**Webservice**

Web service is a way of communication that allows interoperability between different applications on different platforms, for example, a Java based application on Windows can communicate with a .Net based one on Linux. The communication can be done through a set of XML messages over HTTP protocol, Web services are browsers and operating system independent service.

* Web applications are meant for users and to be accessed in browser having human readable format whereas web services are meant for applications to access data in the format of XML, JSON etc.
* Web applications always use HTTP/HTTPS protocol whereas traditional web services use SOAP protocol. Recently REST is getting popularity that is an architecture style and almost all times run on HTTP/HTTPS protocol.
* Web applications are not meant for reusability whereas this is one of the benefit of web services. A single web service can be used by different kinds of applications.
* Web application can access web services to access some data or to perform some tasks, web services can’t access web applications to fetch some data.
* Web applications are capable to maintain user session, web services are stateless.

**Java Web Services**

Java provides it’s own API to create both SOAP as well as REST web services.

1. JAX-WS: JAX-WS stands for Java API for XML Web Services. JAX-WS is XML based Java API to build web services server and client application.
2. JAX-RS: Java API for RESTful Web Services (JAX-RS) is the Java API for creating REST web services. JAX-RS uses annotations to simplify the development and deployment of web services.

# **Service Oriented Architecture (SOA)**

Service Oriented Architecture or SOA is a design pattern. It is designed to provide services to other applications through protocol. It is a concept only and not tied to any programming language or platform.

**Web services is a technology of SOA most likely.**

## Service

A service is well-defined, self-contained function that represents unit of functionality. A service can exchange information from another service. It is not dependent on the state of another service.

## Service Connections

The figure given below illustrates the service oriented architecture. Service consumer sends service request to the service provider and service provider sends the service response to the service consumer. The service connection is understandable to both service consumer and service provider.



# **SOAP Web Services**

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

SOAP is a W3C recommendation for communication between two applications.

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

## Advantages of Soap Web Services

WS Security: SOAP defines its own security known as WS Security.

Language and Platform independent: SOAP web services can be written in any programming language and executed in any platform.

## Disadvantages of Soap Web Services

Slow: SOAP uses XML format that must be parsed to be read. It defines many standards that must be followed while developing the SOAP applications. So it is slow and consumes more bandwidth and resource.

**WSDL dependent**: SOAP uses WSDL and doesn't have any other mechanism to discover the service

**Rest architectural style:**

An architectural style provides a framework for thinking about solutions - in the case of REST, the concept of resources addressable through a URL, a vocabulary for manipulating those resources, statelessness etc.

This is important, because the question "is this a client server architecture" can easily be answered - is there a client, does it connect to the server, are there clear responsibilities for each? The question "is this a RESTful solution" is much harder to answer - it requires you to see if the solution follows the mental model of REST.

## Restful Web Services

Restful Web Services is a **stateless client-server** architecture where web services are resources and can be identified by their URIs.

REST Client applications can use HTTP GET/POST methods to invoke Restful web services. REST doesn’t specify any specific protocol to use, but in almost all cases it’s used over HTTP/HTTPS. When compared to SOAP web services, these are lightweight and doesn’t follow any standard. We can use XML, JSON, text or any other type of data for request and response.

### **Java RESTful Web Services API**

Java API for RESTful Web Services (JAX-RS) is the Java API for creating REST web services. JAX-RS uses annotations to simplify the development and deployment of web services. JAX-RS is part of JDK, so you don’t need to include anything to use it’s annotations.

### **Restful Web Services Annotations**

Some of the important JAX-RS annotations are:

* @Path: used to specify the relative path of class and methods. We can get the URI of a webservice by scanning the Path annotation value.
* @GET, @PUT, @POST, @DELETE and @HEAD: used to specify the HTTP request type for a method.
* @Produces, @Consumes: used to specify the request and response types.
* @PathParam: used to bind the method parameter to path value by parsing it.

### **Restful Web Services and SOAP**

1. SOAP is a protocol whereas REST is an architectural style.
2. SOAP server and client applications are tightly coupled and bind with the WSDL contract whereas there is no contract in REST web services and client.
3. Learning curve is easy for REST when compared to SOAP web services.
4. REST web services request and response types can be XML, JSON, text etc. whereas SOAP works with XML only.
5. JAX-RS is the Java API for REST web services whereas JAX-WS is the Java API for SOAP web services.

### **REST API Implementations**

There are two major implementations of JAX-RS API.

1. **Jersey**: [Jersey](https://jersey.github.io/) is the reference implementation provided by Sun. For using Jersey as our JAX-RS implementation, all we need to configure its servlet in web.xml and add required dependencies. Note that JAX-RS API is part of JDK not Jersey, so we have to add its dependency jars in our application.
2. **RESTEasy**: [RESTEasy](http://resteasy.jboss.org/" \t "_blank) is the JBoss project that provides JAX-RS implementation.

