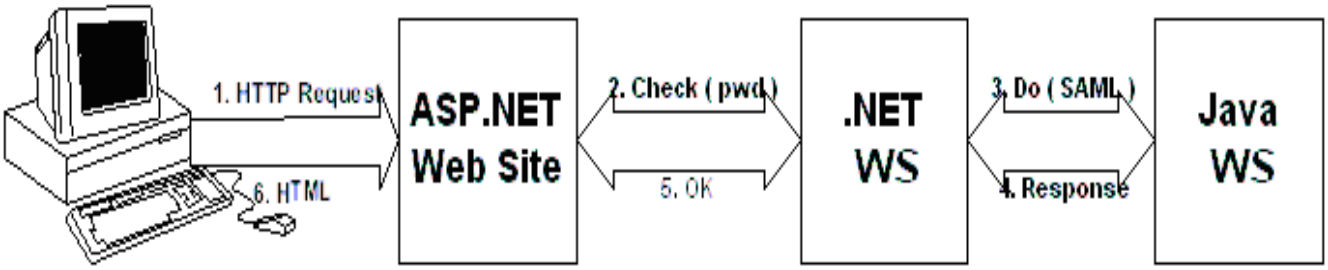
Web services

WebServices:

Web service is a technology to communicate one programming language with another. For example, java programming language can interact with PHP and .Net by using web services. In other words, web service provides a way to achieve interoperability.

Interoperability

The promise of web services lies is the interoperability between services that are running on different platforms and implemented using different programming languages. A web service created in a JAVA platform can be invoked by any client application implemented in different programming language like ANDROID,.NET . As web services are defined in terms of service interfaces based on standards like XML, SOAP and WSDL, implementation of web services using Java or .NET is matter of realization of the specification.



WebServices Introduction:

- For developing a distributed applications in JAVASE firstly introduced socket programming .
- In socket programing at server applicatoin a socket object is created with port no and in client application a serversocket object is created with ip address of the machine and port no .
- With the help of serversocket and socket objects, both client and server applications are communicated and exchanged the data across the network.
- If server ip address is changed from one ip address to other then clients application can't find the server ,or client system connects to more systems then it will burden on developers to

change server ipaddress at each server location.

- To resolve this introduced some other technologies like

  RMI(Remote method invocation),
  CORBA(Common object request broker architecture),
  EJB(enterprise java bean),
  Naming registry etc.....

All these are failed to fulfilling the interaperability.

After they introduced new technology as webservices with support of ws-i organization.

Webservices is an independent distributed technology which doesn't have any link with previous technologies like EJB,CORBA.

Webservices is independent of platforms and technologies and servers.

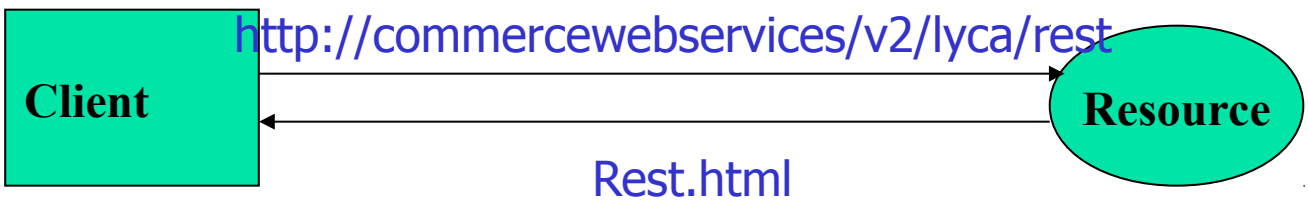
- web services are categeroised into two types
  1.soap based webservices
  2.restful webwebservices

Restful webwebservices (REpresentational State Transfer):

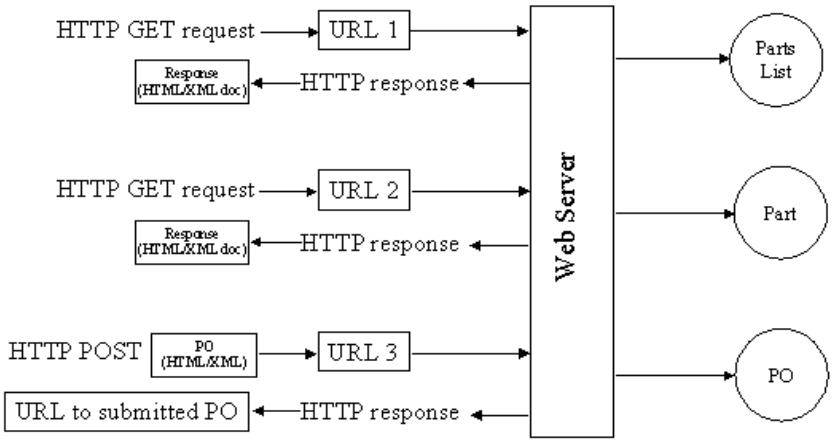
What is REST ?

REST is a term coined by Roy Fielding to describe an **architecture style** of networked systems. REST is an acronym standing for Representational State Transfer.

Why is it called Representational State Transfer ?



REST way of Implementing the web services:



REST is more dynamic, no need for creating and updating UDDI.

- REST is not restricted to XML format. REST web services can send plain text, JSON, and also XML.
- restful is lightweight so no need to write any xml and binding cls's.
- REST **uses URL to expose business logic**.
- **JAX-RS** is the java API for REST web services.

- no any specifacations for rest but in soap we need to follow some specifications.
- SO REST **more preferred** than SOAP.

JAX-RS:

Java API for RESTful Web Services (**JAX-RS**) is a Java programming language API that provides support in creating web services according to the Representational State Transfer (REST) architectural pattern.

Difference between jax 1.0 and jax2.0

In jax rs api 1.0 it provides only webservice for our class but not included the api for creating client if u want create a client api we need to use java networking api.

In jax 2.0 they included both.

In Jax rs 1.0 version:

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

<param-name>com.sun.jersey.config.property.packages</param-name>

In Jax rs 2.0 version:

<servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>

<param-name>jersey.config.server.provider.packages</param-name>

See Reference: Jersey 2.0 Documentation

Rules to make our cls as restfulwebservice :

Every rest cls must contain a default constructor and cls must be public.

we need to use annotations in our classes for restful features

- @Path specifies the relative path for a resource class or method.

- @GET specfies reading the data.
- @POST specfies storing the data.
- @PUT specfies update the data.
- @DELETE specfies deletethe data.
- @Produces specifies the response format.
- @Consumes specifies the accepted request Internet media types.

- @PathParam binds the method parameter to a path segment.
- @QueryParam binds the method parameter to the value of an HTTP query parameter.

Messages in rest client :

200 OK
General success status code. This is the most common code. Used to indicate success.

201 CREATED
Successful creation occurred (via either POST or PUT). Set the Location header to contain a link to the newly-created resource (on POST). Response body content may or may not be present.

204 NO CONTENT
Indicates success but nothing is in the response body, often used for DELETE and PUT operations.

400 BAD REQUEST
General error for when fulfilling the request would cause an invalid state. Domain validation errors, missing data, etc. are some examples.

401 UNAUTHORIZED
Error code response for missing or invalid authentication token.

403 FORBIDDEN
Error code for when the user is not authorized to perform the operation or the resource is unavailable for some reason (e.g. time constraints, etc.).

404 NOT FOUND
Used when the requested resource is not found, whether it doesn't exist or if there was a 401 or 403 that, for security reasons, the service wants to mask.

500 INTERNAL SERVER ERROR
Never return this intentionally. The general catch-all error when the server-side throws an exception. Use this only for errors that the consumer cannot address from their end.

JAX-RS Example Jersey:

We can create JAX-RS example by jersey implementation. To do so, you need to load jersey jar files or use maven framework.

In this example, we are using jersey jar files for using jersey example for JAX-RS.

Download the jersey jars in follow link

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

1)create a dynamic web project

Create the following class

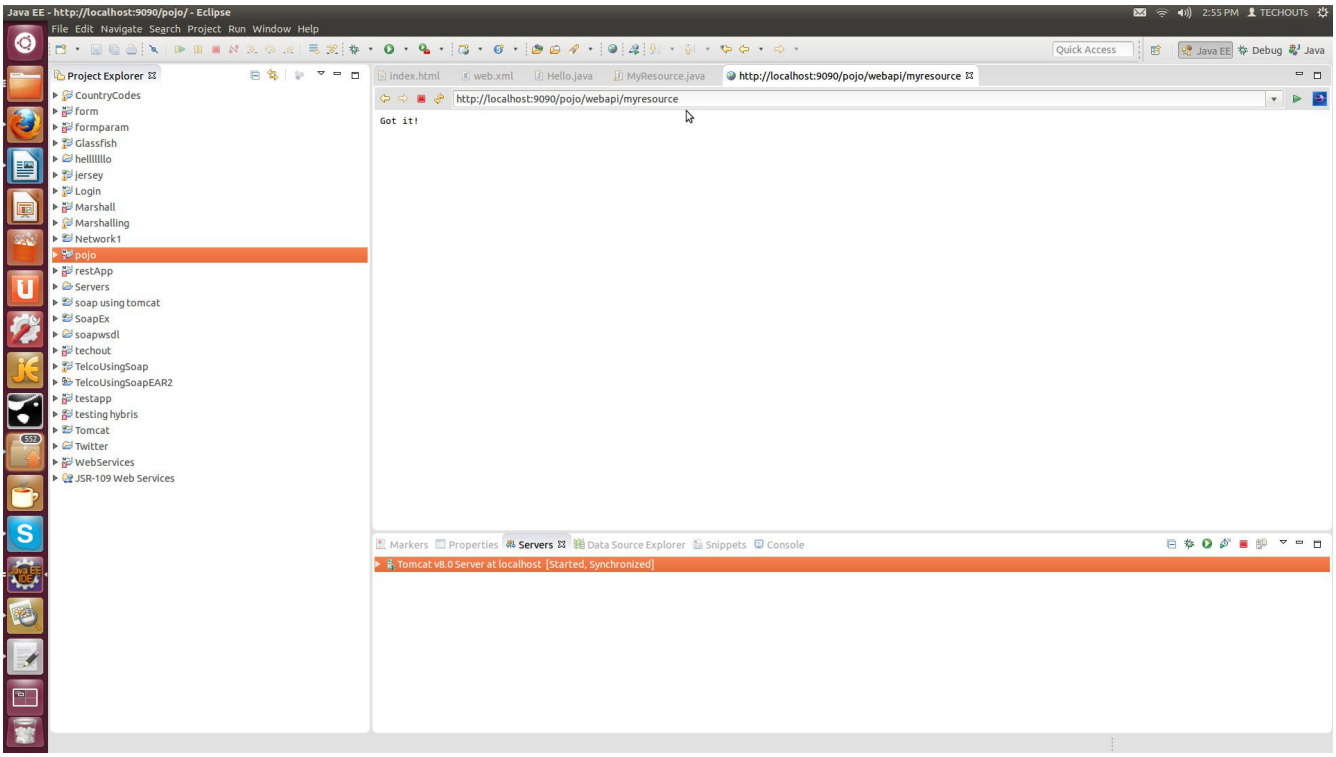
```
package jersey;

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

@Path("/myresource")
public class Hello {



  @GET
  @Produces(MediaType.APPLICATION_XML)
  public String jersey() {
    return "Hello Jersey Plain";
  }

}
```

