## Types of Web Services

On the conceptual level, 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 in the form of self-containing documents that make very few assumptions about the technological capabilities of the receiver.

On a technical level, web services can be implemented in various ways. The two types of web services discussed in this section can be distinguished as “big” web services and “RESTful” web services.

### “Big” Web Services

In Java EE 6, JAX-WS provides the functionality for “big” web services, which are described in [Chapter 19, Building Web Services with JAX-WS](https://docs.oracle.com/javaee/6/tutorial/doc/bnayl.html). 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. Such systems often contain a machine-readable description of the operations offered by the service, written in the Web Services Description Language (WSDL), an XML language for defining interfaces syntactically.

The SOAP message format and the WSDL interface definition language have gained widespread adoption. Many development tools, such as NetBeans IDE, can reduce the complexity of developing web service applications.

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. You may also process SOAP messages in a JAX-WS service without publishing a WSDL.
* The architecture must address complex nonfunctional requirements. Many web service specifications address such requirements and establish a common vocabulary for them. Examples include transactions, security, addressing, trust, coordination, and so on.
* 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.

### RESTful Web Services

In Java EE 6, 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. You can use a development tool such as NetBeans IDE to further reduce the complexity of developing RESTful web services.

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.
* A caching infrastructure can be leveraged for performance. If the data that the web service returns is not dynamically generated and can be cached, the caching infrastructure that web servers and other intermediaries inherently provide can be leveraged to improve performance. However, the developer must take care because such caches are limited to the HTTP GET method for most servers.
* 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.
* Bandwidth is particularly important and needs to be limited. 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 web sites 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 web site architecture. Existing developers will be more productive because they are adding to something they are already familiar with rather than having to start from scratch with new technology.

RESTful web services are discussed in [Chapter 20, Building RESTful Web Services with JAX-RS](https://docs.oracle.com/javaee/6/tutorial/doc/giepu.html). This chapter contains information about generating the skeleton of a RESTful web service using both NetBeans IDE and the Maven project management tool.

## Deciding Which Type of Web Service to Use

Basically, you would want to use RESTful web services for integration over the web and use big web services in enterprise application integration scenarios that have advanced quality of service (QoS) requirements.

* **JAX-WS**: addresses advanced QoS requirements commonly occurring 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 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.

### Chapter 19 Building Web Services with JAX-WS

Java API for XML Web Services (JAX-WS) is a technology for building web services and clients that communicate using XML. JAX-WS allows developers to write message-oriented as well as Remote Procedure Call-oriented (RPC-oriented) web services.

In JAX-WS, a web service operation invocation is represented by an XML-based protocol, such as SOAP. The SOAP specification defines the envelope structure, encoding rules, and conventions for representing web service invocations and responses. These calls and responses are transmitted as SOAP messages (XML files) over HTTP.

Although SOAP messages are complex, the JAX-WS API hides this complexity from the application developer. On the server side, the developer specifies the web service operations by defining methods in an interface written in the Java programming language. The developer also codes one or more classes that implement those methods. Client programs are also easy to code. A client creates a proxy (a local object representing the service) and then simply invokes methods on the proxy. With JAX-WS, the developer does not generate or parse SOAP messages. It is the JAX-WS runtime system that converts the API calls and responses to and from SOAP messages.

With JAX-WS, clients and web services have a big advantage: the platform independence of the Java programming language. In addition, JAX-WS is not restrictive: A JAX-WS client can access a web service that is not running on the Java platform, and vice versa. This flexibility is possible because JAX-WS uses technologies defined by the W3C: HTTP, SOAP, and WSDL. WSDL specifies an XML format for describing a service as a set of endpoints operating on messages.

## Creating a Simple Web Service and Clients with JAX-WS

This section shows how to build and deploy a simple web service and two clients: an application client and a web client. The source code for the service is in the *tut-install*/examples/jaxws/helloservice/ directory, and the clients are in the *tut-install*/examples/jaxws/appclient/ and *tut-install*/examples/jaxws/webclient/ directories.

[Figure 19-1](https://docs.oracle.com/javaee/6/tutorial/doc/bnayn.html#bnayo) illustrates how JAX-WS technology manages communication between a web service and a client.

**Figure 19-1 Communication between a JAX-WS Web Service and a Client**



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:

1. Code the implementation class.
2. Compile the implementation class.
3. Package the files into a WAR file.
4. Deploy the WAR file. The web service artifacts, which are used to communicate with clients, are generated by the GlassFish Server during deployment.
5. Code the client class.
6. Use a wsimport Ant task to generate and compile the web service artifacts needed to connect to the service.
7. Compile the client class.
8. Run the client.

If you use NetBeans IDE to create a service and client, the IDE performs the wsimport task for you.

The sections that follow cover these steps in greater detail.

### Requirements of a JAX-WS Endpoint

JAX-WS endpoints must follow these requirements.

* The implementing class must be annotated with either the javax.jws.WebService or the javax.jws.WebServiceProvider annotation.
* 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.
* The business methods of the implementing class must be public and must not be declared static or final.
* Business methods that are exposed to web service clients must be annotated with javax.jws.WebMethod.
* 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](https://docs.oracle.com/javaee/6/tutorial/doc/bnazc.html).
* The implementing class must not be declared final and must not be abstract.
* The implementing class must have a default public constructor.
* The implementing class must not define the finalize method.
* 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

In this example, the implementation class, Hello, is annotated as a web service endpoint using the @WebService annotation. Hello declares a single method named sayHello, annotated with the @WebMethod annotation, which exposes the annotated method to web service clients. The sayHello method returns a greeting to the client, using the name passed to it to compose the greeting. The implementation class also must define a default, public, no-argument constructor.

package helloservice.endpoint;

import javax.jws.WebService;

import javax.jws.WebMethod;

@WebService

public class Hello {

private String message = new String("Hello, ");

public void Hello() {

}

@WebMethod

public String sayHello(String name) {

return message + name + ".";

}

}

### Building, Packaging, and Deploying the Service

You can use either NetBeans IDE or Ant to build, package, and deploy the helloservice application.

#### To Build, Package, and Deploy the Service Using NetBeans IDE

1. **From the File menu, choose Open Project.**
2. **In the Open Project dialog, navigate to:**

*tut-install*/examples/jaxws/

1. **Select the helloservice folder.**
2. **Select the Open as Main Project check box.**
3. **Click Open Project.**
4. **In the Projects tab, right-click the helloservice project and select Deploy.**

This command builds and packages the application into helloservice.war, located in *tut-install*/examples/jaxws/helloservice/dist/, and deploys this WAR file to the GlassFish Server.

Next Steps

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

#### To Build, Package, and Deploy the Service Using Ant

1. **In a terminal window, go to:**

*tut-install*/examples/jaxws/helloservice/

1. **Type the following command:**

**ant**

This command calls the default target, which builds and packages the application into a WAR file, helloservice.war, located in the dist directory.

1. **Make sure that the GlassFish Server is started.**
2. **Type the following:**

**ant deploy**

Next Steps

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

### Testing the Methods of a Web Service Endpoint

GlassFish Server allows you to test the methods of a web service endpoint.

#### To Test the Service without a Client

To test the sayHello method of HelloService, follow these steps.

1. **Open the web service test interface by typing the following URL in a web browser:**

http://localhost:8080/helloservice/HelloService?Tester

1. **Under Methods, type a name as the parameter to the sayHello method.**
2. **Click the sayHello button.**

This takes you to the sayHello Method invocation page.

Under Method returned, you’ll see the response from the endpoint.

### A Simple JAX-WS Application Client

The HelloAppClient class is a stand-alone application client that accesses the sayHello method of HelloService. This call is made through a port, a local object that acts as a proxy for the remote service. The port is created at development time by the wsimport task, which generates JAX-WS portable artifacts based on a WSDL file.

#### Coding the Application Client

When invoking the remote methods on the port, the client performs these steps:

1. Uses the generated helloservice.endpoint.HelloService class, which represents the service at the URI of the deployed service’s WSDL file:
2. import helloservice.endpoint.HelloService;
3. import javax.xml.ws.WebServiceRef;
4. public class HelloAppClient {
5. @WebServiceRef(wsdlLocation =
6. "META-INF/wsdl/localhost\_8080/helloservice/HelloService.wsdl")

private static HelloService service;

1. Retrieves a proxy to the service, also known as a port, by invoking getHelloPort on the service:

helloservice.endpoint.Hello port = service.getHelloPort();

The port implements the SEI defined by the service.

1. Invokes the port’s sayHello method, passing a string to the service:

return port.sayHello(arg0);

Here is the full source of HelloAppClient, which is located in the following directory:

*tut-install*/examples/jaxws/appclient/src/appclient/

package appclient;

import helloservice.endpoint.HelloService;

import javax.xml.ws.WebServiceRef;

public class HelloAppClient {

@WebServiceRef(wsdlLocation =

"META-INF/wsdl/localhost\_8080/helloservice/HelloService.wsdl")

private static HelloService service;

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

System.out.println(sayHello("world"));

}

private static String sayHello(java.lang.String arg0) {

helloservice.endpoint.Hello port = service.getHelloPort();

return port.sayHello(arg0);

}

}

#### Running the Application Client

You can use either NetBeans IDE or Ant to build, package, deploy, and run the appclient application. To build the client, you must first have deployedhelloservice, as described in [Building, Packaging, and Deploying the Service](https://docs.oracle.com/javaee/6/tutorial/doc/bnayn.html#bnayr).

#### To Run the Application Client Using NetBeans IDE

1. **From the File menu, choose Open Project.**
2. **In the Open Project dialog, navigate to:**

*tut-install*/examples/jaxws/

1. **Select the appclient folder.**
2. **Select the Open as Main Project check box.**
3. **Click Open Project.**
4. **In the Projects tab, right-click the appclient project and select Run.**

You will see the output of the application client in the Output pane.

#### To Run the Application Client Using Ant

1. **In a terminal window, go to:**

*tut-install*/examples/jaxws/appclient/

1. **Type the following command:**

**ant**

This command calls the default target, which runs the wsimport task and builds and packages the application into a JAR file, appclient.jar, located in the dist directory.

1. **Type the following command:**

**ant getclient**

This command deploys the appclient.jar file and retrieves the client stubs.

1. **To run the client, type the following command:**

**ant run**

### A Simple JAX-WS Web Client

HelloServlet is a servlet that, like the Java client, calls the sayHello method of the web service. Like the application client, it makes this call through a port.

#### Coding the Servlet

To invoke the method on the port, the client performs these steps:

1. Imports the HelloService endpoint and the WebServiceRef annotation:
2. import helloservice.endpoint.HelloService;
3. ...

import javax.xml.ws.WebServiceRef;

1. Defines a reference to the web service by specifying the WSDL location:
2. @WebServiceRef(wsdlLocation =

"WEB-INF/wsdl/localhost\_8080/helloservice/HelloService.wsdl")

1. Declares the web service, then defines a private method that calls the sayHello method on the port:
2. private HelloService service;
3. ...
4. private String sayHello(java.lang.String arg0) {
5. helloservice.endpoint.Hello port = service.getHelloPort();
6. return port.sayHello(arg0);

}

1. In the servlet, calls this private method:

out.println("<p>" + sayHello("world") + "</p>");

The significant parts of the HelloServlet code follow. The code is located in the *tut-install*/examples/jaxws/src/java/webclient/ directory.

package webclient;

import helloservice.endpoint.HelloService;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import javax.xml.ws.WebServiceRef;

@WebServlet(name="HelloServlet", urlPatterns={"/HelloServlet"})

public class HelloServlet extends HttpServlet {

@WebServiceRef(wsdlLocation =

"WEB-INF/wsdl/localhost\_8080/helloservice/HelloService.wsdl")

private HelloService service;

/\*\*

\* Processes requests for both HTTP <code>GET</code>

\* and <code>POST</code> methods.

\* @param request servlet request

\* @param response servlet response

\* @throws ServletException if a servlet-specific error occurs

\* @throws IOException if an I/O error occurs

\*/

protected void processRequest(HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

response.setContentType("text/html;charset=UTF-8");

PrintWriter out = response.getWriter();

try {

out.println("<html lang=\"en\">");

out.println("<head>");

out.println("<title>Servlet HelloServlet</title>");

out.println("</head>");

out.println("<body>");

out.println("<h1>Servlet HelloServlet at " +

request.getContextPath () + "</h1>");

out.println("<p>" + sayHello("world") + "</p>");

out.println("</body>");

out.println("</html>");

} finally {

out.close();

}

}

// doGet and doPost methods, which call processRequest, and

// getServletInfo method

private String sayHello(java.lang.String arg0) {

helloservice.endpoint.Hello port = service.getHelloPort();

return port.sayHello(arg0);

}

}

#### Running the Web Client

You can use either NetBeans IDE or Ant to build, package, deploy, and run the webclient application. To build the client, you must first have deployed helloservice, as described in [Building, Packaging, and Deploying the Service](https://docs.oracle.com/javaee/6/tutorial/doc/bnayn.html#bnayr).