### **Java Restful Web Services Tutorial**

Let’s see how easy to create Restful web service using Jersey and then RESTEasy. We will be exposing following methods over HTTP and use Chrome Postman extension to test these.

|  |  |  |
| --- | --- | --- |
| URI | HTTP METHOD | DESCRIPTION |
| /person/{id}/getDummy | GET | Returns a dummy person object |
| /person/add | POST | Adds a person |
| /person/{id}/delete | GET | Delete the person with ‘id’ in the URI |
| /person/getAll | GET | Get all persons |
| /person/{id}/get | GET | Get the person with ‘id’ in the URI |

|  |
| --- |
| <dependencies>  <dependency>  <groupId>com.sun.jersey</groupId>  <artifactId>jersey-server</artifactId>  <version>1.19</version>  </dependency>  <dependency>  <groupId>com.sun.jersey</groupId>  <artifactId>jersey-servlet</artifactId>  <version>1.19</version>  </dependency>  <dependency>  <groupId>com.sun.jersey</groupId>  <artifactId>jersey-client</artifactId>  <version>1.19</version>  </dependency>  </dependencies> |

 the deployment descriptor to learn how to configure Jersey to create our web application

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd" id="WebApp\_ID" version="3.1">  <display-name>JAXRS-Example</display-name>  <!-- Jersey Servlet configurations -->  <servlet>  <servlet-name>Jersey REST Service</servlet-name>  <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>  <init-param>  <param-name>com.sun.jersey.config.property.packages</param-name>  <param-value>com.journaldev</param-value>  </init-param>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>Jersey REST Service</servlet-name>  <url-pattern>/\*</url-pattern>  </servlet-mapping>  <!-- Jersey Servlet configurations -->  </web-app> |

|  |
| --- |
| **service interface**  **public** **interface** PersonService {  **public** Response addPerson(Person p);  **public** Response deletePerson(**int** id);  **public** Person getPerson(**int** id);  **public** Person[] getAllPersons();  }  @Path("/person")  @Consumes(MediaType.APPLICATION\_XML)  @Produces(MediaType.APPLICATION\_XML)  **public** **class** PersonServiceImpl **implements** PersonService {  **private** **static** Map<Integer,Person> *persons* = **new** HashMap<Integer,Person>();      @Override  @POST  @Path("/add")  **public** Response addPerson(Person p) {  Response response = **new** Response();  **if**(*persons*.get(p.getId()) != **null**){  response.setStatus(**false**);  response.setMessage("Person Already Exists");  **return** response;  }  *persons*.put(p.getId(), p);  response.setStatus(**true**);  response.setMessage("Person created successfully");  **return** response;  }  @Override  @GET  @Path("/{id}/delete")  **public** Response deletePerson(@PathParam("id") **int** id) {  Response response = **new** Response();  **if**(*persons*.get(id) == **null**){  response.setStatus(**false**);  response.setMessage("Person Doesn't Exists");  **return** response;  }  *persons*.remove(id);  response.setStatus(**true**);  response.setMessage("Person deleted successfully");  **return** response;  }  @Override  @GET  @Path("/{id}/get")  **public** Person getPerson(@PathParam("id") **int** id) {  **return** *persons*.get(id);  }    @GET  @Path("/{id}/getDummy")  **public** Person getDummyPerson(@PathParam("id") **int** id) {  Person p = **new** Person();  p.setAge(99);  p.setName("Dummy");  p.setId(id);  **return** p;  }  @Override  @GET  @Path("/getAll")  **public** Person[] getAllPersons() {  Set<Integer> ids = *persons*.keySet();  Person[] p = **new** Person[ids.size()];  **int** i=0;  **for**(Integer id : ids){  p[i] = *persons*.get(id);  i++;  }  **return** p;  }  }  Model clasees:  @XmlRootElement  **public** **class** Response {  **private** **boolean** status;  **private** String message;  //setters and getters  }  @XmlRootElement (name="person")  **public** **class** Person {  **private** String name;  **private** **int** age;  **private** **int** id;  //setters and getters  } |

Most of the code is self explanatory, spend some time to familiarize yourself with JAX-RS annotations @Path, @PathParam, @POST, @GET, @Consumes and @Produces.

http://localhost:8080/JAXRS-Example/person/99/getDummy

http://localhost:8080/JAXRS-Example/person/add

http://localhost:8080/JAXRS-Example/person/99/get

http://localhost:8080/JAXRS-Example/person/99/delete

<http://localhost:8080/JAXRS-Example/person/getAll>

JAX-RS Annotations

The **javax.ws.rs** package contains JAX-RS annotations.