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"
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>com.vogella.jersey.first</display-name>
 <servlet>
    <servlet-name>Jersey REST Service</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
     <!-- Register resources and providers under com.vogella.jersey.first package. -->
    <init-param>
       <param-name>jersey.config.server.provider.packages</param-name>
       <param-value>com.vogella.jersey.first</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Jersey REST Service</servlet-name>
    <url-pattern>/webapi/*</url-pattern>
  </servlet-mapping>
```

```
</web-app>
```

run the aboue program :



Using maven:

Example 1:

1)create maven project

add the archetype as

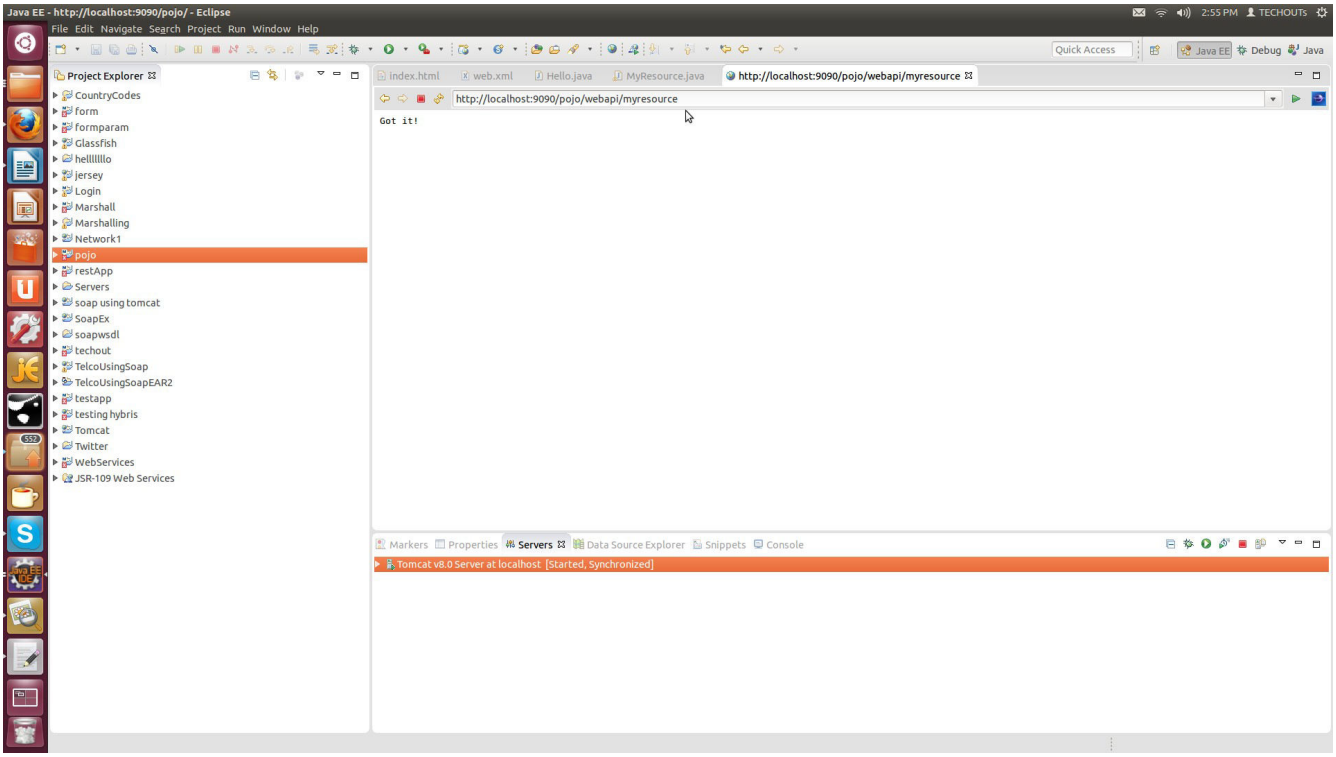
Group Id org.glassfish.jersey.archetypes

Artifact Id:Jersey-quickstart-webapp

Version:2.16

resource class

right clk on your application and run it will display like below



Soap based webservices (Simple Object Access Protocol)

SOAP is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. SOAP uses XML technologies to define an extensible messaging framework, which provides a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other

implementation-specific semantics.

- **JAX-WS** is the java API for SOAP web services.
- SOAP **uses services interfaces to expose the business logic**
- SOAP **defines standards** to be strictly followed.

- Webservices is a distributed technology introduced for developing interaparable communication between the client and the server applications.

- Client and server can be in different languages ,client data can't undderstood by server and server response can't underatood by client directly so xml is taken as a mediator for transfering data .

- If client and server sending their own xml's format one cant understood the other's xml,in order to resolve this WS-I Organization released soap specification.

- In soap based webservices client and server need to accept common soap xml format for transfering the data.
- SOAP stands for Simple Object Access Protocol. It is a XML-based protocol for accessing web services.

Along with the soap specification ,soap namespace also given elements,and attributes and datatypes,which are require for constructing soap request xml and response xml.

A namespace uri for soap is

“http://schemas.xml/soap.org/soap/envelop/”

a soap request and response called as envelop,because root element of soap xml is Envelop.

A soap request xml contains two parts one header and body,header is optional but body is mandatory.

SOAP Request

```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Header/>
   <S:Body>
      <ns2:UserDetails xmlns:ns2="http://tech.com/"/>
   </S:Body>
</S:Envelope>
```

SOAP Response

```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Header/>
   <S:Body>
      <ns2:UserDetailsResponse xmlns:ns2="http://tech.com/">
         <return>welcomenull</return>
      </ns2:UserDetailsResponse>
   </S:Body>
</S:Envelope>
```

WSDL: Web Service Description Language:

what is wsdl:

Web Service Description Language
WSDL is a document written in XML
The document describes a Web service
Specifies the location of the service and the methods the service exposes

The Main Structure of WSDL:

```
<definition namespace = “http/… “>
        <type> xschema types </type>
        <message>  … </message>
        <port> a set of operations </port>
        <binding> communication protocols </binding>
        <service> a list of binding and ports </service>
<definition>
```

Wsdl file contains five sections

- types section
- message section
- porttype section
- binding section
- service section

Types section:

in this types section an xml schema will describe simple elements and complex elements of our service class.
Ex:

```
<types>
  <schema targetNamespace="http://example.com/stockquote.xsd"
    xmlns="http://www.w3.org/2000/10/XMLSchema">

    <element name="TradePriceRequest">
      <complexType>
        <all>
          <element name="tickerSymbol" type="string"/>
        </all>
      </complexType>
    </element>

    <element name="TradePrice">
      <complexType>
        <all>
          <element name="price" type="float"/>
        </all>
      </complexType>
    </element>

  </schema>
</types>
```

Message section :

input and output parameters we need to pass in message section.

Ex:

```
<message name="SayHelloRequest">
  <part name="firstName" type="xsd:string"/>
</message>

<message name="SayHelloResponse">
  <part name="greeting" type="xsd:string"/>
</message>
```

Port type section:

this section of wsdl contains interface and class name ,its methods and their input and output messages.

ex:

```
<portType name="Hello_PortType">
  <operation name="sayHello">
    <input message="tns:SayHelloRequest"/>
    <output message="tns:SayHelloResponse"/>
  </operation>
</portType>
```

Binding section:

binding section tells a service requester about which transport protocal to be used and also which messsage exchanging format is required to call operations of a webservice.

Ex:

```
<binding name="Hello_Binding" type="tns:Hello_PortType">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/ >
  <operation name="sayHello">
    <soap:operation soapAction="sayHello"/>
```

```
    <input>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:examples:helloservice" use="encoded"/>
    </input>

    <output>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:examples:helloservice" use="encoded"/>
    </output>
  </operation>
</binding>
```

Service section:

service section of wsdl file describes a network address location of a webservice.

