JAX-RS –

**JAX-RS** stands for JAVA API for RESTful Web Services

JAX-WS libs are already present in JDK….so that directly by importing …we can use it

But JAX-RS jars , we need to add in lib ..to use in creation of webservices

difference between query param and matrix param

Queryparam:

url ? key=value;

Matrixparam

url; key=value;

What is annotation :

1. What are RESTful webservices?

Web services based on REST Architecture are known as RESTful web services. These web services use HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service,

* REST is stateless, therefore the SERVER has no state (or session data)

|  |  |
| --- | --- |
| SOAP supports XML data format only | REST supports any data type such as XML, JSON, image etc. |
| SOAP is a standard protocol for creating web services. | REST is an architectural style to create web services. |
| 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. |
| * SOAP is slow because payload is large for a simple string message, since it uses XML format. | REST is fatser |

**Accept** headers tells web service what kind of response client is accepting

**Content-Type** header is used to tell server what is the format of data being sent in the request

**How would you choose between SOAP and REST web services?**

Web Services work on client-server model and when it comes to choose between SOAP and REST, it all depends on project requirements. Let’s look at some of the conditions affecting our choice:

* Do you know your web service clients beforehand? If Yes, then you can define a contract before implementation and SOAP seems better choice. But if you don’t then REST seems better choice because you can provide sample request/response and test cases easily for client applications to use later on.
* How much time you have? For quick implementation REST is the best choice. You can create web service easily, test it through browser/curl and get ready for your clients.
* What kind of data format are supported? If only XML then you can go with SOAP but if you think about supporting JSON also in future then go with REST.

Some of the important annotations used in JAX-WS API are:

* @WebService
* @SOAPBinding
* @WebMethod

<http://localhost:8888/testWS?wsdl>

There are two major implementations of JAX-RS API.

1. Jersey: Jersey 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 is the JBoss project that provides JAX-RS implementation.

# web.xml Servlet Configuration

**http://tutorials.jenkov.com/java-servlets/web-xml.html**

import javax.xml.bind.annotation.XmlElement;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "user")

import javax.ws.rs.Produces; -- all annotations

import javax.ws.rs.core.MediaType;

**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>User Management</display-name>

<servlet>

<servlet-name>Jersey RESTful Application</servlet-name>

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

<init-param>

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

<param-value>com.tutorialspoint</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>Jersey RESTful Application</servlet-name>

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

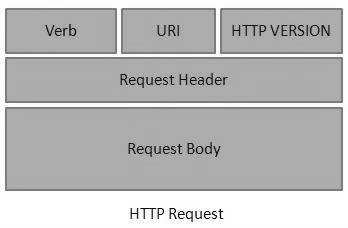
</servlet-mapping>

</web-app>

RESTful Web Services - Messages

A client sends a message in form of a HTTP Request and the server responds in the form of an HTTP Response. This technique is termed as Messaging. These messages contain message data and metadata i.e. information about message itself. Let us have a look on the HTTP Request and HTTP Response messages for HTTP 1.1.

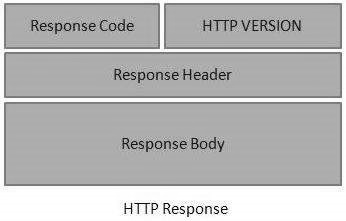
## **HTTP Request**



An HTTP Request has five major parts −

* **Verb** − Indicates the HTTP methods such as GET, POST, DELETE, PUT, etc.
* **URI** − Uniform Resource Identifier (URI) to identify the resource on the server.
* **HTTP Version** − Indicates the HTTP version. For example, HTTP v1.1.
* **Request Header** − Contains metadata for the HTTP Request message as key-value pairs. For example, client (or browser) type, format supported by the client, format of the message body, cache settings, etc.
* **Request Body** − Message content or Resource representation.

## **HTTP Response**



An HTTP Response has four major parts −

* **Status/Response Code** − Indicates the Server status for the requested resource. For example, 404 means resource not found and 200 means response is ok.
* **HTTP Version** − Indicates the HTTP version. For example HTTP v1.1.
* **Response Header** − Contains metadata for the HTTP Response message as keyvalue pairs. For example, content length, content type, response date, server type, etc.
* **Response Body** − Response message content or Resource representation.