#### To Run the Web Client Using NetBeans IDE

1. **From the File menu, choose Open Project.**
2. **In the Open Project dialog, navigate to:**

*tut-install*/examples/jaxws/

1. **Select the webclient folder.**
2. **Select the Open as Main Project check box.**
3. **Click Open Project.**
4. **In the Projects tab, right-click the webclient project and select Deploy.**

This task runs the wsimport tasks, builds and packages the application into a WAR file, webclient.war, located in the dist directory, and deploys it to the server.

1. **In a web browser, navigate to the following URL:**

http://localhost:8080/webclient/HelloServlet

The output of the sayHello method appears in the window.

#### To Run the Web Client Using Ant

1. **In a terminal window, go to:**

*tut-install*/examples/jaxws/webclient/

1. **Type the following command:**

**ant**

This command calls the default target, which runs the wsimport tasks, then builds and packages the application into a WAR file, webclient.war, located in the dist directory.

1. **Type the following command:**

**ant deploy**

This task deploys the WAR file to the server.

1. **In a web browser, navigate to the following URL:**

http://localhost:8080/webclient/HelloServlet

The output of the sayHello method appears in the window.

## Types Supported by JAX-WS

JAX-WS delegates the mapping of Java programming language types to and from XML definitions to JAXB. Application developers don’t need to know the details of these mappings but should be aware that not every class in the Java language can be used as a method parameter or return type in JAX-WS.

The following sections explain the default schema-to-Java and Java-to-schema data type bindings.

### Schema-to-Java Mapping

The Java language provides a richer set of data type than XML schema. [Table 19-1](https://docs.oracle.com/javaee/6/tutorial/doc/bnazc.html#bnazu) lists the mapping of XML data types to Java data types in JAXB.

**Table 19-1 JAXB Mapping of XML Schema Built-in Data Types**

|  |  |
| --- | --- |
| **XML Schema Type** | **Java Data Type** |
| xsd:string | java.lang.String |
| xsd:integer | java.math.BigInteger |
| xsd:int | int |
| xsd.long | long |
| xsd:short | short |
| xsd:decimal | java.math.BigDecimal |
| xsd:float | float |
| xsd:double | double |
| xsd:boolean | boolean |
| xsd:byte | byte |
| xsd:QName | javax.xml.namespace.QName |
| xsd:dateTime | javax.xml.datatype.XMLGregorianCalendar |
| xsd:base64Binary | byte[] |
| xsd:hexBinary | byte[] |
| xsd:unsignedInt | long |
| xsd:unsignedShort | int |
| xsd:unsignedByte | short |
| xsd:time | javax.xml.datatype.XMLGregorianCalendar |
| xsd:date | javax.xml.datatype.XMLGregorianCalendar |
| xsd:g | javax.xml.datatype.XMLGregorianCalendar |
| xsd:anySimpleType | java.lang.Object |
| xsd:anySimpleType | java.lang.String |
| xsd:duration | javax.xml.datatype.Duration |
| xsd:NOTATION | javax.xml.namespace.QName |

### Java-to-Schema Mapping

[Table 19-2](https://docs.oracle.com/javaee/6/tutorial/doc/bnazc.html" \l "bnazx) shows the default mapping of Java classes to XML data types.

**Table 19-2 JAXB Mapping of XML Data Types to Java Classes**

|  |  |
| --- | --- |
| **Java Class** | **XML Data Type** |
| java.lang.String | xs:string |
| java.math.BigInteger | xs:integer |
| java.math.BigDecimal | xs:decimal |
| java.util.Calendar | xs:dateTime |
| java.util.Date | xs:dateTime |
| javax.xml.namespace.QName | xs:QName |
| java.net.URI | xs:string |
| javax.xml.datatype.XMLGregorianCalendar | xs:anySimpleType |
| javax.xml.datatype.Duration | xs:duration |
| java.lang.Object | xs:anyType |
| java.awt.Image | xs:base64Binary |
| javax.activation.DataHandler | xs:base64Binary |
| javax.xml.transform.Source | xs:base64Binary |
| java.util.UUID | xs:string |

## What Are RESTful Web Services?

**RESTful web services** are built to work best on the Web. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability, that enable services to work best on the Web. In the REST architectural style, data and functionality are considered resources and are accessed using **Uniform Resource Identifiers (URIs)**, typically links on the Web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains an architecture to a client/server architecture and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.

The following principles encourage RESTful applications to be simple, lightweight, and fast:

* **Resource identification through URI**: A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery. See [The @Path Annotation and URI Path Templates](https://docs.oracle.com/javaee/6/tutorial/doc/gilik.html#ginpw) for more information.
* **Uniform interface**: Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted by using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource. See [Responding to HTTP Methods and Requests](https://docs.oracle.com/javaee/6/tutorial/doc/gilik.html#gipys) for more information.
* **Self-descriptive messages**: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others. Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control. See [Responding to HTTP Methods and Requests](https://docs.oracle.com/javaee/6/tutorial/doc/gilik.html#gipys) and [Using Entity Providers to Map HTTP Response and Request Entity Bodies](https://docs.oracle.com/javaee/6/tutorial/doc/gilik.html#gipze) for more information.
* **Stateful interactions through hyperlinks**: Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction. See [Using Entity Providers to Map HTTP Response and Request Entity Bodies](https://docs.oracle.com/javaee/6/tutorial/doc/gilik.html#gipze)and “Building URIs” in the JAX-RS Overview document for more information.

## Creating a RESTful Root Resource Class

**Root resource classes** are POJOs that are either annotated with @Path or have at least one method annotated with @Path or a **request method designator**, such as @GET, @PUT, @POST, or @DELETE. **Resource methods** are methods of a resource class annotated with a request method designator. This section explains how to use JAX-RS to annotate Java classes to create RESTful web services.

### Developing RESTful Web Services with JAX-RS

JAX-RS is a Java programming language API designed to make it easy to develop applications that use the REST architecture.

The JAX-RS API uses Java programming language annotations to simplify the development of RESTful web services. Developers decorate Java programming language class files with JAX-RS annotations to define resources and the actions that can be performed on those resources. JAX-RS annotations are runtime annotations; therefore, runtime reflection will generate the helper classes and artifacts for the resource. A Java EE application archive containing JAX-RS resource classes will have the resources configured, the helper classes and artifacts generated, and the resource exposed to clients by deploying the archive to a Java EE server.

[Table 20-1](https://docs.oracle.com/javaee/6/tutorial/doc/gilik.html#ginna) lists some of the Java programming annotations that are defined by JAX-RS, with a brief description of how each is used. Further information on the JAX-RS APIs can be viewed at <http://docs.oracle.com/javaee/6/api/>.

**Table 20-1 Summary of JAX-RS Annotations**

|  |  |
| --- | --- |
| **Annotation** | **Description** |
| @Path | The @Path annotation’s value is a relative URI path indicating where the Java class will be hosted: for example, /helloworld. You can also embed variables in the URIs to make a URI path template. For example, you could ask for the name of a user and pass it to the application as a variable in the URI: /helloworld/{username}. |
| @GET | The @GET annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP GET requests. The behavior of a resource is determined by the HTTP method to which the resource is responding. |
| @POST | The @POST annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP POST requests. The behavior of a resource is determined by the HTTP method to which the resource is responding. |
| @PUT | The @PUT annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP PUT requests. The behavior of a resource is determined by the HTTP method to which the resource is responding. |
| @DELETE | The @DELETE annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP DELETE requests. The behavior of a resource is determined by the HTTP method to which the resource is responding. |
| @HEAD | The @HEAD annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP HEAD requests. The behavior of a resource is determined by the HTTP method to which the resource is responding. |
| @PathParam | The @PathParam annotation is a type of parameter that you can extract for use in your resource class. URI path parameters are extracted from the request URI, and the parameter names correspond to the URI path template variable names specified in the @Path class-level annotation. |
| @QueryParam | The @QueryParam annotation is a type of parameter that you can extract for use in your resource class. Query parameters are extracted from the request URI query parameters. |
| @Consumes | The @Consumes annotation is used to specify the MIME media types of representations a resource can consume that were sent by the client. |
| @Produces | The @Produces annotation is used to specify the MIME media types of representations a resource can produce and send back to the client: for example, "text/plain". |
| @Provider | The @Provider annotation is used for anything that is of interest to the JAX-RS runtime, such as MessageBodyReader andMessageBodyWriter. For HTTP requests, the MessageBodyReader is used to map an HTTP request entity body to method parameters. On the response side, a return value is mapped to an HTTP response entity body by using a MessageBodyWriter. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return aResponse that wraps the entity and that can be built using Response.ResponseBuilder. |

### Overview of a JAX-RS Application

The following code sample is a very simple example of a root resource class that uses JAX-RS annotations:

package com.sun.jersey.samples.helloworld.resources;

import javax.ws.rs.GET;

import javax.ws.rs.Produces;

import javax.ws.rs.Path;

// The Java class will be hosted at the URI path "/helloworld"

@Path("/helloworld")

public class HelloWorldResource {

// The Java method will process HTTP GET requests

@GET

// The Java method will produce content identified by the MIME Media

// type "text/plain"

@Produces("text/plain")

public String getClichedMessage() {

// Return some cliched textual content

return "Hello World";

}

}

The following sections describe the annotations used in this example.

* The @Path annotation’s value is a relative URI path. In the preceding example, the Java class will be hosted at the URI path /helloworld. This is an extremely simple use of the @Path annotation, with a static URI path. Variables can be embedded in the URIs. **URI path templates** are URIs with variables embedded within the URI syntax.
* The @GET annotation is a request method designator, along with @POST, @PUT, @DELETE, and @HEAD, defined by JAX-RS and corresponding to the similarly named HTTP methods. In the example, the annotated Java method will process HTTP GET requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
* The @Produces annotation is used to specify the MIME media types a resource can produce and send back to the client. In this example, the Java method will produce representations identified by the MIME media type "text/plain".
* The @Consumes annotation is used to specify the MIME media types a resource can consume that were sent by the client. The example could be modified to set the message returned by the getClichedMessage method, as shown in this code example:
* @POST
* @Consumes("text/plain")
* public void postClichedMessage(String message) {
* // Store the message

}

### The @Path Annotation and URI Path Templates

The @Path annotation identifies the URI path template to which the resource responds and is specified at the class or method level of a resource. The @Pathannotation’s value is a partial URI path template relative to the base URI of the server on which the resource is deployed, the context root of the application, and the URL pattern to which the JAX-RS runtime responds.

URI path templates are URIs with variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI. Variables are denoted by braces ({ and }). For example, look at the following @Path annotation:

@Path("/users/{username}")

In this kind of example, a user is prompted to type his or her name, and then a JAX-RS web service configured to respond to requests to this URI path template responds. For example, if the user types the user name “Galileo,” the web service responds to the following URL:

http://example.com/users/Galileo

To obtain the value of the user name, the @PathParam annotation may be used on the method parameter of a request method, as shown in the following code example:

@Path("/users/{username}")

public class UserResource {

@GET

@Produces("text/xml")

public String getUser(@PathParam("username") String userName) {

...

}

}

By default, the URI variable must match the regular expression "[^/]+?". This variable may be customized by specifying a different regular expression after the variable name. For example, if a user name must consist only of lowercase and uppercase alphanumeric characters, override the default regular expression in the variable definition:

@Path("users/{username: [a-zA-Z][a-zA-Z\_0-9]\*}")

In this example the username variable will match only user names that begin with one uppercase or lowercase letter and zero or more alphanumeric characters and the underscore character. If a user name does not match that template, a 404 (Not Found) response will be sent to the client.

A @Path value isn’t required to have leading or trailing slashes (/). The JAX-RS runtime parses URI path templates the same whether or not they have leading or trailing spaces.

A URI path template has one or more variables, with each variable name surrounded by braces: { to begin the variable name and } to end it. In the preceding example, username is the variable name. At runtime, a resource configured to respond to the preceding URI path template will attempt to process the URI data that corresponds to the location of {username} in the URI as the variable data for username.

For example, if you want to deploy a resource that responds to the URI path template http://example.com/myContextRoot/resources/{name1}/{name2}/, you must deploy the application to a Java EE server that responds to requests to the http://example.com/myContextRoot URI and then decorate your resource with the following @Path annotation:

@Path("/{name1}/{name2}/")

public class SomeResource {

...

