Web Services:

* **Web services** are client and server applications that communicate over the World Wide Web's (WWW) HyperText Transfer Protocol (HTTP).
* Web services provide a standard means of interoperating between software applications running on a variety of platforms and frameworks.
* Web services can be combined in a loosely coupled way to achieve complex operations.
* A service is a software component provided through a network-accessible endpoint. The service consumer and provider use messages to exchange invocation request and response information
* There are two types of web Services

1. Big Web services
2. RESTful Web Services
3. Big Web services

* JAX-WS provides the functionality for "big" web services
* Big web services use XML messages that follow the Simple Object Access Protocol (SOAP) standard, an XML language defining a message architecture and message formats.
* Web Services Description Language (WSDL), an XML language for defining interfaces syntactically.
* SOAP message format and the WSDL interface definition language
* A SOAP-based design must include the following elements.
  + - A formal contract must be established to describe the interface that the web service offers. WSDL can be used to describe the details of the contract, which may include messages, operations, bindings, and the location of the web service.
    - The architecture needs to handle asynchronous processing and invocation. In such cases, the infrastructure provided by standards, such as Web Services Reliable Messaging (WSRM), and APIs, such as JAX-WS, with their client-side asynchronous invocation support, can be leveraged out of the box.

1. RESTful Web Services:

* JAX-RS provides the functionality for Representational State Transfer (RESTful) web services.
* REST is well suited for basic, ad hoc integration scenarios.
* RESTful web services, often better integrated with HTTP than SOAP-based services are, do not require XML messages or WSDL service-API definitions.
* Project Jersey is the production-ready reference implementation for the JAX-RS specification. Jersey implements support for the annotations defined in the JAX-RS specification, making it easy for developers to build RESTful web services with Java and the Java Virtual Machine (JVM).
* Because RESTful web services use existing well-known W3C and Internet Engineering Task Force (IETF) standards (HTTP, XML, URI, MIME) and have a lightweight infrastructure that allows services to be built with minimal tooling, developing RESTful web services is inexpensive and thus has a very low barrier for adoption.
* A RESTful design may be appropriate when the following conditions are met.
  + - The web services are completely stateless. A good test is to consider whether the interaction can survive a restart of the server.
    - The service producer and service consumer have a mutual understanding of the context and content being passed along. Because there is no formal way to describe the web services interface, both parties must agree out of band on the schemas that describe the data being exchanged and on ways to process it meaningfully. In the real world, most commercial applications that expose services as RESTful implementations also distribute so-called value-added toolkits that describe the interfaces to developers in popular programming languages.
    - REST is particularly useful for limited-profile devices, such as PDAs and mobile phones, for which the overhead of headers and additional layers of SOAP elements on the XML payload must be restricted.
    - Web service delivery or aggregation into existing websites can be enabled easily with a RESTful style. Developers can use such technologies as JAX-RS and Asynchronous JavaScript with XML (Ajax) and such toolkits as Direct Web Remoting (DWR) to consume the services in their web applications. Rather than starting from scratch, services can be exposed with XML and consumed by HTML pages without significantly refactoring the existing website architecture.

Deciding Which Type of Web Service to Use

* **JAX-WS**: Addresses advanced QoS requirements that commonly occur in enterprise computing. When compared to JAX-RS, JAX-WS makes it easier to support the WS-\* set of protocols, which provide standards for security and reliability, among other things, and interoperate with other WS-\* conforming clients and servers.
* **JAX-RS**: Makes it easier to write web applications that apply some or all of the constraints of the REST style to induce desirable properties in the application, such as loose coupling (evolving the server is easier without breaking existing clients), scalability (start small and grow), and architectural simplicity (use off-the-shelf components, such as proxies or HTTP routers). You would choose to use JAX-RS for your web application because it is easier for many types of clients to consume RESTful web services while enabling the server side to evolve and scale. Clients can choose to consume some or all aspects of the service and mash it up with other web-based services.
* You can use a development tool such as NetBeans IDE to further reduce the complexity of developing SOAP Based web services, RESTful web services.