**Addressing**

Addressing refers to locating a resource or multiple resources lying on the server. It is analogous to locate a postal address of a person.

Each resource in REST architecture is identified by its URI (Uniform Resource Identifier). A URI is of the following format −

<protocol>://<service-name>/<ResourceType>/<ResourceID>

# Statelessness

RESTful Web Service should not keep a client state on the server. This restriction is called Statelessness. It is the responsibility of the client to pass its context to the server and then the server can store this context to process the client's further request

### @GET

Annotate your Get request methods with @GET.

|  |  |
| --- | --- |
| 1  2  3  4 | @GET  public String getHTML() {    ...  } |

### @Produces

@Produces annotation specifies the type of output this method (or web service) will produce.

|  |  |
| --- | --- |
| 1  2  3  4  5 | @GET  @Produces("application/xml")  public Contact getXML() {    ...  } |
| 1  2  3  4 | @GET  @Produces("application/json")  public Contact getJSON() {    ...  }  More than one media type may be declared in the same @Produces declaration. The following code example shows how this is done:  @Produces({"application/xml", "application/json"}) |

### @Path

@Path annotation specify the URL path on which this method will be invoked.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @GET  @Produces("application/xml")  @Path("xml/{firstName}")  public Contact getXML() {    ...  } |

### @PathParam

We can bind REST-style URL parameters to method arguments using @PathParam annotation as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @GET  @Produces("application/xml")  @Path("xml/{firstName}")  public Contact getXML(@PathParam("firstName") String firstName) {    Contact contact = contactService.findByFirstName(firstName);    return contact;  } |
| 1  2  3  4  5  6  7 | @GET  @Produces("application/json")  @Path("json/{firstName}")  public Contact getJSON(@PathParam("firstName") String firstName) {    Contact contact = contactService.findByFirstName(firstName);    return contact;  } |

### @QueryParam

Request parameters in query string can be accessed using @QueryParam annotation as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @GET  @Produces("application/json")  @Path("json/companyList")  public CompanyList getJSON(@QueryParam("start") int start, @QueryParam("limit") int limit) {    CompanyList list = new CompanyList(companyService.listCompanies(start, limit));    return list;  } |

### @POST

Annotate POST request methods with @POST.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @POST  @Consumes("application/json")  @Produces("application/json")  public RestResponse<Contact> create(Contact contact) {  ...  } |

### @Consumes

The @Consumes annotation is used to specify the MIME media types a REST resource can consume.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @PUT  @Consumes("application/json")  @Produces("application/json")  @Path("{contactId}")  public RestResponse<Contact> update(Contact contact) {  ...  } |

### @FormParam

The REST resources will usually consume XML/JSON for the complete Entity Bean. Sometimes, you may want to read parameters sent in POST requests directly and you can do that using @FormParam annotation. GET Request query parameters can be accessed using [@QueryParam](http://www.techferry.com/articles/RESTful-web-services-JAX-RS-annotations.html#QueryParam) annotation.

|  |  |
| --- | --- |
| 1  2  3  4  5 | @POST  public String save(@FormParam("firstName") String firstName,      @FormParam("lastName") String lastName) {        ...    } |

### @PUT

Annotate PUT request methods with @PUT.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @PUT  @Consumes("application/json")  @Produces("application/json")  @Path("{contactId}")  public RestResponse<Contact> update(Contact contact) {  ...  } |

### @DELETE

Annotate DELETE request methods with @DELETE.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @DELETE  @Produces("application/json")  @Path("{contactId}")  public RestResponse<Contact> delete(@PathParam("contactId") int contactId) {  ...  } |

Sample Code :  href="rest/hello/2014/12/05"

1. **package** com.javatpoint.rest;
2. **import** javax.ws.rs.GET;
3. **import** javax.ws.rs.Path;
4. **import** javax.ws.rs.PathParam;
5. **import** javax.ws.rs.core.Response;
6. @Path("/hello")
7. **public** **class** HelloService{
8. @GET
9. @Path("{year}/{month}/{day}")
10. **public** Response getDate(
11. @PathParam("year") **int** year,
12. @PathParam("month") **int** month,
13. @PathParam("day") **int** day) {
15. String date = year + "/" + month + "/" + day;
17. **return** Response.status(200)
18. .entity("getDate is called, year/month/day : " + date)
19. .build();
20. }
21. }