}

In this example, the URL pattern for the JAX-RS helper servlet, specified in web.xml, is the default:

<servlet-mapping>

<servlet-name>My JAX-RS Resource</servlet-name>

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

</servlet-mapping>

A variable name can be used more than once in the URI path template.

If a character in the value of a variable would conflict with the reserved characters of a URI, the conflicting character should be substituted with percent encoding. For example, spaces in the value of a variable should be substituted with %20.

When defining URI path templates, be careful that the resulting URI after substitution is valid.

[Table 20-2](https://docs.oracle.com/javaee/6/tutorial/doc/gilik.html#gipym) lists some examples of URI path template variables and how the URIs are resolved after substitution. The following variable names and values are used in the examples:

* name1: james
* name2: gatz
* name3:
* location: Main%20Street
* question: why

**Note -**The value of the name3 variable is an empty string.

**Table 20-2 Examples of URI Path Templates**

|  |  |
| --- | --- |
| **URI Path Template** | **URI After Substitution** |
| http://example.com/{name1}/{name2}/ | http://example.com/james/gatz/ |
| http://example.com/{question}/{question}/{question}/ | http://example.com/why/why/why/ |
| http://example.com/maps/{location} | http://example.com/maps/Main%20Street |
| http://example.com/{name3}/home/ | http://example.com//home/ |

### Responding to HTTP Methods and Requests

The behavior of a resource is determined by the HTTP methods (typically, GET, POST, PUT, DELETE) to which the resource is responding.

#### The Request Method Designator Annotations

Request method designator annotations are runtime annotations, defined by JAX-RS, that correspond to the similarly named HTTP methods. Within a resource class file, HTTP methods are mapped to Java programming language methods by using the request method designator annotations. The behavior of a resource is determined by which HTTP method the resource is responding to. JAX-RS defines a set of request method designators for the common HTTP methods @GET, @POST, @PUT, @DELETE, and @HEAD; you can also create your own custom request method designators. Creating custom request method designators is outside the scope of this document.

The following example, an extract from the storage service sample, shows the use of the PUT method to create or update a storage container:

@PUT

public Response putContainer() {

System.out.println("PUT CONTAINER " + container);

URI uri = uriInfo.getAbsolutePath();

Container c = new Container(container, uri.toString());

Response r;

if (!MemoryStore.MS.hasContainer(c)) {

r = Response.created(uri).build();

} else {

r = Response.noContent().build();

}

MemoryStore.MS.createContainer(c);

return r;

}

By default, the JAX-RS runtime will automatically support the methods HEAD and OPTIONS if not explicitly implemented. For HEAD, the runtime will invoke the implemented GET method, if present, and ignore the response entity, if set. For OPTIONS, the Allow response header will be set to the set of HTTP methods supported by the resource. In addition, the JAX-RS runtime will return a Web Application Definition Language (WADL) document describing the resource; see <http://www.w3.org/Submission/wadl/> for more information.

Methods decorated with request method designators must return void, a Java programming language type, or a javax.ws.rs.core.Response object. Multiple parameters may be extracted from the URI by using the @PathParam or @QueryParam annotations as described in [Extracting Request Parameters](https://docs.oracle.com/javaee/6/tutorial/doc/gilik.html#gipyw). Conversion between Java types and an entity body is the responsibility of an entity provider, such as MessageBodyReader or MessageBodyWriter. Methods that need to provide additional metadata with a response should return an instance of the Response class. The ResponseBuilder class provides a convenient way to create a Response instance using a builder pattern. The HTTP PUT and POST methods expect an HTTP request body, so you should use a MessageBodyReader for methods that respond to PUT and POST requests.

Both @PUT and @POST can be used to create or update a resource. POST can mean anything, so when using POST, it is up to the application to define the semantics. PUT has well-defined semantics. When using PUT for creation, the client declares the URI for the newly created resource.

PUT has very clear semantics for creating and updating a resource. The representation the client sends must be the same representation that is received using a GET, given the same media type. PUT does not allow a resource to be partially updated, a common mistake when attempting to use the PUT method. A common application pattern is to use POST to create a resource and return a 201 response with a location header whose value is the URI to the newly created resource. In this pattern, the web service declares the URI for the newly created resource.

#### Using Entity Providers to Map HTTP Response and Request Entity Bodies

**Entity providers** supply mapping services between representations and their associated Java types. The two types of entity providers are MessageBodyReader and MessageBodyWriter. For HTTP requests, the MessageBodyReader is used to map an HTTP request entity body to method parameters. On the response side, a return value is mapped to an HTTP response entity body by using a MessageBodyWriter. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a Response that wraps the entity and that can be built by using Response.ResponseBuilder.

[Table 20-3](https://docs.oracle.com/javaee/6/tutorial/doc/gilik.html#gkccg) shows the standard types that are supported automatically for HTTP request and response entity bodies. You need to write an entity provider only if you are not choosing one of these standard types.

**Table 20-3 Types Supported for HTTP Request and Response Entity Bodies**

|  |  |
| --- | --- |
| **Java Type** | **Supported Media Types** |
| byte[] | All media types (\*/\*) |
| java.lang.String | All text media types (text/\*) |
| java.io.InputStream | All media types (\*/\*) |
| java.io.Reader | All media types (\*/\*) |
| java.io.File | All media types (\*/\*) |
| javax.activation.DataSource | All media types (\*/\*) |
| javax.xml.transform.Source | XML media types (text/xml, application/xml, and application/\*+xml) |
| javax.xml.bind.JAXBElement and application-supplied JAXB classes | XML media types (text/xml, application/xml, and application/\*+xml) |
| MultivaluedMap<String, String> | Form content (application/x-www-form-urlencoded) |
| StreamingOutput | All media types (\*/\*), MessageBodyWriter only |

The following example shows how to use MessageBodyReader with the @Consumes and @Provider annotations:

@Consumes("application/x-www-form-urlencoded")

@Provider

public class FormReader implements MessageBodyReader<NameValuePair> {

The following example shows how to use MessageBodyWriter with the @Produces and @Provider annotations:

@Produces("text/html")

@Provider

public class FormWriter implements

MessageBodyWriter<Hashtable<String, String>> {

The following example shows how to use ResponseBuilder:

@GET

public Response getItem() {

System.out.println("GET ITEM " + container + " " + item);

Item i = MemoryStore.MS.getItem(container, item);

if (i == null)

throw new NotFoundException("Item not found");

Date lastModified = i.getLastModified().getTime();

EntityTag et = new EntityTag(i.getDigest());

ResponseBuilder rb = request.evaluatePreconditions(lastModified, et);

if (rb != null)

return rb.build();

byte[] b = MemoryStore.MS.getItemData(container, item);

return Response.ok(b, i.getMimeType()).

lastModified(lastModified).tag(et).build();

}

### Using @Consumes and @Produces to Customize Requests and Responses

The information sent to a resource and then passed back to the client is specified as a MIME media type in the headers of an HTTP request or response. You can specify which MIME media types of representations a resource can respond to or produce by using the following annotations:

* javax.ws.rs.Consumes
* javax.ws.rs.Produces

By default, a resource class can respond to and produce all MIME media types of representations specified in the HTTP request and response headers.

#### The @Produces Annotation

The @Produces annotation is used to specify the MIME media types or representations a resource can produce and send back to the client. If @Produces is applied at the class level, all the methods in a resource can produce the specified MIME types by default. If applied at the method level, the annotation overrides any @Produces annotations applied at the class level.

If no methods in a resource are able to produce the MIME type in a client request, the JAX-RS runtime sends back an HTTP “406 Not Acceptable” error.

The value of @Produces is an array of String of MIME types. For example:

@Produces({"image/jpeg,image/png"})

The following example shows how to apply @Produces at both the class and method levels:

@Path("/myResource")

@Produces("text/plain")

public class SomeResource {

@GET

public String doGetAsPlainText() {

...

}

@GET

@Produces("text/html")

public String doGetAsHtml() {

...

}

}

The doGetAsPlainText method defaults to the MIME media type of the @Produces annotation at the class level. The doGetAsHtml method’s @Producesannotation overrides the class-level @Produces setting and specifies that the method can produce HTML rather than plain text.

If a resource class can produce more than one MIME media type, the resource method chosen will correspond to the most acceptable media type as declared by the client. More specifically, the Accept header of the HTTP request declares what is most acceptable. For example, if the Accept header is Accept: text/plain, the doGetAsPlainText method will be invoked. Alternatively, if the Accept header is Accept: text/plain;q=0.9, text/html, which declares that the client can accept media types of text/plain and text/html but prefers the latter, the doGetAsHtml method will be invoked.

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"})

public String doGetAsXmlOrJson() {

...

}

The doGetAsXmlOrJson method will get invoked if either of the media types application/xml and application/json is acceptable. If both are equally acceptable, the former will be chosen because it occurs first. The preceding examples refer explicitly to MIME media types for clarity. It is possible to refer to constant values, which may reduce typographical errors. For more information, see the constant field values of MediaType at <http://jsr311.java.net/nonav/releases/1.0/javax/ws/rs/core/MediaType.html>.

#### The @Consumes Annotation

The @Consumes annotation is used to specify which MIME media types of representations a resource can accept, or consume, from the client. If @Consumes is applied at the class level, all the response methods accept the specified MIME types by default. If applied at the method level, @Consumes overrides any @Consumesannotations applied at the class level.

If a resource is unable to consume the MIME type of a client request, the JAX-RS runtime sends back an HTTP 415 (“Unsupported Media Type”) error.

The value of @Consumes is an array of String of acceptable MIME types. For example:

@Consumes({"text/plain,text/html"})

The following example shows how to apply @Consumes at both the class and method levels:

@Path("/myResource")

@Consumes("multipart/related")

public class SomeResource {

@POST

public String doPost(MimeMultipart mimeMultipartData) {

...

}

@POST

@Consumes("application/x-www-form-urlencoded")

public String doPost2(FormURLEncodedProperties formData) {

...

}

}

The doPost method defaults to the MIME media type of the @Consumes annotation at the class level. The doPost2 method overrides the class level @Consumesannotation to specify that it can accept URL-encoded form data.

If no resource methods can respond to the requested MIME type, an HTTP 415 (“Unsupported Media Type”) error is returned to the client.

The HelloWorld example discussed previously in this section can be modified to set the message by using @Consumes, as shown in the following code example:

@POST

@Consumes("text/plain")

public void postClichedMessage(String message) {

// Store the message

}

In this example, the Java method will consume representations identified by the MIME media type text/plain. Note that the resource method returns void. This means that no representation is returned and that a response with a status code of HTTP 204 (“No Content”) will be returned.

### Extracting Request Parameters

Parameters of a resource method may be annotated with parameter-based annotations to extract information from a request. A previous example presented the use of the @PathParam parameter to extract a path parameter from the path component of the request URL that matched the path declared in @Path.

You can extract the following types of parameters for use in your resource class:

* Query
* URI path
* Form
* Cookie
* Header
* Matrix

**Query parameters** are extracted from the request URI query parameters and are specified by using the javax.ws.rs.QueryParam annotation in the method parameter arguments. The following example, from the sparklines sample application, demonstrates using @QueryParam to extract query parameters from the Query component of the request URL:

@Path("smooth")

@GET

public Response smooth(

@DefaultValue("2") @QueryParam("step") int step,

@DefaultValue("true") @QueryParam("min-m") boolean hasMin,

@DefaultValue("true") @QueryParam("max-m") boolean hasMax,

@DefaultValue("true") @QueryParam("last-m") boolean hasLast,

@DefaultValue("blue") @QueryParam("min-color") ColorParam minColor,

@DefaultValue("green") @QueryParam("max-color") ColorParam maxColor,

@DefaultValue("red") @QueryParam("last-color") ColorParam lastColor

) { ... }

If the query parameter step exists in the query component of the request URI, the value of step will be extracted and parsed as a 32-bit signed integer and assigned to the step method parameter. If step does not exist, a default value of 2, as declared in the @DefaultValue annotation, will be assigned to the step method parameter. If the step value cannot be parsed as a 32-bit signed integer, an HTTP 400 (“Client Error”) response is returned.

User-defined Java programming language types may be used as query parameters. The following code example shows the ColorParam class used in the preceding query parameter example:

public class ColorParam extends Color {

public ColorParam(String s) {

super(getRGB(s));

}

private static int getRGB(String s) {

if (s.charAt(0) == '#') {

try {

Color c = Color.decode("0x" + s.substring(1));

return c.getRGB();

} catch (NumberFormatException e) {

throw new WebApplicationException(400);

}

} else {

try {

Field f = Color.class.getField(s);

return ((Color)f.get(null)).getRGB();

} catch (Exception e) {

throw new WebApplicationException(400);

}

}

}

}

The constructor for ColorParam takes a single String parameter.