|  |  |
| --- | --- |
| **Annotation** | **Description** |
| Path | It identifies the URI path. It can be specified on class or method. |
| PathParam | represents the parameter of the URI path. |
| GET | specifies method responds to GET request. |
| POST | specifies method responds to POST request. |
| PUT | specifies method responds to PUT request. |
| HEAD | specifies method responds to HEAD request. |
| DELETE | specifies method responds to DELETE request. |
| OPTIONS | specifies method responds to OPTIONS request. |
| FormParam | represents the parameter of the form. |
| QueryParam | represents the parameter of the query string of an URL. |
| HeaderParam | represents the parameter of the header. |
| CookieParam | represents the parameter of the cookie. |
| Produces | defines media type for the response such as XML, PLAIN, JSON etc. It defines the media type that the methods of a resource class or MessageBodyWriter can produce. |
| Consumes | It defines the media type that the methods of a resource class or MessageBodyReader can produce. |

## JAX-RS @FormParam and @POST Annotation

|  |
| --- |
| @Path("/product")  **public** **class** ProductService{  @POST  @Path("/add")  **public** Response addUser(  @FormParam("id") **int** id,  @FormParam("name") String name,  @FormParam("price") **float** price) {    **return** Response.status(200)  .entity(" Product added successfuly!<br> Id: "+id+"<br> Name: " + name+"<br> Price: "+price)  .build();  }  }  <form action="rest/product/add" method="post">  Enter Id:<input type="text" name="id"/><br/><br/>  Enter Name:<input type="text" name="name"/><br/><br/>  Enter Price:<input type="text" name="price"/><br/><br/>  <input type="submit" value="Add Product"/>  </form> |

# **JAX-WS Tutorial**

SOAP Web Services

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

SOAP is a W3C recommendation for communication between two applications.

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

### **SOAP**

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

### **WSDL**

WSDL stands for Web Service Description Language. WSDL is an XML based document that provides technical details about the web service. Some of the useful information in WSDL document are: method name, port types, service end point, binding, method parameters etc.

### **UDDI**

UDDI is acronym for Universal Description, Discovery and Integration. UDDI is a directory of web services where client applications can lookup for web services. Web Services can register to the UDDI server and make them available to client applications.

## Top-Down vs. Bottom-Up

There are two ways of building SOAP web services. We can go with a top-down approach or a bottom-up approach.

In a top-down (contract-first) approach, a WSDL document is created, and the necessary Java classes are generated from the WSDL.

In a bottom-up (contract-last) approach, the Java classes are written, and the WSDL is generated from the WSDL.

Writing a WSDL file can be quite difficult depending on how complex your web service is. This makes the bottom-up approach an easier option. On the other hand, since your WSDL is generated from the Java classes, any change in code might cause a change in the WSDL. This is not the case for the top-down approach.

**Advantages of Soap Web Services**

WS Security: SOAP defines its own security known as WS Security.

Language and Platform independent: SOAP web services can be written in any programming language and executed in any platform

**Disadvantages of Soap Web Services**

Slow: SOAP uses XML format that must be parsed to be read. It defines many standards that must be followed while developing the SOAP applications. So it is slow and consumes more bandwidth and resource.

WSDL dependent: SOAP uses WSDL and doesn't have any other mechanism to discover the service.

**Example:**

|  |
| --- |
| **import** javax.jws.WebMethod;  **import** javax.jws.WebService;  **import** javax.jws.soap.SOAPBinding;  **import** com.journaldev.jaxws.beans.Person;  @WebService  @SOAPBinding(style = SOAPBinding.Style.***RPC***)  **public** **interface** PersonService {  @WebMethod  **public** **boolean** addPerson(Person p);    @WebMethod  **public** **boolean** deletePerson(**int** id);    @WebMethod  **public** Person getPerson(**int** id);    @WebMethod  **public** Person[] getAllPersons();  }  Endpoint service:  **import** java.util.HashMap;  **import** java.util.Map;  **import** java.util.Set;  **import** javax.jws.WebService;  **import** com.journaldev.jaxws.beans.Person;  @WebService(endpointInterface = "com.journaldev.jaxws.service.PersonService")  **public** **class** PersonServiceImpl **implements** PersonService {  **private** **static** Map<Integer,Person> *persons* = **new** HashMap<Integer,Person>();    @Override  **public** **boolean** addPerson(Person p) {  **if**(*persons*.get(p.getId()) != **null**) **return** **false**;  *persons*.put(p.getId(), p);  **return** **true**;  }  @Override  **public** **boolean** deletePerson(**int** id) {  **if**(*persons*.get(id) == **null**) **return** **false**;  *persons*.remove(id);  **return** **true**;  }  @Override  **public** Person getPerson(**int** id) {  **return** *persons*.get(id);  }  @Override  **public** Person[] getAllPersons() {  Set<Integer> ids = *persons*.keySet();  Person[] p = **new** Person[ids.size()];  **int** i=0;  **for**(Integer id : ids){  p[i] = *persons*.get(id);  i++;  }  **return** p;  }  }  Publish Class:  **import** javax.xml.ws.Endpoint;  **public** **class** SOAPPublisher {  **public** **static** **void** main(String[] args) {  Endpoint.*publish*("http://localhost:8888/ws/person", **new** PersonServiceImpl());  }  }  Pojo class:  **import** java.io.Serializable;  **public** **class** Person **implements** Serializable{  **private** **static** **final** **long** ***serialVersionUID*** = -5577579081118070434L;    **private** String name;  **private** **int** age;  **private** **int** id;  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }    @Override  **public** String toString(){  **return** id+"::"+name+"::"+age;  }  } |

