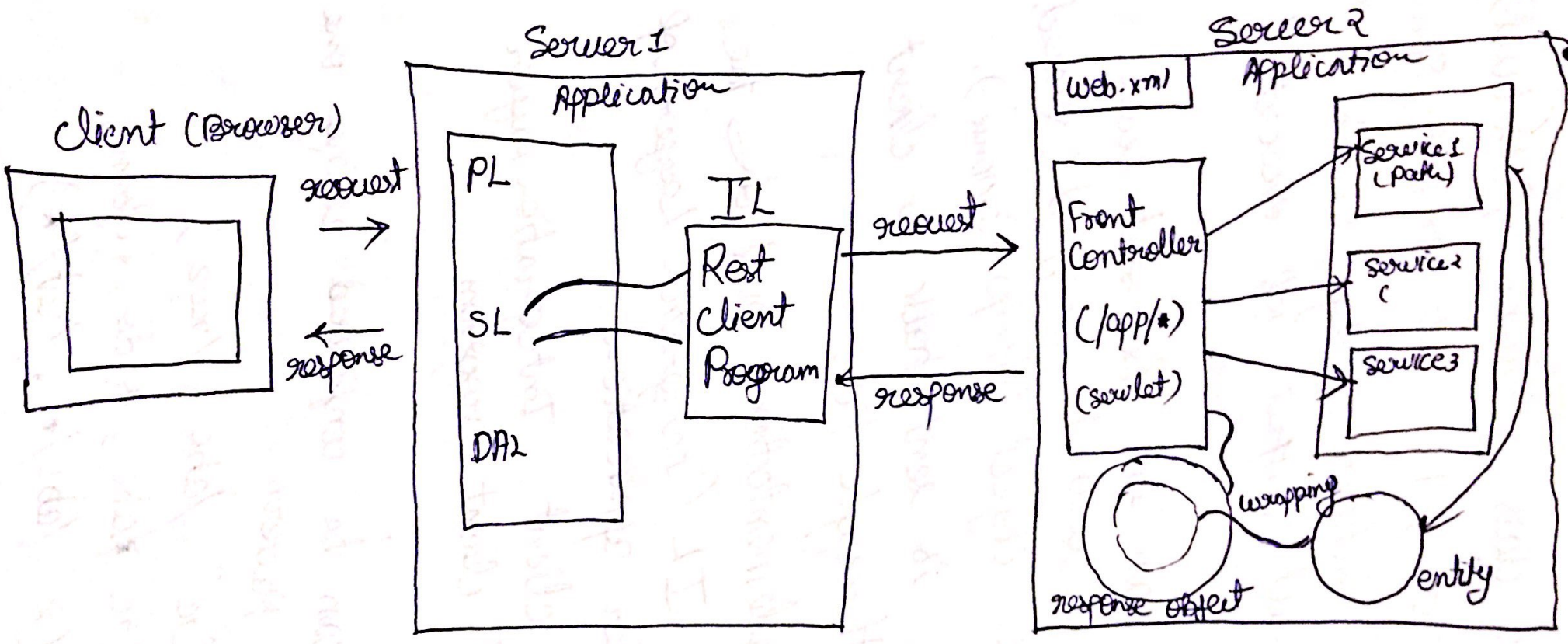
* FC is servlet in case of Java
* It must be configured under web.xml file.
* It should follow directory match URL pattern (ex: /rest/*) in web.xml).
* Every request that contains directory pattern will be given to FC only.
* It will decide which service needs to be called. Here Service is a simple

* Every service class contains Path (url).
* Based on path, FC will execute Service.
* Service will return entity, which contains result/output of executed service.
* This entity can not be sent over network, so it will be wrapped (placed) into response object (HTTP Response).
* This response is sent back to client application through FC.
* This FC implementation must be done under IL (Integration Layer) of service provider application.
* Consumer or client Integration layer contains ReST client program.

URL patterns

* A servlet can be configured using one of below URL Pattern.

i) exact match (ie /abc, /xyz)
ii) extension match (ie *.abc, *.do, *.form)
iii) Directory Match (ie /abc/*, /rest/*)



Type of Http Error response

1xx — Information
2xx → success
3xx → Redirection
4xx → client-side Error
5xx → Server Errors

400 → Syntax error (data type conversion problem)
404 → Resource Not Available (url is not valid)
405 → request response (GET/POST mismatch)
415 → method request type mismatch (like JSON, but request is xml)

Whenever a request is made server returns one of above response code for sure.

* 200 : Request process and executed successfully executed.
* 400 Data type conversion problems like String to int
* 405 method not allowed, If request is GET type and Service is POST type they are not matched.
* 415 Unsupported Media type. Request sent in text format but provider needs xml.

* 500 : Server error, unable to create service provider object, version problem bad request gateway.

XML : extensible Markup Language

* This is used for data representation and exchange of data between two heterogeneious applications.
* In case of webservices two applications exchanges request and response in xml format only.
* XML types :-

i-> Simple XML (welformed)
ii-> Standard XML (valid)

Simple XML :

* This XML follows only below rules XML should have root tag and it should not be duplicated.
* XML is case sensitive accepts both cases but open tag must match with close tag.

<empId> 44 </empId> (invalid)

Tag name can be any thing and it can be in any case.

Every open tag must have paired closing tag or self closing tag.

Ex: <eid> 55 </eid> valid
<eid> ss <eid/> invalid
<eid /> valid

Attributes

* It must be defined in a open tag only
* These values should be quoted (either single or double)

ex <emp id=45 > invalid
<emp id="45" > (valid)
<emp id='45' > (valid)

In case of multiple attributes order is not required while ~~using~~ defining at tag level.

Ex: <emp id='ss' name='A' > } valid
<emp name='A' id='ss' > }

tag level and attributes are also case sensitive.

ex: <emp id='s' ID='ss'>
(valid)

<emp id='s' id='7'>
(invalid).

Standard xml

An xml follows either DTD or XSD to maintain a standard.

DTD or XSD provides below rules

i:> pre-defined tag set
ii:> Order of tags
iii:> Attributes of tags
iv:> Occurance of tags

DTD (Document Type Definition)
XSD (xml Schema Design)

Realtime example of xml standard

web.xml
hibernate-config.xml
hibernate-mapping.xml

In web sevices consumer and provider are connected together and exchanges xml data, that must follow DTD/XSD.

* This standard file special name is WSDL.
  It will be genorated after publish operation
  also called as document.

<!ELEMENT Book (bid, bname, bAuth)>

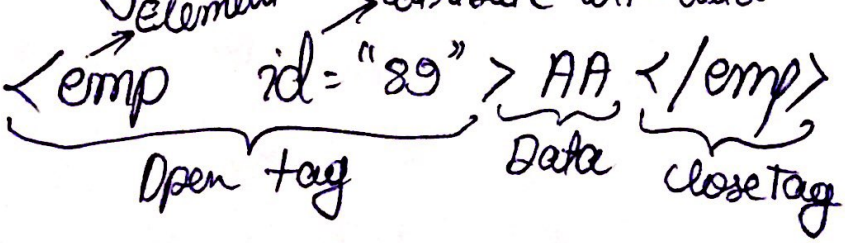
<!ELEMENT bid (#PCdata)>
<!ELEMENT bname (#pcdata)>
<!ELEMENT bAuth (fn, ln)>
<!ELEMENT fn (#PCDATA)>
<!ELEMENT ln (#PCdata)>

book
  bid = 95
  bname = AA
  bAuth
    fn = AA , ln = BB

DTD (Document Type Definition)

DTD provides rules for xml file, they are
  i) pre-defined Tag set
A tag can contain one element + Zero-to-n Attributes.

Tag = Element + Attributes
(element, attribute with data)



* Here, all elements are divided into 2 types they are

i> Leaf Element: which contains only data no child element.

ex: <empId> 88 </empId>
<sal> 83.6 </sal>
<fn> SAM </fn>

ii> Non-Leaf Elements:

Element which contains child elements are known as non-leaf elements.

ex: <book>
<bid> 84 </bid>
</book>

Here book is non leaf.

<auth>
<fn> AA </fn>
<ln> AB </ln>
</auth>

Here auth is non-leaf and fn, ln are leaf elements.

Syntax: To define element

i> Non-leaf (contains childs)

<!ELEMENT element-name (child elements, ...)>

ii) leaf (contains data)

<!ELEMENT element-name (#PCDATA)>

Example: Define DTD for book contains bid, bname and auth, where bid contains only data and where bname contains lang, mode. auth contains fn, ln. Here bid, lang, mode, fn, ln are leafs.

book.dtd

```
<!ELEMENT book (bid, bname, auth)>
<!ELEMENT bname (lang, mode)>
<!ELEMENT auth (fn, ln)>
<!ELEMENT lang (#PCDATA)>
<!ELEMENT mode (#PCDATA)>
<!ELEMENT fn (#PCDATA)>
<!ELEMENT ln (#PCDATA)>
<!ELEMENT bid (#PCDATA)>
```

iii) Order of tags:

In predefined tags whatever order is followed it should be in same order, while using in xml.

iii) Occurance of tags

Every element by default must occur one time (min = max = 1)

To specify some other occurance, use occurance table.

Occurance Table

Symbol	min	max
*	0	n
+	1	n
?	0	1
No symbol	1	1

Pipe (|) either or

If min = 0 then element is optional

Here, n indicates non-limit.

Example DTD

<!ELEMENT book (bid ?, bname *, auth +,
(amt | mrp | cost))>

<!ELEMENT bid (#PCDATA)>

// bname, auth, amt, cost mrp are #PCDATA

Create DTD and XML files in eclipse

i.> Open eclipse in java, create a new project.

File -> new → Java project → Enter name → Finish

ii.> DTD file :

Right click on src → new → other → type DTD search → choose DTD file → Next → Enter name.

example name books.dtd → Finish

iii.> XML using DTD :

src → right click → new → Other → type XML → choose XML file → Next → Create XML file from a DTD file → chose DTD from src → Next → finish.

Here <! DOCTYPE --> is used as link to connect XML with DTD.

<! DOCTYPE root-element PUBLIC/SYSTEM "description" "file name with location .dtd">

ex : DTD :- books.dtd

<!ELEMENT book (bid?, bname*)>
<!ELEMENT bid (#PCDATA)>
<!ELEMENT bname (#PCDATA)>

example xml → follows DTD :-

<!DOCTYPE book SYSTEM "books.dtd">
<book>
<bid> 79 </bid>
<bname> AA </bname>
<bname> BB </bname>
</book>

web.xml DTD Link

<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.
//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

System indicates paid source and PUBLIC indicates open source.

Here Description is optional. If it is given contains format Below :

File provider, File type file version, language.

* File location can be URL or Local FOLDER file

Define Simple Types in XSD

<xs: element name = " " type = "xs: __ " />

bookId :

<xs: element name = "bookId" type = "xs: int" />

bookAuth

<xs: element name = "bookAuth" type = "xs: String" />

cost

<xs: element name = "cost" ~~type = "xs: double" />~~

type = "xs: double / decimle" />

XSD : XML schema Design

It is an advanced format of DTD, which provides

→ Data Type supports
→ data Restrictions
→ condition re-usable

→ Multi - configuration

* Every element in xsd can be SimpleTyp or complexType.

i-> SimpleType Element : An element which contains only data, no child or no attribute.

* Every simpleType must have data type.
* ComplexType Element :
  An element which contains either child or attribute, some times both.

It will never follow data type.

Syntax for simple type :

<xs:element name="____" type="xs:___"/>

Or

<xs:element name="____" type="xs:___">

</xs:element>

Example data types:

xs:int , xs:double , xs:string , xs:decimal

xs:integer

empAge → int

<xs:element name="empAge"
type="xs:int" />

empName → String

<xs:element name="empName"
type="xs:string" />

Example bookId min=1 max=1500

<xs:element name="bookId">
<xs:simpleType>
<xs:restriction base="xs:int">
<xs:minInclusive value="1" />
<xs:maxExclusive value="1500" />
</xs:simpleType>
</xs:restriction>
</xs:simpleType>
</xs:element>

```
<xs:element name="bookId">
  <xs:simpleType>
    <xs:restriction base="xs:int">
      <xs:minExclusive value="0"/>
      <xs:maxExclusive value="1501"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

-494 -301

```
<xs:element name="bookId">
  <xs:simpleType>
    <xs:restriction base="xs:int">
      <xs:minExclusive value="~~201~~ 299"/>
      <xs:maxExclusive value="~~301~~ 321"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

-321 -129

~~<xs:element~~

minExclusive -322
maxExclusive -128

XSD Elements

1) SimpleType with Restriction

* A Restriction is a limit over a data type to an element data
* In case of int restrictions are min, max.

i-> minInclusive <=

ii -> minExclusive <

iii -> maxInclusive >=

iv -> maxExclusive >

v -> ~~minE~~

* for SimpleTypes restrictions can be given in below syntax:

```
<xs: element name="  ">
  <xs: simpleType>
    <xs: restriction base="  ">
      <xs: minInclusive value="  " />
    </xs: restriction>
  </xs: simpleType>
</xs: element>
```

Here name = " " indicates element name and base = " " data type.

If restriction are given with element type should not be defined.

Example

define XSD element with name empAge min = 22, max = 60.

```
<xs: element name="empAge" >
```

```
<xs:simpleType>
  <xs:restriction base="xs:int">
   <xs:minExclusive value="21"/>
   <xs:maxExclusive value="61"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
```

* Above code can be define using Inclusive ~~Exclusive~~ as below :

```
<xs:element name="empAge">
  <xs:simpleType>
   <xs:restriction base="xs:int">
    <xs:minInclusive value="22"/>
    <xs:maxInclusive value="60"/>
   </xs:restriction>
  </xs:simpleType>
</xs:element>
```

```
<xs:element name="empId">
  <xs:simpleType>
    <xs:restriction base="xs:int">
      <xs:minExclusive value="21"/>
      <xs:maxInclusive value="60"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

& <xs: minInclusive value="25" />
<xs: maxInclusive value="13" />
invalid combination.

Either only min or max can also be specified for an element.

<xs: element ~~type~~ name="password">
<xs: simpleType>
<xs: restriction base="xs: string">
<xs: minlength value="6" />
<xs: maxlength value="10" />
</xs: restriction>
</xs: simpleType>
</xs: element>

<xs: element name="modeNum">
<xs: simpleType>
<xs: restriction base="xs: string">
<xs: length value="6" />
</xs: restriction>
</xs: simpleType>
</xs: element>

<xs: element name="serviceId">
<xs: simpleType>
<xs: restriction base="xs: string">
<xs: maxlength value="14" />
</xs: restriction>
</xs: simpleType>

xs: string type restrictions

xs: string type provide range restriction as.

i-> minLength / maxLength exact length restriction

ii-> length

* If we do not specify minLength, then default value is zero.
* Only length indicates exact length (min-max)

Syntax

```
<xs: minLength value="" />
<xs: maxLength value="" />

Or
<xs: length value="" />
```

* length should never be negative value, also min < = max length.

Defined xsd element of authentication of string restriction

```
<xs: element name="Authentication">
  <xs: simpleType>
    <xs: restriction base="xs: string">
      <xs: minLength value="12" />
    </xs: restriction>
  </xs: simpleType>
</xs: element>
```

empsal decimal

minInclusive value = -314

maxInclusive value = -219 -210 -314

totalDigit value = 6

fractionDigit value = 3

i> Invalid -217.312

ii> Invalid -369.417

iii> valid -302.19

iv> Invalid -202.919

v> Invalid -220.2345

vi> Invalid -96.31

vii> Invalid -97.367

viii> Invalid 400.36

ix> Invalid 346.34

x> valid -280.9

Decimal Restrictions

decimal type provides restrictions are

i> minInclusive

ii> maxInclusive

iii> minExclusive

iv> maxExclusive and also,

v.> totalDigits
vi.> fractionDigits

* totalDigits : total count (total digits) in given number should not exceed with totalDigits.

It will never count symbol like - (minus) . (dot)

* fractionDigits : count (digits count) after decimal point should not exceed fractionDigits.

* for example :-

Define empSal with totalDigits = 6, fractionDigits = 2.