Creating a Simple Web Service and Clients with JAX-WS

* The starting point for developing a JAX-WS web service is a Java class annotated with the javax.jws.WebService annotation.
* The @WebService annotation defines the class as a web service endpoint.
* A **service endpoint interface** or **service endpoint implementation** (SEI) is a Java interface or class,respectively, that declares the methods that a client can invoke on the service. An interface is not required when building a JAX-WS endpoint. The web service implementation class implicitly defines an SEI.
* You may specify an explicit interface by adding the endpointInterface element to the @WebService annotation in the implementation class.
* You must then provide an interface that defines the public methods made available in the endpoint implementation class.
* The basic steps for creating a web service and client are as follows:
  + - Code the implementation class.
    - Compile the implementation class.
    - Package the files into a WAR file.
    - Deploy the WAR file. The web service artifacts, which are used to communicate with clients, are generated by the GlassFish Server during deployment.
    - Code the client class.
    - Use a wsimport Ant task to generate and compile the web service artifacts needed to connect to the service.
    - Compile the client class.
    - Run the client.

Requirements of a JAX-WS Endpoint

1. The implementing class must be annotated with either the javax.jws.WebService or the javax.jws.WebServiceProvider annotation.
2. The implementing class may explicitly reference an SEI through the endpointInterface element of the @WebService annotation but is not required to do so. If no endpointInterface is specified in @WebService, an SEI is implicitly defined for the implementing class.
3. The business methods of the implementing class must be public and must not be declared static or final.
4. Business methods that are exposed to web service clients must be annotated with javax.jws.WebMethod.
5. Business methods that are exposed to web service clients must have JAXB-compatible parameters and return types. See the two tables of JAXB default data type bindings in [Types Supported by JAX-WS](http://docs.oracle.com/javaee/6/tutorial/doc/bnazc.html).
6. The implementing class must not be declared final and must not be abstract.
7. The implementing class must have a default public constructor.
8. The implementing class must not define the finalize method.
9. The implementing class may use the javax.annotation.PostConstruct or the javax.annotation.PreDestroy annotations on its methods for lifecycle event callbacks.
   * + The @PostConstruct method is called by the container before the implementing class begins responding to web service clients.
     + The @PreDestroy method is called by the container before the endpoint is removed from operation.

Coding the Service Endpoint Implementation Class

**package** com.test.webservice;

**import** javax.jws.WebMethod;

**import** javax.jws.WebService;

@WebService

**publicclass** HelloService {

**private** String message = "Hello, ";

**private** String message1 = "Weclcome to Rich & Poor Bank!";

@WebMethod

**public**String sayHello(String name) {

**return**message + name + ". \n" + message1;

}}

Building, Packaging, and Deploying the Service

* 1. Using Tomcat and Eclipse.

Deploy the project over tomcat.

* Go to Window > Preferences. Click on Java > Installed JREs and make sure, that a jdk-version is installed. Choose the jdk-version and click on Edit.... Check, if there is a file named tools.jar. If it does not, click on Add External JARs... and choose the file {sdk\_folder}/lib/tools.jar.

Next Steps

You can view the WSDL file of the deployed service by requesting the URLhttp://localhost:8080/helloservice/HelloService?wsdl in a web browser. Now you are ready to create a client that accesses this service.

java web services and its specifications such as JAX-WS and JAX-RS.

There are two ways to write the code for JAX-WS by RPC style and Document style. Like JAX-WS, JAX-RS can be written by Jersey and RESTeasy. We will learn all these technologies later.

There are three major web service components.

1. SOAP
2. WSDL
3. UDDI

# **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.

# **RESTful Web Services**

REST stands for REpresentational State Transfer.

REST is an architectural style not a protocol.

## Advantages of RESTful Web Services

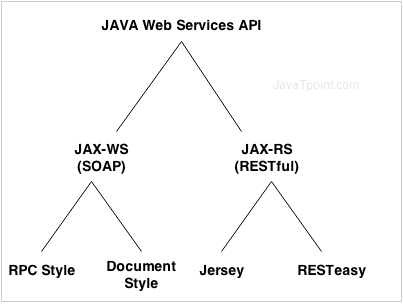
**Fast**: RESTful Web Services are fast because there is no strict specification like SOAP. It consumes less bandwidth and resource.