```
Ex:
<service name="Hello_Service">
  <documentation>WSDL File for HelloService</documentation>
  <port binding="tns:Hello_Binding" name="Hello_Port">
     <soap:address
       location="http://www.examples.com/SayHello/">
  </port>
</service>
```

Example WSDL file:

```
<!-- Published by JAX-WS RI (http://jax-ws.java.net). RI's version is JAX-WS RI 2.2.9-
b130926.1035 svn-revision#5f6196f2b90e9460065a4c2f4e30e065b245e51e. -->
<!-- Generated by JAX-WS RI (http://jax-ws.java.net). RI's version is JAX-WS RI 2.2.9-
b130926.1035 svn-revision#5f6196f2b90e9460065a4c2f4e30e065b245e51e. -->
```

```
<definitions targetNamespace="http://rest.com/" name="HelloWorldImplService">
<types>
<xsd:schema>
<xsd:import namespace="http://rest.com/" schemaLocation="http://localhost:4111/hai?xsd=1"/>
</xsd:schema>
</types>
<message name="getHelloWorldAsString">
<part name="parameters" element="tns:getHelloWorldAsString"/>
</message>
<message name="getHelloWorldAsStringResponse">
<part name="parameters" element="tns:getHelloWorldAsStringResponse"/>
</message>
<portType name="HelloWorld">
<operation name="getHelloWorldAsString">
<input wsam:Action="http://rest.com/HelloWorld/getHelloWorldAsStringRequest"
message="tns:getHelloWorldAsString"/>
<output wsam:Action="http://rest.com/HelloWorld/getHelloWorldAsStringResponse"
message="tns:getHelloWorldAsStringResponse"/>
</operation>
</portType>
<binding name="HelloWorldImplPortBinding" type="tns:HelloWorld">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
<operation name="getHelloWorldAsString">
<soap:operation soapAction=""/>
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
</operation>
</binding>
<service name="HelloWorldImplService">
<port name="HelloWorldImplPort" binding="tns:HelloWorldImplPortBinding">
<soap:address location="http://localhost:4111/hai"/>
</port>
</service>
</definitions>
```

Soap annotations:

@WebService

This JAX-WS annotation can be used in 2 ways. If we are annotating this over a class, it means that we are trying to mark the class as the implementing the Web Service, in other words Service Implementation Bean (SIB). Or we are marking this over an interface, it means that we are defining a Web Service Interface (SEI), in other words Service Endpoint Interface.
Now lets see the java program demonstrating both of the mentioned ways:

Example:

```
@WebService
public interface soap {
   @WebMethod
   @SOAPBinding(style = Style.RPC)
    float celsiusToFarhenheit(float celsius);
   }
```

In the above program we can see that we haven't provided any optional element along with the @WebService annotation. And here it is used to define SEI. Regarding the other annotations used in the above program, we shall see their description a little ahead.

```
@WebService(endpointInterface="com.techouts.soap")
    public class rest implements soap {
    public float celsiusToFarhenheit(float celsius)
    {
       return ((celsius – 32)*5)/9;
      }
    }
```

In the above program we can see that we have provided an optional element endpointInterface along with the @WebService annotation. And here it is used to define SIB. endpointInterface optional element describes the SEI that the said SIB is implementing.
While implementing a web service as in above example, it is not mandatory for rest to implement soap, this just serves as a check. Also, it is not mandatory to use an SEI, however, as a basic design principle "We should program to interface", hence we have adapted this methodology in above program.
Other optional elements to @WebService can be like wsdlLocation that defines location of pre-defined wsdl defining the web service, name that defines name of the web service etc.

@WebMethod
@WebMethod JAX-WS annotation can be applied over a method only. This specified that the method represents a web service operation

@SOAPBinding:

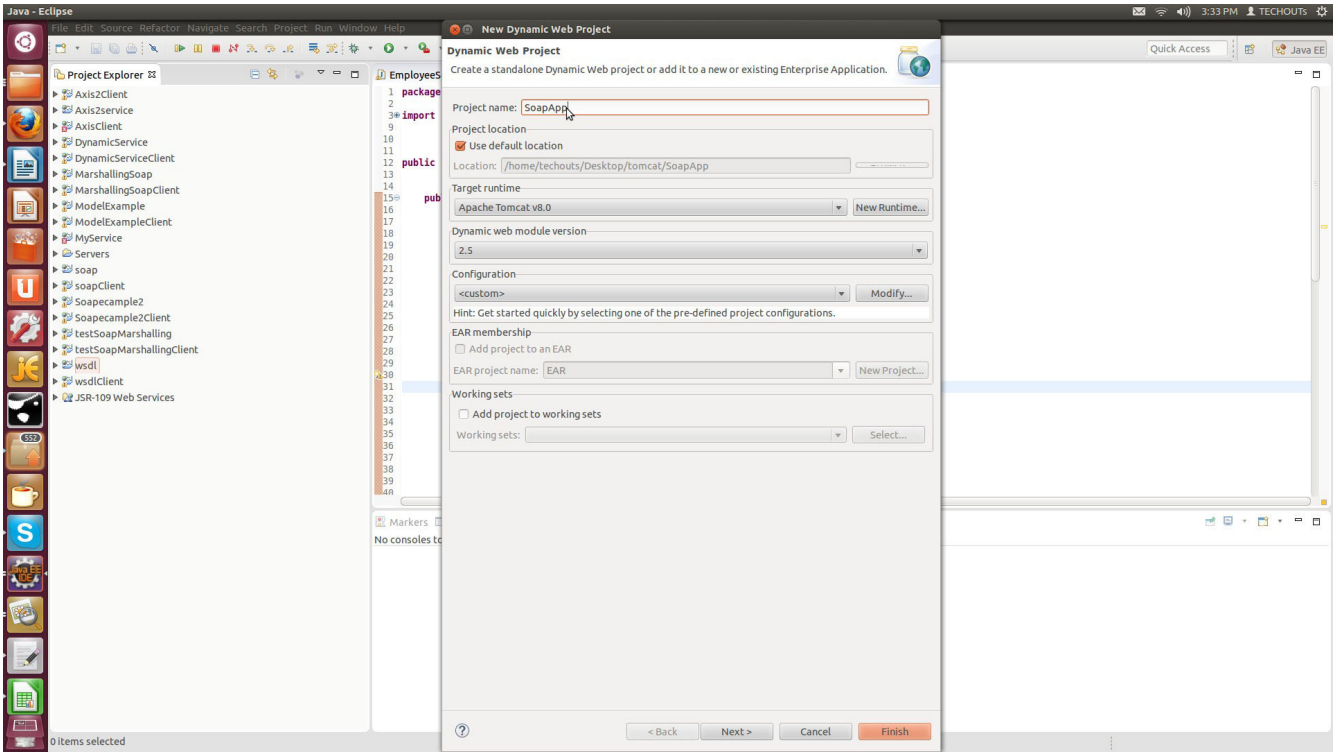
This annotation is used to specify the SOAP messaging style which can either be RPC or DOCUMENT. This style represents the encoding style of message sent to and fro while using the web service.

Java API for XML Web Services (JAX-WS), is a set of APIs for creating web services in XML format (SOAP). JAX-WS provides many annotation to simplify the development and deployment for both web service clients and web service providers (endpoints).

Soap can implement using glassfish or tomcat.

Using tomcat :

create dynamic web project

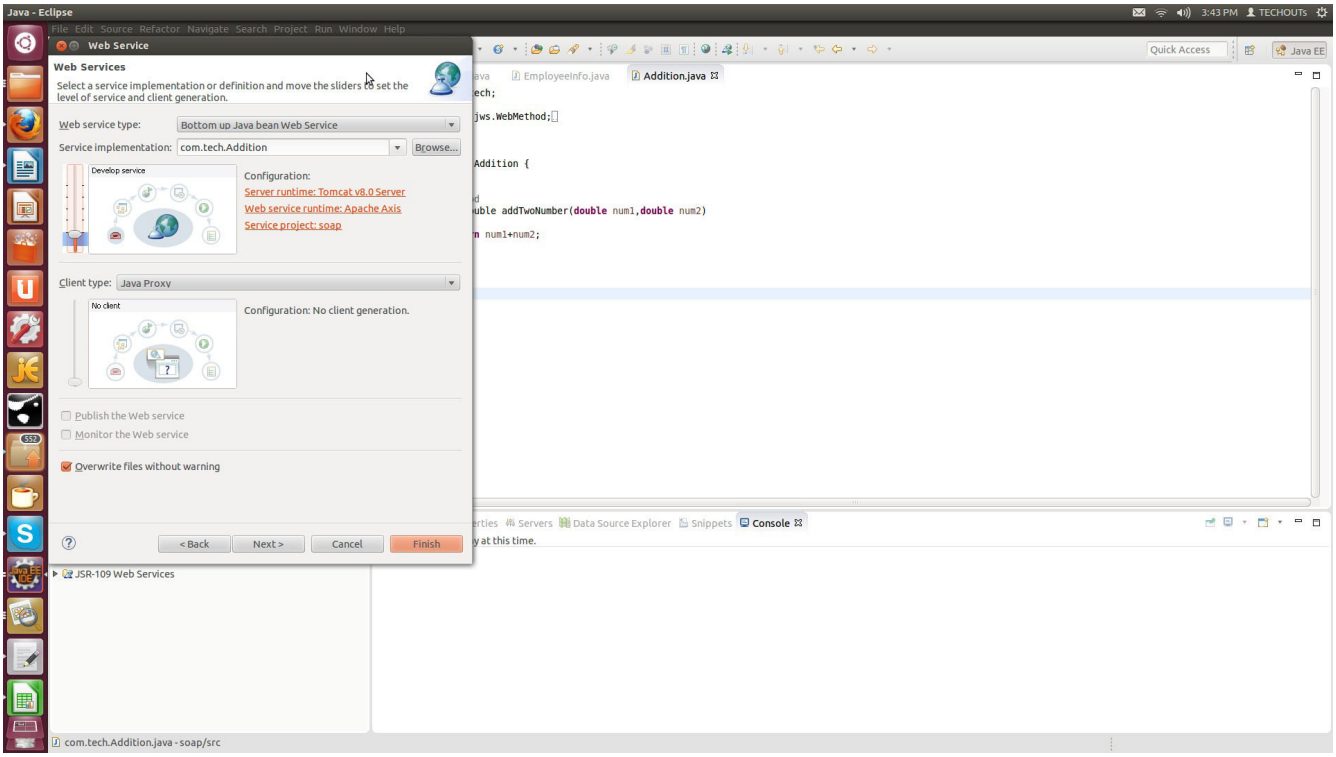


use dynamic module version as 2.5 .

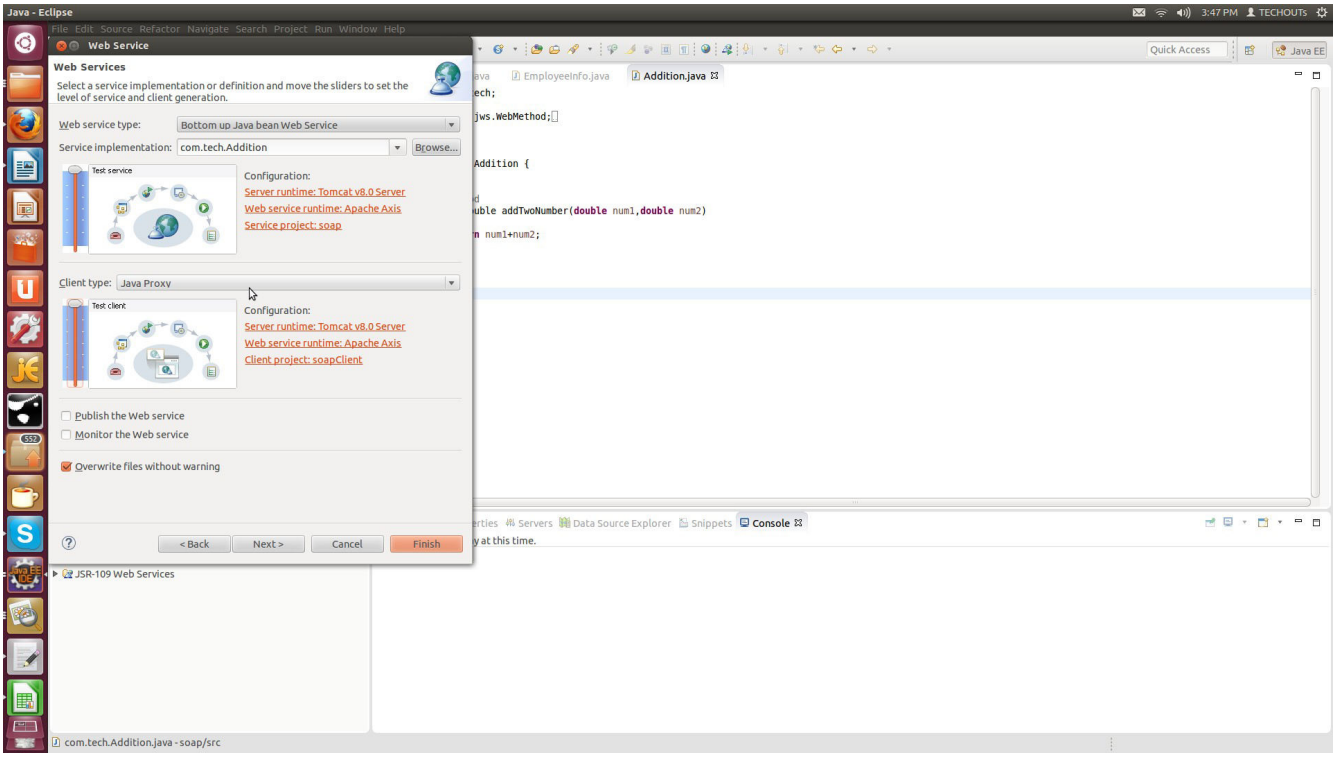
create a class like below