```
<xs:element name="empsal">
<xs:simpleType>
    <xs:restriction
        base="xs:decimal">
    <xs:totalDigits value="6" />
    <xs:fractionDigits value="2" />
</xs:restriction>
</xs:simpleType>
```

Ex:

i -> -23.22 (valid)

ii -> 232323.2 (Invalid)

iii -> 23.543 (Invalid)

iv -> 1111 3.2 (valid)

v -> -8782.433 (Invalid)

* Define xsd element bookCost min = -883 max = 26 total digits = 6, fraction digits = 3

```
<xs:element name="bookCost">
  <xs:simpleType>
    <xs:restriction base="xs:decimal">
      <xs:minInclusive value="-883"/>
      <xs:maxInclusive value="26"/>
      <xs:totalDigits value="6"/>
      <xs:fractionDigits value="3"/>
    </xs:simpleType>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

-3245.44	invalid
-782.4	valid
-853.233	valid
-0.4343	invalid
0.0	valid
-953.43	invalid
-894.3	invalid
-872.55	valid
124.45	invalid
753.2	invalid

enum Types (xs: enumeration):

* Some times possible values set is represented over data type. No other values are accepted in that case. To represent them we use <xs:enumeration value=""/>
* This can be created over any datatype.
* In case of string value is case-sensitive.
* Example
  ticketStatus [CNF, WL, RAC]

XSD:

```
<xs:element name="ticketStatus">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="CNF"/>
      <xs:enumeration value="WL"/>
      <xs:enumeration value="RAC"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

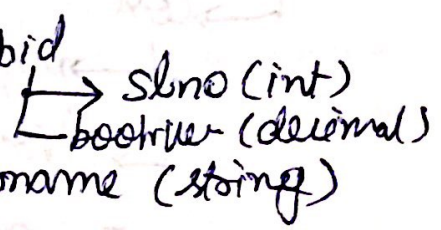
example: define empDesg with possible values as 1, 2, 5, 10, 15

example: define JdkVersion with posible value 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 (xs:dec

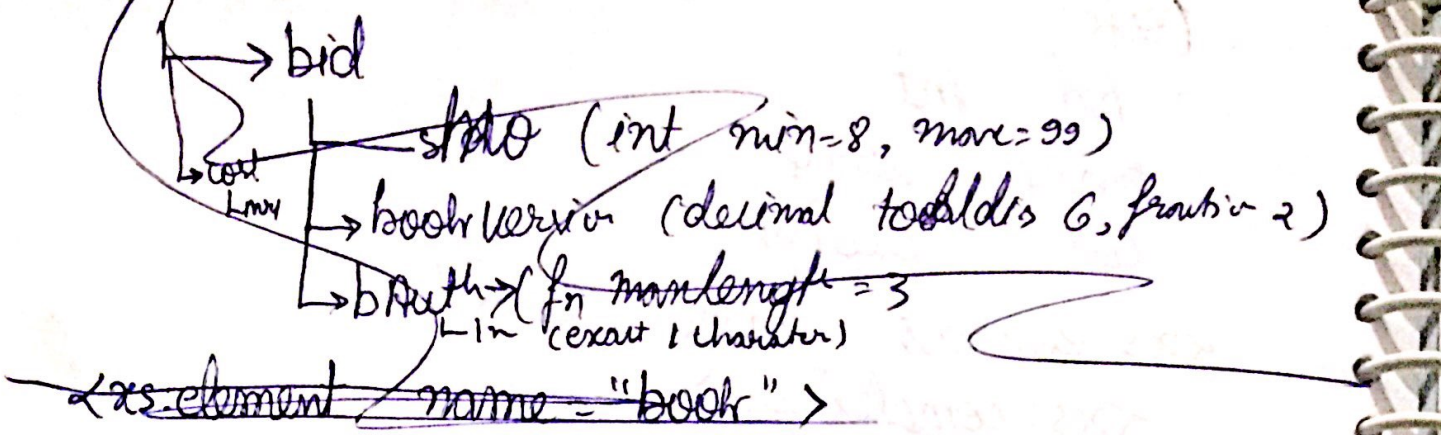
```
<xs:element name="empDesg">
  <xs:simpleType>
    <xs:restriction base="xs:int">
      <xs:enumeration value="1" />
      <xs:enumeration value="2" />
      <xs:enumeration value="5" />
      <xs:enumeration value="10" />
      <xs:enumeration value="15" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>

<xs:element name="tdhVersion">
  <xs:simpleType>
    <xs:restriction base="xs:decimal">
      <xs:enumeration value="1.0" />
      <xs:enumeration value="2.0" />
      <xs:enumeration value="3.0" />
      <xs:enumeration value="6.0" />
      <xs:enumeration value="7.0" />
      <xs:enumeration value="8.0" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

book
bid int
bname string
cost decimal

```
<xs:element name = "book" >
  <xs:complexType >
    <xs:sequence >
      <xs:element name = "bid">
        <xs:simpleType >
          <xs:restriction base = "xs:int" >
            <xs:element
      <xs:element name = "bid" type = "xs:int"/>
      <xs:element name = "bname" type = "xs:string"/>
      <xs:element name = "cost" type = "xs:decimal"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```



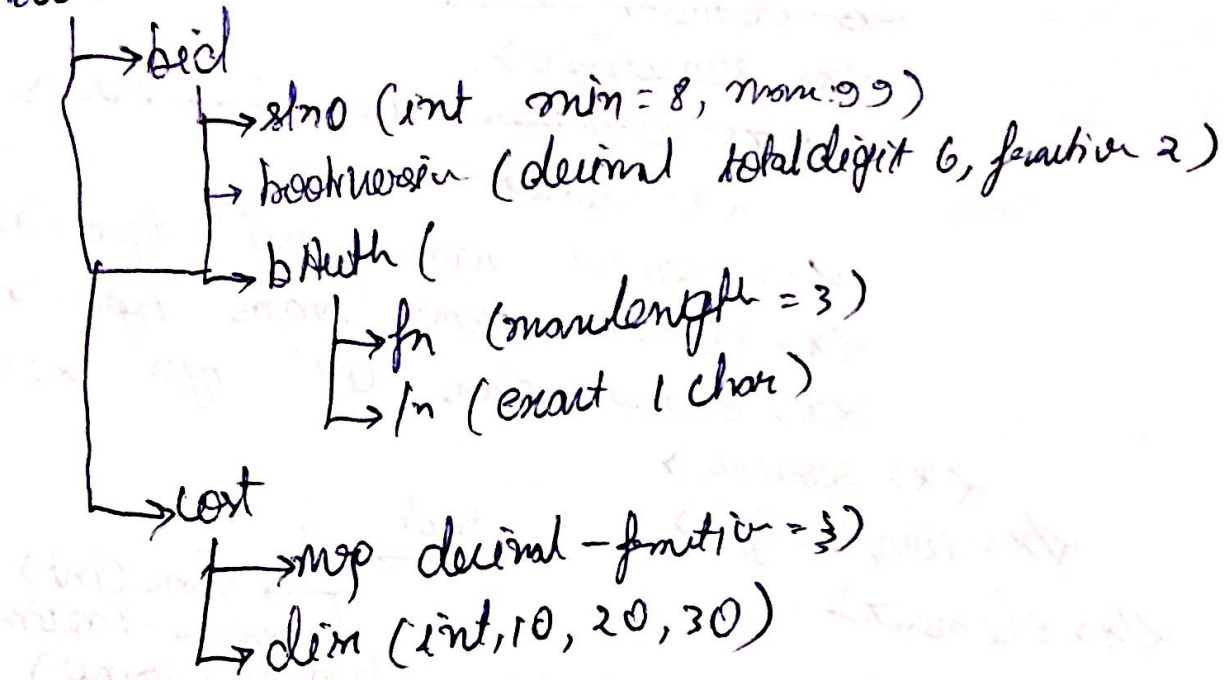

```
<xs:element name = "bookr">
  <xs:complexType >
    <xs:sequence >
      <xs:element name = "bid" >
        <xs:complexType >
          <xs:sequence>
            <xs:element name = "slno" type="xs:int"/>
            <xs:element name = "bookrice" type = "xs:decimal"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name = "bname" type = "xs:string"/>
    </xs:sequence>
  </xs:complexType>
```



~~<xs:element name = "book">~~

<xs:

book




```
<x:element name = "book">
<xs: complexType>
  <xs: sequence>
    <xs: element name = "slno">
      <xs: simpleType>
        <xs: restriction base = "xs: int">
          <xs: minInclusive value = "8" />
          <xs: maxInclusive value = "99" />
        </xs: restriction>
      </xs: simpleType>
    </xs: element>
    <xs: element name = "bookVersion">
      <xs: simpleType>
        <xs: restriction base = "xs: decimal">
          <xs: totalDigits value = "6" />
```

```
<xs:fractionDigits value="99" />
</xs:restriction>
</xs:simpleType>
</xs:element>
<xs:element name="bAuth">
<xs:complexType>
<xs:sequence>
<xs:element name="fn">
<xs:simpleType>
<xs:restriction base="xs:string">
<xs:maxLength value="3" />
</xs:restriction>
</xs:simpleType>
</xs:element>
<xs:element name="ln">
<xs:simpleType>
<xs:restriction base="xs:string">
<xs:length value="1" />
</xs:restriction>
</xs:simpleType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="cost">
<xs:complexType>
<xs:sequence>
<xs:element name="mrp">
<xs:restriction base="xs:decimal">
<xs:fractionDigit value="3" />
</xs:restriction>
</xs:element>
<xs:element name="discount">
<xs:simpleType>
<xs:restriction base="xs:int">
<xs:enumeration value="10" /> /20 30/>
</xs:restriction></xs:simpleType></xs:element></xs:sequence></xs:complexType></xs:element>
```

</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>

Complex Type elements:

An element which contains either child or attribute, some time both is known as complex Type element.

Syntax:

```
<xs:element name="_____" >
  <xs:complexType>
    <xs:sequence>
      // child element
    </xs:sequence>
    // attributes
  </xs:complexType>
</xs:element>
```

Syntax for simple Type element

i-> Without Restriction

```
<xs:element name="___" type="_____" />
```

ii-> With Restriction

```
<xs:element name="____" >
  <xs:simpleType>
    <xs:restriction base="____" >
      <xs:____ value="____" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Example Define XSD for student (sid, sname, sfee)

And sname (fn, ln)

```
<xs:element name = "Student">
  <xs:complexType>
    <xs:sequence>
      <xs:element name = "sid" type = "xs:int" />
      <xs:element name = "sfee" type = "xs:int" />
      <xs:element name = "sname">
        <xs: ~~element~~ complexType>
          <xs:sequence>
            <xs:element name = "fn" type = "xs:string" />
            <xs:element name = "ln" type = "xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Attributes: If an element contains attribut then element is treated as complexType

* Attribute contains data, so to create this provide name = " ", and type = " "
* Some time it may have restrictions also

Syntax 1

Without restriction :—

```
<xs:attribute name="" type="" />
```

Syntax 2

With restriction

```
<xs:attribute name="">
  <xs:simpleType>
    <xs:restriction base="">
      ———
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
```

* It must be declared under </xs:sequence> and </xs:complexType>

example: define

```
<empName eid="44">
    <fn> AA </fn>
</empName>
```

Here eid min value = 10, max = 50.

```
<xs:element name="empName">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="fn" type="xs:string" />
    </xs:sequence>
    <xs:attribute name="eid">
      <xs:simpleTypes>
```

```
<xs:restriction base="xs:int">
<xs:minInclusive value="10"/>
<xs:maxInclusive value="50"/> </
</xs:restriction> </xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:element>
```

Type Declarations :

If an element is simpleType or complexType we can declare them in 2 ways

i) Internal declaration

ii) external declaration

Internal declaration: Defining SimpleType or complexType inside the element (without name for type) is known as internal declaration.

ex: empId (int, min=1, max=20)

XSD code for internal declaration:

```
<xs:element name="empId">
<xs:simpleType>
<xs:restriction base="xs:int">
<xs:minInclusive value="1"/>
<xs:maxInclusive value="20"/>
</xs:restriction>
</xs:element>
```

ii) External declaration :

In this case we make simpleType or complexType as reusable

ie : declare simpleType outside of element and provide a name. This outside simpleType behaves as user defined data type. To access this we have to write tns:name

(tns => target name space)

* If above example is converted to external type then, xsd code is :

```
<xs:simpleType name="abc">
    <xs:restriction base="xs:int">
        <xs:minInclusive value="1"/>
        <xs:maxInclusive value="20"/>
    </xs:restriction>
</xs:simpleType>
```

using at element

```
<xs:element name="EmpId" type="tns:abc"/>
```

* This internal and external declarations also applicable to complexType.

Syntax

```
<xs:element name="">
    <xs:complexType>
        ----------
    </xs:complexType>
</xs:element>
```

external

<xs: complexType name="p">

</xs: complexType>

<xs: element name="emp" type="p">

Define permAddress (hno, loc) and presentAddr (hno, loc)

as external declaration

<xs: complexType name="addrType">
<xs: sequence>
<xs: element name="hno" type="xs: int" />
<xs: element name="loc" type="xs: String" />
</xs: sequence>
</xs: complexType>

<xs: element name="presAddr" type="tns: addrType" />
<xs: element name="permAddr" type="tns: addrType" />

JAXB (Java API for XML BINDING)

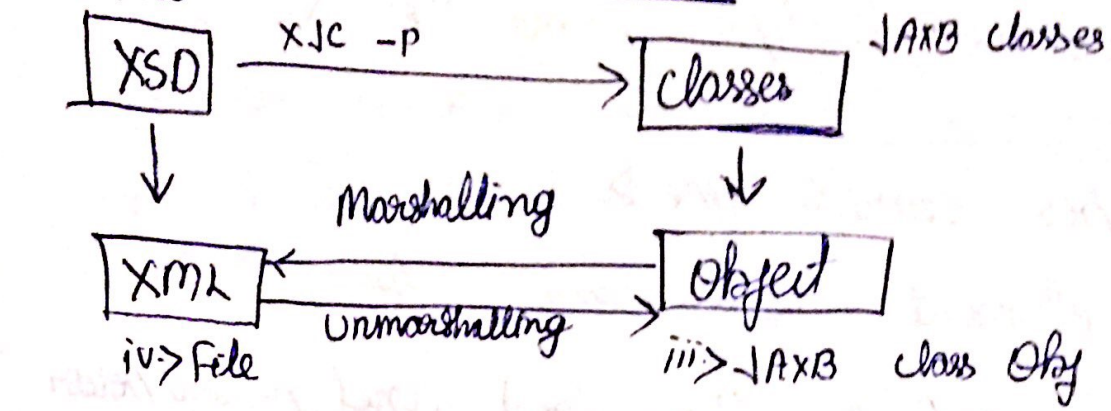
This API is given from Sun Microsystem.

This is used to convert Object (JAXB class Obj) to XML file and XML to Object.

i) Marshalling → Object to XML

ii) UnMarshalling → XML → Object

** JAXB Process



Step I Define XSD with ComplexType & SimpleType elements.

~~Step II~~

ex: employee (eid, ename)
eid : int, ename : string

```
<xs:element name="employee">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="eid" type="xs:int"/>
      <xs:element name="ename" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

JAXB classes

place above xsd in some folder location

ex: D:/abc/xyz.xsd

Shift + Right click

choose "open command window here".