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

**Can use SOAP**: RESTful web services can use SOAP web services as the implementation.

**Permits different data format**: RESTful web service permits different data format such as Plain Text, HTML, XML and JSON.

|  |  |  |
| --- | --- | --- |
| **SOAP** | **REST** |  |
| SOAP is a **protocol**. | REST is an **architectural style**. |  |
| SOAP stands for **Simple Object Access Protocol**. | REST stands for **REpresentational State Transfer**. |  |
| 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. |  |
| SOAP **uses services interfaces to expose the business logic**. | REST **uses URI to expose business logic**. |  |
| **JAX-WS** is the java API for SOAP web services. | **JAX-RS** is the java API for RESTful web services. |  |
| SOAP **defines standards**to be strictly followed. | REST does not define too much standards like SOAP. |  |
| SOAP **requires more bandwidth** and resource than REST. | REST **requires less bandwidth** and resource than SOAP. |  |
| SOAP **defines its own security**. | RESTful web services **inherits security measures** from the underlying transport. |  |
| SOAP **permits XML** data format only. | REST **permits different** data format such as Plain text, HTML, XML, JSON etc. |  |
| SOAP is **less preferred** than REST. | REST **more preferred** than SOAP. |  |



# **Difference between RPC and Document web services**

There are many differences between RPC and Document web services. The important differences between RPC and Document are given below:

## RPC Style

1) RPC style web services use method name and parameters to generate XML structure.

2) The generated WSDL is **difficult to be validated** against schema.

3) In RPC style, SOAP **message is sent as many elements**.

4) RPC style message is **tightly coupled**.

5) In RPC style, SOAP message **keeps the operation name**.

6) In RPC style, parameters are sent as **discrete values**.

Let's see the RPC style generated WSDL file.

**WSDL file:**

In WSDL file, it doesn't specify the types details.

1. <types/>

For message part, it defines name and type attributes.

1. <message name="getHelloWorldAsString">
2. <part name="arg0" type="xsd:string"/>
3. </message>
4. <message name="getHelloWorldAsStringResponse">
5. <part name="return" type="xsd:string"/>
6. </message>

For soap:body, it defines use and namespace attributes.

1. <binding name="HelloWorldImplPortBinding" type="tns:HelloWorld">
2. <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
3. <operation name="getHelloWorldAsString">
4. <soap:operation soapAction=""/>
5. <input>
6. <soap:body use="literal" namespace="http://javatpoint.com/"/>
7. </input>
8. <output>
9. <soap:body use="literal" namespace="http://javatpoint.com/"/>
10. </output>
11. </operation>
12. </binding>

## Document Style

1) Document style web services **can be validated against predefined schema**.

2) In document style, SOAP message is **sent as a single document**.

3) Document style message is **loosely coupled**.

4) In Document style, SOAP message **loses the operation name**.

5) In Document style, parameters are sent in **XML format**.

Let's see the Document style generated WSDL file.

**WSDL file:**

In WSDL file, it specifies types details having namespace and schemaLocation.

1. <types>
2. <xsd:schema>
3. <xsd:**import** namespace="http://javatpoint.com/" schemaLocation="http://localhost:7779/ws/hello?xsd=1"/>
4. </xsd:schema>
5. </types>

For message part, it defines name and element attributes.

1. <message name="getHelloWorldAsString">
2. <part name="parameters" element="tns:getHelloWorldAsString"/>
3. </message>
4. <message name="getHelloWorldAsStringResponse">
5. <part name="parameters" element="tns:getHelloWorldAsStringResponse"/>
6. </message>

For soap:body, it defines use attribute only not namespace.

1. <binding name="HelloWorldImplPortBinding" type="tns:HelloWorld">
2. <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
3. <operation name="getHelloWorldAsString">
4. <soap:operation soapAction=""/>
5. <input>
6. <soap:body use="literal"/>
7. </input>
8. <output>
9. <soap:body use="literal"/>
10. </output>
11. </operation>
12. </binding>

# **JAX-WS Example RPC Style**

Creating JAX-WS example is a easy task because it requires no extra configuration settings.

JAX-WS API is inbuilt in JDK, so you don't need to load any extra jar file for it. Let's see a simple example of JAX-WS example in RPC style.

There are created 4 files for hello world JAX-WS example:

1. HelloWorld.java
2. HelloWorldImpl.java
3. Publisher.java
4. HelloWorldClient.java

The first 3 files are created for server side and 1 application for client side.