```
package com.tech;

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


@WebService

public class Addition {


        @WebMethod
        public Double addTwoNumber(double num1,double num2)
   {
      return num1+num2;
   }
```

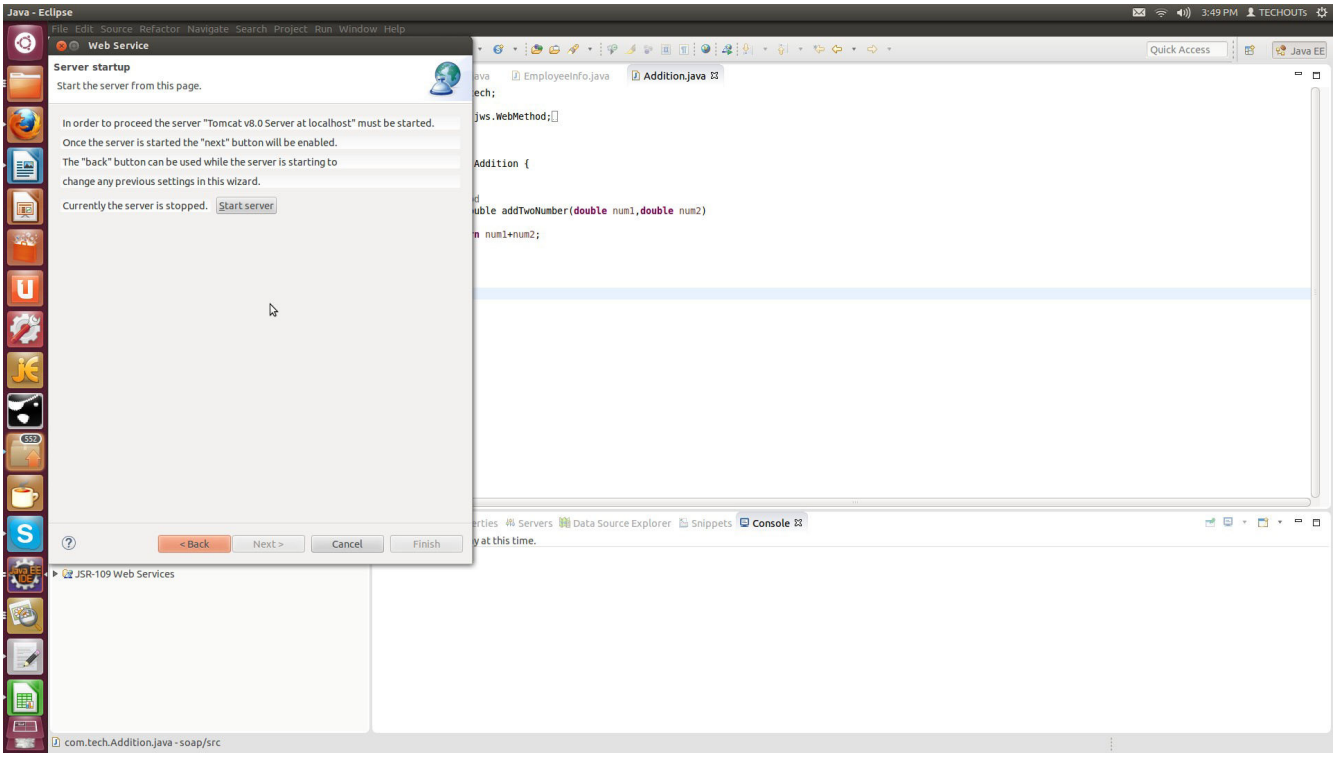
right click on your class new->other->webservice->webservices->



browse service implementation add ur class and change testclient and test service like below



click next->next



click on start server button -> finish

tomcat admin console will open click ur method and revoke check the result.

It generate wsdl file in ur webcontent folder.

Example2:

create simple java project

HelloWorld.java

```
package com.rest;

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


@WebService
@SOAPBinding(style=Style.DOCUMENT)
public interface HelloWorld {
	@WebMethod
	String getHelloWorldAsString(String name);

}
```

HelloWorldImpl.java

```
package com.rest;

import javax.jws.WebService;

@WebService(endpointInterface = "com.rest.HelloWorld")
public class HelloWorldImpl implements HelloWorld {


	@Override
	public String getHelloWorldAsString(String name) {
		return "Hello World JAX-WS " + name;
	}
}
```

create a main class

```
package com.rest;

import javax.xml.ws.Endpoint;


public class HelloWorldPublisher {

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

```
                Endpoint endpoint=Endpoint.create(new HelloWorldImpl());
                endpoint.publish("http://localhost:4111/hai");
                System.out.println("welcome to soap world");
    }

}
```

The service endpoint interface (SEI) is a Java interface class that defines the methods to be exposed as a Web service

You can use the endpointInterface attribute to separate between the implementing class and the interface. Basically, this determines what will be mapped to your wsdl:portType when you deploy the service and the wsdl:definition is generated.

if you *do not* define the endpointInterface all public methods of the annotated class will be mapped to wsdl:operation (as long as you do not influence this behaviour with @WebMethod annotations)

Use the javax.xml.ws.Endpoint create() method to create the endpoint, specify the implementor (that is, the Web Service implementation) to which the endpoint is associated, and optionally specify the binding type. If not specified, the binding type defaults to SOAP1.1/HTTP. The endpoint is associated with only one implementation object and one javax.xml.ws.Binding, as defined at runtime; these values cannot be changed.

For example, the following example creates a Web Service endpoint for the CallbackWS() implementation.

```
Endpoint endpoint=Endpoint.create(new HelloWorldImpl());
```

Use the javax.xml.ws.Endpoint publish() method to specify the server context, or the address and optionally the implementor of the Web Service endpoint.

Note: If you wish to update the metadata documents (WSDL or XML schema) associated with the endpoint, you must do so before publishing the endpoint.

For example, the following example publishes the Web Service endpoint created in Step 1 using the server context.

```
endpoint.publish("http://localhost:4111/hai");
```

chech if the wsdl file is generated or not to ur service

http://localhost:4111/hai?wsdl

OCCcommerseservices:

In hybris we are using OCCcommerseservices extension for webservices.
The Omni Commerce Connect (OCC) is based on **RESTful web services**.To maximize the list of potential API clients, the OCC supports both XML and JSON representations.