Both @QueryParam and @PathParam can be used only on the following Java types:

* All primitive types except char
* All wrapper classes of primitive types except Character
* Any class with a constructor that accepts a single String argument
* Any class with the static method named valueOf(String) that accepts a single String argument
* List<T>, Set<T>, or SortedSet<T>, where *T* matches the already listed criteria. Sometimes, parameters may contain more than one value for the same name. If this is the case, these types may be used to obtain all values

If @DefaultValue is not used in conjunction with @QueryParam, and the query parameter is not present in the request, the value will be an empty collection for List, Set, or SortedSet; null for other object types; and the default for primitive types.

**URI path parameters** are extracted from the request URI, and the parameter names correspond to the URI path template variable names specified in the @Pathclass-level annotation. URI parameters are specified using the javax.ws.rs.PathParam annotation in the method parameter arguments. The following example shows how to use @Path variables and the @PathParam annotation in a method:

@Path("/{username}")

public class MyResourceBean {

...

@GET

public String printUsername(@PathParam("username") String userId) {

...

}

}

In the preceding snippet, the URI path template variable name username is specified as a parameter to the printUsername method. The @PathParam annotation is set to the variable name username. At runtime, before printUsername is called, the value of username is extracted from the URI and cast to a String. The resulting String is then available to the method as the userId variable.

If the URI path template variable cannot be cast to the specified type, the JAX-RS runtime returns an HTTP 400 (“Bad Request”) error to the client. If the @PathParamannotation cannot be cast to the specified type, the JAX-RS runtime returns an HTTP 404 (“Not Found”) error to the client.

The @PathParam parameter and the other parameter-based annotations (@MatrixParam, @HeaderParam, @CookieParam, and @FormParam) obey the same rules as @QueryParam.

**Cookie parameters**, indicated by decorating the parameter with javax.ws.rs.CookieParam, extract information from the cookies declared in cookie-related HTTP headers. **Header parameters**, indicated by decorating the parameter with javax.ws.rs.HeaderParam, extract information from the HTTP headers. **Matrix parameters**, indicated by decorating the parameter with javax.ws.rs.MatrixParam, extract information from URL path segments.

**Form parameters**, indicated by decorating the parameter with javax.ws.rs.FormParam, extract information from a request representation that is of the MIME media type application/x-www-form-urlencoded and conforms to the encoding specified by HTML forms, as described in <http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.1>. This parameter is very useful for extracting information sent by POST in HTML forms.

The following example extracts the name form parameter from the POST form data:

@POST

@Consumes("application/x-www-form-urlencoded")

public void post(@FormParam("name") String name) {

// Store the message

}

To obtain a general map of parameter names and values for query and path parameters, use the following code:

@GET

public String get(@Context UriInfo ui) {

MultivaluedMap<String, String> queryParams = ui.getQueryParameters();

MultivaluedMap<String, String> pathParams = ui.getPathParameters();

}

The following method extracts header and cookie parameter names and values into a map:

@GET

public String get(@Context HttpHeaders hh) {

MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();

Map<String, Cookie> pathParams = hh.getCookies();

}

In general, @Context can be used to obtain contextual Java types related to the request or response.

For form parameters, it is possible to do the following:

@POST

@Consumes("application/x-www-form-urlencoded")

public void post(MultivaluedMap<String, String> formParams) {

// Store the message

}

## Example Applications for JAX-RS

This section provides an introduction to creating, deploying, and running your own JAX-RS applications. This section demonstrates the steps that are needed to create, build, deploy, and test a very simple web application that uses JAX-RS annotations.

### A RESTful Web Service

This section explains how to use NetBeans IDE to create a RESTful web service. NetBeans IDE generates a skeleton for the application, and you simply need to implement the appropriate methods. If you do not use an IDE, try using one of the example applications that ship with Jersey as a template to modify.

You can find a version of this application at *tut-install*/examples/jaxrs/HelloWorldApplication/.

#### To Create a RESTful Web Service Using NetBeans IDE

1. **In NetBeans IDE, create a simple web application. This example creates a very simple “Hello, World” web application.**
   1. **From the File menu, choose New Project.**
   2. **From Categories, select Java Web. From Projects, select Web Application. Click Next.**

**Note -**For this step, you could also create a RESTful web service in a Maven web project by selecting Maven as the category and Maven Web Project as the project. The remaining steps would be the same.

* 1. **Type a project name, HelloWorldApplication, and click Next.**
  2. **Make sure that the Server is GlassFish Server (or similar wording).**
  3. **Click Finish.**

The project is created. The file index.jsp appears in the Source pane.

1. **Right-click the project and select New; then select RESTful Web Services from Patterns.**
   1. **Select Simple Root Resource and click Next.**
   2. **Type a Resource Package name, such as helloWorld.**
   3. **Type helloworld in the Path field. Type HelloWorld in the Class Name field. For MIME Type, select text/html.**
   4. **Click Finish.**

The REST Resources Configuration page appears.

* 1. **Click OK.**

A new resource, HelloWorld.java, is added to the project and appears in the Source pane. This file provides a template for creating a RESTful web service.

1. **In HelloWorld.java, find the getHtml() method. Replace the //TODO comment and the exception with the following text, so that the finished product resembles the following method.**

**Note -**Because the MIME type produced is HTML, you can use HTML tags in your return statement.

/\*\*

\* Retrieves representation of an instance of helloWorld.HelloWorld

\* @return an instance of java.lang.String

\*/

@GET

@Produces("text/html")

public String getHtml() {

return "<html lang=\"en\"><body><h1>Hello, World!!</body></h1></html>";

}

1. **Test the web service. To do this, right-click the project node and click Test RESTful Web Services.**

This step deploys the application and brings up a test client in the browser.

1. **When the test client appears, select the helloworld resource in the left pane, and click the Test button in the right pane.**

The words Hello, World!! appear in the Response window below.

1. **Set the Run Properties:**
   1. **Right-click the project node and select Properties.**
   2. **In the dialog, select the Run category.**
   3. **Set the Relative URL to the location of the RESTful web service relative to the Context Path, which for this example is resources/helloworld.**

**Tip -**You can find the value for the Relative URL in the Test RESTful Web Services browser window. In the top of the right pane, after Resource, is the URL for the RESTful web service being tested. The part following the Context Path (http://localhost:8080/HelloWorldApp) is the Relative URL that needs to be entered here.

If you don’t set this property, the file index.jsp will appear by default when the application is run. As this file also contains Hello World as its default value, you might not notice that your RESTful web service isn’t running, so just be aware of this default and the need to set this property, or update index.jsp to provide a link to the RESTful web service.

1. **Right-click the project and select Deploy.**
2. **Right-click the project and select Run.**

A browser window opens and displays the return value of Hello, World!!

See Also

For other sample applications that demonstrate deploying and running JAX-RS applications using NetBeans IDE, see [The rsvp Example Application](https://docs.oracle.com/javaee/6/tutorial/doc/gipzz.html#gjvbc) and *Your First Cup: An Introduction to the Java EE Platform* at <http://docs.oracle.com/javaee/6/firstcup/doc/>. You may also look at the tutorials on the NetBeans IDE tutorial site, such as the one titled “Getting Started with RESTful Web Services” at <http://www.netbeans.org/kb/docs/websvc/rest.html>. This tutorial includes a section on creating a CRUD application from a database. Create, read, update, and delete (CRUD) are the four basic functions of persistent storage and relational databases.

### The rsvp Example Application

The rsvp example application, located in the *tut-install*/examples/jaxrs/rsvp/ directory, allows invitees to an event to indicate whether they will attend. The events, people invited to the event, and the responses to the invite are stored in a Java DB database using the Java Persistence API. The JAX-RS resources in rsvpare exposed in a stateless session enterprise bean.

#### Components of the rsvp Example Application

The three enterprise beans in the rsvp example application are rsvp.ejb.ConfigBean, rsvp.ejb.StatusBean, and rsvp.ejb.ResponseBean.

ConfigBean is a singleton session bean that initializes the data in the database.

StatusBean exposes a JAX-RS resource for displaying the current status of all invitees to an event. The URI path template is declared as follows:

@Path("/status/{eventId}/")

The URI path variable eventId is a @PathParam variable in the getResponse method, which responds to HTTP GET requests and has been annotated with @GET. The eventId variable is used to look up all the current responses in the database for that particular event.

ResponseBean exposes a JAX-RS resource for setting an invitee’s response to a particular event. The URI path template for ResponseBean is declared as follows:

@Path("/{eventId}/{inviteId}")

Two URI path variables are declared in the path template: eventId and inviteId. As in StatusBean, eventId is the unique ID for a particular event. Each invitee to that event has a unique ID for the invitation, and that is the inviteId. Both of these path variables are used in two JAX-RS methods in ResponseBean: getResponse and putResponse. The getResponse method responds to HTTP GET requests and displays the invitee’s current response and a form to change the response.

An invitee who wants to change his or her response selects the new response and submits the form data, which is processed as an HTTP PUT request by the putResponse method. One of the parameters to the putResponse method, the userResponse string, is annotated with @FormParam("attendeeResponse"). The HTML form created by getResponse stores the changed response in the select list with an ID of attendeeResponse. The annotation@FormParam("attendeeResponse") indicates that the value of the select response is extracted from the HTTP PUT request and stored as the userResponsestring. The putResponse method uses userResponse, eventId, and inviteId to update the invitee’s response in the database.

The events, people, and responses in rsvp are encapsulated in Java Persistence API entities. The rsvp.entity.Event, rsvp.entity.Person, and rsvp.entity.Response entities respectively represent events, invitees, and responses to an event.

The rsvp.util.ResponseEnum class declares an enumerated type that represents all the possible response statuses an invitee may have.

#### Running the rsvp Example Application

Both NetBeans IDE and Ant can be used to deploy and run the rsvp example application.

#### To Run the rsvp Example Application in NetBeans IDE

1. **From the File menu, choose Open Project.**
2. **In the Open Project dialog, navigate to:**

*tut-install*/examples/jaxrs/

1. **Select the rsvp folder.**
2. **Select the Open as Main Project check box.**
3. **Click Open Project.**
4. **Right-click the rsvp project in the left pane and select Run.**

The project will be compiled, assembled, and deployed to GlassFish Server. A web browser window will open to http://localhost:8080/rsvp.

1. **In the web browser window, click the Event Status link for the Duke’s Birthday event.**

You’ll see the current invitees and their responses.

1. **Click on the name of one of the invitees, select a response, and click Submit response; then click Back to event page.**

The invitee’s new status should now be displayed in the table of invitees and their response statuses.

#### To Run the rsvp Example Application Using Ant

**Before You Begin**

You must have started the Java DB database before running rsvp.

1. **In a terminal window, go to:**

*tut-install*/examples/jaxrs/rsvp/

1. **Type the following command:**

**ant all**

This command builds, assembles, and deploys rsvp to GlassFish Server.

1. **Open a web browser window to http://localhost:8080/rsvp.**
2. **In the web browser window, click the Event Status link for the Duke’s Birthday event.**

You’ll see the current invitees and their responses.

1. **Click on the name of one of the invitees, select a response, and click Submit response, then click Back to event page.**

The invitee’s new status should now be displayed in the table of invitees and their response statuses.

### Real-World Examples

Most blog sites use RESTful web services. These sites involve downloading XML files, in RSS or Atom format, that contain lists of links to other resources. Other web sites and web applications that use REST-like developer interfaces to data include Twitter and Amazon S3 (Simple Storage Service). With Amazon S3, buckets and objects can be created, listed, and retrieved using either a REST-style HTTP interface or a SOAP interface. The examples that ship with Jersey include a storage service example with a RESTful interface. The tutorial at <http://netbeans.org/kb/docs/websvc/twitter-swing.html> uses NetBeans IDE to create a simple, graphical, REST-based client that displays Twitter public timeline messages and lets you view and update your Twitter status.

## Annotations for Field and Bean Properties of Resource Classes

JAX-RS annotations for resource classes let you extract specific parts or values from a Uniform Resource Identifier (URI) or request header.

JAX-RS provides the annotations listed in [Table 21-1](https://docs.oracle.com/javaee/6/tutorial/doc/gkkrb.html#gkobo).

**Table 21-1 Advanced JAX-RS Annotations**

|  |  |
| --- | --- |
| **Annotation** | **Description** |
| @Context | Injects information into a class field, bean property, or method parameter |
| @CookieParam | Extracts information from cookies declared in the cookie request header |
| @FormParam | Extracts information from a request representation whose content type is application/x-www-form-urlencoded |
| @HeaderParam | Extracts the value of a header |
| @MatrixParam | Extracts the value of a URI matrix parameter |
| @PathParam | Extracts the value of a URI template parameter |
| @QueryParam | Extracts the value of a URI query parameter |

### Extracting Path Parameters

URI path templates are URIs with variables embedded within the URI syntax. The @PathParam annotation lets you use variable URI path fragments when you call a method.