## JAX-WS Server Code

*File: HelloWorld.java*

1. **package** com.javatpoint;
2. **import** javax.jws.WebMethod;
3. **import** javax.jws.WebService;
4. **import** javax.jws.soap.SOAPBinding;
5. **import** javax.jws.soap.SOAPBinding.Style;
6. //Service Endpoint Interface
7. @WebService
8. @SOAPBinding(style = Style.RPC)
9. **public** **interface** HelloWorld{
10. @WebMethod String getHelloWorldAsString(String name);
11. }

*File: HelloWorldImpl.java*

1. **package** com.javatpoint;
2. **import** javax.jws.WebService;
3. //Service Implementation
4. @WebService(endpointInterface = "com.javatpoint.HelloWorld")
5. **public** **class** HelloWorldImpl **implements** HelloWorld{
6. @Override
7. **public** String getHelloWorldAsString(String name) {
8. **return** "Hello World JAX-WS " + name;
9. }
10. }

*File: Publisher.java*

1. **package** com.javatpoint;
2. **import** javax.xml.ws.Endpoint;
3. //Endpoint publisher
4. **public** **class** HelloWorldPublisher{
5. **public** **static** **void** main(String[] args) {
6. Endpoint.publish("http://localhost:7779/ws/hello", **new** HelloWorldImpl());
7. }
8. }

## How to view generated WSDL

After running the publisher code, you can see the generated WSDL file by visiting the URL:

1. http://localhost:7779/ws/hello?wsdl

## JAX-WS Client Code

*File: HelloWorldClient.java*

1. **package** com.javatpoint;
2. **import** java.net.URL;
3. **import** javax.xml.namespace.QName;
4. **import** javax.xml.ws.Service;
5. **public** **class** HelloWorldClient{
6. **public** **static** **void** main(String[] args) **throws** Exception {

    URL url = **new** URL("http://localhost:7779/ws/hello?wsdl");

        //1st argument service URI, refer to wsdl document above

    //2nd argument is service name, refer to wsdl document above

        QName qname = **new** QName("http://javatpoint.com/", "HelloWorldImplService");

        Service service = Service.create(url, qname);

        HelloWorld hello = service.getPort(HelloWorld.**class**);

        System.out.println(hello.getHelloWorldAsString("javatpoint rpc"));

     }

 }

Output:

Hello World JAX-WS javatpoint rpc

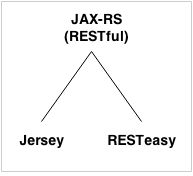
|  |
| --- |
| JAX-WS RPC style generated wsdl  <?xmlversion=*"1.0"*encoding=*"UTF-8"*?>  <!-- Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.1.6 in JDK 6. -->  <!-- Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.1.6 in JDK 6. -->  <definitionsname=*"HelloWorldImplService"*targetNamespace=*"http://javawsrpc.test.com/"*xmlns=*"http://schemas.xmlsoap.org/wsdl/"*xmlns:xsd=*"http://www.w3.org/2001/XMLSchema"*xmlns:tns=*"http://javawsrpc.test.com/"*xmlns:soap=*"http://schemas.xmlsoap.org/wsdl/soap/"*>  <types/>    <messagename=*"getHelloWorldAsString"*>  <partname=*"arg0"*type=*"xsd:string"*/>  </message>  <messagename=*"getHelloWorldAsStringResponse"*>  <partname=*"return"*type=*"xsd:string"*/>  </message>    <portTypename=*"HelloWorld"*>  <operationname=*"getHelloWorldAsString"*>  <inputmessage=*"tns:getHelloWorldAsString"*/>  <outputmessage=*"tns:getHelloWorldAsStringResponse"*/>  </operation>  </portType>  <bindingname=*"HelloWorldImplPortBinding"*type=*"tns:HelloWorld"*>  <soap:bindingstyle=*"rpc"*transport=*"http://schemas.xmlsoap.org/soap/http"*/>  <operationname=*"getHelloWorldAsString"*>  <soap:operationsoapAction=*""*/>  <input>  <soap:bodynamespace=*"http://javawsrpc.test.com/"*use=*"literal"*/>  </input>  <output>  <soap:bodynamespace=*"http://javawsrpc.test.com/"*use=*"literal"*/>  </output>  </operation>  </binding>    <servicename=*"HelloWorldImplService"*>  <portname=*"HelloWorldImplPort"*binding=*"tns:HelloWorldImplPortBinding"*>  <soap:addresslocation=*"http://localhost:9191/ws/hello"*/>  </port>  </service>  </definitions> |
| **JAX-WS Document style generated wsdl**  <?xmlversion=*"1.0"*encoding=*"UTF-8"*?>  <!-- Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.1.6 in JDK 6. -->  <!-- Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.1.6 in JDK 6. -->  <definitions name=*"HelloWorldImplService"* targetNamespace=[*http://javawsrpc.test.com/*](http://javawsrpc.test.com/)xmlns=[*http://schemas.xmlsoap.org/wsdl/*](http://schemas.xmlsoap.org/wsdl/)xmlns:xsd=[*http://www.w3.org/2001/XMLSchema*](http://www.w3.org/2001/XMLSchema)xmlns:tns=[*http://javawsrpc.test.com/*](http://javawsrpc.test.com/)xmlns:soap=*"http://schemas.xmlsoap.org/wsdl/soap/"*>  <types>  <xsd:schema>  <xsd:import namespace=[*http://javawsrpc.test.com/*](http://javawsrpc.test.com/)schemaLocation=*"http://localhost:9191/ws/hello?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 message=*"tns:getHelloWorldAsString"*/>  <output message=*"tns:getHelloWorldAsStringResponse"*/>  </operation>  </portType>  <binding name=*"HelloWorldImplPortBinding"* type=*"tns:HelloWorld"*>  <soap:binding style=*"document"* transport=*"http://schemas.xmlsoap.org/soap/http"*/>  <operation name=*"getHelloWorldAsString"*>  <soap:operation soapAction=*""*/>  <input>  <soap:bodyuse=*"literal"*/>  </input>  <output>  <soap:body use=*"literal"*/>  </output>  </operation>  </binding>    <service name=*"HelloWorldImplService"*>  <port name=*"HelloWorldImplPort"* binding=*"tns:HelloWorldImplPortBinding"*>  <soap:address location=*"http://localhost:9191/ws/hello"*/>  </port>  </service>  </definitions> |