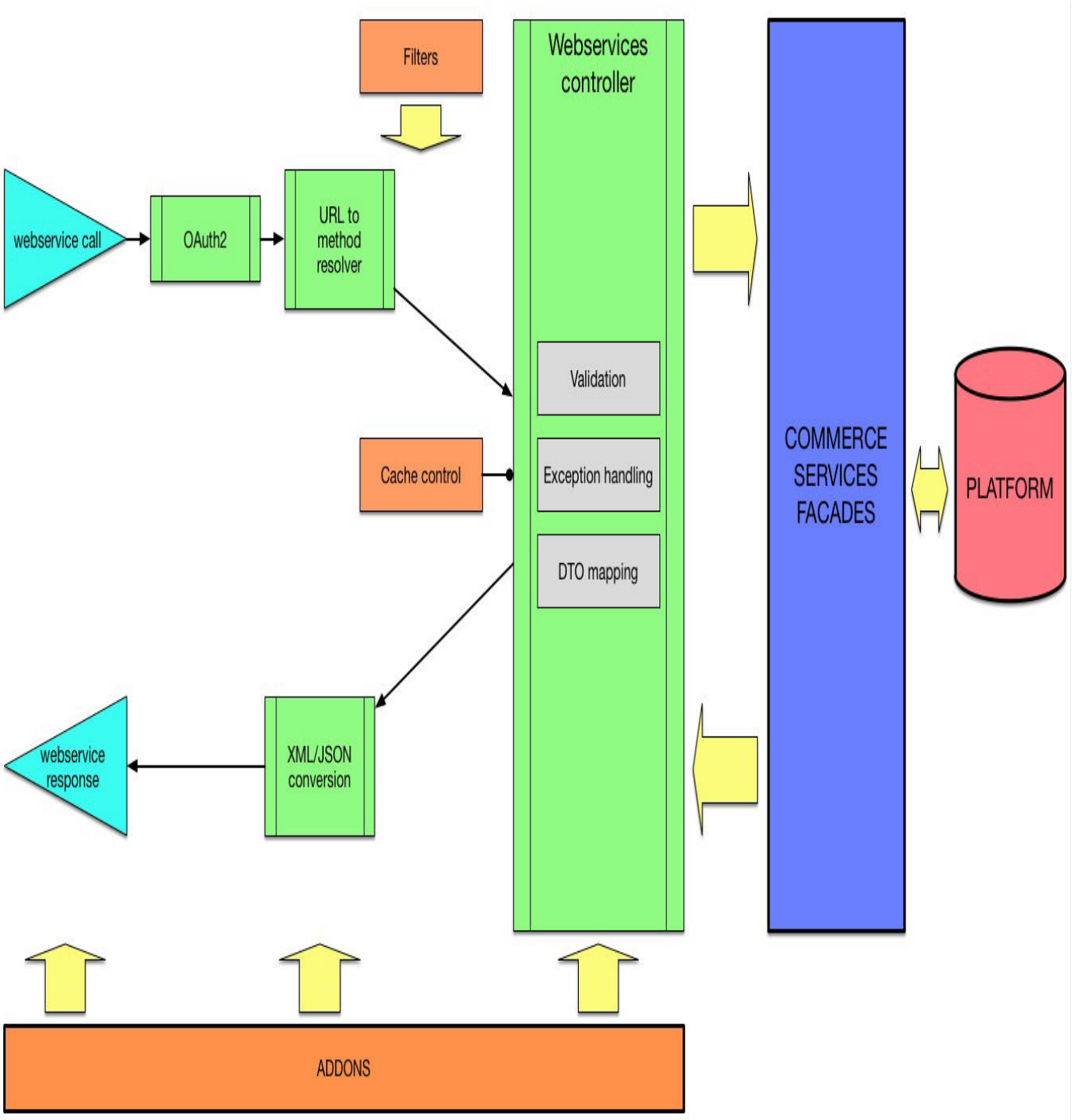
Before 4.6 we had only the platformwebservices extension, and yes, it provides more or less CRUD access the the data model. The problem is you are completely bypassing the business logic, so doing a real checkout etc is pain. The new OCC web services in 4.6 are based on the commercefacades, e.g. tightly focused on commerce flows: registration, cart handling, checkout, orders...

Commerce Web Services Extensions

The Commerce Web Services module contains following extensions:

- **commercewebservicescommons** extension
- **ycommercewebservices** extension
- **ycommercewebserviceshmc** extension
- **ycommercewebservicestest** extension

Most of these extensions are templates which can be adjusted to requirements provided by the client. Along with the hybris version 5.4 a **ycommercewebservicesaddons** folder was added, containing the **acceleratorwebservicesaddon** AddOn**.** This specific AddOn depends on **acceleratorservices** extension and is required to ensure advanced payment functionality using CIS services.



OCC Architecture

ycommercewebservices Extension

This is the main template extension of the OCC module. The most important part of this extension is advanced webservices application build on the Spring MVC framework. The calls to the specific resources are executed by a method using a request to the controller. A standard flow is presented below:

1.The request comes to commerce web services controller and is in most cases passed directly to commerce facades (sometimes additional validation is required). The types of requests include:

- **GET** - a request for data which triggers facade methods that look for and retrieve proper information.
- **POST, PUT, PATCH** - requests for creating and updating items which can be send either as separate URL parameters or with the use of RequestBody approach. The create/update requests are usually additionally validated in the OCC module.
- **HEAD** - a request that can be used to retrieve only the number of requested items information - this number is set in the response header.

2. Once the data object is retrieved from the commerce facades, it is converted to predefined DTOs (this way web services are isolated from data object changes in the commerce layer).

3. Data objects then send a response to the call in XML or JSON format (assuming there is no errors or exceptions).

The **ycommercewebservices** extension has also some additional features, described below.

he **ycommercewebservices** template has been developed in two versions: v1 and v2 respectively. Both versions are available to the users, however **v2 is the default one**.

Overview

The **ycommercewebservices** template has two versions of web services available: **v1** and **v2**.

- **v1** is the old version, released along with Release 5.1.
- **v2** was introduced with Release 5.4 and is a new, configurable and stateless implementation of RESTful web services.

Version 2 (default)

The key features of Version 2 of the ycommercewebservices extension are as follows:

1.Stateless

The calls are now stateless so the customer data is no longer preserved between subsequent requests. Each time the customer needs to provide all the required data such as customer id or cart id.

2.RESTful implementation

URL resources have been refactored and are now more RESTful.
The customer needs to provide resource type and resource identifier of the resource he would like to work with.

3.Data creation

Executed using the URL parameter list or RequestBody.

Version 1

The key features of Version 1 of the ycommercewebservices extension are as follows:

1.Stateful

A stateful way of communication with commerce layer
Flows some extent similar to storefront flows, based on the acceleratorservices extension (e.g. the checkout process).
The customer cart and other customer data are stored in the session, and are preserved between subsequent requests (basing on the assumption the JSESSIONID is stored).
The customer can be more focused on the actual actions he wants to perform and not on holding and providing the whole context data.

2.Response format

Responses are provided in XML or JSON format., depending on the request.

For each version, there is a separate servlet defined in the **web.xml** file. These servlets have different java-based configurations. There is also a servlet defined for OAuth authorization - in this case it is a feature common for both versions.

RESTful Implementation in v2

OCC **v2** became more RESTful and fully stateless from the end user perspective. This document describes some of the new ideas and approaches regarding URLs and access control in **v2**.

Users

The user resources are available under the following path:

https://localhost:9002/rest/v2/{baseSiteID}/users/{userID}

example:

https://localhost:9002/rest/v2/b2ctelco/users/anil@gmail.com

Carts

The cart resource is available under the following path:

https://localhost:9002/rest/v2/{baseSiteID}/users/{userID}/carts/{cartID}

example:

https://localhost:9002/rest/v2/b2ctelco/users/anil@gmail.com/carts/0111111

OAuth :

OAuth is an open standard for authorization, commonly used as a way for Internet users to log into third party websites using their Microsoft, Google, Facebook, Twitter, etc. accounts without exposing their password. Generally, OAuth provides to clients a "secure delegated access" to server resources on behalf of a resource owner. It specifies a process for resource owners to authorize third-party access to their server resources without sharing their credentials. Designed specifically to work with Hypertext Transfer Protocol (HTTP), OAuth essentially allows access tokens to be issued to third-party clients by an authorization server, with the approval of the resource owner. The third party then uses the access token to access the protected resources hosted by the resource server

OAuth 2.0 :

OAuth 2.0 is the next evolution of the OAuth protocol and is not backwards compatible with OAuth 1.0. OAuth 2.0 focuses on client developer simplicity while providing specific authorization flows for web applications, desktop applications, mobile phones, and living room devices.

In order to simplify authentication and authorization, OCC uses a standard **OAuth2** protocol. The main purpose is to enable long-term access to the principal and differentiate security rules depending on the type of client application.

- **Authentication** means checking provided credentials. If credentials are valid, then the proper roles are assigned to a **principal**.
- **Authorization** means deciding if a principal can perform a given action. This is determined based on the assigned roles of the **principal** and also on other constraints, for example secure communication channel.

OAuth 2.0 is the next evolution of the OAuth protocol which was originally created in late 2006. OAuth 2.0 focuses on client developer simplicity while providing specific authorization flows for web applications, desktop applications, mobile phones, and living room devices.

This document provides an introduction to the **OAuth 2.0** used in the hybris OCC web services.
I find the following diagrams very useful. They illustrate the difference in communication between parties with OAuth2 and OAuth1.

How is OAuth 2 different from OAuth 1:

- **More OAuth Flows to allow better support for non-browser based applications.** This is a main criticism against OAuth from client applications that were not browser based. For example, in OAuth 1.0, desktop applications or mobile phone applications had to direct the user to open their browser to the desired service, authenticate with the service, and copy the token from the service back to the application. The main criticism here is against the user experience. With OAuth 2.0, there are now new ways for an application to get authorization for a user.

- **OAuth 2.0 no longer requires client applications to have cryptography.** This hearkens back to the old Twitter Auth API, which didn't require the application to HMAC hash tokens and request strings. With OAuth 2.0, the application can make a request using only the issued token over HTTPS.