Now type there

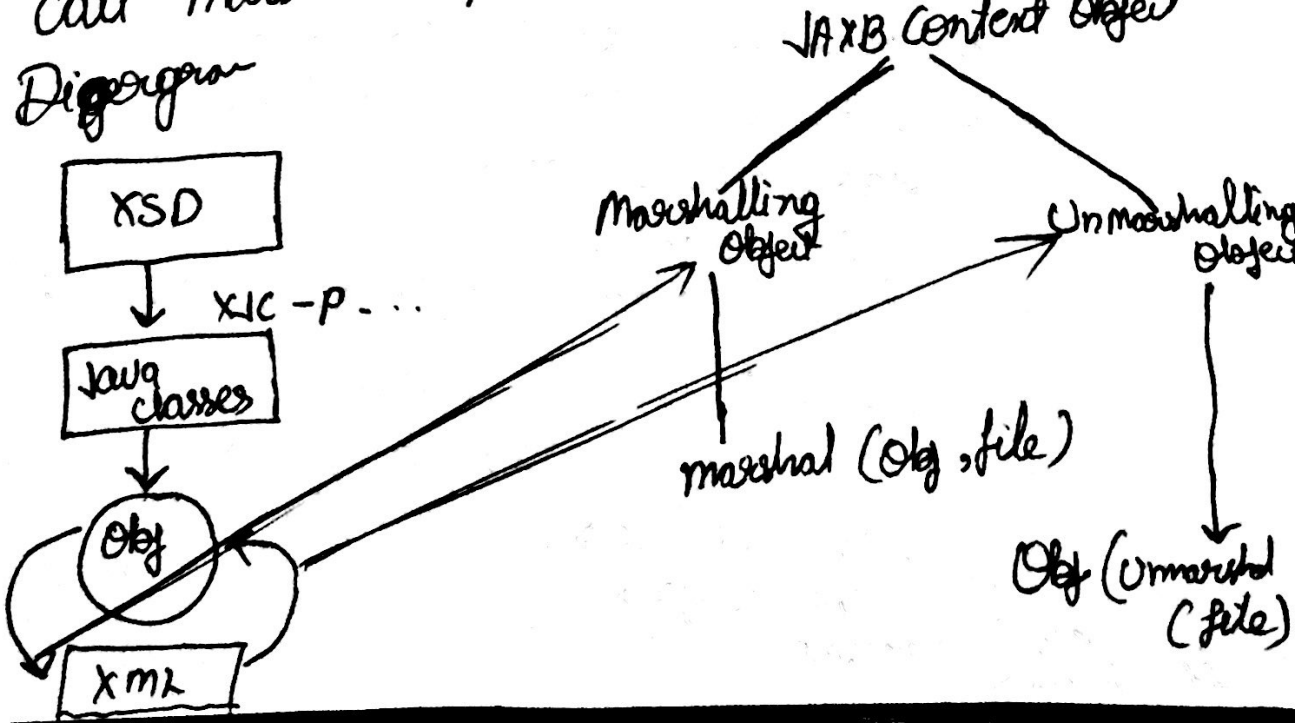
xjc -p com.app emp.xsd -d .

* It will generate JAXB classes.

-p : package
-d directory / destination / location
. present directory / folder

Marshalling Operation

* After generating JAXB classes, we can create object to them and those can be converted to xml format
* To perform any operation (marshalling/unmarshalling) we have to create JAXB context object using factory method ie newInstance(Class).
* Then, using context object create Marshller/unmarshaller object.
* call marshal() / unmarshal() method Flow Diagram



Step 1: create java project in eclipse - file → new → java project → enter name example sample → finish

Step 2: Create XSD file under src.
src → right click → new → other → type xml → choose xml schema file → Next → Enter name
example: emp.xsd → finish.

Step 3 | open XSD and add prefix "xs" to schema

When you open file looks as below :-

```
<xs:schema    xmlns:xs="_____"
</xs:schema>
```

Also define example xsd like

```
<xs:schema    xmlns:xs="_____">
  <xs:element name="employee">
    <xs:complexType>
      <xs:sequence>
<xs:element name="eid" type="xs:int"/>
<xs:element name="ename"
            type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Step 4: Generate JAXB classes using "xjc"

→ copy above xsd to any folder (ex: d/test)

→ shift + Right click

→ Open cmd from window here and then type

xjc cmd command

xjc -p com.app emp.xsd -d .

Step 5 copy "com" folder to project src folder.

Step 6 create Test class for marshalling/unmarshalling.

Code : (Test class)

```
public class Test
{
    void main(String ... s)
    {
    //Employee emp = new Employee();
      emp.setEid(856);
      emp.setEname("AA");
      try{
        //JAXB context Object
    JAXBContext c = JAXBContext.newInstance(Employee.class);
        //marshaller obj
    Marshaller m = c.createMarshaller();
    // marshall method
```

```
m.marshal(emp, new File("d:/a/ab.xml"));
Sop("done");
}catch(Exception e){ }
}
}
```

UnMarshalling example

```
public class Test {
    main( )
    {
    File f = new File("d:/a/a.xml")// location of xml.

    try{
    JAXBContent c = JAXBContext.newInstance(Employee.class);

    UnMarshaller um = c.createUnMarshaller( );

Employee obj = (Employee)um.unmarshal(f);
    }catch(Exception e)
        {
        }
    }
}
```

* Here unmarshal() return Object in java.lang.Object, downcast to our class

If given file is not available or not accessible to program then JAXB throws

Setting Schema to marshaller/Unmarshaller Object :-

* => This schema object contains XSD details.
* This is mainly used to validate XML conversions.

Step I => create SchemaFactory object.

SchemaFactory sf = SchemaFactory.newInstance ("~~www.~~http://www.w3.org/2001/XMLSchema");

Step II creating Schema object using sf obj.

Here, we need to pass XSD location as file object.

Schema s = sf.newSchema (new File ("src/emp.xsd"));

Step III Set Schema obj to marshaller obj or Unmarshaller Object based on operation chosen.

ex: m.setSchema (s);
um.setSchema (s);

In XSD to classes conversion,

* "every complexType is represented as class."
* "Every simpleType is represented as variable
* Example :- consider below XSD and observe generated class:

Employee

- eid : int
- ename
  - fn string
  - ln string

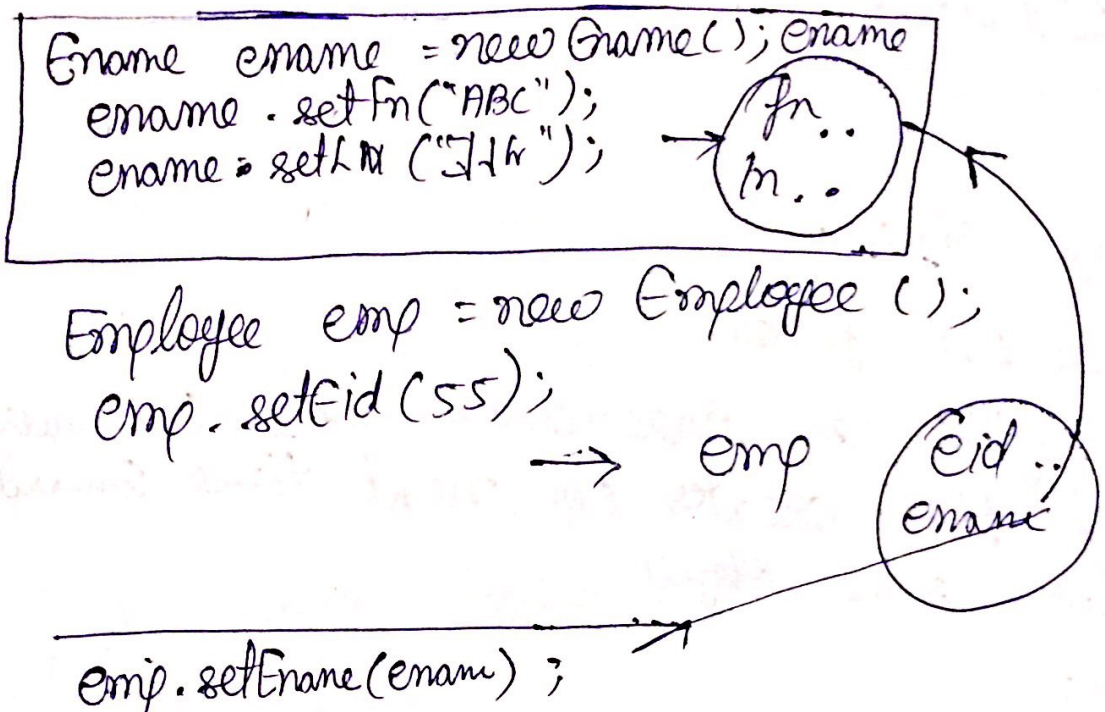
```
<xs : element name = "employee" >
    <xs:complexType>
        <xs:sequence>
            <xs: ename element name = "ename" >
                <xs: complexType>
                    <xs: sequenceType>
                        <xs: element name = "fn" type = "xs: string" />
                        <xs: element name = "ln" type = "xs: string" />
                    </xs: sequence>
                </xs: complexType>
            </xs: element>
        <xs: element name = "eid" type = "xs: int" />
    </xs: sequence>
</xs: complexType>
</xs: element>
```

- After generating classes using xsc. from XSD code will be looking as

```
public class Employee {
    private int eid;
    private Employee.Ename ename;
    static class Ename {
        String fn;
        String ln;
    }
}
```

* In case of Test class to convert obj to XML, create objects in below order (child first, parent next and set child to parent)

Code:



* If any child element contains max occurs = "unbounded" then it is going to created as List type variable in JAXB class.

Ex: employee contains eid (int), version: string (max = unbounded)

```
<xs:element name="employee">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="eid" type="xs:int" />
            <xs:element name="version" type="xs:string" maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>
```

* After generating classes using xjc code looks as below

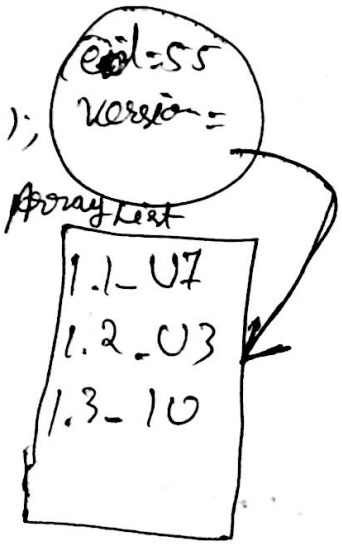
```
public class Employee {
    int eid;
    List<String> version;
}
```

* Creating Object :

Here, List follows ArrayList as implementation class on reading first time creates obj, next time onwards returns same object.

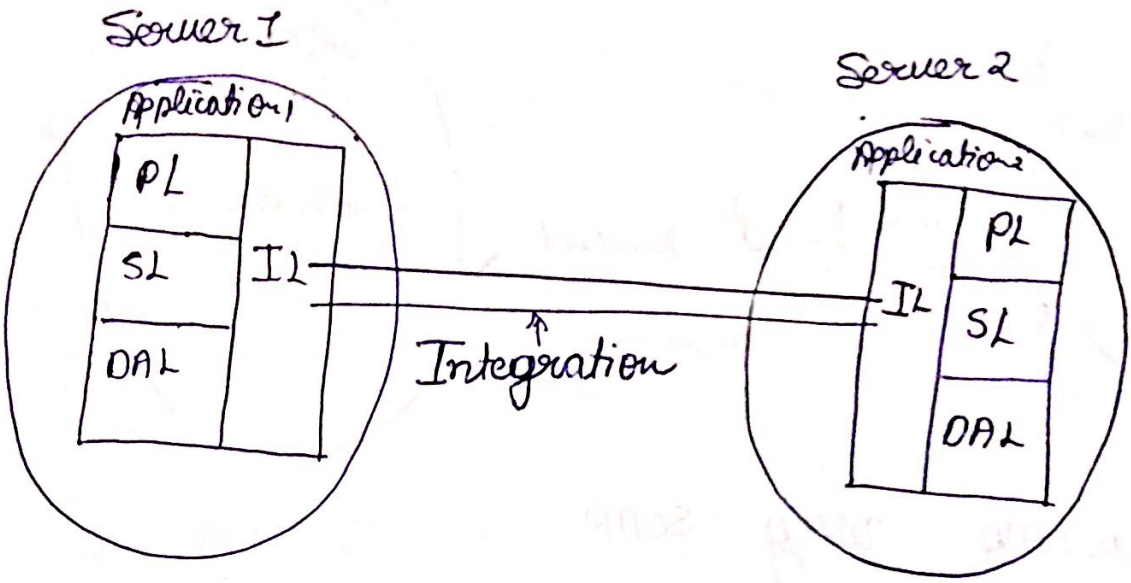
ex:

```
Employee emp = new Employee();
emp.setEid(55);
List<String> verList = emp.getVersion();
verList.add("1.1_U7");
verList.add("1.2_U3");
verList.add("1.3_10");
```



Web Services

Web services are introduced for integration of web applications using IL (Integration Layer)



Web service Implementation

i-> SOAP (SOA)

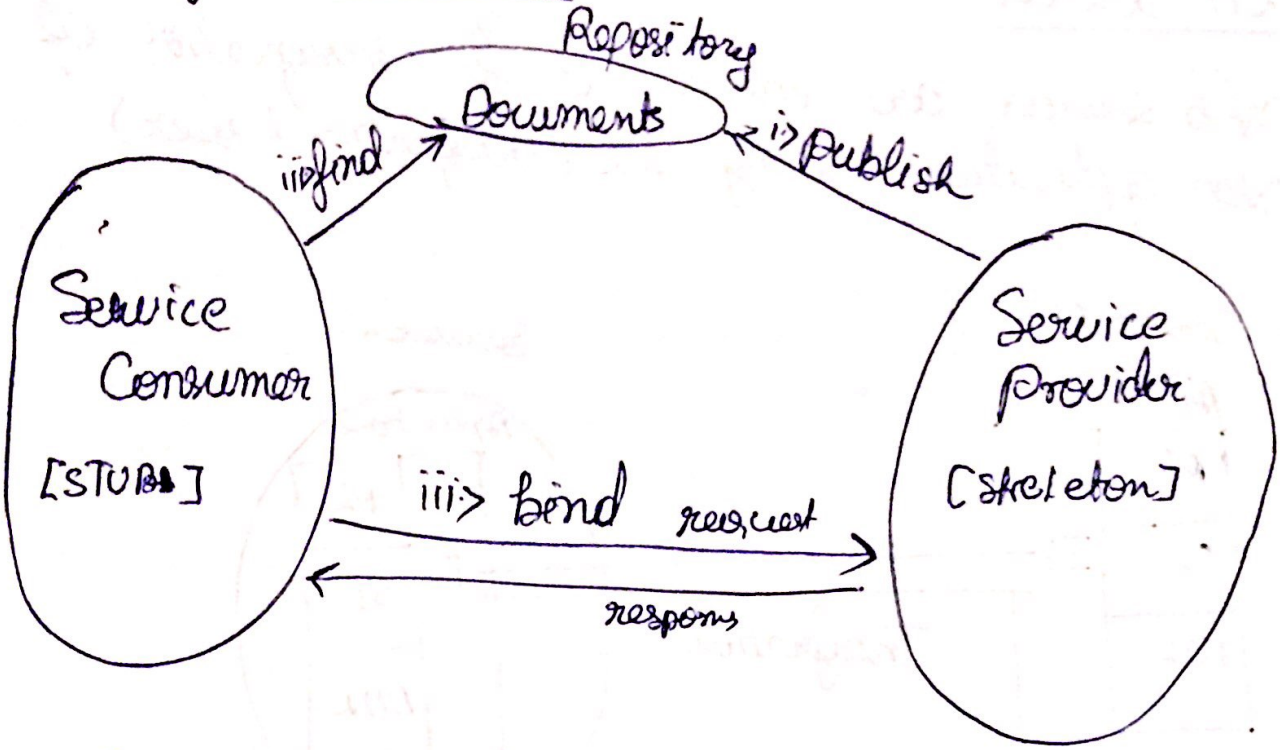
ii-> ReST (client-server)

Simple Object Access Protocol:

It uses XML concept to make connection between 2 applications and to exchange data also.

* XML (XSD/DTD, XML and JAXB) concepts are used in document creation, request, response data sending input data types and order, Skeleton details like class, method, input and output (params, return type.
* It follows SOA. (Service Oriented Architecture)

Design of SOA:



Programming using SOAP

Step 1 :> Define Service Provider

Step 2 :> Generate WSDL document

Step 3 :> Generate Stubs

Step 4 :> Bind Program

Step 1 and 2 are related to service provider application.

Step 3 and 4 are related to service consumer application.

Step I Defining service provider

Go To Eclipse → create a new Java project that represents service provider application

File → New → Java project → enter Name
(Example : provider → Finish

* Create a class under src folder and define methods in class.
* Convert this normal class to webservice by adding 2 annotation.

@WebService (class level)
@WebMethod (method level)

This class is also known as skeleton.

Example:

```
package com.app
@WebService
public class Sample
{
@WebMethod
public String doMsg (int id, String name)
    {
    String msg = "welcome to"+name+" id is "+id);
    return msg;
    }
}
```

* Above code will be used to generate Document (WSDL).

Endpoint is the class, provides a method publish (address, object)

address : String type. Also known as where to publish.

* Obj : object of sevice class, also known as what to publish

Example publisher code:

```
publier class Test
{
public static void main (String ... s)
    {
    Sample s = new Sample ();
    String address = "http://localhost:9988/abcd";
    Endpoint.publish (address, s);
    System.out.println ("done");
    }
}
```

* To view generated WSDL use address and add ?wsdl at end, enter this in web browser.

http://localhost:9988/abcd?wsdl

Address:

* WSDL will be generated using publish operation and it will be placed under given address.
  This address should follow below format
  protocol : //IP: PORT / folder
  example:
  http:// localhost : 9999 /sample
* To identify any service in a system connected to network, we use combination of IP and port Number.
* Os provides logical number to every service to identify in unique manner. This logical number starts from 0 to 65535. Total 65536.
* Zero to 1024 are known as reserved ports by OS.
* One port Number will be given to only one service at a time.
* If service is a server, we can run child sub-service using folder concept.
* recommended port number for publishing is 79000
* To check WSDL is published or not type with ?wsdl in web browser. If it

is running then we can create service consumer.

Service Consumer

- Create a new Java project with name consumer
  → file → new → Java project → enter name
  → ex consumer → finish

Generating Stubs

- Open cmd with src location
  → src → right click - property → copy the location → win + R → cmd →
  cd (location of src)
  change driver if it is different (ex d: enter)
- Now use wsimport to generate stubs
  ex: d:/ wsimport -keep http://localhost:9988/sample abcd?wsdl
- wsimport tool read WSDL file into its own format known as parsing file
- Then it will generate classes and compiles them using Javac.

* go to eclipse project (refresh) to View Stubs.

Step 4 Bind program

Create a class under src and define main method

* follow below steps to write bind program.

Create object of service class (in WSDL)

```
<service  name = "xyz" >
    xyz  x = new xyz();
```

Create object to portType using service object (In WSDL)

```
< service
    <port  name = "_____"
    <portType  name = "abc" >
```

ex

```
portType ob = serviceObj.get<PortName>();
abc ob = x.get<PortName>();
```

call operation using portType obj and get response

print response.

WSDL (Wizz-dull)

WSDL (Web Service Description Language)

It provides details of skeleton (class).

class name, method name, parameters, return type, data types of parameters and return type also order of and count of parameters.

It will be created on publish operation.

It is used to implement service Consumer application.

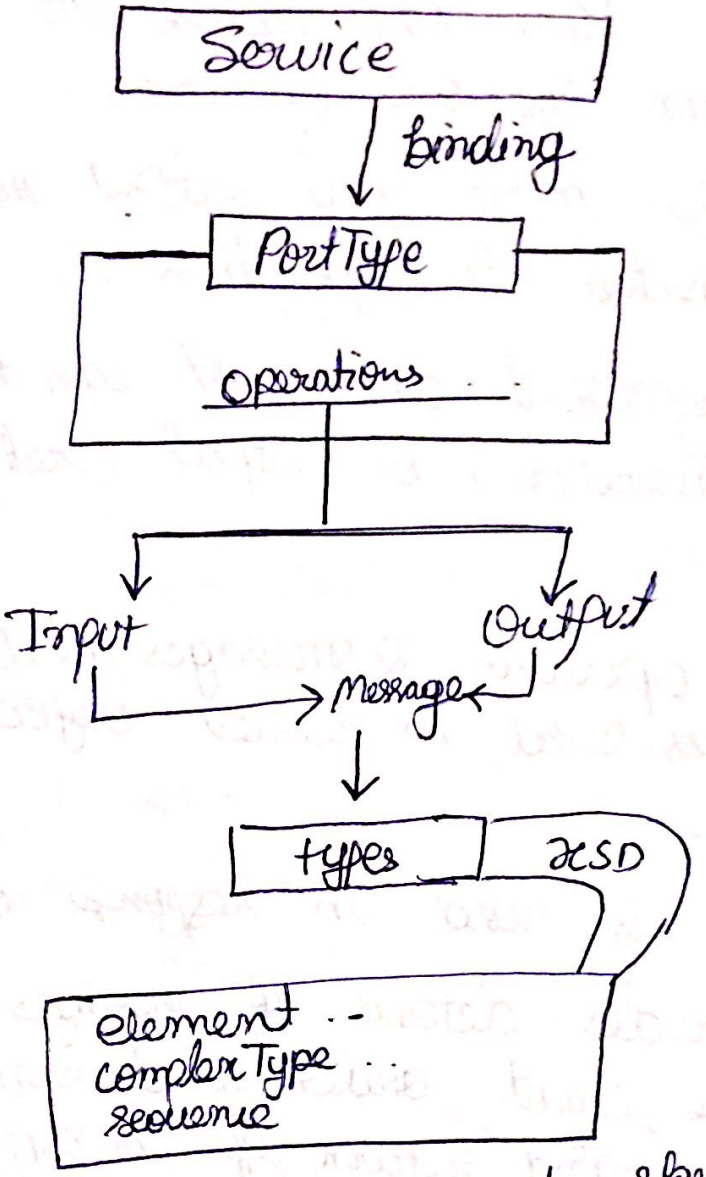
It is used in

- → Stubs Generation
- → bind program implementation
- → request and response construction.

* On modifying any code in service provider we need to re-publish WSDL
* re-generate stubs
* re-write Bind program.

```
<definitions>
  <types> </types>
  <message> </message>
  <portType> </portType>
  <binding> </binding>
  <service> </service>
</definition>
```

Flow of WSDL :-



* Service : It represents skeleton class. It is the starting point of WSDL. This one contains details of binding link to get it's related operation set (portType)
* Binding : It is used to link Service and its operation set.
* It also provides implementation style and encoding support.

* It directly links to portType.
* PortType :- It is used to represent operation set. Here one method in Java class is shown as one operation.

For one operation input and output message links are connected to operation.

* message : It represent data, it can be input (parameters) or output (return type)

→ for one operation, 2 messages will be provided. One is used in request object and (input message).

Output message is used in response object.

* types : It provides details of messages like data type, count, order and name of parameters and return type in XSD format.

* After performing find operation successfully it will generate stubs for
  → service
  → portType
  → message
* These are classes used to define bind program.

* Consider below example

→ Calculation.java
(portType)

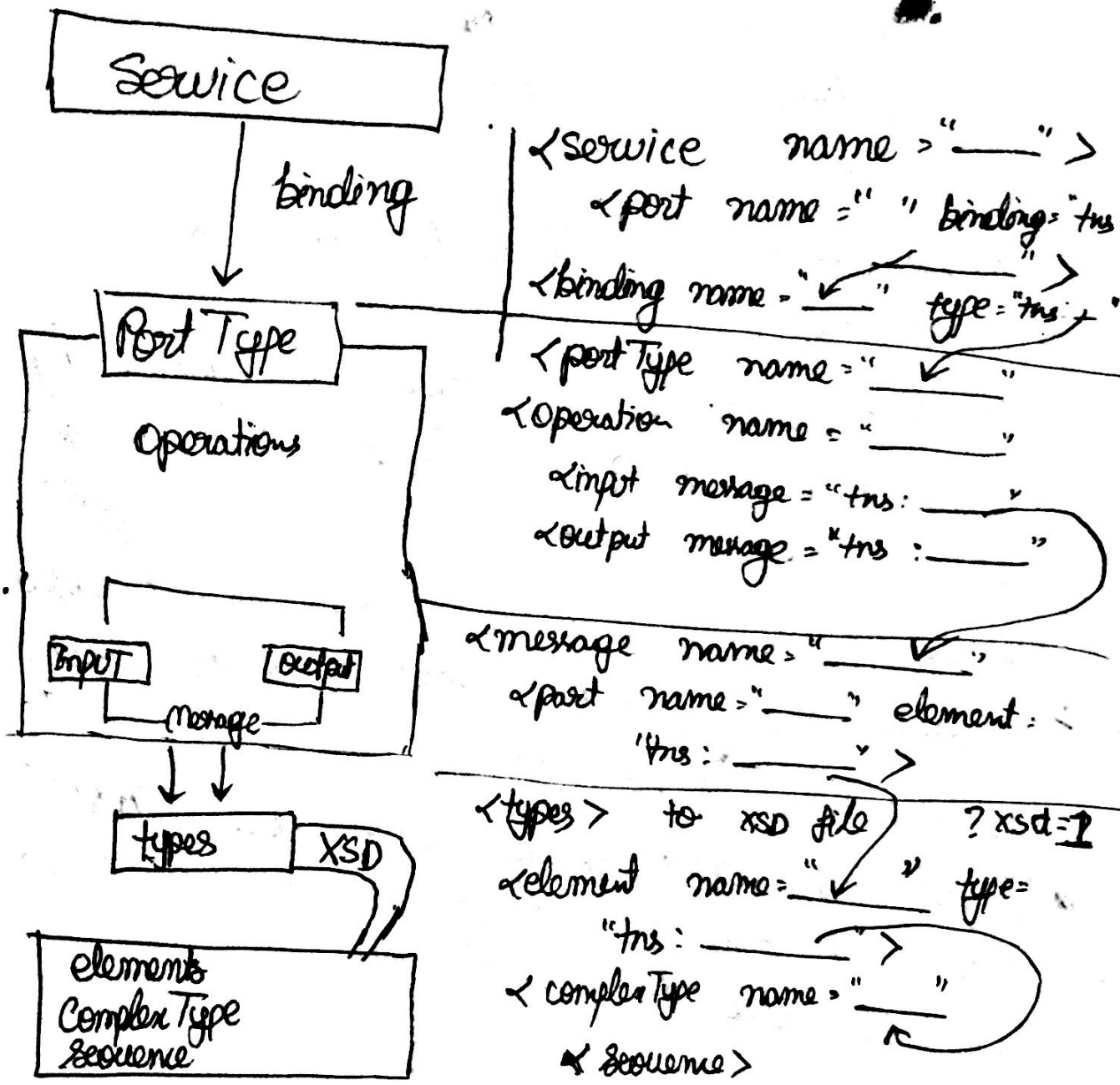
→ CalculationService.java
(Service)

→ DoSum.java and DoSumResponse.java
(message)

for one operation 2 messages are given by WSDL and these 2 are classes when we generate stubs

Input message class is used in request object creation and output message class is used in creating response object.

* Input and output message classes are JAXB classes so that request or response objects can be converted to XML formats



Step 1 Create Object to service class using
< service name = "_____"

Step 2 Create Object to portType using
< port ___ binding = "tns: _____"
to < binding name = "_____"
type = "tns: _____". and compare type
with < portType Name = "_____".

Syntax :
portTypename ob = servic.get[port name]();

Step 3 call operation using portType obj.
< portType name = "_____"
< operation name = "_____"

Step 4 specify params and return type with below flow:
input / output → message → element →
complexType

** → To get types XSD, copy schemaLocation (or address ? xsd=1) and paste in new tab.

* Here one element represent parameter and another one is return type.
* no. of child elements equals to number of params to operation/method.

* sequence indicates order of parameters.

```
class Details {
    double getSum (int a, int b)
    {
    }
}
```

GetSum getSum = new GetSum();
getSum.setArg0 = 100;
getSum.setArg1 = 200;

```
<Envelope>
  <Head> </Head>
  <Body>
    <getSum>
      <arg0> 100 </arg0>
      <arg0> 200 </arg0>
    </getSum>
  </Body>
</Envelope>
```

GetSumResponse getSumResponse = new GetSumResponse();
getSumResponse.setReturn (2000);

```
<Envelope>
  <Head> </Head>
  <Body>
    <getSumResponse>
      <return> 2000 </return>
    </getSumResponse>
  </Body>
</Envelope>
```

class Message

{

String proces (int, double, String)

{

}

}

@ Test

Proces proces = new Proces();

proces.setArg0(200);

proces.setArg1(300.01);

proces.setArg2("RAM");

<Envelope>

<Head/>

<Body>

<Proces>

<arg0> 200 </arg0>

<arg1> 300.01 </arg1>

<arg2> RAM </arg2>

</proces>

</Body>

</Envelope>

Response

ProcesResponse procesResponse = new ProcesResponse();

procesResp.setProcesReturn("RAM 400.01");

<Envelope

<Head/>

<Body>

<proces> <procesResponse>

<return> "RAM400.01" </return>

</procesResponse>

</Body>

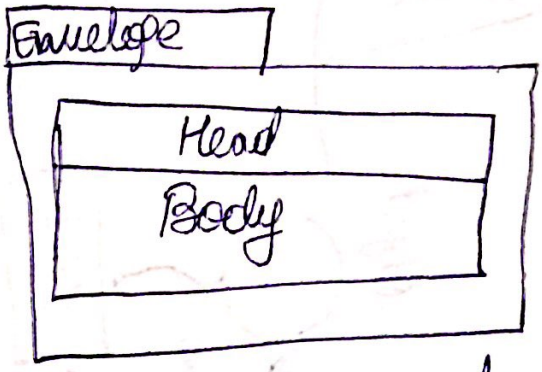
</Envelope>

Bind Flow

* On execution of Bind program, it creates object to input message class and sets data as args (ex: arg0, arg1 ---);
* This message class object also known as request object. This is JAXB class object, so it can be converted to XML
* This XML will be formatted as SOAP request XML and sent to service provider using HTTP request Object.
* -> provider side, reads XML from HTTP and converts again into object format. This object is given to skeleton code.
* Skeleton returns response after processing. This is output message class object, which contains returned value.
* Response object also JAXB class object, so it will be converted to XML and later placed into SOAP body.
* This SOAP XML will be sent to consumer using HTTP response. ~~using HTTPResponse~~
* This XML converted back to response object and returned as response value to Bind program.

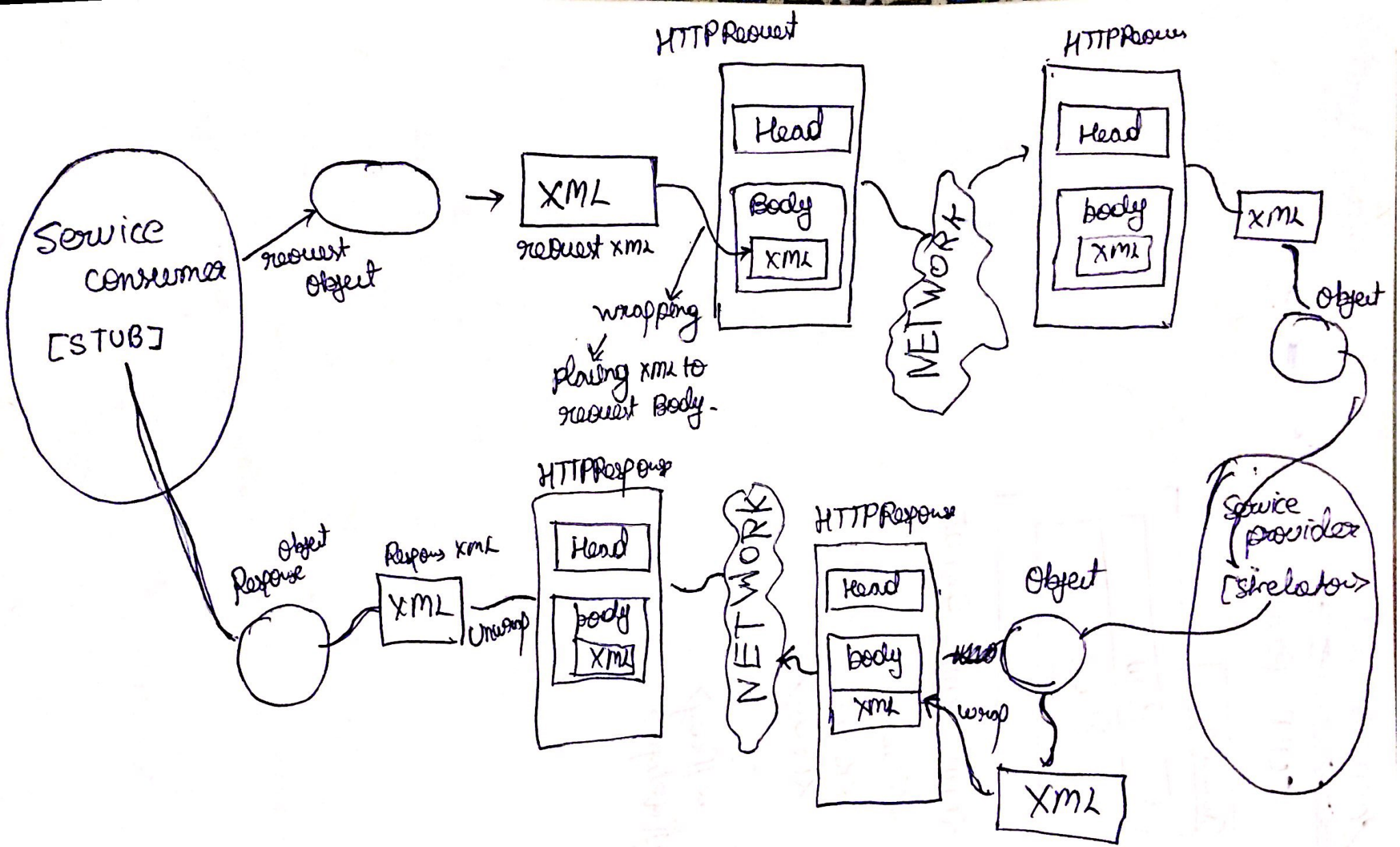
SOAP Format

SOAP Format



SOAP XML Format

```
<Envelope>
  <Head> </Head>
  <Body>
    - - - - -
  </Body>
</Envelope>
```



SOAP + Servlet + JSP Integration

We have to create

i.> Service Provider application (Java application)

ii.> Service Consumer application (dynamic Web project)

i.> Service provider :

Create a project

File → New → Java Project → enter name → finish

ii.> Define Service

Class under src and Publish the service

```
example package com.app;
@WebService
public class Calculation
{
@WebMethod
public int doSum(int x, int y)
{
return x,y;
}
}
```

* publish above code using One address.