# **JAX-RS Tutorial**

**JAX-RS tutorial** is provides concepts and examples of JAX-RS API. This JAX-RS tutorial is designed for beginners and professionals.

There are two main implementation of JAX-RS API.

1. Jersey
2. RESTEasy



J2SE 8=52,

J2SE 7=51,

J2SE 6.0=50,

J2SE 5.0=49,

JDK 1.4=48,

JDK 1.3=47,

JDK 1.2=46,

JDK 1.1=45

RESTful Web service can be implemented with

* JAX-RS specification,
* Jersey is the one implementation for the JAX-RS specification.

RESTful application generally build in one of below two ways:

1. With Jersey implementation
2. With JAXB
3. With Jersey implementation

* Below jars are required for jdk 1.6 for the service code and the client code.

**jersey-bundle-1.9.jar (Jersey implementation)**

**jsr311-api-1.1.1.jar (JAX-RS specification)**

Example:

1. Hello.java
2. **Hello.java**

**package** com.test.restfuljersey;

**import** javax.ws.rs.GET;

**import** javax.ws.rs.Path;

**import** javax.ws.rs.Produces;

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

@Path("/hello")

**publicclass** Hello {

@GET

@Produces(MediaType.*TEXT\_PLAIN*)

**public** String sayPlainTextHello() {

**return**"Hello Jersey Plain";

}

// This method is called if XML is request

@GET

@Produces(MediaType.*TEXT\_XML*)

**public** String sayXMLHello() {

**return**"<?xml version=\"1.0\"?>" + "<hello> Hello Jersey" + "</hello>";

}

// This method is called if HTML is request

@GET

@Produces(MediaType.*TEXT\_HTML*)

**public** String sayHtmlHello() {

**return**"<html> " + "<title>" + "Hello Jersey" + "</title>"

+ "<body><h1>" + "Hello Jersey HTML" + "</h1></body>"

+ "</html> ";

}

}

1. **web.xml**

<?xmlversion=*"1.0"*encoding=*"UTF-8"*?>

<web-appxmlns: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>restfuljersey1</display-name>

<servlet>

<servlet-name>Jersey REST Service</servlet-name>

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

<!-- Register resources and providers under com.vogella.jersey.first package. -->

<init-param>

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

<param-value>com.test.restfuljersey</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>Jersey REST Service</servlet-name>

<url-pattern>/rest/\*</url-pattern>

</servlet-mapping>

</web-app>

1. **ClientTest.java**

**package** com.test.restclient;

**import** java.net.URI;

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

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

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

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

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

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

**publicclass** ClientTest {

**publicstaticvoid** main(String[] args) {

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

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

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

System.*out*.println(service.path("rest").path("hello").accept(MediaType.*TEXT\_PLAIN*).get(String.**class**));

System.*out*.println(service.path("rest").path("hello").accept(MediaType.*TEXT\_XML*).get(String.**class**));

System.*out*.println(service.path("rest").path("hello").accept(MediaType.*TEXT\_HTML*).get(String.**class**));

}

**privatestatic** URI getBaseURI() {

**return** UriBuilder.*fromUri*("http://localhost:9090/restfuljersey1").build();

}

}

1. With JAXB

* Below jar is required for jdk 1.6 for the service code.