The following code snippet shows how to extract the last name of an employee when the employee’s email address is provided:

@Path(/employees/"{firstname}.{lastname}@{domain}.com")

public class EmpResource {

@GET

@Produces("text/xml")

public String getEmployeelastname(@PathParam("lastname") String lastName) {

...

}

}

In this example, the @Path annotation defines the URI variables (or path parameters) {firstname} , {lastname}, and {domain}. The @PathParam in the method parameter of the request method extracts the last name from the email address.

If your HTTP request is GET /employees/john.doe@example.com, the value “doe” is injected into {lastname}.

You can specify several path parameters in one URI.

You can declare a regular expression with a URI variable. For example, if it is required that the last name must consist only of lower and upper case characters, you can declare the following regular expression:

@Path(/employees/{"firstname}.{lastname[a-zA-Z]\*}@{domain}.com")

If the last name does not match the regular expression, a 404 response is returned.

### Extracting Query Parameters

Use the @QueryParam annotation to extract query parameters from the query component of the request URI.

For instance, to query all employees who have joined within a specific range of years, use a method signature like the following:

@Path(/employees/")

@GET

public Response getEmployees(

@DefaultValue("2002") @QueryParam("minyear") int minyear,

@DefaultValue("2010") @QueryParam("maxyear") int maxyear)

{...}

This code snippet defines two query parameters, minyear and maxyear. The following HTTP request would query for all employees who have joined between 1999 and 2009:

GET /employees?maxyear=2009&minyear=1999

The @DefaultValue annotation defines a default value, which is to be used if no values are provided for the query parameters. By default, JAX-RS assigns a null value for Object values and zero for primitive data types. You can use the @DefaultValue annotation to eliminate null or zero values and define your own default values for a parameter.

### Extracting Form Data

Use the @FormParam annotation to extract form parameters from HTML forms. For example, the following form accepts the name, address, and manager’s name of an employee:

<FORM action="http://example.com/employees/" method="post">

<p>

<fieldset>

Employee name: <INPUT type="text" name="empname" tabindex="1">

Employee address: <INPUT type="text" name="empaddress" tabindex="2">

Manager name: <INPUT type="text" name="managername" tabindex="3">

</fieldset>

</p>

</FORM>

Use the following code snippet to extract the manager name from this HTML form:

@POST

@Consumes("application/x-www-form-urlencoded")

public void post(@FormParam("managername") String managername) {

// Store the value

...

}

To obtain a map of form parameter names to values, use a code snippet like the following:

@POST

@Consumes("application/x-www-form-urlencoded")

public void post(MultivaluedMap<String. String> formParams) {

// Store the message

}

### Extracting the Java Type of a Request or Response

The javax.ws.rs.core.Context annotation retrieves the Java types related to a request or response.

The javax.ws.rs.core.UriInfo interface provides information about the components of a request URI. The following code snippet shows how to obtain a map of query and path parameter names to values:

@GET

public String getParams(@Context UriInfo ui) {

MultivaluedMap<String, String> queryParams = ui.getQueryParameters();

MultivaluedMap<String, String> pathParams = ui.getPathParameters();

}

The javax.ws.rs.core.HttpHeaders interface provides information about request headers and cookies. The following code snippet shows how to obtain a map of header and cookie parameter names to values:

@GET

public String getHeaders(@Context HttpHeaders hh) {

MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();

MultivaluedMap<String, Cookie> pathParams = hh.getCookies();

}

## Subresources and Runtime Resource Resolution

You can use a resource class to process only a part of the URI request. A root resource can then implement subresources that can process the remainder of the URI path.

A resource class method that is annotated with @Path is either a subresource method or a subresource locator:

* A subresource method is used to handle requests on a subresource of the corresponding resource.
* A subresource locator is used to locate subresources of the corresponding resource.

### Subresource Methods

A **subresource method** handles an HTTP request directly. The method must be annotated with a request method designator such as @GET or @POST, in addition to @Path. The method is invoked for request URIs that match a URI template created by concatenating the URI template of the resource class with the URI template of the method.

The following code snippet shows how a subresource method can be used to extract the last name of an employee when the employee’s email address is provided:

@Path("/employeeinfo")

Public class EmployeeInfo {

public employeeinfo() {}

@GET

@Path("/employees/{firstname}.{lastname}@{domain}.com")

@Produces("text/xml")

public String getEmployeeLastName(@PathParam("lastname") String lastName) {

...

}

}

The getEmployeeLastName method returns doe for the following GET request:

GET /employeeinfo/employees/john.doe@example.com

### Subresource Locators

A **subresource locator** returns an object that will handle an HTTP request. The method must not be annotated with a request method designator. You must declare a subresource locator within a subresource class, and only subresource locators are used for runtime resource resolution.

The following code snippet shows a subresource locator:

// Root resource class

@Path("/employeeinfo")

public class EmployeeInfo {

// Subresource locator: obtains the subresource Employee

// from the path /employeeinfo/employees/{empid}

@Path("/employees/{empid}")

public Employee getEmployee(@PathParam("empid") String id) {

// Find the Employee based on the id path parameter

Employee emp = ...;

...

return emp;

}

}

// Subresource class

public class Employee {

// Subresource method: returns the employee's last name

@GET

@Path("/lastname")

public String getEmployeeLastName() {

...

return lastName

}

}

In this code snippet, the getEmployee method is the subresource locator that provides the Employee object, which services requests for lastname.

If your HTTP request is GET /employeeinfo/employees/as209/, the getEmployee method returns an Employee object whose id is as209. At runtime, JAX-RS sends a GET /employeeinfo/employees/as209/lastname request to the getEmployeeLastName method . The getEmployeeLastName method retrieves and returns the last name of the employee whose id is as209.

## Integrating JAX-RS with EJB Technology and CDI

JAX-RS works with Enterprise JavaBeans technology (enterprise beans) and Contexts and Dependency Injection for the Java EE Platform (CDI).

In general, for JAX-RS to work with enterprise beans, you need to annotate the class of a bean with @Path to convert it to a root resource class. You can use the @Path annotation with stateless session beans and singleton POJO beans.

The following code snippet shows a stateless session bean and a singleton bean that have been converted to JAX-RS root resource classes.

@Stateless

@Path("stateless-bean")

public class StatelessResource {...}

@Singleton

@Path("singleton-bean")

public class SingletonResource {...}

Session beans can also be used for subresources.

JAX-RS and CDI have slightly different component models. By default, JAX-RS root resource classes are managed in the request scope, and no annotations are required for specifying the scope. CDI managed beans annotated with @RequestScoped or @ApplicationScoped can be converted to JAX-RS resource classes.

The following code snippet shows a JAX-RS resource class.

@Path("/employee/{id}")

public class Employee {

public Employee(@PathParam("id") String id) {...}

}

@Path("{lastname}")

public final class EmpDetails {...}

The following code snippet shows this JAX-RS resource class converted to a CDI bean. The beans must be proxyable, so the Employee class requires a non-private constructor with no parameters, and the EmpDetails class must not be final.

@Path("/employee/{id}")

@RequestScoped

public class Employee {

public Employee() {...}

@Inject

public Employee(@PathParam("id") String id) {...}

}

@Path("{lastname}")

@RequestScoped

public class EmpDetails {...}

## Conditional HTTP Requests

JAX-RS provides support for conditional GET and PUT HTTP requests. Conditional GET requests help save bandwidth by improving the efficiency of client processing.

A GET request can return a Not Modified (304) response if the representation has not changed since the previous request. For example, a web site can return 304 responses for all its static images that have not changed since the previous request.

A PUT request can return a Precondition Failed (412) response if the representation has been modified since the last request. The conditional PUT can help avoid the lost update problem.

Conditional HTTP requests can be used with the Last-Modified and ETag headers. The Last-Modified header can represent dates with granularity of one second.

@Path("/employee/{joiningdate}")

public class Employee {

Date joiningdate;

@GET

@Produces("application/xml")

public Employee(@PathParam("joiningdate") Date joiningdate,

@Context Request req,

@Context UriInfo ui) {

this.joiningdate = joiningdate;

...

this.tag = computeEntityTag(ui.getRequestUri());

if (req.getMethod().equals("GET")) {

Response.ResponseBuilder rb = req.evaluatePreconditions(tag);

if (rb != null) {

throw new WebApplicationException(rb.build());

}

}

}

}

In this code snippet, the constructor of the Employee class computes the entity tag from the request URI and calls the request.evaluatePreconditions method with that tag. If a client request returns an If-none-match header with a value that has the same entity tag that was computed, evaluate.Preconditions returns a pre-filled-out response with a 304 status code and an entity tag set that may be built and returned.

## Runtime Content Negotiation

The @Produces and @Consumes annotations handle static content negotiation in JAX-RS. These annotations specify the content preferences of the server. HTTP headers such as Accept, Content-Type, and Accept-Language define the content negotiation preferences of the client.

For more details on the HTTP headers for content negotiation, see [HTTP /1.1 - Content Negotiation](http://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html).

The following code snippet shows the server content preferences:

@Produces("text/plain")

@Path("/employee")

public class Employee {

@GET

public String getEmployeeAddressText(String address) { ... }

@Produces("text/xml")

@GET

public String getEmployeeAddressXml(Address address) { ... }

}

The getEmployeeAddressText method is called for an HTTP request that looks as follows:

GET /employee

Accept: text/plain

This will produce the following response:

500 Oracle Parkway, Redwood Shores, CA

The getEmployeeAddressXml method is called for an HTTP request that looks as follows:

GET /employee

Accept: text/xml

This will produce the following response:

<address street="500 Oracle Parkway, Redwood Shores, CA" country="USA"/>

With static content negotiation, you can also define multiple content and media types for the client and server.

@Produces("text/plain", "text/xml")

In addition to supporting static content negotiation, JAX-RS also supports runtime content negotiation using the javax.ws.rs.core.Variant class and Requestobjects. The Variant class specifies the resource representation of content negotiation. Each instance of the Variant class may contain a media type, a language, and an encoding. The Variant object defines the resource representation that is supported by the server. The Variant.VariantListBuilder class is used to build a list of representation variants.

The following code snippet shows how to create a list of resource representation variants:

List<Variant> vs =

Variant.mediatypes("application/xml", "application/json")

.languages("en", "fr").build();

This code snippet calls the build method of the VariantListBuilder class. The VariantListBuilder class is invoked when you call the mediatypes, languages, or encodings methods. The build method builds a series of resource representations. The Variant list created by the build method has all possible combinations of items specified in the mediatypes, languages, and encodings methods.

In this example, the size of the vs object as defined in this code snippet is 4, and the contents are as follows:

[["application/xml","en"], ["application/json","en"],

["application/xml","fr"],["application/json","fr"]]

The javax.ws.rs.core.Request.selectVariant method accepts a list of Variant objects and chooses the Variant object that matches the HTTP request. This method compares its list of Variant objects with the Accept, Accept-Encoding, Accept-Language, and Accept-Charset headers of the HTTP request.

The following code snippet shows how to use the selectVariant method to select the most acceptable Variant from the values in the client request.

@GET

public Response get(@Context Request r) {

List<Variant> vs = ...;

Variant v = r.selectVariant(vs);

if (v == null) {

return Response.notAcceptable(vs).build();

} else {

Object rep = selectRepresentation(v);

return Response.ok(rep, v);

}

}

The selectVariant method returns the Variant object that matches the request, or null if no matches are found. In this code snippet, if the method returns null, a Response object for a non-acceptable response is built. Otherwise, a Response object with an OK status and containing a representation in the form of an Objectentity and a Variant is returned.

## he customer Example Application

This section describes how to build and run the customer sample application. This example application is a RESTful web service that uses JAXB to perform the Create, Read, Update, Delete (CRUD) operations for a specific entity.

The customer sample application is in the *tut-install*/examples/jaxrs/customer/ directory. See [Chapter 2, Using the Tutorial Examples](https://docs.oracle.com/javaee/6/tutorial/doc/gfiud.html) for basic information on building and running sample applications.

### Overview of the customer Example Application

The source files of this application are at *tut-install*/examples/jaxrs/customer/src/java/. The application has three parts:

* The Customer and Address entity classes. These classes model the data of the application and contain JAXB annotations. See [The Customer and AddressEntity Classes](https://docs.oracle.com/javaee/6/tutorial/doc/gkoib.html#gmgcm) for details.
* The CustomerService resource class. This class contains JAX-RS resource methods that perform operations on Customer instances represented as XML or JSON data using JAXB. See [The CustomerService Class](https://docs.oracle.com/javaee/6/tutorial/doc/gkoib.html#gklgt) for details.
* The CustomerClientXML and CustomerClientJSON client classes. These classes test the resource methods of the web service using XML and JSON representations of Customer instances. See [The CustomerClientXML and CustomerClientJSON Classes](https://docs.oracle.com/javaee/6/tutorial/doc/gkoib.html#gkqjq) for details.