```
class Publish
{
public static void main (String..s)
{
Endpoint.publish ("http://localhost:2016/abcd", new Calculation());
}}
```

2.> Service consumer

i.> Configure tomcat server

window → open perspective → other

→ Choose Java EE

→ Observe at bottom of eclipse → server tab

→ right click → new → add server

→ choose tomcat → browse for location

→ finish

ii.> Create a dynamic web project.

file → new → dynamic web project → ex consumer → finish

iii.> Generate stubs under src (use wsimport)

iv.> Define client class

ex package com.app;

public class ClientSample

{

public static int executeRes (int x, int y)

{

CalculationService serv = new CalculationService();

Calculation con = serv.getCalculationPort();

int res = con.doSum(x,y);

return res;

}

}

V> Create a Jsp file under WebContent

→ WebContent → right click → new → Jsp file → Enter name (index.jsp) → finish

ex: <html> <body>
<form action = "sample" method = "post">
<pre>
Id 1 <input type = "text" name = "val1" />
Id 2 <input type = "text" value = "val2" />
<input type = "submit" value = "calculate" />
</pre>
</form> ${key}
</body>
</html>

Vi> Create a servlet class that handles request from UI and provides response through client (Bind);

src → right click → new → servlet → enter package and class name → Next → next → finish

ex: package com.app
public class SampleServ extends HttpServlet
{
public void doPost(req, res) throws IOExn servlet
{

```
int x = Integer.parseInt(req.getParameter("val1"));
int y = Integer.parseInt(req.getParameter("val2"));
int result = clientSample.executeSt(x, y);
req.setAttribute("key", result);
req.getRequestDispatcher("index.jsp").forward(req, res);
}
}
```

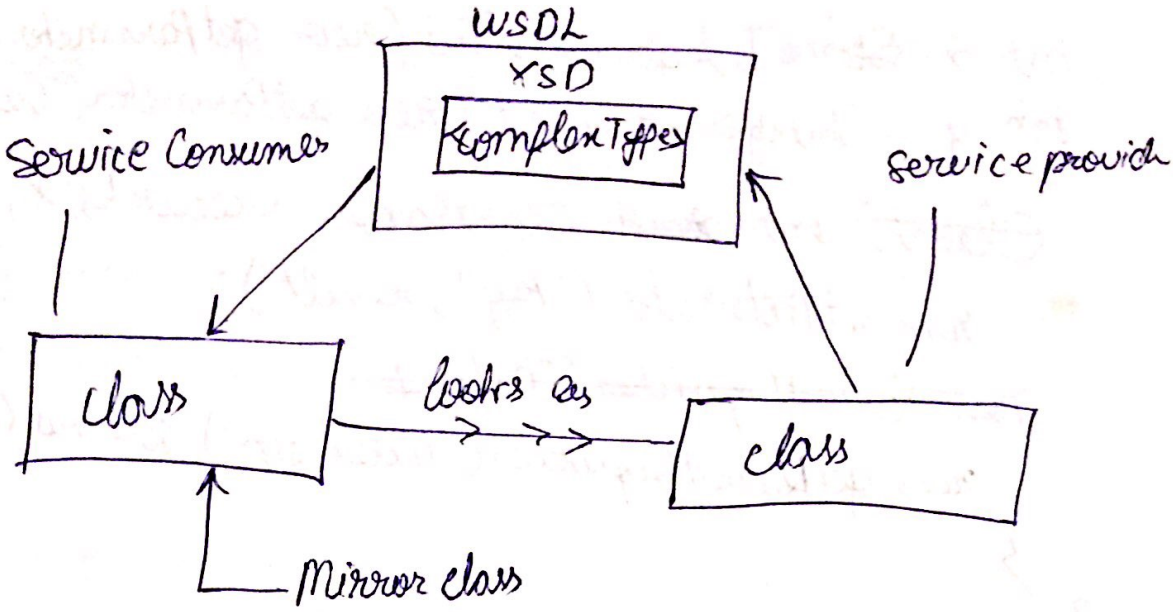
Mirror class: In service provider application for a web method if we choose parameter or return type as class representation, then for this class an equal class will be generated at consumer side as copy to original. This class is known as Mirror class.

* Service provider class will be converted as XSD code (complexType) in WSDL on publishing
* This complexType is re-converted to class at consumer side on stub generation.

Example:

Service provider code:

```
package com.app
@WebService
public class Message {
    @WebMethod
```




```
public String getMsg ( Employee emp)
    {return "welcome to : " + emp.getempId()
        " " +emp.getEmpName () + " " +
        emp.getEmpSal ();
}
}
```

Here Employee is a class. Define as below at provider side :—

```
package com.app
public class Employee {
    private int empId;
    private String empName;
    private double empSal;
// set, get --- to String }
```

* On publishing above code under WSDL → XSD <complexType> will be generated as below

```
<xs:complexType name="employee">
    <xs:sequence>
        <xs:element name="empId" type="xs:int"/>
        <xs:element name="empName" type="xs:string"/>
        <xs:element name="empSal" type="xs:double"/>
    </xs:sequence>
</xs:complexType>
```

Here observe the conversion, class → complexType
primitive variable → element collection → element unbounded
inner class → complexType inside sequence.

* On generating stubs, an equal class will be generated at consumer application side along with other stub classes.

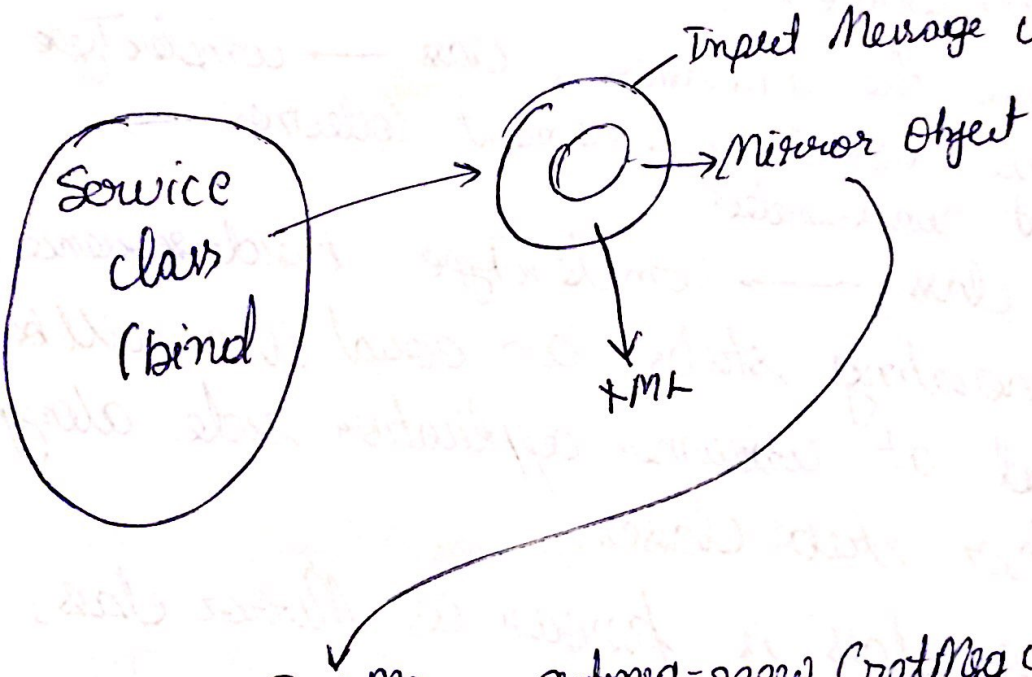
* Here, this class is known as Mirror class, used while calling operation.

Example Bind program

```
package com.app
public class Test
{
    main (String...s)
    {
        Employee emp = new Employee();
        emp.setEmpId(1001);
        emp.setEmpName("Ram");
        emp.setEmpSal(10000);
```

```
MessageService ms = new MessageService();
Message m = ms.getMessagePort();
String res = m.getMsg(emp);
sopln(res);
}
}
```

* Request Object and XML conversion for Mirror




```
GetMsg getmsg = new GetMsg();
Employee emp = new Employee();
    emp.setEmpId(10);
    emp.setEmpName(AA);
    emp.setEmpSal(71.76);
```

```
<getMsg>
    <arg0>
        <empId>10</empId>
        <empName>AA</empName>
    </arg0><empSal>71.76</empSal>
</getMsg>
```

Soap Request

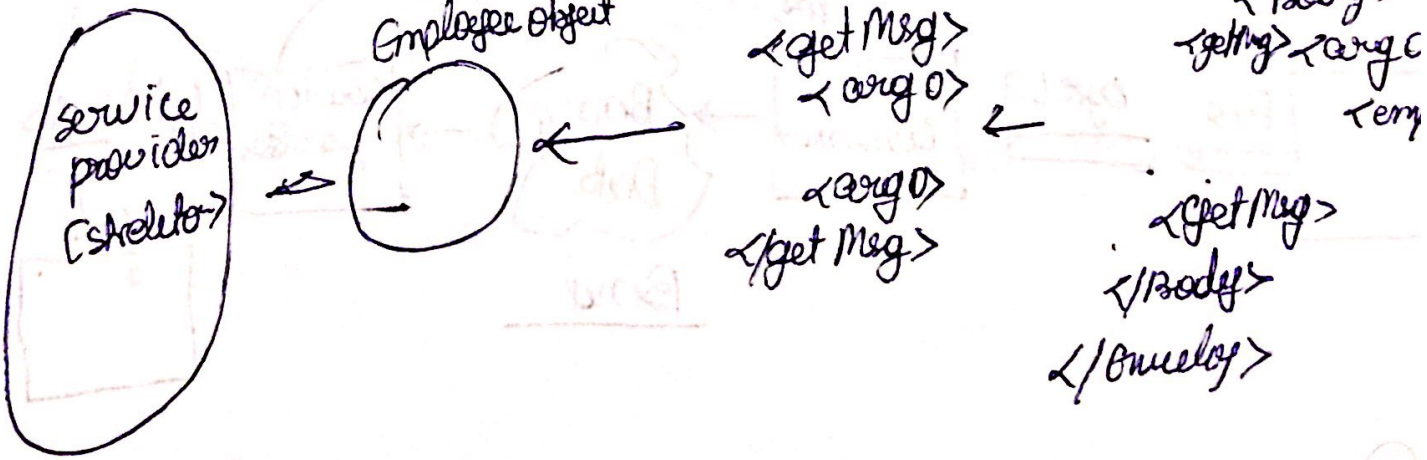
```
<Envelop>
  <Head/>
  <Body>
    <getMsg>
      <arg0>
        <empId> 10 </empId>
        <empName> AA </empName>
        <empSal> 71.76 </empSal>
      </arg0>
    </getMsg>
  </Body>
</Envelop>
```

i:> request object will be generated on executing bind program.

ii:> Mirror object will be created and placed inside request object.

iii:> request object will be converted to equal XML code.

iv:> Converted XML will be placed under SOAP XML Req <Body> (service provide class object)



v:> From Request xml, mirror class xml will be read out by provider app.

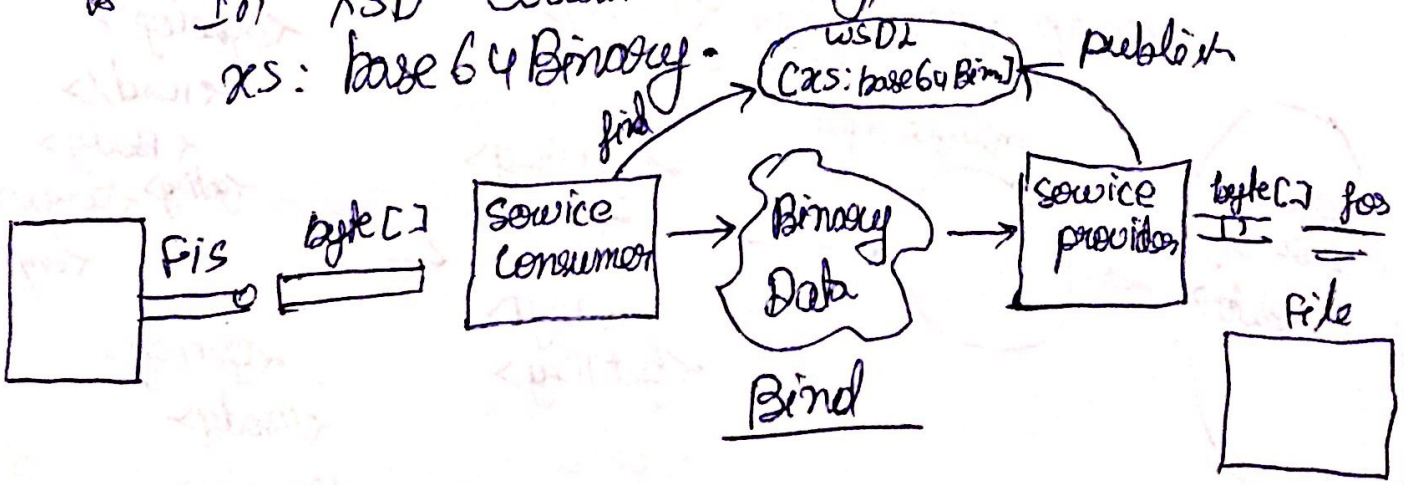
vi:> This xml will be unmarshalled to Provider class object.
(ie converted to original object)

vii:> This object is given to provider to process. (To call operation)

Sending files using SOAP

Soap supports sending binary data using byte[].

* byte[] can be used either param or return type in provider application.
* This type mainly used to send large objects like images, audio, video, documents and lengthy string.
* In XSD equalio datatype for byte[] is xs: base64Binary.



package com.app
Service provider Logic

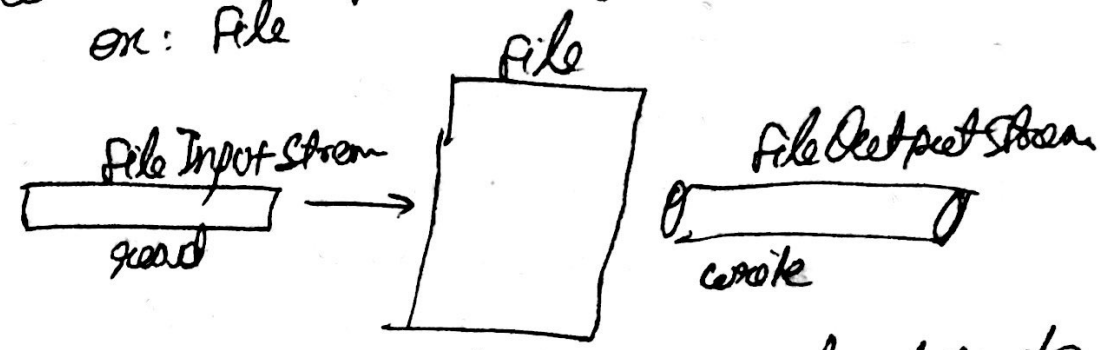
```
@WebService
    public class SampleDate
    {
      @WebMethod
        public String doProcess(byte[] aar, String
                                fname)
        {
          try
          {
            File f = new File("D:/sample/" + fileName);
            FileOutputStream fos = new FileOutputStream(f);
            fos.write(aar);
            fos.flush();
            fos.close();
          }catch(Exception e)
          {
            sop(e);
          }
        return "done";
    }
}
```

* Publish above code and observe addr?xsd=1 data type for byte[].
* Create consumer application and generate stubs.

ServiceConsumer Bind