**jsr311-api-1.1.1.jar**

* Below jar is required for jdk 1.6 for the client code.

**jsr311-api-1.1.1.jar**

**jersey-bundle-1.9.jar**

Example:

1. Todo.java
2. TodoResource.java
3. web.xml
4. TestRest.java
5. Todo.java

**package** com.test.restfuljaxb;

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

@XmlRootElement

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

// JSON

// Isn't that cool?

**public class** Todo {

**private** String summary;

**private** String description;

**public** String getSummary() {

**return**summary;

}

**public void** setSummary(String summary) {

**this**.summary = summary;

}

**public** String getDescription() {

**return**description;

}

**public void** setDescription(String description) {

**this**.description = description;

}

}

**2.TodoResource.java**

**package** com.test.restfuljaxb;

**import** javax.ws.rs.GET;

**import** javax.ws.rs.Path;

**import** javax.ws.rs.Produces;

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

@Path("/todo")

**publicclass** TodoResource {

// This method is called if XMLis request

@GET

@Produces({ MediaType.*APPLICATION\_XML*, MediaType.*APPLICATION\_JSON* })

**public** Todo getXML() {

Todo todo = **new** Todo();

todo.setSummary("This is my first todo");

todo.setDescription("This is my first todo");

**return** todo;

}

// This can be used to test the integration with the browser

@GET

@Produces({ MediaType.*TEXT\_XML* })

**public** Todo getHTML() {

Todo todo = **new** Todo();

todo.setSummary("This is my first todo");

todo.setDescription("This is my first todo");

**return** todo;

}

}

1. web.xml

<?xmlversion=*"1.0"*encoding=*"UTF-8"*?>

<web-appxmlns: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>restfuljaxb</display-name>

<servlet>

<servlet-name>Jersey REST Service</servlet-name>

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

<!-- Register resources and providers under com.vogella.jersey.first package. -->

<init-param>

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

<param-value>com.test.restfuljaxb</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>Jersey REST Service</servlet-name>

<url-pattern>/rest/\*</url-pattern>

</servlet-mapping>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

</web-app>

1. TestRest.java

**package** com.restfuljaxb.client;

**import** java.net.URI;

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

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

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

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

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

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

**publicclass** TestRest {

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

Client Config config = **new** DefaultClientConfig();

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

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

System.*out*.println(service.path("rest").path("todo").accept(MediaType.*TEXT\_XML*).get(String.**class**));

}

**private static** URI getBaseURI() {

**return** UriBuilder.*fromUri*("http://localhost:9090/restfuljaxb").build();

}

}

**RESTful web service**

A concrete implementation of a REST Web service follows four basic design principles:

* Use HTTP methods explicitly.
* Be stateless.
* Expose directory structure-like URIs.
* Transfer XML, JavaScript Object Notation (JSON), or both.

**Use HTTP methods explicitly**

One of the key characteristics of a RESTful Web service is the explicit use of HTTP methods in a way that follows the protocol as defined by RFC 2616. HTTP GET, for instance, is defined as a data-producing method that's intended to be used by a client application to retrieve a resource, to fetch data from a Web server, or to execute a query with the expectation that the Web server will look for and respond with a set of matching resources.

REST asks developers to use HTTP methods explicitly and in a way that's consistent with the protocol definition. This basic REST design principle establishes a one-to-one mapping between create, read, update, and delete (CRUD) operations and HTTP methods. According to this mapping:

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

An unfortunate design flaw inherent in many Web APIs is in the use of HTTP methods for unintended purposes. The request URI in an HTTP GET request, for example, usually identifies one specific resource. Or the query string in a request URI includes a set of parameters that defines the search criteria used by the server to find a set of matching resources. At least this is how the HTTP/1.1 RFC describes GET. But there are many cases of unattractive Web APIs that use HTTP GET to trigger something transactional on the server—for instance, to add records to a database. In these cases the GET request URI is not used properly or at least not used RESTfully. If the Web API uses GET to invoke remote procedures, it looks like this:  
  
GET /adduser?name=Robert HTTP/1.1

It's not a very attractive design because the Web method above supports a state-changing operation over HTTP GET. Put another way, the HTTP GET request above has side effects. If successfully processed, the result of the request is to add a new user—in this example, Robert—to the underlying data store. The problem here is mainly semantic. Web servers are designed to respond to HTTP GET requests by retrieving resources that match the path (or the query criteria) in the request URI and return these or a representation in a response, not to add a record to a database. From the standpoint of the intended use of the protocol method then, and from the standpoint of HTTP/1.1-compliant Web servers, using GET in this way is inconsistent.