The customer sample application shows you how to model your data entities as Java classes with JAXB annotations. The JAXB schema generator produces an equivalent XML schema definition file (.xsd) for your entity classes. The resulting schema is used to automatically marshal and unmarshal entity instances to and from XML or JSON in the JAX-RS resource methods.

In some cases you may already have an XML schema definition for your entities. See [Modifying the Example to Generate Entity Classes from an Existing Schema](https://docs.oracle.com/javaee/6/tutorial/doc/gkoib.html#gmgbv) for instructions on how to modify the customer example to model your data starting from an .xsd file and using JAXB to generate the equivalent Java classes.

### The Customer and Address Entity Classes

The following class represents a customer’s address:

@XmlRootElement(name="address")

@XmlAccessorType(XmlAccessType.FIELD)

public class Address {

@XmlElement(required=true)

protected int number;

@XmlElement(required=true)

protected String street;

@XmlElement(required=true)

protected String city;

@XmlElement(required=true)

protected String state;

@XmlElement(required=true)

protected String zip;

@XmlElement(required=true)

protected String country;

public Address() { }

// Getter and setter methods

// ...

}

The @XmlRootElement(name="address") annotation maps this class to the address XML element. The @XmlAccessorType(XmlAccessType.FIELD)annotation specifies that all the fields of this class are bound to XML by default. The @XmlElement(required=true) annotation specifies that an element must be present in the XML representation.

The following class represents a customer:

@XmlRootElement(name="customer")

@XmlAccessorType(XmlAccessType.FIELD)

public class Customer {

@XmlAttribute(required=true)

protected int id;

@XmlElement(required=true)

protected String firstname;

@XmlElement(required=true)

protected String lastname;

@XmlElement(required=true)

protected Address address;

@XmlElement(required=true)

protected String email;

@XmlElement (required=true)

protected String phone;

public Customer() { }

// Getter and setter methods

// ...

}

The Customer class contains the same JAXB annotations as the previous class, except for the @XmlAttribute(required=true) annotation, which maps a property to an attribute of the XML element representing the class.

The Customer class contains a property whose type is another entity, the Address class. This mechanism allows you to define in Java code the hierarchical relationships between entities without having to write an .xsd file yourself.

JAXB generates the following XML schema definition for the two classes above:

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

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

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

<xs:complexType name="address">

<xs:sequence>

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

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

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

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

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

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

</xs:sequence>

</xs:complexType>

<xs:complexType name="customer">

<xs:sequence>

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

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

<xs:element ref="address"/>

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

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

</xs:sequence>

<xs:attribute name="id" type="xs:int" use="required"/>

</xs:complexType>

</xs:schema>

The file sample-input.xml in the top-level directory of the project contains an example of an XML representation of a customer:

<?xml version="1.0" encoding="UTF-8"?>

<customer id="1">

<firstname>Duke</firstname>

<lastname>OfJava</lastname>

<address>

<number>1</number>

<street>Duke's Way</street>

<city>JavaTown</city>

<state>JA</state>

<zip>12345</zip>

<country>USA</country>

</address>

<email>duke@example.com</email>

<phone>123-456-7890</phone>

</customer>

The file sample-input.json contains an example of a JSON representation of a customer:

{

"@id": "1",

"firstname": "Duke",

"lastname": "OfJava",

"address": {

"number": 1,

"street": "Duke's Way",

"city": "JavaTown",

"state": "JA",

"zip": "12345",

"country": "USA"

},

"email": "duke@example.com",

"phone": "123-456-7890"

}

### The CustomerService Class

The CustomerService class has a createCustomer method that creates a customer resource based on the Customer class and returns a URI for the new resource. The persist method emulates the behavior of the JPA entity manager. This example uses a java.util.Properties file to store data. If you are using the default configuration of GlassFish Server, the properties file is at *domain-dir*/CustomerDATA.txt.

@Path("/Customer")

public class CustomerService {

public static final String DATA\_STORE = "CustomerDATA.txt";

public static final Logger logger =

Logger.getLogger(CustomerService.class.getCanonicalName());

...

@POST

@Consumes({"application/xml", "application/json"})

public Response createCustomer(Customer customer) {

try {

long customerId = persist(customer);

return Response.created(URI.create("/" + customerId)).build();

} catch (Exception e) {

throw new WebApplicationException(e,

Response.Status.INTERNAL\_SERVER\_ERROR);

}

}

...

private long persist(Customer customer) throws IOException {

File dataFile = new File(DATA\_STORE);

if (!dataFile.exists()) {

dataFile.createNewFile();

}

long customerId = customer.getId();

Address address = customer.getAddress();

Properties properties = new Properties();

properties.load(new FileInputStream(dataFile));

properties.setProperty(String.valueOf(customerId),

customer.getFirstname() + ","

+ customer.getLastname() + ","

+ address.getNumber() + ","

+ address.getStreet() + ","

+ address.getCity() + ","

+ address.getState() + ","

+ address.getZip() + ","

+ address.getCountry() + ","

+ customer.getEmail() + ","

+ customer.getPhone());

properties.store(new FileOutputStream(DATA\_STORE),null);

return customerId;

}

...

}

The response returned to the client has a URI to the newly created resource. The return type is an entity body mapped from the property of the response with the status code specified by the status property of the response. The WebApplicationException is a RuntimeException that is used to wrap the appropriate HTTP error status code, such as 404, 406, 415, or 500.

The @Consumes({"application/xml","application/json"}) and @Produces({"application/xml","application/json"}) annotations set the request and response media types to use the appropriate MIME client. These annotations can be applied to a resource method, a resource class, or even an entity provider. If you do not use these annotations, JAX-RS allows the use of any media type ("\*/\*").

The following code snippet shows the implementation of the getCustomer and findbyId methods. The getCustomer method uses the @Produces annotation and returns a Customer object, which is converted to an XML or JSON representation depending on the Accept: header specified by the client.

@GET

@Path("{id}")

@Produces({"application/xml", "application/json"})

public Customer getCustomer(@PathParam("id") String customerId) {

Customer customer = null;

try {

customer = findById(customerId);

} catch (Exception ex) {

logger.log(Level.SEVERE,

"Error calling searchCustomer() for customerId {0}. {1}",

new Object[]{customerId, ex.getMessage()});

}

return customer;

}

private Customer findById(String customerId) throws IOException {

properties properties = new Properties();

properties.load(new FileInputStream(DATA\_STORE));

String rawData = properties.getProperty(customerId);

if (rawData != null) {

final String[] field = rawData.split(",");

Address address = new Address();

Customer customer = new Customer();

customer.setId(Integer.parseInt(customerId));

customer.setAddress(address);

customer.setFirstname(field[0]);

customer.setLastname(field[1]);

address.setNumber(Integer.parseInt(field[2]));

address.setStreet(field[3]);

address.setCity(field[4]);

address.setState(field[5]);

address.setZip(field[6]);

address.setCountry(field[7]);

customer.setEmail(field[8]);

customer.setPhone(field[9]);

return customer;

}

return null;

}

### The CustomerClientXML and CustomerClientJSON Classes

Jersey is the reference implementation of JAX-RS (JSR 311). You can use the Jersey client API to write a test client for the customer example application. You can find the Jersey APIs at <http://jersey.java.net/nonav/apidocs/latest/jersey/>.

The CustomerClientXML class calls Jersey APIs to test the CustomerService web service:

package customer.rest.client;

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

import com.sun.jersey.api.client.ClientResponse;

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

import customer.data.Address;

import customer.data.Customer;

import java.util.logging.Level;

import java.util.logging.Logger;

import javax.ws.rs.core.MediaType;

public class CustomerClientXML {

public static final Logger logger =

Logger.getLogger(CustomerClientXML.class.getCanonicalName());

public static void main(String[] args) {

Client client = Client.create();

// Define the URL for testing the example application

WebResource webResource =

client.resource("http://localhost:8080/customer/rest/Customer");

// Test the POST method

Customer customer = new Customer();

Address address = new Address();

customer.setAddress(address);

customer.setId(1);

customer.setFirstname("Duke");

customer.setLastname("OfJava");

address.setNumber(1);

address.setStreet("Duke's Drive");

address.setCity("JavaTown");

address.setZip("1234");

address.setState("JA");

address.setCountry("USA");

customer.setEmail("duke@java.net");

customer.setPhone("12341234");

ClientResponse response =

webResource.type("application/xml").post(ClientResponse.class,

customer);

logger.info("POST status: {0}" + response.getStatus());

if (response.getStatus() == 201) {

logger.info("POST succeeded");

} else {

logger.info("POST failed");

}

// Test the GET method using content negotiation

response = webResource.path("1").accept(MediaType.APPLICATION\_XML)

.get(ClientResponse.class);

Customer entity = response.getEntity(Customer.class);

logger.log(Level.INFO, "GET status: {0}", response.getStatus());

if (response.getStatus() == 200) {

logger.log(Level.INFO, "GET succeeded, city is {0}",

entity.getAddress().getCity());

} else {

logger.info("GET failed");

}

// Test the DELETE method

response = webResource.path("1").delete(ClientResponse.class);

logger.log(Level.INFO, "DELETE status: {0}", response.getStatus());

if (response.getStatus() == 204) {

logger.info("DELETE succeeded (no content)");

} else {

logger.info("DELETE failed");

}

response = webResource.path("1").accept(MediaType.APPLICATION\_XML)

.get(ClientResponse.class);

logger.log(Level.INFO, "GET status: {0}", response.getStatus());

if (response.getStatus() == 204) {

logger.info("After DELETE, the GET request returned no content.");

} else {

logger.info("Failed, after DELETE, GET returned a response.");

}

}

}

This Jersey client tests the POST, GET, and DELETE methods using XML representations.

All of these HTTP status codes indicate success: 201 for POST, 200 for GET, and 204 for DELETE. For details about the meanings of HTTP status codes, see <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>.

The CustomerClientJSON class is similar to CustomerClientXML but it uses JSON representations to test the web service. In the CustomerClientJSON class "application/xml" is replaced by "application/json", and MediaType.APPLICATION\_XML is replaced by MediaType.APPLICATION\_JSON.

### Modifying the Example to Generate Entity Classes from an Existing Schema

This section describes how you can modify the customer example if you provide an XML schema definition file for your entities instead of providing Java classes. In this case JAXB generates the equivalent Java entity classes from the schema definition.

For the customer example you provide the following .xsd file:

<?xml version="1.0"?>

<xs:schema targetNamespace="http://xml.customer"

xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"

xmlns:ora="http://xml.customer">

<xs:element name="customer" type="ora:Customer"/>

<xs:complexType name="Address">

<xs:sequence>

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

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

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

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

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

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

</xs:sequence>

</xs:complexType>

<xs:complexType name="Customer">

<xs:sequence>

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

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

<xs:element name="address" type="ora:Address"/>

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

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

</xs:sequence>

<xs:attribute name="id" type="xs:int" use="required"/>

</xs:complexType>

</xs:schema>

You can modify the customer example as follows:

#### To Modify the customer Example to Generate Java Entity Classes from an Existing XML Schema Definition

1. **Create a JAXB binding to generate the entity Java classes from the schema definition. For example, in NetBeans IDE, follow these steps:**
   1. **Right click on the customer project and select New > Other...**
   2. **Under the XML folder, select JAXB Binding and click Next.**
   3. **In the Binding Name field, type CustomerBinding.**
   4. **Click Browse and choose the .xsd file from your file system.**
   5. **In the Package Name field, type customer.xml.**
   6. **Click Finish.**

This procedure creates the Customer class, the Address class, and some JAXB auxiliary classes in the package customer.xml.

1. **Modify the CustomerService class as follows:**
   1. **Replace the customer.data.\* imports with customer.xml.\* imports and import the JAXBElement and ObjectFactory classes:**
   2. import customer.xml.Customer;
   3. import customer.xml.Address;
   4. import customer.xml.ObjectFactory;

import javax.xml.bind.JAXBElement;

* 1. **Replace the return type of the getCustomer method:**
  2. public JAXBElement<Customer> getCustomer(
  3. @PathParam("id") String customerId) {
  4. ...
  5. return new ObjectFactory().createCustomer(customer);

}

1. **Modify the CustomerClientXML and CustomerClientJSON classes as follows:**
   1. **Replace the customer.data.\* imports with customer.xml.\* imports and import the JAXBElement and ObjectFactory classes:**
   2. import customer.xml.Address;
   3. import customer.xml.Customer;
   4. import customer.xml.ObjectFactory;

import javax.xml.bind.JAXBElement;

* 1. **Create an ObjectFactory instance and a JAXBElement<Customer> instance at the beginning of the main method:**
  2. public static void main(String[] args) {
  3. Client client = Client.create();
  4. ObjectFactory factory = new ObjectFactory();
  5. WebResource webResource = ...;
  6. ...
  7. customer.setPhone("12341234");
  8. JAXBElement<Customer> customerJAXB = factory.createCustomer(customer);
  9. ClientResponse response = webResource.type("application/xml")
  10. .post(ClientResponse.class, customerJAXB);
  11. ...