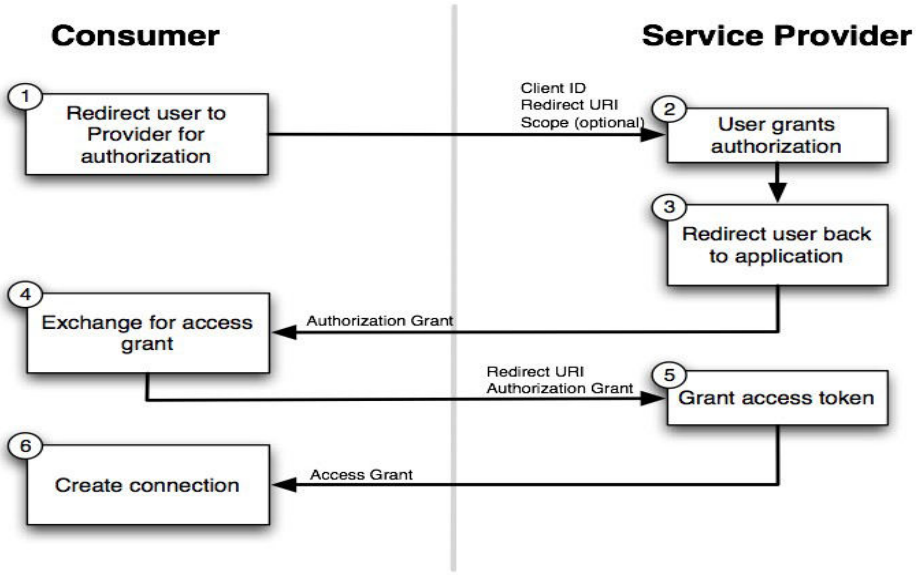
- **OAuth 2.0 signatures are much less complicated.** No more special parsing, sorting, or encoding.

- **OAuth 2.0 Access tokens are "short-lived".** Typically, OAuth 1.0 Access tokens could be stored for a year or more (Twitter never let them expire). OAuth 2.0 has the notion of refresh tokens. While I'm not entirely sure what these are, my guess is that your access tokens can be short lived (i.e. session based) while your refresh tokens can be "life time". You'd use a refresh token to acquire a new access token rather than have the user re-authorize your application.

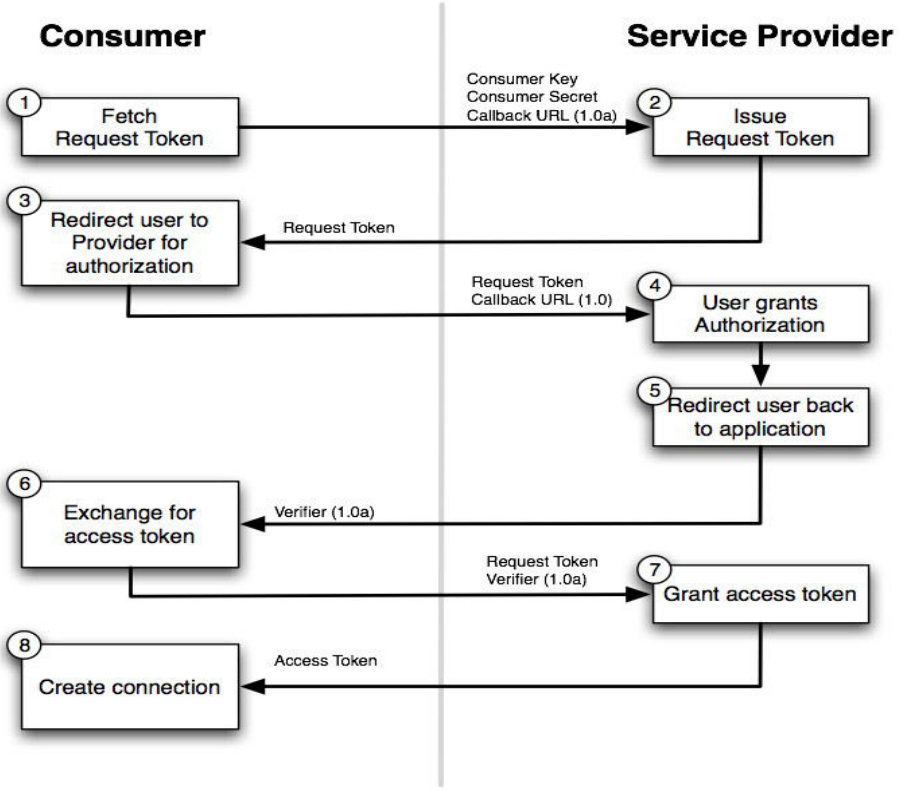
- **Finally, OAuth 2.0 is meant to have a clean separation of roles between the server responsible for handling OAuth requests and the server handling user authorization.** More information about that is detailed in the aforementioned article.

- I find the following diagrams very useful. They illustrate the difference in communication between parties with OAuth2 and OAuth1.

OAuth 2



OAuth 1



Overview

- In the traditional client-server authentication model, the client requests an access-restricted resource
- (in other words, protected resource) on the server by authenticating with the server using the resource owner's credentials.
- In order to provide third-party applications access to the restricted resources, the resource owner shares its credentials with the third-party application.
- This creates several issues and limitations:

- Third-party applications are required to store the resource owner's credentials for future use, typically a password in clear text form.
- Servers are required to support password authentication, despite the security weaknesses inherent in passwords.

- Third-party applications gain overly broad access to the owner's protected resources, leaving

resource owners without any ability to restrict duration or access to a limited subset of resources.

- Resource owners cannot revoke access to an individual third-party without revoking access to all third-parties, and must do so by changing their password.

- Compromise of any third-party application results in compromise of the end-user's password and all of the data protected by it.

- OAuth 2.0 addresses these issues by introducing an authorization layer and separating the role of the client from that layer.

The OAuth 2.0 authorization framework is the default authorization framework for the commerce driven OCC (Omni Commerce Connect) Web Services under the ycommercewebservices extension.

The key benefit of using OAuth 2.0 (compared to basic authentication, even over HTTPS) is that the API client does not have to save or, in some cases, even obtain the user's credentials. (username,passwords)
Instead, access tokens are returned to the client that can use refresh tokens to obtain new access tokens once they have expired.

See ur OAuth Configuration Files

The OAuth configuration file names and their location may differ depending on the chosen release version:

- for version 5.1: **ycommercewebservices-web-spring.xml** file.
- for versions 5.2, 5.3 and 5.4: security-spring.xml file located in the ycommercewebservices/web/webroot/WEB-INF/config/ directory.
- for version 5.5 and higher: **security-spring.xml** file located in the ycommercewebservices/web/webroot/WEB-INF/config/common directory

oauth2.0

1. **Step (A)**: The **client** receives the **username** and **password**. In this step, the user enters this information directly into the client application. Note that users must have a way to identify the application as being the official application they can trust.
2. **Step (B)**: Next, the client application makes a request to the **Authorization Server**, for example the **/oauth/token** endpoint. There are two ways the **client_id** and **client_secret** can be sent along: either in a regular basic authentication request header, or as a part of the parameters passed in the request payload (that is, the request body). See the list of parameters to be passed:
    - **client_id** and **client_secret**: Either passed as parameters or as a basic authentication header. Basic authentication means that **client_id** and **client_secret** are treated as username and password, concatenated using a colon (:) and then **Base64** encoded. This value is then used as a part of the authorization request header, for example: **Authorization: Basic <Base64-encoded username:password>**

- **username** and **password**: Credentials for the resource owner, the user's real credentials.
- **grant_type**: Needs to be set to **password** for this flow.

3. **Step (C)**: The authentication server returns the **access_token** with an optional **refersh_token**.

Detailed description of the presented flow:

1. **Step (A)**: Redirect the user to the authorization server (typical endpoint should be **/oauth/authorize**). All communication should occur through HTTPS so that **client_id** contained in the URL is safe. The following parameters have to be included:
   - **response_type**: **code**.
   - **client_id**: For hybris' web services, the OAuth 2.0 client has to be manually set up in the Spring Security OAuth2 XML configuration.
   - **client_secret**: The **secret** for the **client_id**, and it needs to be on the server-side so that users cannot see it.
   - **redirect_uri**: Optional, but recommended. When a user is redirected to the authorization endpoint, this value needs to be passed, and it has to match the server configuration settings.
   - **scopes**: Used to allow different access levels.
   - **state**: Optional, but recommended. Used to overcome CSRF (cross-site request forgery) attacks.

     The following code sample uses the described parameters.

```
import java.net.URLEncoder

def client_id = 'mobile_android'
def client_secret = 'secret'
def redirect_uri = 'http://localhost:8080/oauth2_callback'

def scopes = [
'customer'
]

def state = new Random(System.currentTimeMillis()).nextInt().toString()
request.getSession(true).setAttribute('state', state)

redirect "http://localhost:9001/rest/oauth/authorize?client_id=${client_id}&redirect_uri=$
{URLEncoder.encode(redirect_uri, 'UTF-8')}&response_type=code&scope=$
{URLEncoder.encode(scopes.join(' '), 'UTF-
def client_id = 'mobile_android'
def client_secret = 'secret'
def redirect_uri = 'http://localhost:8080/oauth2_callback'
```

```
def scopes = [
    'customer'
]

def state = new Random(System.currentTimeMillis()).nextInt().toString()
request.getSession(true).setAttribute('state', state)

redirect "http://localhost:9001/rest/oauth/authorize?client_id=${client_id}&redirect_uri=${URLEncoder.encode(redirect_uri, 'UTF-8')}&response_type=code&scope=${URLEncoder.encode(scopes.join(' '), 'UTF-8')}&state=${URLEncoder.encode(state, 'UTF-8')}"
```

Telco accelerator:

The hybris Telco Accelerator is a ready-to-use Web framework that enables you to sell more online and offline in an natural and cost-effective way.