```
package com.app
public class Bind
{
    public static void Main(String...s)
    {
    SampleDataSevice
    try{
    File f = new File("D:/abc.txt");
FileInputStream fis = new FileInputStream(f);
byte[] arr = new byte[fis.available()];
fis.read(arr);
fis.close();
// Sevice Object
SampleDataService srs = new SampleDataSevice();
SampleData sd = srs.getSampleDataPort();
String res = sd.doProcess(arr, "abc.txt");
sop(res);
}catch(Exception e)
{
sop(e);
}
}
}
```

Note : IO streams are used to read and write data from a object or file



Fis provides available() as int type to find file available size to create byte[].

SOAP using Tools :-

(Eclipse + TOMCAT + AXIS2)

* Apache Axis 2 is designed to automate process of SOAP Implementation.
* It will create as web applications
* Using tools makes process automated like
  - → removes annotations
  - → No manual publishing
  - → No stubs generation by programmer
  - → AXIS 2 provides single stub process with input and output message classes as inner classes
* In bind program, while calling operation, create obj to input message class and pass as param to operation
* operation returns output Message class object. To this call get_return() method, to get returned

value or response.

1> Configuration :

i> download and install jdk 1.7 & tomcat 1.7

ii> open eclipse in JavaEE perspective. Crate server tab → server → right click → new → server → choose tomcat → next → browse for loc → finish.

iii> Download axis2 binary or war distribution from http://axis.apache.org/axis2/java/core/download.html

iv> extract .zip in case binary distribution. Crate eclipse → window → preference → type axis2 → axis2 preferences → browse → select extracted folder → observe succes msg → apply → ok

---

Service Provider Project

File → new → dynamic web project → Enter name and click on config "modify" button → select Axis2 web service check box → OK → next → next → finish.

* create a skeleton class - Example

package com.app

public class MessageSample

{

public String showMsg (int id)

{

return "Hello, this is no:" + id ;

}

}

* publish service:

Right click on project
→ new → other → type web service
→ choose web service → next → Enter skeleton class name or browse and select class → change preference from axis → axis-2
→ next → start server → finish.

* To see WSDL, run as → run on server
→ select service → choose Message sample
→ copy WSDL link.

3) Service Consumer creation

* create consumer project same as provider project with custom (modify) config.
* generate stubs: right click on project → new → other → type web service client → choose this → next → enter WSDL → select Axis2
→ next → next → finish.

* Bind program: ex:

```
public class Bind
{
main(string arg[])
{
try {
MessageSampleStub stub = new MessageSampleStub();
//create input/ message class obj
showMsg input = new showMsg();
input.setId(7878);
//call operation and get output
ShowMsgResponse res = stub.showMsg(input);
//get return value from output obj
String msg = res.get_return();
sop(msg);
}
}
```

* Define UI (JSP and servelets)

Java Equal XSD Datatypes

Java Type	XSD TYPE
byte	xs: byte
short	xs: short
int	xs: int
long	xs: long
float	xs: float
double	xs: double
boolean	xs: boolean
char	xs: unsigned short
Object	xs: any Type
byte[]	xs: base64Binary
class	xs: complex Type

minOccurs = 0
maxOccurs = unbounded
nillable = true

Arr[] (array) — minOccurs=0, maxOccurs=unbounded

List / Set — If no generic

element - entry
minOccurs = 0
maxOccurs = unbounded

Map < — entry (key, value) is not provided for object type as default type.

* If datatype is <generic> collection then consider

BigDecimal — xs: decimal

BigInteger — xs: integer

* If we do not specify any name for parameter of service method, then xsd provides names as arg0, arg1 ----

* To provide meaningful name use
  @WebParam (name = "__")

exampl

```
package com.app;
@WebService
public class sample {
@WebMethod
public String doProcess (@WebParam (name = "id")
    int empId, @WebParam (name = "data") Object ob,
                @WebParam (name = "models") List<Integer> lit,
                @WebParam (name = "mode") char c1,
                @WebParam (name = "vals") double [] arr)
    {
    return "Done";
    }
}
```

xsd service code

```
<xs:element name = "id" type = "xs:int" />
<xs:element name = "data" type = "xs:anyType" />
<xs:element name = "models" type = "xs:int" minOccurs = 0
                maxOccurs = unbounded />
<xs:element name = "mode" type = "xs:~~char~~" unsignedShort" />
<xs:element name = "vals" type = "xs:double" minOccurs = 0
                nillable = "true" />
```

```
public class Sample
{
    @WebMethod
    public String doProcess(@WebParam(name = "id"
      Map<Integer, List> empData,
    @WebParam(name = "data") Map<Integer, Double> d
    @WebParam(name = "fileData") byte[] data)
    {
    return "Done";
    }
}
```

```
<xs:element name="id" type="xs:int"
            minOccurs="0"
            maxOccurs="unbounded" />
<xs:element name="data" type="xs:complexType">
    <xs:sequence>
        <xs:element name="entry"
            minOccurs="0"
            maxOccurs="unbounded">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="key" minOccurs="0" type="xs:int" />
                    <xs:element name="value" minOccurs="0" type="xs:double" />
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>
</xs:element>
```

<xs:element name="fileData"
type="xs:base64Binary"
nillable="true" minOccurs="0" />

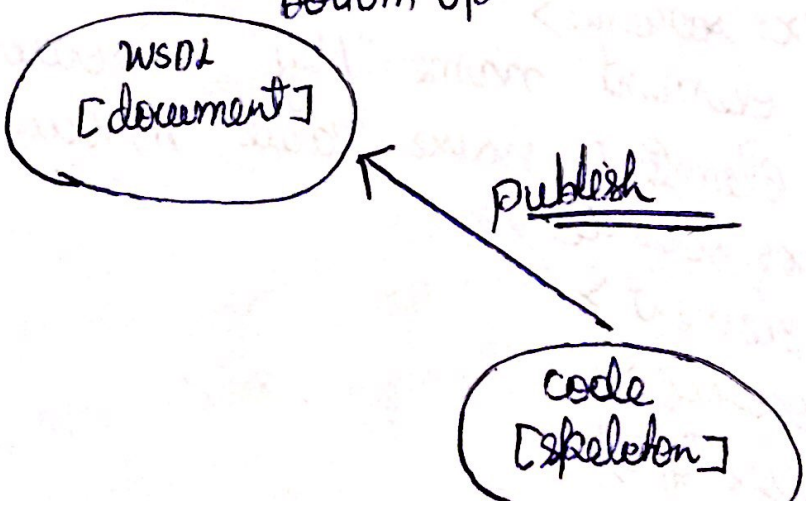
* SOAP Implementation Approaches

SOAP supports implementation service provider in 2 ways

i> Bottom-Up Approach

ii> Top-Down Approach

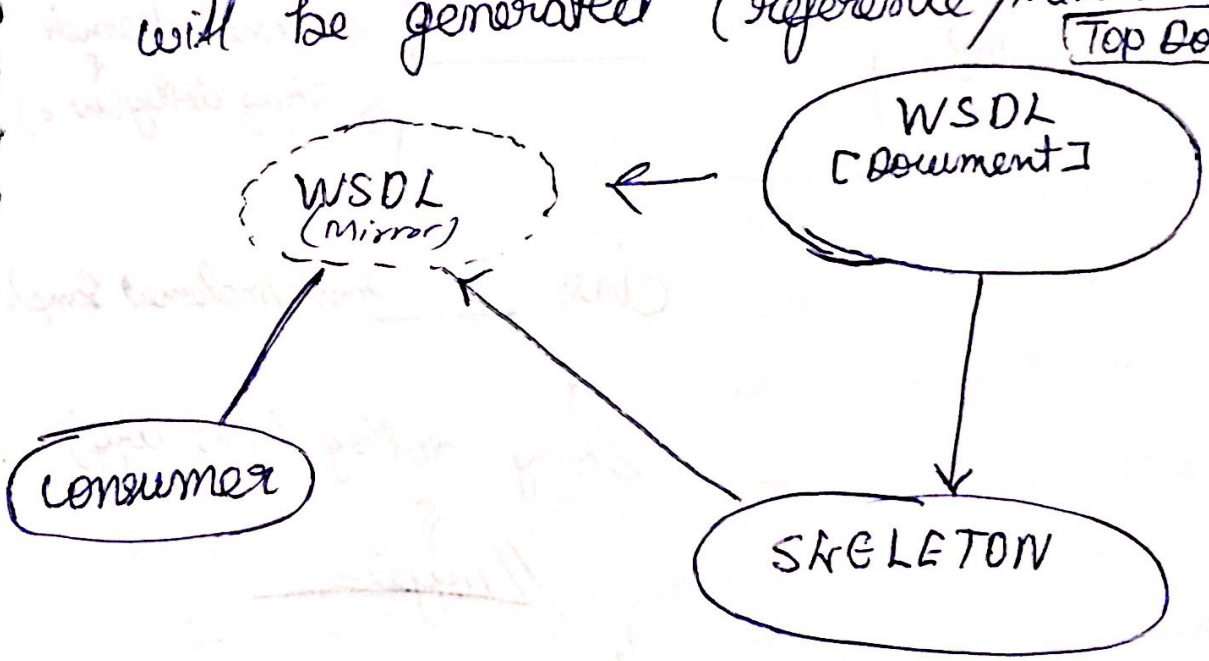
* Bottom-up approach: In this case programmer needs to define code (skeleton) first, this will be used to generate WSDL (wizedull) file.
* This is also called as service first approach.
* In this approach, if skeleton is incomplete or not compiled properly or .class files are missed, then WSDL will not be generated.



ii) Top-down

It is also known as Contract (Document) first. In this approach, we define WSDL (by programmer) and using this generates classes and interfaces with no logic.
(Empty Impl class)

We need to provide logic in the Impl class at same time and copy of WSDL file will be generated (reference / mirror WSDL file).



WSD2 To Code Example

* In WSDL operations are provided with input message and output message.
* This operation will be represented as abstract method in skeleton.

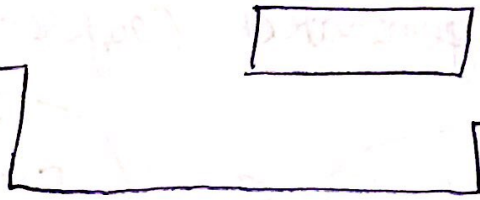
* Remote interface is defined by Sun (oracle) and extended by Service interface which contains operations abstract method.

This interface will be implemented by (__portBindingImpl) Impl class

Example

WSDL ⟶ Interface Remote

(operation) → doMsg



class ____Impl implement Sample

{

String doMsg (int arg)

{

//Logic

}

}

Implementation using Eclipse + Apache Axis

Top Down

provider App

* create new Dynamic web project → file → new → dynamic web project → Enter Name → finish
* Place WSDL file under Web Content folder.
* ~~Right~~ Generate ~~Skeletons~~ Impl and wsdl (also publish at same Time)

Right click on project → new → other → type web service → choose web service → next → choose Type as Top-down → click on browse → browse → select WSDL file → next → start server → finish.

* observe ___PortBindingImpl class and copy WSDL under webContent/wsdl folder.
* Define logic in method which impl from service interface.

Consumer

* create Dynamic web project (ex: consumer)
* Right click on project → new → other → choose web service client → select WSDL from wsdl folder (or enter WSDL address)

* Drag Bar show upto (Test) status as Test client.

→ next → finish.

→ Run project → Run as → Run On Server

→ choose operation link

SOAP UI Tool

* It supports making request using XML, with HTTP, which constructs auto Test cases and request paces.

Download link

https://www.soapui.org/download/soapui.html

* Create and publish one service provider

copy wsdl link

* open soap UI, goto file → new → soap project
  Enter name and wsdl link → click ok
* Observe project created under Navigate pane.
* Expand this to get default request and
* double click on this to open request test window.
* Enter values and click on submit request (▷)
* Observe response in XML or Raw format.
* Response format

```
<Envelope>
    <Header> </Header>
    <Body>
        <Fault>
            <faultcode></faultcode>
            <faultstring></faultstring>
```

Or

```
<...Response>
    <return> -- </return>
</Response>
</Body></Envelope>
```

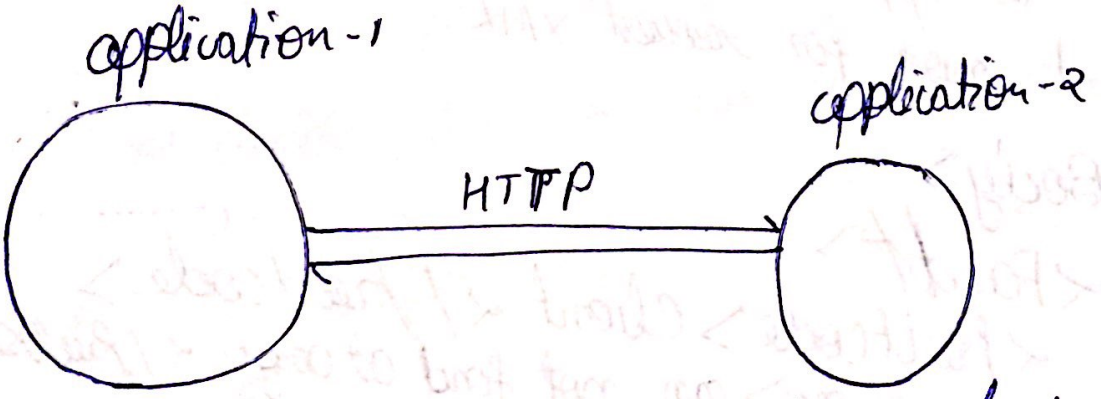
- Here Fault indicates, problem in request processing, it can be client side or server side. It provide rootcause as fault string. If request processed without any problem then Fault will never be shown response XML.
- Fault is applicable only for ~~root~~ Response XML format, not for request XML.