Beyond the semantics, the other problem with GET is that to trigger the deletion, modification, or addition of a record in a database, or to change server-side state in some way, it invites Web caching tools (crawlers) and search engines to make server-side changes unintentionally simply by crawling a link. A simple way to overcome this common problem is to move the parameter names and values on the request URI into XML tags. The resulting tags, an XML representation of the entity to create, may be sent in the body of an HTTP POST whose request URI is the intended parent of the entity (see Listings 1 and 2).

**Listing 1. Before**

GET /adduser?name=Robert HTTP/1.1

**Listing 2. After**

POST /users HTTP/1.1

Host: myserver

Content-Type: application/xml

<?xml version="1.0"?>

<user>

<name>Robert</name>

</user>

The method above is exemplary of a RESTful request: proper use of HTTP POST and inclusion of the payload in the body of the request. On the receiving end, the request may be processed by adding the resource contained in the body as a subordinate of the resource identified in the request URI; in this case the new resource should be added as a child of /users. This containment relationship between the new entity and its parent, as specified in the POST request, is analogous to the way a file is subordinate to its parent directory. The client sets up the relationship between the entity and its parent and defines the new entity's URI in the POST request.

**Common MIME types used by RESTful services**

| **MIME-Type** | **Content-Type** |
| --- | --- |
| **JSON** | application/json |
| **XML** | application/xml |
| **XHTML** | application/xhtml+xml |

### Statelessness

A RESTful service is stateless and does not maintain the application state for any client. A request cannot be dependent on a past request and a service treats each request independently. HTTP is a stateless protocol by design and you need to do something extra to implement a stateful service using HTTP. But it is really easy to implement stateful services with current technologies. We need a clear understanding of a stateless and stateful design so that we can avoid misinterpretation.

A stateless design looks like so:

Request1: GET http://MyService/Persons/1 HTTP/1.1

Request2: GET http://MyService/Persons/2 HTTP/1.1

Each of these requests can be treated separately.

A stateful design, on the other hand, looks like so:

Request1: GET http://MyService/Persons/1 HTTP/1.1

Request2: GET http://MyService/NextPerson HTTP/1.1

To process the second request, the server needs to remember the last PersonID that the client fetched. In other words, the server needs to remember the current state — otherwise Request2 cannot be processed. Design your service in a way that a request never refers to a previous request. Stateless services are easier to host, easy to maintain, and more scalable. Plus, such services can provide better response time to requests, as it is much easier to load balance them.

<definitions>

<types>

definition of types........

</types>

<message>

definition of a message....

</message>

<portType>

<operation>

definition of a operation.......

</operation>

</portType>

<binding>

definition of a binding....

</binding>

<service>

definition of a service....

</service>

</definitions>

WSDL Structure:

There are five parts of WSDL file.

1. types
2. message
3. portType
4. binding
5. service
6. Define the data types used by your services.
7. Define the messages used by your services.
8. Define the interfaces for your services.
9. Define the bindings between the messages used by each interface and the concrete representation of the data on the wire.
10. Define the transport details for each of the services.

**<types>**

**<schema**...**>**

...

**<element**name="personalLookup"**>**

**<complexType>**

**<sequence>**

**<element**name="empID"type="xsd:int" **/>**

**</sequence>**

**</complexType>**

**</element>**

**<element**name="personalLookupResponse"**>**

**<complexType>**

**<sequence>**

**<element**name="return"type="personalInfo" **/>**

**</sequence>**

**</complexType>**

**</element>**

**</schema>**

**</types>**

**<message**name="personalLookupRequest"**>**

**<part**name="empId"element="xsd1:personalLookup"**/>**

**<message/>**

**<message**name="personalLookupResponse"**>**

**<part**name="return"element="xsd1:personalLookupResponse"**/>**

**<message/>**

A WSDL document is, at its simplest, a collection of elements contained within a root definition element. These elements describe a service and how an endpoint implementing that service is accessed.

A WSDL document has two distinct parts:

* [A logical part](https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.0/html/Writing_WSDL_Contracts/files/WSDLStructure.html#WSDLIntroLogicalPart) that defines the service in implementation neutral terms
* [A concrete part](https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.0/html/Writing_WSDL_Contracts/files/WSDLStructure.html#WSDLIntroPhysicalPart) that defines how an endpoint implementing the service is exposed on a network

### The logical part

The logical part of a WSDL document contains the types, the message, and the portType elements.