Only hybris delivers a commerce platform specifically designed to maximize average revenue per user (ARPU) and deliver an multichannel customer experience that streamlines a complex purchase process.

The hybris Telco Accelerator delivers all this and enables you to get to market fast, so you can stay ahead of the competition.

The hybris Telco Accelerator is a single platform to sell more phones, plans, and cloud services integrating online and offline marketing.

The hybris Telco Accelerator package answers customer inquiries by enabling:

- Product package sale with promotions and customer discount
- Customer personalization
- Contract bundles
- Contract renewal
- Access to digital products and services

The hybris Telco Accelerator with the Subscriptions and Bundling Module enables you to enhance customer satisfaction in an intuitive and cost-effective way. By conducting customer's transactions, Telco can have an overview of trends in the market.

In addition, the functional e-commerce storefront customized with languages and currencies can be easily updated frequently with information regarding the products, accessories, and services.

Assessing Business Opportunities

Within the hybris Telco Accelerator, customer maintaince is:

- Convenient, useful, and suitable. It allows you to focus on customer preferences guaranteeing a higher average lifetime value.
- All-embracing. It leads to customer satisfaction offering a phone with contract discounts, accessories, data plan, voice, text messaging, and all the other products and services.
- Efficient. It gives you a competitive advantage in the business and real customer insight.
- Comprehensive. It ensures the salespeople follow all sales process steps.

As a company with the highest level service, Telco needs to diversify their offerings. The hybris Telco Accelerator brings a new and innovative way to shop and choose services the customer prefers. Having the ability to track customer behaviors and patterns in buying gives a company priceless information about the way to further market a particular product.

- OCC-related REST API in the Telco Accelerator :

When we introduced the subscription functionality, we modified the behaviour of OCC REST API such that it fits the needs of the Telco Accelerator.

Subscription Product vs. Regular Product
Subscription products differ from regular products in that subscription products consist of several parts that constitute an entire subscription product only when they are put together. Such parts may include a portable device, accessroies ,a cell phone service plan, and some additional entitlements. Also, unlike regular products, subscription products are not entirely paid for during checkout.

- Subscription Product :

The Subscription Billing Gateway (SBG) acts as a translation interface web service between the hybris Commerce Suite and external subscription billing management provider systems, offering an end-to-end subscription-based sales solution.

- Subscription Products Features:

There are a few features related to subscription products:

- **Entitlements** - an entitlement defines what is the value that a customer receives with a subscription product.
- **Subscription Terms** - they specify what are terms of a subscription and length of a customer's commitment. You can also define the conditions of renewal of a subscription or whether a customer is eligible to cancel the commitment and under which conditions. Subscription terms are related to **Billing Cycle Plan** where you can specify what is the frequency of payments, of what cycle type they are and what is the cycle day, that is, the day when a customer is to pay for the subscription.
- **Price Plan** - which defines what are the periodic charges depending on length of the commitment.

- Omni Commerce Connect-related API Changes:

API changes made along with the introduction of the subscription functionality have considerable impact on the API of the **ycommercewebservices** extension. Below, you can find how the changes are introduced and which functionality they are related to.

Subscription-related OCC:

The Subscription module exposes its full functionality through the REST API calls simply by enriching the existing API of the **ycommercewebservices** extension. In order to use the new functionality, the resources act in the following way:

Resource Products gives all the details about the produts like creation time,available facets, available sort options, and pagination options.

Bundle-related OCC:

The Bunding module does not expose its full functionality through REST API calls, but it extends the existing API of the **ycommercewebservices** extension in order to use the new functionality. The resources act in the following way:

in Resource products **SubscriptionProducts** have more data to expose in order to indicate if they can be sold individually or only in bundle.

For webservices by default hybris extension is ycommerceservices.
Dont use the default extension for webservices.
Default extension can be reusable and any upgrade came use it and create a new extension.

Creating Customized Extension Using extgen:

Create a new extension for webservices.

1)ant extgen

modulegen:
    [input]
    [input] Please choose a template for generation.
    [input] Press [Enter] to use the default value ([accelerator], b2baccelerator, chinaaccelerator, commercewebservices, acceleratorordermanagement)

commercewebservices
    [input]
    [input] Please choose the package name of your extension. It has to fulfill java package name convention.
    [input] Press [Enter] to use the default value [org.training]

com.mycompany.mycommercewebservices

    [input]
    [input] Please choose a template for generation.
    [input] Press [Enter] to use the default value ([yempty], ycockpit, ybackoffice, yaddon, yacceleratorfulfilmentprocess, ycommercewebservices)

ycommercewebservices

     [echo] Using extension template source: C:\workspace\bin/ext-accelerator/ycommercewebservices
   [delete] Deleting directory C:\workspace\temp\hybris\extgen_final
   [delete] Deleting directory C:\workspace\temp\hybris\extgen
    [mkdir] Created dir: C:\workspace\temp\hybris\extgen
     [echo] Copying template files from C:\workspace\bin/ext-accelerator/ycommercewebservices to C:\workspace\temp\hybris/extgen
     [copy] Copying 298 files to C:\workspace\temp\hybris\extgen
     [copy] Copying 298 files to C:\workspace\temp\hybris\extgen_final
    [mkdir] Created dir: C:\workspace\bin\custom\mycommercewebservices\lib
     [copy] Copying 215 files to C:\workspace\bin\custom\mycommercewebservices
     [echo]
     [echo]
     [echo]   Next steps:
     [echo]
     [echo] 1) Add your extension to your C:\workspace\config/localextensions.xml
     [echo]
     [echo]   <extension dir="C:\workspace\bin/custom/mycommercewebservices"/>
     [echo]
     [echo] 2) Make sure the applicationserver is stopped before you build the extension the first time.
     [echo]
     [echo] 3) Perform 'ant' in your hybris/platform directory.

[echo]
[echo] 4) Restart the applicationserver
[echo]
[echo]

Follow the **Next steps** hints to include your new extension in your hybris platform instalation. Replace the **ycommercewebservices** with **mycommercewebservices** in the **localextensions.xml** file.

```
<!-- <extension name='ycommercewebservices' /> -->
```

```
<extension name="mycommercewebservices"/>
```

Edit the **extensioninfo.xml** file for your customized commerce web services extension to provide a **webroot** context that best fits your project, for example:

```
<extension abstractclassprefix="Generated" classprefix="Ycommercewebservices" name="occcommercewebservices" usemaven="false">
<requires-extension name="commercefacades"/>
        <requires-extension name="commerceservices"/>

        <requires-extension name="commercewebservicescommons"/>

        <coremodule generated="true" manager="de.hybris.platform.jalo.extension.GenericManager" packageroot="com.lycamobile"/>


        <webmodule jspcompile="false" webroot="/occcommercewebservices"/>


        <meta key="modulegen-name" value="commercewebservices"/>
    </extension>
```

List Of Default Exposes(produces) Services in Hybris:
for catalogs(w)

https://localhost:9002/rest/v2/b2ctelco/catalogs

for productcatalogs(w)
https://localhost:9002/rest/v2/b2ctelco/catalogs/b2ctelcoProductCatalog

for particular catalog version (w)
https://localhost:9002/rest/v2/b2ctelco/catalogs/b2ctelcoProductCatalog/Online

for single product(w)
https://localhost:9002/rest/v2/b2ctelco/products/Y_Lyca_100_1Y

for product reviews(w)
https://localhost:9002/rest/v2/b2ctelco/products/Y_Lyca_100_1Y/reviews

for languages and countries
https://localhost:9002/rest/v2/b2ctelco/languages

https://localhost:9002/rest/v2/b2ctelco/deliverycountries

for currencies

https://localhost:9002/rest/v2/b2ctelco/currencies

for cardtypes and titles

https://localhost:9002/rest/v2/b2ctelco/titles

https://localhost:9002/rest/v2/b2ctelco/cardtypes

for stores
https://localhost:9002/rest/v2/b2ctelco/stores

for all categories
https://localhost:9002/rest/v2/b2ctelco/catalogs/b2ctelcoProductCatalog/Online/categories/1

for users

https://localhost:9002/rest/v2/b2ctelco/users/anil@gmail.com

for customergroups
https://wiki.hybris.com/display/release5/Resource+Customergroup

for orders
https://localhost:9002/rest/v2/b2ctelco/orders

Integration using restful

1)create service for the application using restful Webservices.

Marshalling :

Marshalling is used for converting javaObject into xml format.

Unmarshalling:

Unmarshalling is used for converting xml into javaObject format.

Example1)

1)create one pojo class

```
public class EmployeeInfo {
	private int employeeId;

    private String employeeName;
    private String employeeCity;

}
```

2)create a service for the pojo

```
public class EmployeeService {

	public void testClient() throws JAXBException {

		try{
			EmployeeInfo employee=new EmployeeInfo(1218, "Abc", "Delhi");

			JAXBContext jaxbcontext=JAXBContext.newInstance(EmployeeInfo.class);

			Marshaller ms=jaxbcontext.createMarshaller();

			ms.setProperty(ms.JAXB_FORMATTED_OUTPUT, true);

			ms.marshal(employee, System.out);

			ms.marshal(employee,new File("src\\data\\customer.xml") );

		}
```

```
                        catch(Exception e){
                                System.out.println(e.getMessage());
                        }
```

make it ur application as a jar and add this jar into in hybris library.