```
<Body>
  <Fault>
    <faultcode> Client </faultcode>
    <faultstring> can not find at cargo </faultstring>
  </Fault>
</Body>
```

ReST Web Services

* Web Services are used to connect two different applications.
* Those applications can be defined in any language.
* Here HTTP is used to exchange data between two different application.
* It supports not only xml data, also plain text, JSON (Java Script Object Notation), Images, files etc....

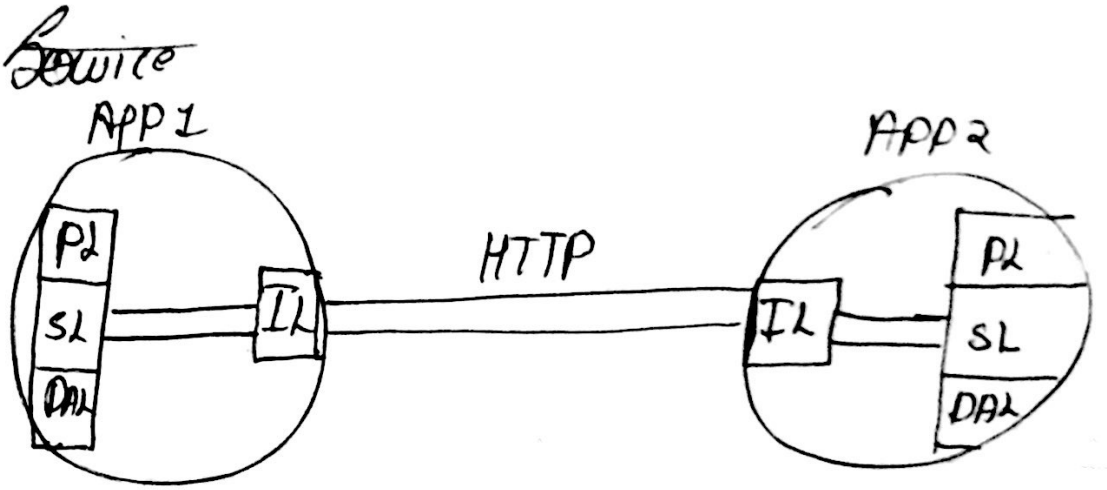
Design:



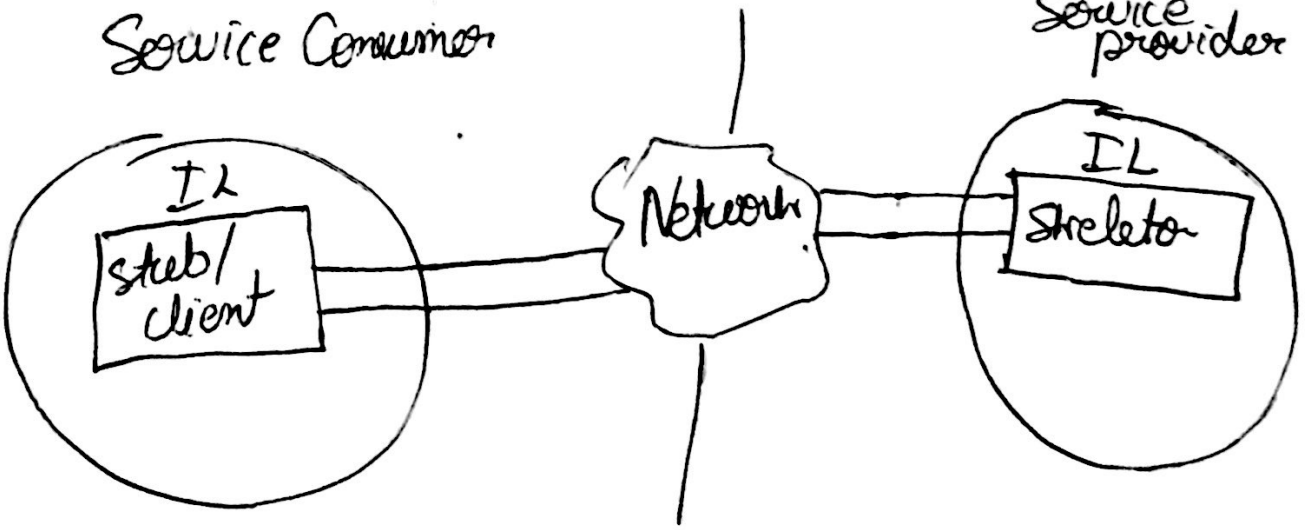
* To create communication link between 2 different application one need another layer required. ie I2 (Integration Layer)
* But only for development of application 3 layers required. Those are
  i-> P2 (presentation Layer)

ii => SL (Service Layer)
iii => DAL (DataAccessLayer)

* Two applications must have IL to get connected with each other.
* In those one is called as service provider and another one is called as service consumer.



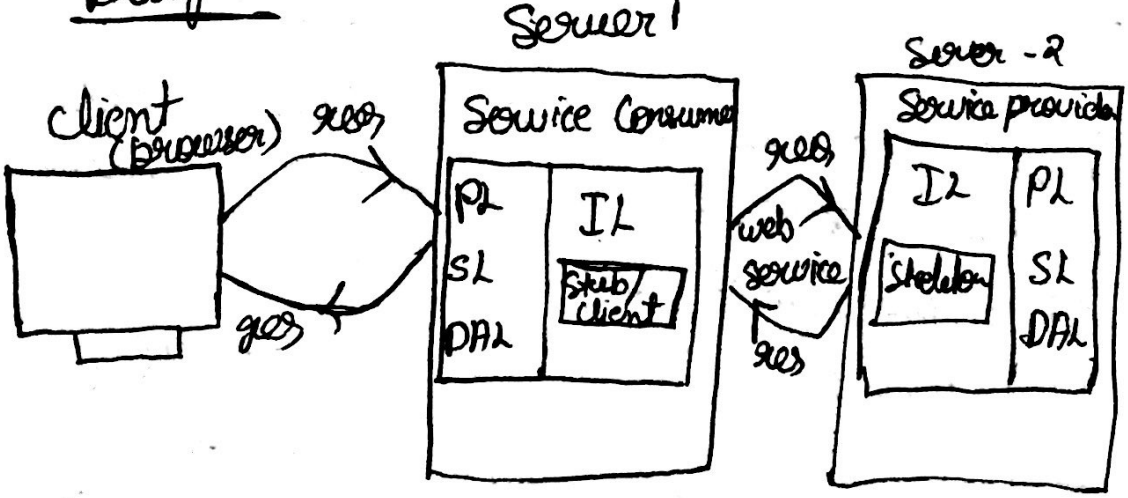
* Service provide Container skeleton code in IL, where service consumer contains stub code/client code in IL.



Web Sowice Architecture

- In this design web sowices are used between two applications placed under two different sowers.
- Client application makes request to sowice consumer application, consumer makes request to sowice provider application. Provider returns back a response to consumer, at last consumer returns response to client.
- This design also named as client-sower or consumer-sowice.

Design:-



Rest :

Re : Representation

S : state

T : Transfer

* It is a webService used to connect two or more application developed in any language.

* It uses HTTP based design :

  Using HTTP, it will send request and gets response with different types of Data

* ReST supports sending data in xml, JSON (Java script object Notation) also primitive using parameters concept.

* Parameters in ReST :

  It support 5 different types of parameters

They are :-

i.> QueryParameters (? &)
ii.> MatrixParameters (;)
iii> PathParameters (/)
iv.> HeaderParameters (HTTP)
v.> FormParameters (HTML - PhysicalForm, ClientClass - LogicalForm).

Form

Here Header Parameters are used to send secure data in Encoded format (Unreadable format)

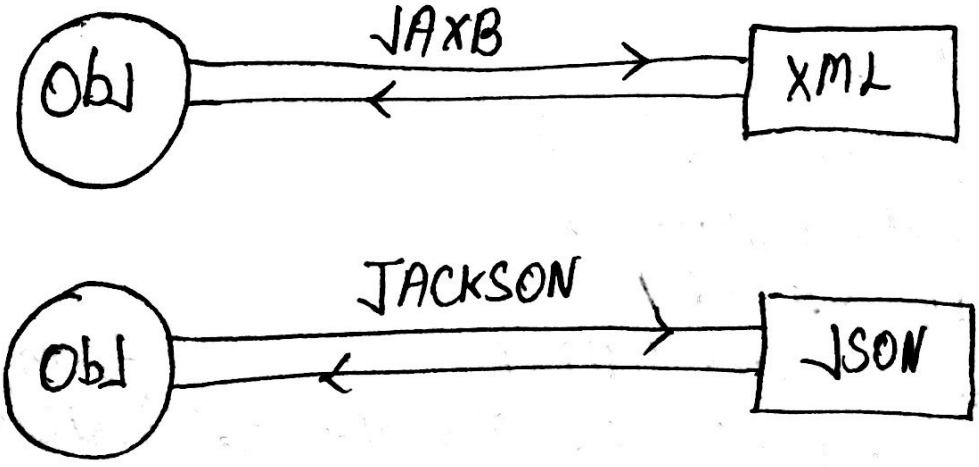
To convert XML to Object format and Obj to XML format, Java provides API.

ies JAXB.

(Java API for XML Binding)

JSON is a light weight concept (less memory) used with ReST to create Integration.

- JSON mainly used to send data from application to application.
- To convert JSON to Object or object to JSON API is Jackson.
- ReST supports integration with other technologies like Jdbc, Sevulets, JSP, Mail Encoding & Decoding etc.



Front Controller Design:

It is used to develop a central request processor design, which implements Light-weight application.

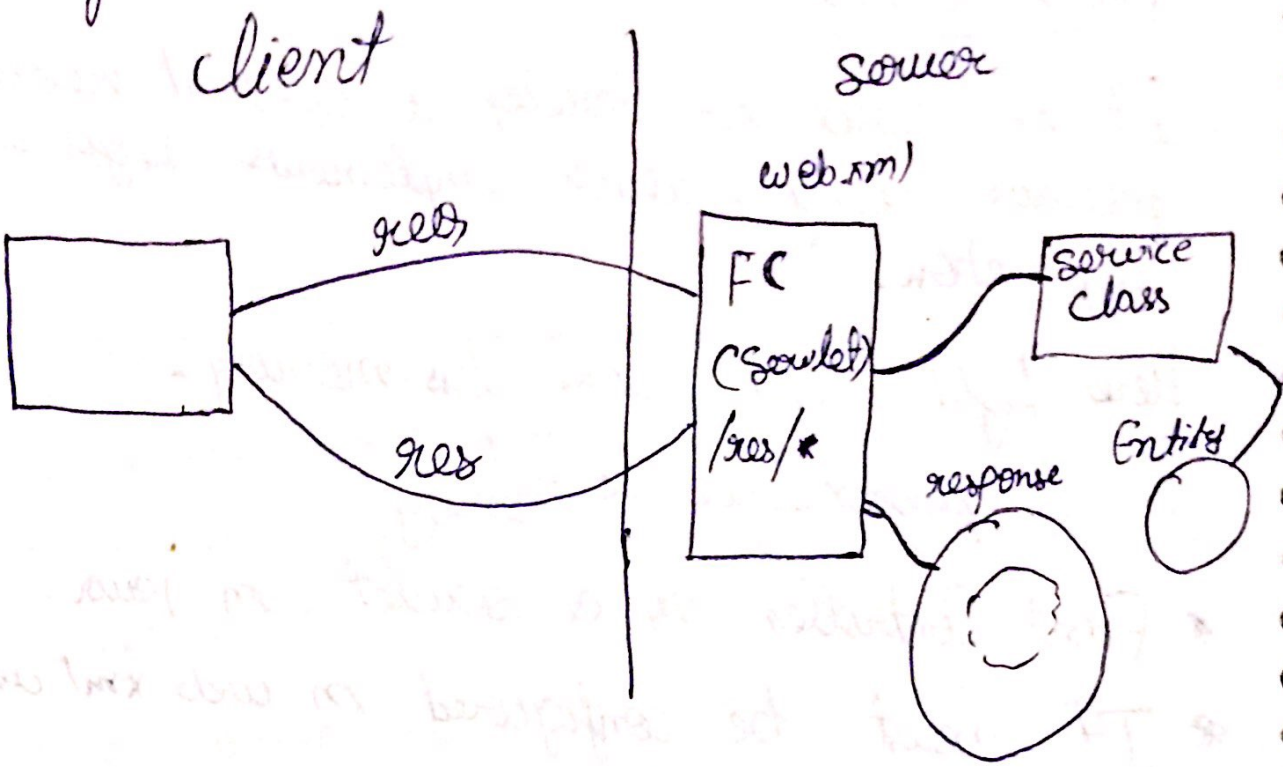
Here, Light-weight means less-memory.

$$\text{performance} \propto \frac{1}{\text{memory}}$$

* Front Controller is a servlet in Java.
* It must be configured in web.xml using directory url pattern.
* Incase of RestT fc is pre-defined servlet name is:

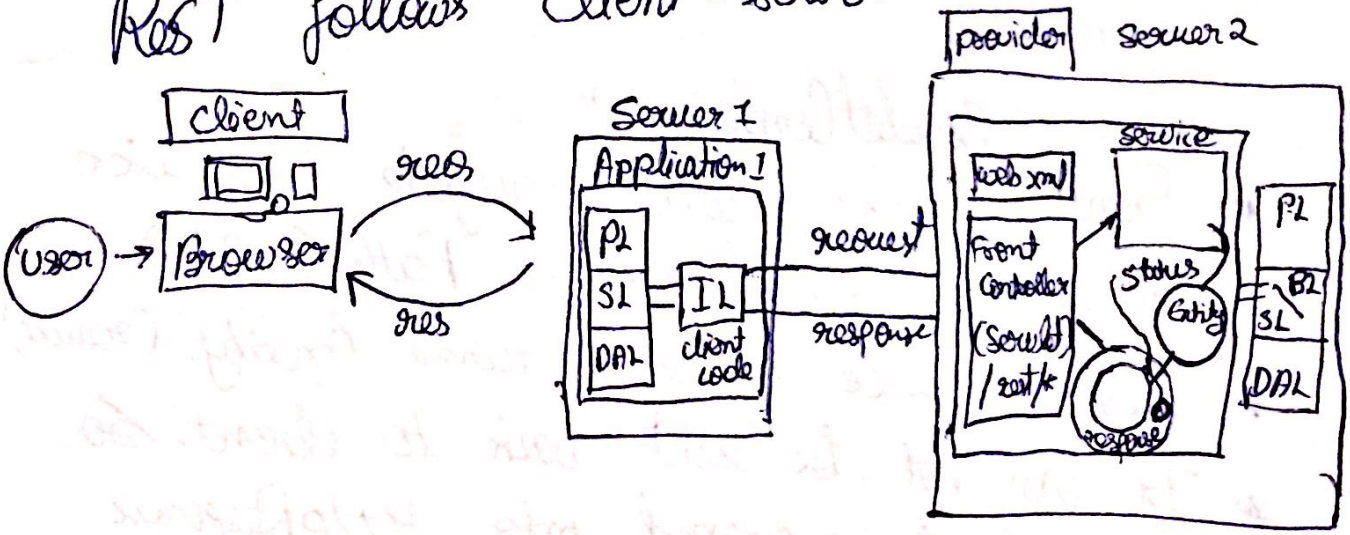
  "ServletContainer"
* FrontController will navigate to Service (provider) class based on Path (URL)
* These service classes returns Entity (result)
* It can not be sent back to client. So, it will be wrapped into HttpResponse Object.
* This Response object sent through FC to client.

Design.



Rest Architecture :-

ReST follows client-server.



Front Controller in case of ReST webservice is a predefined servlet, name is Servlet Container. It must be configured under web.xml using directory panel.

Service classes also known as Service provider classes contains methods to define webservice logic

* Every class and method will be mapped with URL (path).
* Servlet Container identifies Service provider class based on request URL (path)
* Service class method returns entity along with HTTP Status.
* This status indicates action done by server.

ex 200 - Success
400 - Syntax Error
300 - Redirected URL
500 - Internal Server
405 → Method not Allowed.

* Respons object is constructed by FC in HTTP Response format.
* This Response will be return application.
* client application must have REST client program to make request and to get response.
* In above design application-1 placed under server-1 is behaving as client by making request to application-2 placed under Server2.

* Here ReST webservice is used to create link between IL of App1 to App2 IL. It can not be used to develop PL, SL, PL.

Service Provider class / service class :-

* To convert a normal class to webservice provider class use annotations @Path("url")
* @GET/@POST
* @Path can be applied at class and method level. @GET/@POST can be applied at method level.
  These are not default for method type (GET/POST)
* Programmer must specify GET/POST type at every method level.
* Request can be made from browser in 3 ways
  i-> Entering URL in address bar (GET).
  ii -> Form submissions (GET-default/POST)
  iii > clicking on Hyper Links (<a href="..."> (GET).

* If request URL is not matched with Service classes, then front Controller return 404 Resource Not Found.
* If Url is matched, but methodType is not matched, then FC returns 405- Method not allowed
ex: /abc/show req made with type GET, but /abc/show is defined with type POST. Then error is 405.
* Incase of no errors FC returns 200- Success message to client.

Sun Rest Programming

Sun provided API for ReST programming with basic annotations in a package

javax.ws.rs

* frontController of ReST annotations are done using different vendors.
  * Some vendors are Jersey, RestEasy, CXF, RestServers etc
  * On changing vendor web.xml must be modified and also Jars but not Service provider classes.
* Most used vendor in development is 'Jersey'
* Download link of Jersey

  https://jersey.java.net/download.html
* In this page search for

  Jersey 1.19.1 Zip bundle or Jar bundle

Setup:-

i> Open Eclipse in Java EE perspective

windows → open perspective → other → Java EE

ii> Create Web Project

file → new → choose Dynamic web project

→ Enter name → finish

ex: Provider

Coding:

i> Configure FrontController in web.xml using directory url pattern.

ii> In case of Jersey fc servlet class name is com.sun.jersey.spi.container.servlet.

ServletContainer

ex web.xml

```
<web-app>
  <servlet>
    <servlet-name> one </servlet-name>
    <servlet-class> com.sun.jersey.spi.container.servlet.
        ServletContainer </servlet-name>
  </servlet>
  <servlet-mapping>
      <servlet-name> one </servlet.name>
      <url-pattern> /rest/* </url-pattern>
  </servlet-mapping>
</web-app>
```

1) Service Provider class

This class must have been mapped with URLs using Annotation @Path and method must be of type GET/POST

ex: package com.app;