}

* 1. **Modify the GET request after testing the DELETE method:**
  2. response = webResource.path("1").accept(MediaType.APPLICATION\_XML)
  3. .get(ClientResponse.class);
  4. entity = response.getEntity(Customer.class);
  5. logger.log(Level.INFO, "GET status: {0}", response.getStatus());
  6. try {
  7. logger.info(entity.getAddress().getCity());
  8. } catch (NullPointerException ne) {
  9. // null after deleting the only customer
  10. logger.log(Level.INFO, "After DELETE, city is: {0}", ne.getCause());

}

1. The instructions for building, deploying, and running the example are the same for the original customer example and for the modified version using this procedure.

### Running the customer Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the customer application.

#### To Build, Package, and Deploy the customer Example Using NetBeans IDE

This procedure builds the application into the *tut-install*/examples/jax-rs/customer/build/web/ directory. The contents of this directory are deployed to the GlassFish Server.

1. **From the File menu, choose Open Project.**
2. **In the Open Project dialog, navigate to:**

*tut-install*/examples/jaxrs/

1. **Select the customer folder.**
2. **Select the Open as Main Project check box.**
3. **Click Open Project.**

It may appear that there are errors in the source files, because the files refer to JAXB classes that will be generated when you build the application. You can ignore these errors.

1. **In the Projects tab, right-click the customer project and select Deploy.**

#### To Build, Package, and Deploy the customer Example Using Ant

1. **In a terminal window, go to:**

*tut-install*/examples/jaxrs/customer/

1. **Type the following command:**

**ant**

This command calls the default target, which builds and packages the application into a WAR file, customer.war, located in the dist directory.

1. **Type the following command:**

**ant deploy**

Typing this command deploys customer.war to the GlassFish Server.

#### To Run the customer Example Using the Jersey Client

1. **In NetBeans IDE, expand the Source Packages node.**
2. **Expand the customer.rest.client node.**
3. **Right-click the CustomerClientXML.java file and select Run File.**

The output of the client looks like this:

run:

Jun 12, 2012 2:40:20 PM customer.rest.client.CustomerClientXML main

INFO: POST status: 201

Jun 12, 2012 2:40:20 PM customer.rest.client.CustomerClientXML main

INFO: POST succeeded

Jun 12, 2012 2:40:20 PM customer.rest.client.CustomerClientXML main

INFO: GET status: 200

Jun 12, 2012 2:40:20 PM customer.rest.client.CustomerClientXML main

INFO: GET succeeded, city is JavaTown

Jun 12, 2012 2:40:20 PM customer.rest.client.CustomerClientXML main

INFO: DELETE status: 204

Jun 12, 2012 2:40:20 PM customer.rest.client.CustomerClientXML main

INFO: DELETE succeeded (no content)

Jun 12, 2012 2:40:20 PM customer.rest.client.CustomerClientXML main

INFO: GET status: 204

Jun 12, 2012 2:40:20 PM customer.rest.client.CustomerClientXML main

INFO: After DELETE, the GET request returned no content.

BUILD SUCCESSFUL (total time: 5 seconds)

The output is slightly different for the modified customer example:

run:

Jun 12, 2012 2:40:20 PM customer.rest.client.CustomerClientXML main

INFO: POST status: 201

[...]

Jun 12, 2012 2:40:20 PM customer.rest.client.CustomerClientXML main

INFO: DELETE succeeded (no content)

Jun 12, 2012 2:40:20 PM customer.rest.client.CustomerClientXML main

INFO: GET status: 200

Jun 12, 2012 2:40:20 PM customer.rest.client.CustomerClientXML main

INFO: After DELETE, city is: null

BUILD SUCCESSFUL (total time: 5 seconds)

#### To Run the customer Example Using the Web Services Tester

1. **In NetBeans IDE, right-click the customer node and select Test RESTful Web Services.**

**Note -**The Web Services Tester works only with the modified version of the customer example.

1. **In the Configure REST Test Client dialog, select Web Test Client in Project and click Browse.**
2. **In the Select Project dialog, choose the customer project and click OK.**
3. **In the Configure REST Test Client dialog, click OK.**
4. **When the test client appears in the browser, select the Customer resource node in the left pane.**
5. **Paste the following XML code into the Content text area, replacing “Insert content here”:**
6. <?xml version="1.0" encoding="UTF-8"?>
7. <customer id="1">
8. <firstname>Duke</firstname>
9. <lastname>OfJava</lastname>
10. <address>
11. <number>1</number>
12. <street>Duke's Way</street>
13. <city>JavaTown</city>
14. <state>JA</state>
15. <zip>12345</zip>
16. <country>USA</country>
17. </address>
18. <email>duke@example.com</email>
19. <phone>123-456-7890</phone>

</customer>

You can find the code in the file customer/sample-input.xml.

1. **Click Test.**

The following message appears in the window below:

Status: 201 (Created)

1. **Expand the Customer node and click {id}.**
2. **Type 1 in the id field and click Test to test the GET method.**

The following status message appears:

Status: 200 (OK)

The XML output for the resource appears in the Response window:

<?xml version="1.0" encoding="UTF-8"?>

<customer xmlns="http://xml.customer" id="1">

<firstname>Duke</firstname>

<lastname>OfJava</lastname>

<address>

<number>1</number>

<street>Duke's Way</street>

<city>JavaTown</city>

<state>JA</state>

<zip>12345</zip>

<country>USA</country>

</address>

<email>duke@example.com</email>

<phone>123-456-7890</phone>

</customer>

A GET for a nonexistent ID also returns a 200 (OK) status, but the output in the Response window shows no content:

<?xml version="1.0" encoding="UTF-8"?>

<customer xmlns="http://xml.customer"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:nil="true"/>

You can test other methods as follows:

* + Select PUT, type the input for an existing customer, modify any content except the id value, and click Test to update the customer fields. A successful update returns the following status message:

Status: 303 (See Other)

* + Select DELETE, type the ID for an existing customer, and click Test to remove the customer. A successful delete returns the following status message:

Status: 204 (See Other)

#### Using Curl to Run the customer Example Application

Curl is a command-line tool you can use to run the customer application on UNIX platforms. You can download Curl from [http://curl.haxx.se](http://curl.haxx.se/) or add it to a Cygwin installation.

Run the following commands in the directory *tut-install*/examples/jaxrs/customer/ after deploying the application.

To add a new customer and test the POST method using XML data, use the following command:

curl -i --data @sample-input.xml \

--header Content-type:application/xml \

http://localhost:8080/customer/rest/Customer

To add a new customer using JSON data instead, use the following command:

curl -i --data @sample-input.json \

--header Content-type:application/json \

http://localhost:8080/customer/rest/Customer

A successful POST returns HTTP Status: 201 (Created).

To retrieve the details of the customer whose ID is 1, use the following command:

curl -i -X GET http://localhost:8080/customer/rest/Customer/1

To retrieve the details of the same customer represented as JSON data, use the following command:

curl -i --header Accept:application/json

-X GET http://localhost:8080/customer/rest/Customer/1

A successful GET returns HTTP Status: 200 (OK).

To delete a customer record, use the following command:

curl -i -X DELETE http://localhost:8080/customer/rest/Customer/1

A successful DELETE returns HTTP Status: 204.

The customer example and the modified version respond differently to a GET request for a customer ID that does not exist. The original customer example returns HTTP Status: 204 (No content), whereas the modified version returns HTTP Status: 200 (OK) with a response that contains the XML header but no customer data.

**The most frequently asked RESTful Web services interview questions:**

Web services, a very well-known term when we talk about exchanging some sort of data between multiple applications or say software. Based on the client-server model, these services can be used by multiple software application written in various languages and it has an advantage of running on various platforms.

Likewise, REST, Representational State Transfer is also based on stateless client-server style architecture, which can be easily accessed over the network and is identified by URIs i.e. Uniform Resource Identifier.

The main aim of describing the definition of Web services as well as REST above is to help you relate to the term ‘RESTful web services’ because RESTful web services are defined as web services which use HTTP method and is based on the architecture of REST. It has useful features like high scalability and maintainability, the creation of APIs, etc.

In this article, you will find the collection of question and answer which will clear your basics and help develop the better understanding of the subject.

**Also read:**

* [**Top Web Services Interview Questions**](http://www.softwaretestinghelp.com/web-services-interview-questions/)
* [**Top SoapUI interview questions**](http://www.softwaretestinghelp.com/soapui-interview-questions-and-answers/)

## RESTful web services Question and Answers

Let’s start.

**Q #1) What is your understanding of what are RESTful web services?**

Just like SOAP (Simple Object Access Protocol), which is used to develop web services by XML method, RESTful web services use web protocol i.e. HTTP protocol method. They have the feature like scalability, maintainability, help multiple application communication built on various programming languages etc.

RESTful web service implementation defines the method of accessing various resources which are required by the client and he has sent the request to the server through the web browser. The important aspects of this implementation include:

* Resources
* Request Headers
* Request Body
* Response Body
* Status codes

**Q #2) Name the protocol which is used by RESTful web services.**

RESTful web services use a famous web protocol i.e. HTTP protocol. This serves as a medium of data communication between client and server. HTTP standard methods are used to access resources in RESTful web service architecture.

**Q #3) Explain the term ‘Addressing’ with respect to RESTful WEB service.**

Just like we require address with postal code to reach any person, in the same way, ‘Addressing’ locates resources that are present on the server for the purpose of hosting web services. This is usually done with URI i.e. Unified Resource Identifier.

**Q #4) Enlist features of RESTful web services.**

Every RESTful web services should have following features and characteristics that are enlisted below:

* Based on the Client Server representation.
* Use of HTTP protocol for performing functions like fetching data from the web service, retrieving resources, execution of any query, etc.
* The communication between the server and client is performed through the medium known as ‘messaging’.
* Addressing of resources available on the server through URIs.
* Based on the concept of statelessness where every client request and the response is independent of the other with complete assurance of providing required information.
* Uses the concept of caching.
* Works on Uniform interface.

**Q #5) Explain messaging technique.**

Messages are the mode of exchanging data for any type of communication to take place. In the same way, HTTP protocol plays the role of message communication between the client and server through HTTP Request and Response methods. HTTP request is sent by the client who contains information about the data and in turn, receives HTTP Response from the server.

Messages are the collection of information about the data i.e. Metadata.

**Q #6) What are the core components of HTTP request and HTTP response?**

The core components that come under HTTP Request are:

* **Verb:** Includes methods like GET, PUT, POST, etc.
* Uniform Resource Identifier for identifying the resources available on the server.
* HTTP Version for specifying the HTTP version.
* HTTP Request header for containing the information about the data.
* HTTP Request body that contains the representation of the resources in use.

**The core components that come under HTTP Response are:**

* **Request Code:** This contains various codes which determine the status of the server response.
* HTTP Version for specifying the HTTP version.
* HTTP Response header for containing the information about the data.
* HTTP Response body that contains the representation of the resources in use.

**Q #7) Explain the term ‘Statelessness’ with respect to RESTful WEB service.**

In REST, ST itself defines State Transfer and Statelessness means complete isolation. This means, the state of the client’s application is never stored on the server and is passed on. In this process, the clients send all the information that is required for the server to fulfill the HTTP request that has been sent. Thus every client request and the response is independent of the other with complete assurance of providing required information.

Every client passes a ‘session identifier’ which also acts as an identifier for each session.

**Q #8) Enlist advantages and disadvantages of ‘Statelessness’.**

In the above question, we have understood the meaning of statelessness with respect to the client-server communication. Now, let us see some of its advantages and disadvantages.

**Advantages:**

* Every method required for communication is identified as an independent method i.e. there are no dependencies to other methods.
* Any previous communication with the client and server is not maintained and thus the whole process is very much simplified.
* If any information or metadata used earlier in required in another method, then the client sends again that information with HTTP request.
* HTTP protocol and REST web service, both shares the feature of statelessness.

**Disadvantages:**

* In every HTTP request from the client, the availability of some information regarding the client state is required by the web service.