* portType describes the service’s interface and
* The portType element contains one or more operation elements that define the messages sent by the operations exposed by the service
* message describes the messages exchanged by the service.
* Within the types element, XML Schema is used to define the structure of the data that makes up the messages.

A number of message elements are used to define the structure of the messages used by the service.

### The concrete part

The concrete part of a WSDL document contains the binding and the service elements.

* It describes how an endpoint that implements the service connects to the outside world.
* The binding elements describe how the data units described by the message elements are mapped into a concrete, on-the-wire data format, such as SOAP.
* The service elements contain one or more port elements which define the endpoints implementing the service.

A WSDL document is made up of the following elements:

* definitions — The root element of a WSDL document. The attributes of this element specify the name of the WSDL document, the document’s target namespace, and the shorthand definitions for the namespaces referenced in the WSDL document.
* types — The XML Schema definitions for the data units that form the building blocks of the messages used by a service. For information about defining data types see [Defining Logical Data Units](https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.0/html/Writing_WSDL_Contracts/files/WSDLTypes.html).
* message — The description of the messages exchanged during invocation of a services operations. These elements define the arguments of the operations making up your service. For information on defining messages see [Defining Logical Messages Used by a Service](https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.0/html/Writing_WSDL_Contracts/files/WSDLMessages.html).
* portType — A collection of operation elements describing the logical interface of a service. For information about defining port types see [Defining Your Logical Interfaces](https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.0/html/Writing_WSDL_Contracts/files/WSDLInterfaces.html).
* operation — The description of an action performed by a service. Operations are defined by the messages passed between two endpoints when the operation is invoked. For information on defining operations see [Operations](https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.0/html/Writing_WSDL_Contracts/files/WSDLInterfaces.html#WSDLOperationElement).
* binding — The concrete data format specification for an endpoint. A binding element defines how the abstract messages are mapped into the concrete data format used by an endpoint. This element is where specifics such as parameter order and return values are specified.
* service — A collection of related port elements. These elements are repositories for organizing endpoint definitions.
* port — The endpoint defined by a binding and a physical address. These elements bring all of the abstract definitions together, combined with the definition of transport details, and they define the physical endpoint on which a service is exposed.

<types> -> what data type will be transmitted

<messsage> -> What message will be transmitted may say input or output parameters for the function

<portType> -> what operation or function has been exposed or can be called

<binding> -> how the operations or functions will be exposed or called: what protocol would be used.

<service> -> where is the service located: on which port, url

Some other tags:

<import> -> to import other wsdl documents or xml schemas

<message>

<part> => function parameter

<sequence> => a list of sub element (it also gives the order of its appearance).

Operation Types:

The wsdl port describes the interfaces(legal operations) exposed by the web service.

Types of operation:

1. one-way => receive a message but not return message (response).
2. Request-response => receive a request and return a response.
3. Solicit-response => send a request and wait for response.
4. Notification => send a message but not wait for response.
5. One-way operation:

<portType name = “”>

<operation name = “”>

<input name = “” message = “” />

</operation>

</portType>

1. Request-Response Operation

<portType name = “”>

<operation name = “”>

<input name = “” message = “” />

<output message = “”/>

</operation>

<portType>

WSDL Binding:

Wsdl binding defines the message format and protocol details for the web service.

binding elements has two attributes

* 1. Name => name of the binding
  2. Type => port for the binding

<soap:binding> element has two attributes: style (rpc or document) and transport (soap protocol to be used)

Operation:

Operation element defines each operation that has been exposed at the port.

Example:

<message name = “getTermRequest”>

<part name = “term” type = “xs:string”/>

</message>

<message name = “getTermResponse”>

<part name = “value” type = “xs:string”/>

</message>

<portType name = “glossaryTerms”>

<operation name = “getTerm”>

<input message = “getTermRequest” />

<output message = “getTermResponse” />

</operation>

</portType>

<binding type = “glossaryTerms” name = “b1”>

<soap:binding style = “document”

transport = “http://schemas.xmlsoap.org/soap/http” /> <operation>

<soap:operation soapAction =

“http://example.com/getTerm” />

<input><soap:body use = “literal” /></input>

<output><soap:body use = “literal” /></output> </operation>

</binding>

Note:

Question: From eclipse cannot start the tomcat server

Answer:

Change the Server Location in Eclipse from Use workspace metadata to Use Tomcat installation