```
@Path("/home")
public class Provider
  {
  @Path("/msg")
  @GET
  public String getValue()
      {
      return "Hello" + new Date();
      }
  }
```

- Execution

Right click on project → Run as → Run On Server → Enter URL upto method level

http://localhost:2016/provider/rest/home/msg

Rules to write Service provider class

* @Path is used at class and method level to specify URL.
* @Path is required at class level.
* @Path is optional for max 2 method, in that one method should be of type GET, another one should be type POST.
* If more than 2 methods are in Service class, then 3rd method onwards must have @Path.
* One Path (url) can be used at max for 2 methods but one should be GET, another one should be POST.

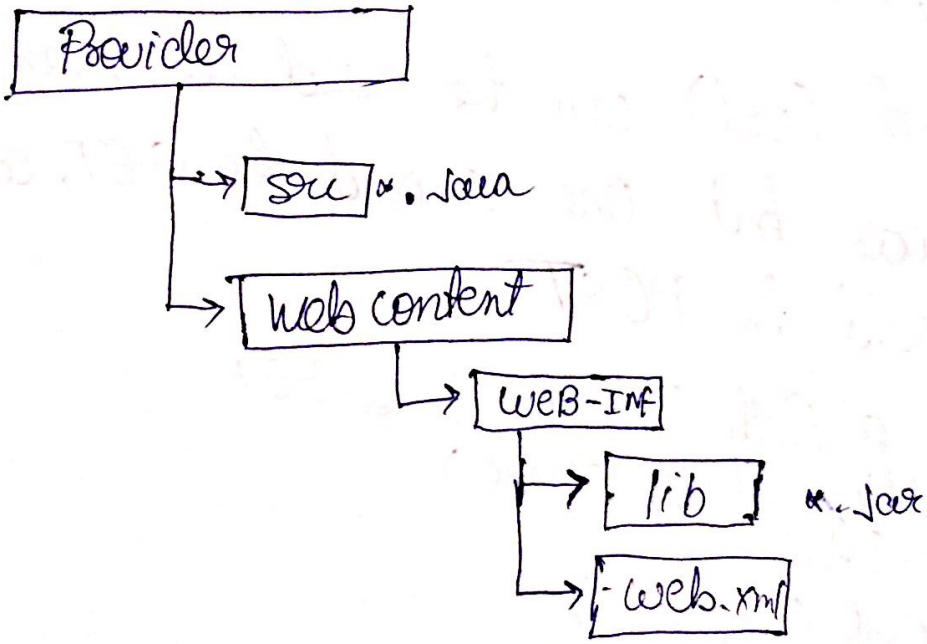
example
```
@Path ("/show")  @GET
public String show()
{ }
@Path ("/show")
@POST
public String cover()
{
}
```

path at class level can also be used at method level.

example:

```
@Path ("/msg")
public class Sample {
 @Path ("/msg")
@GET
public String show()
{
}
```

Folder structure of REST provider :-



All jars must be added in lib folder but not as build path.

Rest client Application (Service Consumer App)

Using Jersey

Step 1 create empty client Object.

Client c = Client.create();

Step 2 Add resource to client object, that returns WebResource.

Here resource (url) type is String

WebResource res = c.resource("http://...");

Step 3 Using resource object make get/post method call and return type should be ClientResponse with Entity.

ClientResponse cr = r.get(ClientResponse.class);

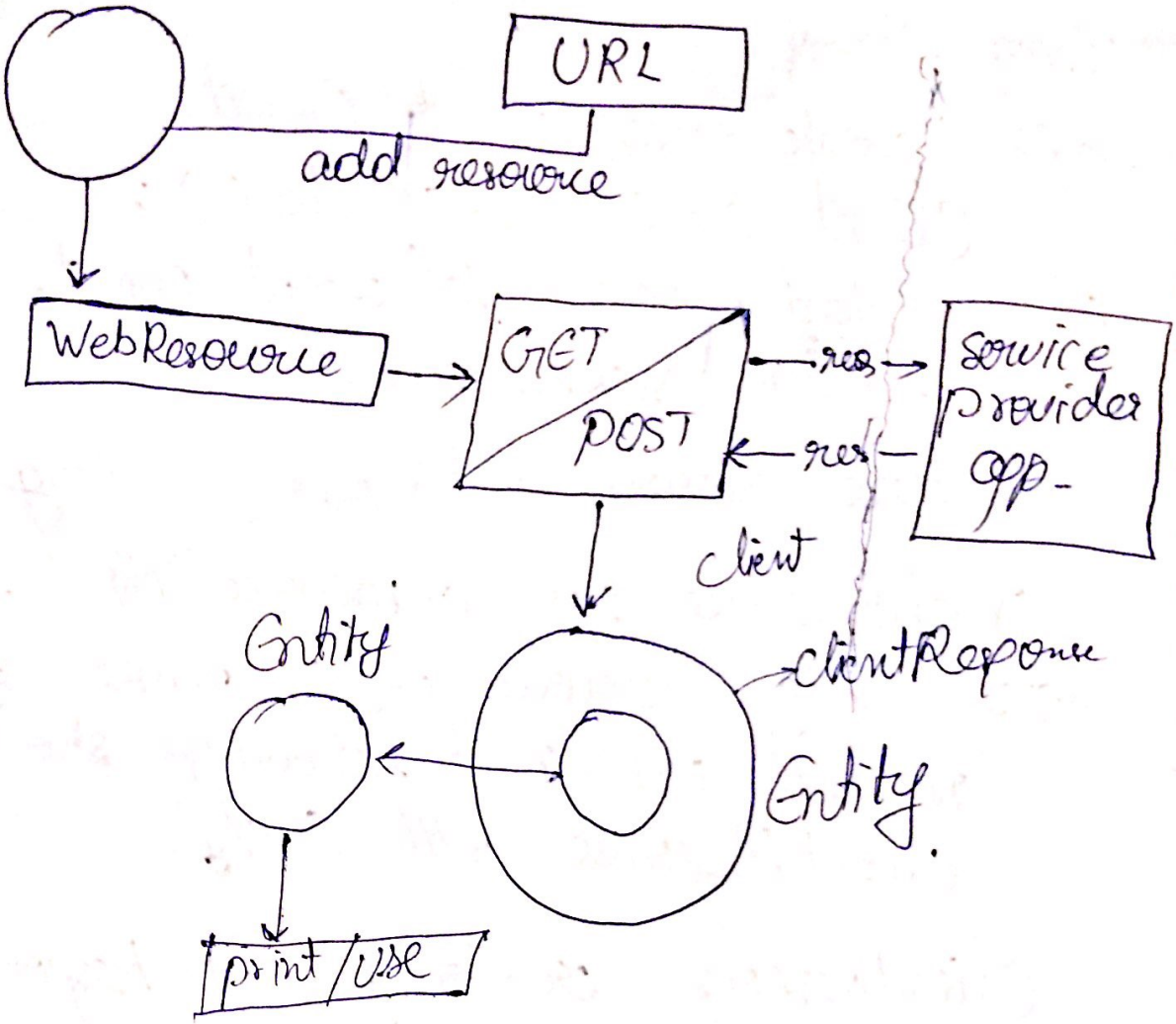
Step 4 read entity as a type from cr obj

String s = cr.getEntity(String.class);

Step 5 print/use entity in application.

- Client Design :

client obj



* Service Consumer example project:

Step 1: create a new Java Project

File → new → Java Project → Enter name

Step 2 add Jars as buildPath

→ right click on project → buildpath → Library tab → Add External Jar → Select Jersey Jars → Finish

Step 3: Define TestClient with provider url.

Error

```
public class Test {
    main (String ... s)
    {
    String url = "http://localhost:8086/provider/rest
                  home/show";

    Client c = Client.create ( );
    WebResource r = C.resource (url);
    ClientResponse cr = r.get (ClientResponse.class)
    String s = cr.getEntity (String.class);
    sop (s);
    }
}
```

* To execute above program press ctrl+F11 or Run (Menu) → Run.
* Server should be started before making request.

http://facebook.com/groups/thejavatemple
javabyraghu@gmail.com.

Response status

On making request from client Application, provider return client response which contains Entity and also "status"

Status is int type, also known as HttpStatus

For client Application

```
// client Object
// resource object
// client Response Object
// int status = cr.getStatus();
```

Status 200 → success

Status 404 → URL Not exist or case is not matched (Upper/lower)

Status 405 – service is GET, But request is POST (or service-POST req-GET)

If provider application is not reachable due to invalue

IP: PORT or server is down then client throws Exception as connectionException connection refused.

Parameters

Parameter concept is used to send data from consumer to provider only it will send data along with URL

i:> QueryParameter

Data is sent using url in key-value pair, both are of type string by default.

Reading this value at provider side done using

@QueryParam("key") localvariable

@QueryParam always support type conversion like String → int

@QueryParam("eid") int id

* Support sending multiple parameter in any order, binding is done based on keys not based or order
* If we donot send any key-value, FC provides default values to a local variable.
* If extra param are sent then are ignored by FC
* If key.value is sent is non-convertable format then FC throws 404, Resource not found.

example Service provider

com.app

```
@Path ("/home")
public class sample {
  @Path ("/msg")
    public String show (@QueryParam("eid") int id,
        @QueryParam("ename") String name,
        @QueryParam("esal") double sal)
    {
    return "Hello:" + id + "," + name + "," + sal;
    }
}
```

Request

→ http: - - - /home/msg

output Hello

→ - - - /home/msg? ename=A & esal=9.6 & eid=7

output Hello 7 A 9.6

: ... /home/msg? EID=7 & ENAME=AA & esal=9

output Hello: 0, null, 9.2

URL case sensitive (even parameter also)

Eid, eid are differ, Eid is ignored

// how / msg ? eid = 98

output 404

```
public class Test
{
  public static void main (String ... s)
  {
    String S;
```

```
client c= Test.create();
WebResource rs = c.resource(url);
ClientResponse rs = rs.get(Client.Reponse.class)
String s: rs.getEntity(String.class)
  sop(s);
```

Sending query param request from client application

= we resource object and call queryparam(key, value) webResource to add one param to request it must be added before get/post call and after resource object creation.

```
public class Test
{
public static void main()
{
String url: "http://_____/hem/msg";
```

Step 1

```
client c= client.create();
WebResource r = c.resource(url);
        r = r.queryParam("eid", "100");
        r = r.queryParam("ename", "AA");
        r = r.queryParam("esal", "98.6");
ClientResponse cr = r.get(ClientResponse.class)
String s: cr.getEntity(String.class)
   sop(s);
}
```

2.7 Matrix Parameters

These are from restful web services, introduced to increase performance of application.
Query param symbols ? & are overloaded.
Where ; is used for terminations
* MatrixParams reduces time execution for parameter processing. Other than this matrix param is similar like Query Param

→ Note:-
Matrix param sends data in key-value pair and both are strings.
Supports type conversion also.
Order is not required while sending multiple parameters. Binding is done based on keys but not on order.
Return 404 if types is mismatched.
Returns default values if no data is sent in request.
Extra key-values are just ignored by FC.
Example program:
Service provider class

```
package com.app;
@Path("/home")
public class Sample
{
@Path("/msg")
@GET
public String show(@MatrixParam("eid") int id,
@MatrixParam("ename") String name, @MatrixParam
("esal") double sal) {
```

return "Hello +" + id + "," + name + "," + sal;

}}

example req:

i-> http: /hom/msg

output: 0, null, 0.0

ii-> ./home/msg ; ename = A; esal = 9.3, eid = 8

output: 8, A, 9.3

iii-> /home/msg ; eid = 9

output: ~~9, null, 0.0~~ 404

iv-> /home/msg ; sid = 87

output 0, null, 0.0

v-> /home/msg ; eid = AA

out: 404

```
@Path("/msg/{id}/{name}")
public String show(@PathParam
("id") int sid, @PathParam("name")
String name)
{
return ename + " " + sid;
}
}
```

req

i-> /home/msg/98

output null, 98

ii-> /home/msg/7/AA ~~0~~, AA 7

/hom/msg

~~0, null~~ 404

Hello: 0

m1 = /abc/id/xyz

m2 = /abc/{id}/xyz

m3 = /abc/{id}/{xyz}

m4 = /{abc}/{id}/xyz

m5 = /{abc}/id/{xyz}

i-> /300/id/xyz — m4, (m5)

ii-> /abc/31/xyz — m2, m3, m4

iii-> /45/51/91 — ~~m4~~ 404

iv-> /400/id/985 — m5

v-> /abc/abc/abc — ~~404~~ m3

vi-> /id/abc/xyz — ~~404~~ m4

vii-> /id/abc/abc — 404

viii-> /id/id/id — m5

ix-> /xyz/abc/id — 404

/{abc}/id/xyz — m4, m5

Path Parameters: Data can be sent along with Url as path which represents data without key sent in order.

* It uses / as separator to send data.
* Path represents static format ex: /id
  dynamic format ex: /{id}

Dynamic path indicates data will be sent in that position

* Static path must be same as request including case (upper & lower)

example @Path("/id/{name}")

Here id is static, name is dynamic path.

* If 2 paths are matched then priority is given to highest static count. Even count is same the first static path is selected.
* If type not matches or not path is matched then FC returns 404
* Path Param must follow order. If not data will be copied into wrong variables or type mismatch may occur.

example service provider class

```
Package com.app;
@Path("/home")
public class Sample
{
@Path("/msg/id")
@GET
public String show()
{
return "Hello:O";
}
}
```

```
@Path("/msg/{id}")
@GET
public String show(
@PathParam("id") int id)
{
sop("this is sid" + id);
return "id";
}
```

* Join Query are used to select data from multiple Tables by executing a single select operation.

* In hibernate application we can write SQL Joins statements or HQL Join statements

* Joins are 4 types

i-> inner Join
ii-> Left / left outer Join
iii -> right / right outer Join
iv-> full Join

Inner Join returns only matched data from both sides of a Join statement

Left Join reads matched data and also unmatched data from left side of Join statement

Right Join reads matched data and unmatched data from Join statement

Full Join reads both matched data and unmatched data from both sides of a Join statement.

Example 1

The following query reads Doctor name and patient name using One To Many relationship with SQL statement.

SQL
```
Select doctor.doctorname, patient.patientName from
doctor inner Join patient on doctor.doctorid
    = patient.doctorid-fk
```

The following statement is a HQL To read Doctorname and patientname using One To Many

example 2
```
Select d.doctorName, p.patientName from Doctor d
inner Join d.patients p
```
patients → collection

example 3 The following Join is HQL with Many To One to read DoctorName and patientName
```
Select d.doctorName, p.patientName from Patient
p Join p.doctor d
```
parent reference

example 4 The following HQL and SQL query reads DoctorName and PatientName using One To Many with left Join.

select doctor.doctorname , patient.patientname from doctor left Join patient on doctor.doctorid = patient.doctorid-fk

Select d.doctorName , p.patientName from Doctor d left Join d.patients p.

SQL Joins statement can be executed by constructing SQLQuery object

HQL Join statement can be executed by constructing Query Object

SQLQuery query = session.createSQLQuery ("select doctordoctorName — — ");

List list = query.list();
Iterator it = list.iterator();
while (it.hasNext())
{
Object ob[] = (Object[]) it.next();
sop (ob[0]); } doctor name
sop (ob[1]); } patient Name
}
}

Query query = session.createQuery (select d. doctorName _____)
List list = query.list();

```
@Entity
public class Student
{
  @Id
  @Column(name="sid")
    private int studentId;
  @Column(length=10)
  private String studentName;
  private int marks
  @Version
    private int version;
  setters // getters
}
public class ClientOfUser:
  {
    main(String..s)
    {
    Student s = (Student) sessio.get(Student.class,100);
    String str = JOptionPane.showInputDialog("Enter new m");
    int m = Integer.parseInt(str);
    s.setMarks(m);
    tx.commit();
  }
```

Copy the same files to other workspace also and create a student table with 4 columns and insert 1 record and commit.

```
Create table student (sid int primary key, sname
                      varchar(10), marks int, version int);
insert into student values(1010,'a',500,0);
commit;
```

execute client of user1.java, it opens a dialogue box, execute client of user2.java, it opens a dialogue box. Enter input in User1 dialogue box then click Ok. Now, the transaction is commited successfully.