### **JAX-WS Client Program**

If you look at the above program, we are using the server code itself. However web services just expose WSDL and third party applications don’t have access to these classes. So in that case, we can use wsimport utility to generate the client stubs. This utility comes with standard installation of JDK. Below image shows what all java classes we get when we run this utility.

# https://cdn.journaldev.com/wp-content/uploads/2015/10/wsimport-utility-parse-wsdl.png

|  |
| --- |
| **import** java.util.Arrays;  **import** com.journaldev.jaxws.service.Person;  **import** com.journaldev.jaxws.service.PersonService;  **import** com.journaldev.jaxws.service.PersonServiceImplService;  **public** **class** TestPersonService {  **public** **static** **void** main(String[] args) {    PersonServiceImplService serviceImpl = **new** PersonServiceImplService();    PersonService service = serviceImpl.getPersonServiceImplPort();    Person p1 = **new** Person(); p1.setName("Pankaj"); p1.setId(1); p1.setAge(30);  Person p2 = **new** Person(); p2.setName("Meghna"); p2.setId(2); p2.setAge(25);    System.***out***.println("Add Person Status="+service.addPerson(p1));  System.***out***.println("Add Person Status="+service.addPerson(p2));    //get person  System.***out***.println(service.getPerson(1));    //get all persons  System.***out***.println(Arrays.*asList*(service.getAllPersons()));    //delete person  System.***out***.println("Delete Person Status="+service.deletePerson(2));    //get all persons  System.***out***.println(Arrays.*asList*(service.getAllPersons()));    }  } |

# **SOAP vs REST Web Services**

There are many differences between SOAP and REST web services. The important 10 differences between SOAP and REST are given below:

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

| **SOAP** | **REST** |
| --- | --- |
| Simple Object Access Protocol (SOAP) serves as a standard protocol for web service creation. | Representational State Transfer (REST) is an architectural style for web service creation. |
| Web services and clients are tightly coupled and define some standards that are to be strictly followed. | It does not follow too many standards and is loosely coupled. |
| It requires more bandwidth and resource as well as uses service interfaces for exposing business logic. | It requires less bandwidth and resource as well as uses URI (Uniform Resource Identifiers) for exposing business logic. |
| It is usually less preferred and permits XML data format only. | It is usually more preferred and permits data formats like Plain text, HTML, JSON, etc. |
| Java API for SOAP web service is JAX-WS. | Java API for RESTFUL web service is JAX-RS. |
| SOAPUI can be used for testing SOAP web services. | Browsers and extensions such as Chrome postman are used for testing REST web services. |
| It defines its own security and uses WSDL contract for binding web services and client programs. | It does not have any defined contract as well as does not have its own security methods. |
| SOAP | REST |
| SOAP is a standard protocol for creating web services. | REST is an architectural style to create web services. |
| SOAP is acronym for Simple Object Access Protocol. | REST is acronym for REpresentational State Transfer. |
| SOAP uses WSDL to expose supported methods and technical details. | REST exposes methods through URIs, there are no technical details. |
| SOAP web services and client programs are bind with WSDL contract | REST doesn’t have any contract defined between server and client |
| SOAP web services and client are tightly coupled with contract. | REST web services are loosely coupled. |
| SOAP learning curve is hard, requires us to learn about WSDL generation, client stubs creation etc. | REST learning curve is simple, POJO classes can be generated easily and works on simple HTTP methods. |
| SOAP supports XML data format only | REST supports any data type such as XML, JSON, image etc. |
| SOAP web services are hard to maintain, any change in WSDL contract requires us to create client stubs again and then make changes to client code. | REST web services are easy to maintain when compared to SOAP, a new method can be added without any change at client side for existing resources. |
| SOAP web services can be tested through programs or software such as Soap UI. | REST can be easily tested through CURL command, Browsers and extensions such as Chrome Postman. |

**Transfer protocol(s):**

Soap supports transfer protocols: HTTP, SMTP, UDP, and others.3

Rest supports : Only HTTP

**Recommended for**

Soap: Enterprise apps, high-security apps, distributed environment, financial services, payment gateways, telecommunication services.

Rest: Public APIs for web services, mobile services, social networks.