2)create client in hybris

```
public class ExampleConsumeController
{
        private static final String targetURL =   "http://localhost:9090/techout/webapi/myresource";



//here we are consuming third party details


        @RequestMapping(value = "/{baseSiteId}/resttest")
        public void testClient() throws JAXBException
        {
                System.out.println("Hello");

                final EmployeeService service = new EmployeeService();

                service.testClient();

                service.unMarshalling();

                mars.unMarshallingList();
                mars.marshallingList();


                try
                {

                        final URL restServiceURL = new URL(targetURL);

                        final HttpURLConnection httpConnection = (HttpURLConnection)
restServiceURL.openConnection();
                        httpConnection.setRequestMethod("GET");
                        httpConnection.setRequestProperty("Accept", "application/xml");

                        if (httpConnection.getResponseCode() != 200)
                        {
                                throw new RuntimeException("HTTP GET Request Failed with Error code : " +
httpConnection.getResponseCode());
                        }
```

```
                        final BufferedReader responseBuffer = new BufferedReader(new
InputStreamReader((httpConnection.getInputStream())));

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

                        while ((output = responseBuffer.readLine()) != null)
                        {
                                System.out.println(output);
                        }

                        httpConnection.disconnect();

                }
                catch (final MalformedURLException e)
                {

                        e.printStackTrace();

                }
                catch (final IOException e)
                {

                        e.printStackTrace();

                }

        }



}
```

Integration using soap

1)create service for the application using soap

make ur application as a jar and add this jar into in hybris library.

2)create client in hybris

```
@Controller
public class SoapUnmarshalling
{
	private static final String targetURL = "http://localhost:9082/MarshallingSoap/services/Main?wsdl";



	@RequestMapping(value = "/{baseSiteId}/soapUnmarshalling")
	public void testClient() throws JAXBException
	{
		System.out.println("Hello");

		try
		{

			final URL restServiceURL = new URL(targetURL);

			final HttpURLConnection httpConnection = (HttpURLConnection)
restServiceURL.openConnection();
			httpConnection.setRequestMethod("GET");
			httpConnection.setRequestProperty("Accept", "application/xml");

			if (httpConnection.getResponseCode() != 200)
			{
				throw new RuntimeException("HTTP GET Request Failed with Error code : " +
httpConnection.getResponseCode());
			}

			final BufferedReader responseBuffer = new BufferedReader(new
InputStreamReader((httpConnection.getInputStream())));

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

			while ((output = responseBuffer.readLine()) != null)
			{
				System.out.println(output);
			}

			httpConnection.disconnect();

		}
		catch (final MalformedURLException e)
		{

			e.printStackTrace();

		}
		catch (final IOException e)
		{

			e.printStackTrace();

		}
```

```
        }


}
```

Save the conumed details into hybris CUSTOMER ITEMTYPE.

Or

need to create a new model EXTENDS USER.

create a model class in hybris (same as ur service model):

```
<itemtype code="EmployeeInfo"
              extends="User"
              jaloclass="com.lycamobile.employee.EmployeeInfo"
              autocreate="true"
              generate="true">
         <attributes>
            <!--  auto ID which is generated by NumberSeries -->
            <attribute autocreate="true" qualifier="employeeId" type="int">
               <modifiers read="true" write="true" search="true" optional="true"/>
               <persistence type="property"/>
            </attribute>
             <attribute autocreate="true" qualifier="employeeName" type="java.lang.String">
               <modifiers read="true" write="true" search="true" optional="true"/>
               <persistence type="property"/>
            </attribute>
             <attribute autocreate="true" qualifier="employeeCity" type="java.lang.String">
               <modifiers read="true" write="true" search="true" optional="true"/>
               <persistence type="property"/>
            </attribute>
         </attributes>
      </itemtype>
```

do ant build
use model.save()

for save details into hybris db.
Change ur client code like as below

```
@Controller
public class ConsumingSoapMarshalling
{
	private static final String targetURL = "http://localhost:9082/MarshallingSoap/services/Main";


	private static Logger Log = Logger.getLogger(ConsumingSoapMarshalling.class);




	@Resource
	private CustomerFacade customerFacade;
	@Resource
	private ModelService modelService;

	@RequestMapping(value = "/{baseSiteId}/soapMarshalling")
	public void testClient() throws JAXBException
	{


		Log.info("-----------------------hellooooooooo----------");



		final EmployeeService service = new EmployeeService();

		service.testClient();

		service.unMarshalling();



		final JAXBContext jaxbcontext = JAXBContext.newInstance(new Class[]
		{ EmployeeInfo.class });

		final Unmarshaller um = jaxbcontext.createUnmarshaller();
```

```
                final EmployeeInfo c = (EmployeeInfo) um.unmarshal(new File("src\\data\\customer.xml"));

                final EmployeeInfoModel employee = new EmployeeInfoModel();


                employee.setEmployeeId(c.getEmployeeId());

                employee.setEmployeeCity(c.getEmployeeCity());

                employee.setEmployeeName(c.getEmployeeName());

                employee.setUid("sssssss@gmail.com");


                Log.info("employee id is-------" + c.getEmployeeCity());

                Log.info("employee id is-------" + c.getEmployeeId());

                Log.info("employee id is-------" + c.getEmployeeName());


                Log.info("employee id is-------" + employee.getEmployeeCity());


                Log.info("employee id is-------" + employee.getEmployeeName());


                Log.info("employee id is-------" + employee.getUid());


                modelService.save(employee);


                try
                {

                        final URL restServiceURL = new URL(targetURL);

                        final HttpURLConnection httpConnection = (HttpURLConnection)
restServiceURL.openConnection();
                        httpConnection.setRequestMethod("GET");
                        httpConnection.setRequestProperty("Accept", "application/xml");

                        if (httpConnection.getResponseCode() != 200)
                        {

                                System.out.println("error report");
                                throw new RuntimeException("HTTP GET Request Failed with Error code : " +
httpConnection.getResponseCode());


                        }

                        final BufferedReader responseBuffer = new BufferedReader(new
InputStreamReader((httpConnection.getInputStream())));
```

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

                        while ((output = responseBuffer.readLine()) != null)
                        {
                                System.out.println(output);
                        }

                        httpConnection.disconnect();

                }
                catch (final MalformedURLException e)
                {

                        e.printStackTrace();

                }
                catch (final IOException e)
                {

                        e.printStackTrace();

                }

        }


}
```

goto hmc and check ur details using flexiblesearchquery:

select * from {EmployeeInfo}

Twitter Integration:

here we need to integrate twitter in hybris.

step 1.Go to https://dev.twitter.com/apps create an account

---

step 2.Create app(fill up the form)

---

step 3.Change permissions if necessary(depending if you want to just read,write or execute)

---

step 4.Go To API keys section and click generate ACCESS TOKEN.

---

download twitter4j jar or jtwitter 4j jar add twitter core-4j jar in hybris lib.

and write client in hybris like below

```
@Controller
public class TwitterIntegration
{
	private static Logger Log = Logger.getLogger(TwitterIntegration.class);

	@RequestMapping(value = "/twitterLogin")
	public void twitter() throws TwitterException
	{
		final ConfigurationBuilder conf = new ConfigurationBuilder();

		conf.setDebugEnabled(true)

		.setOAuthConsumerKey("9NCBELRuw9WlvkDobQ4H4EYm7")
				.setOAuthConsumerSecret("0Y71Wdn6EMJoHMRXs8CoTyYo7NqQFLMQ4Mqb5jEVtxyvxVjK62")
				.setOAuthAccessToken("3251436956-zSYche7Y4Mrku9Cnm7XK1vhh5HEr7ponpUZDShy")
				.setOAuthAccessTokenSecret("qv8yM1m0wFVnyRM8q0yKjQCaZuoAPwMwXWZKR3OQjwf5w");


		final TwitterFactory fa = new TwitterFactory(conf.build());

		final twitter4j.Twitter twitter = fa.getInstance();

		Log.info("Twitter Id is" + twitter.getId());

		Log.info("Twitter languages are " + twitter.getLanguages());




		final List<Status> tweets = twitter.getHomeTimeline();

		for (final Status t : tweets)
		{
```

```
                System.out.println(t.getId() + " - " + t.getCreatedAt() + ": " + t.getText());

                Log.info("id issssssssss" + t.getId());


                Log.info("id issssssssss" + t.getInReplyToUserId());


                Log.info("id issssssssss" + t.getUser());


                Log.info("id issssssssss" + t.getURLEntities());

                Log.info("id issssssssss" + t.getPlace());


                Log.info("id issssssssss" + t.getLang());

            }

        }
}
```

save the consumed details into the hybris database

use customermodel to save the consumed details.

```
final CustomerModel model1 = new CustomerModel();

                model1.setUid(String.valueOf(twitter.getId()));


                model1.setDN(twitter.getScreenName());









                modelService.save(model1);
Log.info("Twitter authrization Id is" + twitter.getAuthorization());


                Log.info("Twitter class is" + twitter.getClass());


                Log.info("myTwitter  favorites is" + twitter.getFavorites());
```

```
Log.info("Twitter Id is" + twitter.getId());

Log.info("Twitter languages are " + twitter.getLanguages());
```

here we are unable to consume email details in hybris to access an email we need permission for twitter.

Go through this link for permission

Go to https://support.twitter.com/forms/platform

- Select "I need access to special permissions"
- Enter Application Name and ID. These can be obtained via https://apps.twitter.com/ -- the application ID is the numeric part in the browser's address bar after you click your app.
- Permissions Request: "Email address"
- Submit & wait for response
- After your request is granted, an addition permission setting is added in your twitter app's "Permission" section. Go to "Additional Permissions" and just tick the checkbox for "Request email addresses from users".

Header Link Component:

create a new link in header for integrations.

Cms-content.impex:

```
# CMS Link Components
INSERT_UPDATE CMSLinkComponent;$contentCV[unique=true];uid[unique=true];name
;url;&linkRef;&componentRef;target(code)[default='newWindow'];$category;$product;
                    ;                ;TwitterLink     ;Twitter Link
;/twitterLogin;TwitterLink;TwitterLink;;;;
                    ;                ;YoutubeLink     ;Youtube Link   ;/youtube
;YoutubeLink;YoutubeLink;;;;
```

Content Slots

```
INSERT_UPDATE ContentSlot;$contentCV[unique=true];uid[unique=true];cmsComponents(uid,
$contentCV)
                ;                ;HeaderLinksSlot ;ContactInfo,TwitterLink,YoutubeLink
```