**Q #9) Enlist some important constraints for RESTful web services.**

Every constraint has positive as well as negative impacts and to produce an overall architecture, there should be the balance between both. Below mentioned are some important constraints for RESTful web service:

* There should be separate concerns for each server and client which will help to maintain the modularity within the application. This will also reduce the complexity and increase the scalability.
* The client-server communication should be stateless, which means no previous information is used and the complete execution is done in isolation. In cases of failure, it also helps the client to recover.
* In client-server communication, the HTTP response should be cacheable so that when required cached copy can be used which in turn enhances the scalability and performance of the server.
* The fourth constraint is the uniform interface which allows client-server interaction to be easily understood. This constraint is further divided into four sub-constraints as:
  + Resource Identification
  + Resource manipulation
  + Each message is easily understood and is self-descriptive.
  + Hypermedia, which is defined as the text with hyperlinks and when clicked it moves to another application state.
* Client-server communication should be done on a layered system and thus the client should only have knowledge about the intermediate level with which communication is being done,

**Q #10) What is a ‘Resource’?**

Just like the ‘Object’ instance, we have learned in object orient programming Language, in the same way, ‘Resource’ is defined as an object of a type which can be an image, HTML file, text data, and any type of dynamic data. There are varieties of representation formats available in order to represent a resource.

**Some most common are enlisted below:**

* JSON
* YAML
* XML
* HTML

**Q #11) Why proper representation of Resource is required?**

Representation is very important because it determines the easy identification of resources. With proper representations of resource in the proper format, allows the client to easily understand the format.

**Q #12) Enlist some important points that should be kept in mind while designing Resources representation for RESTful web services.**

As there are no restrictions on the format in which the resource representation is done but just that the main requirement is the format of the representation should be as per the client requirement. A good resource representation is designed by considering the following main points:

* The resource representation format should be easily understood by the client and server.
* The representation should be complete regardless of its format structure, which may be complex or simple.
* In the case of the link of the resources to other resources, such cases should also be considered and handled.

**Q #13) What is Caching?**

Caching is the process in which server response is stored so that a cached copy can be used when required and there is no need of generating the same response again. This process not only reduces the server load but in turn increase the scalability and performance of the server. Only the client is able to cache the response and that too for a limited period of time.

Mentioned below are the header of the resources and their brief description so that they can be identified for the caching process:

* Time and Date of resource creation
* Time and date of resource modification that usually stores the last detail.
* Cache control header
* Time and date at which the cached resource will expire.
* The age which determines the time from when the resource has been fetched.

**Q #14) Explain Cache-control header.**

A standard Cache control header can help in attaining cache ability. Enlisted below is the brief description of various cache control header:

* **Public:** Resources that are marked as the public can be cached by any intermediate components between the client and server.
* **Private:** Resources that are marked as private can only be cached by the client.
* No cache means that resource cannot be cached and thus the whole process is stopped.

**Q #15) What are the best practices that are to be followed while designing RESTful web services?**

To design a secure RESTful web service, there are some best practices or say points that should be considered. These are explained as follows:

* Every input on the server should be validated.
* Input should be well formed.
* Never pass any sensitive data through URL.
* For any session, the user should be authenticated.
* Only HTTP error messages should be used for indicating any fault.
* Use message format that is easily understood and is required by the client.
* Unified Resource Identifier should be descriptive and easily understood.

**Q #16) What is Payload?**

The request data which is present in the body part of every HTTP message is referred as ‘Payload’.  In Restful web service, the payload can only be passed to the recipient through POST method.

There is no limit of sending data as payload through POST method but the only concern is that more data with consuming more time and bandwidth. This may consume much of user’s time also.

**Q #17) Enlist some of the HTTP methods with description.**

Mentioned below is the list of HTTP methods with their descriptions:

* **GET:** This is a read only operation which fetches the list of users on the server.
* **PUT:** This operation is used for the creation of any new resource on the server.
* **POST:** This operation is used for updating an old resource or for creating a new resource.
* **DELETE:** As the name suggests, this operation is used for deleting any resource on the server.
* **OPTIONS:** This operation fetches the list of any supported options of resources that are available on the server.

**Q #18) What is the difference between PUT method and POST method?**

The major difference between the PUT and POST method is that the result generated with PUT method is always same no matter how many times the operation is performed. On the other hand, the result generated by POST operation is always different every time.

**Q #19) What is your understanding about JAX-RS?**

JAX-RS is defined as the Java API for RESTful web service. Among multiple libraries and framework, this is considered as the most suitable Java programming language based API which supports RESTful web service.

**Some of the implementations of JAX-RS are:**

* Jersey
* RESTEasy
* Apache CFX
* Play

Among these, Jersey is the most popular framework.

**Q #20) What are HTTP status codes? Enlist few with meaning.**

HTTP status codes basically are the representation of the status of the task that has been performed on the server, with the mode of some codes. Every code has their own meaning.

**Some of the HTTP status codes with their meaning are as follows:**

* **Code 200:** This indicates success.
* **Code 201:** This indicates resource has been successfully created.
* **Code 204:** This indicates that there is no content in the response body.
* **Code 404:** This indicates that there is no method available.

There are few more such codes that indicate the status.

### Conclusion:

This article will help you prepare for the RESTful web services interview and help you to understand the concept in the simple and easy way. I have tried to cover all the areas which are very necessary for having the complete knowledge about RESTful Web services.

Just remember, it may be possible that you are not able to answer all questions in the interview but whatever you answer should be accurate. Your basic concept should be strong and your confidence level should be high.

**What does REST stand for?**(answer)  
REST stands for REpresentational State Transfer, which uses HTTP protocol to send data from client to server e.g. a book in the server can be delivered to the client using JSON or XML.  
  
  
**What is a resource?**(answer)  
A resource is how data is represented in REST architecture. By exposing entities as the resource it allows a client to read, write, modify, and create resources using HTTP methods e.g. [GET](http://javarevisited.blogspot.sg/2012/03/get-post-method-in-http-and-https.html), [POST](http://www.java67.com/2014/08/difference-between-post-and-get-request.html), [PUT](http://www.java67.com/2016/09/when-to-use-put-or-post-in-restful-web-services.html), DELETE etc.  
  
  
**What are safe REST operations?**(answer)  
REST API uses HTTP methods to perform operations. Some of the HTTP operations which doesn't modify the resource at the server is known as safe operations e.g. GET and HEAD. On the other hand, [PUT](http://javarevisited.blogspot.sg/2016/10/difference-between-put-and-post-in-restful-web-service.html), POST, and DELETE are unsafe because they modify the resource on the server.  
  
 **What are idempotent operations? Why is idempotency important?**([answer](http://javarevisited.blogspot.sg/2016/05/what-are-idempotent-and-safe-methods-of-HTTP-and-REST.html))  
There are some HTTP methods e.g. GET which produce same response no matter how many times you use them e.g. sending multiple GET request to the same URI will result in same response without any side-effect hence it is known as idempotent.  
  
On the other hand, the POST is not idempotent because if you send multiple POST request, it will result in multiple resource creation on the server, but again, PUT is idempotent if you are using it to update the resource.  
  
Even, multiple PUT request to update a resource on a server will give same end result. You can further take [HTTP Fundamentals](http://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fxhttp-fund) course by Pluralsight to learn more about idempotent methods of HTTP protocol and HTTP in general.

**Is REST scalable and/or interoperable?**(answer)  
Yes, REST is Scalable and interoperable. It doesn't mandate a specific choice of technology either at client or server end. You can use Java, C++, Python or JavaScript to create RESTful Web Services and Consume them at the client end. I suggest you read a good book on REST API e.g. [RESTful Web Services](http://javarevisited.blogspot.sg/2017/02/top-5-books-to-learn-rest-and-restful-web-services-in-java.html) to learn more about REST.  
  
  
**What are the advantages of the RestTemplate?**([answer](http://javarevisited.blogspot.sg/2017/02/how-to-consume-json-from-restful-web-services-Spring-RESTTemplate-Example.html))  
The RestTemplate class is an implementation of Template method pattern in Spring framework. Similar to other popular template classes e.g. JdbcTemplate or JmsTempalte, it also simplifies the interaction with RESTful Web Services on the client side. You can use it to consume a RESTful Web Servicer very easily as shown in this example.  
  
  
**Which HTTP methods does REST use?**([answer](http://javarevisited.blogspot.sg/2016/04/what-is-purpose-of-http-request-types-in-RESTful-web-service.html#axzz56WGunSwy))  
REST can use any HTTP methods but the most popular ones are GET for retrieving a resource, POST for creating a resource, PUt for updating resource and DELETE for removing a resource from the server.  
  
  
  
**What is an HttpMessageConverter in Spring REST?**(answer)  
An HttpMessageConverter is a [Strategy interface](http://www.java67.com/2014/12/strategy-pattern-in-java-with-example.html) that specifies a converter that can convert from and to HTTP requests and responses. Spring REST uses this interface to convert HTTP response to various formats e.g. JSON or XML.  
  
Each HttpMessageConverter implementation has one or several MIME Types associated with it. Spring uses the "Accept" header to determine the content type client is expecting.  
  
It will then try to find a registered HTTPMessageConverter that is capable of handling that specific content-type and use it to convert the response into that format before sending to the client.  
  
  
  
**How to create a custom implementation of HttpMessageConverter to support a new type of request/responses?**(answer)  
You just need to create an implementation of AbstractHttpMessageConverter and register it using the WebMvcConfigurerAdapter#extendMessageConverters() method with the classes which generate a new type of request/response.  
  
  
**Is REST normally stateless?**([answer](http://javarevisited.blogspot.sg/2015/08/difference-between-soap-and-restfull-webservice-java.html))  
Yes, REST API should be stateless because it is based on HTTP which is also stateless. A Request in REST API should contain all the details required it to process i.e. it should not rely on previous or next request or some data maintained at the server end e.g. Sessions. REST specification put a constraint to make it stateless and you should keep that in mind while designing your REST API.  
  
 **What does @RequestMapping annotation do?**([answer](http://javarevisited.blogspot.sg/2017/06/how-spring-mvc-framework-works-web-flow.html#axzz55vF5ugU8))  
The @RequestMapping annotation is used to map web requests to Spring Controller methods. You can map request based upon HTTP methods  e.g. GET and POST and various other parameters. For examples, if you are developing RESTful Web Service using Spring then you can use produces and consumes property along with media type annotation to indicate that this method is only used to produce or consumers JSON as shown below:

@RequestMapping (**method** **=** RequestMethod.POST, consumes**=**"application/json")

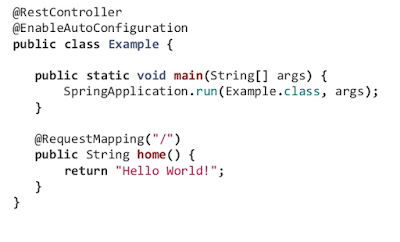
**public** Book save(@RequestBody Book aBook) {

**return** bookRepository.save(aBook);

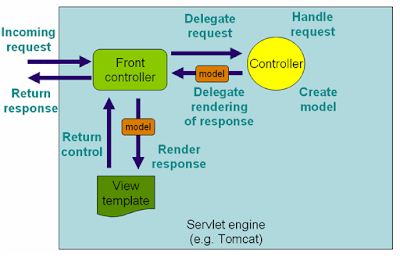
}

You can similarly create other handler methods to produce JSON or XML. If you are not familiar with these annotations then I suggest you join [**Spring MVC For Beginners**](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https%3A%2F%2Fwww.udemy.com%2Fspring-mvc-tutorial-for-beginners-step-by-step%2F) course on Udemy to learn from scratch.

**Is @Controller a stereotype? Is @RestController a stereotype?**([answer](http://javarevisited.blogspot.sg/2017/08/difference-between-restcontroller-and-controller-annotations-spring-mvc-rest.html))  
Yes, both @Controller and @RestController are stereotypes. The @Controller is actually a specialization of Spring's @Component stereotype annotation. This means that class annotated with @Controller will also be automatically be detected by Spring container as part of container's component scanning process.  
  
And, @RestController is a specialization of @Controller for RESTful web service. It not only combines @ResponseBody and @Controller annotation but also gives more meaning to your controller class to clearly indicate that it deals with RESTful requests.  
  
Spring Framework may also use this annotation to provide some more useful features related to REST API development in future.  
 **What is the difference between @Controller and @RestController?**([answer](http://javarevisited.blogspot.sg/2017/08/difference-between-restcontroller-and-controller-annotations-spring-mvc-rest.html))  
There are many differences between @Controller and @RestController as discussed in my earlier article (see the answer) but the most important one is that with @RestController you get the @ResponseBody annotation automatically, which means you don't need to separately annotate your handler methods with @ResponseBody annotation. This makes the development of RESTful web service easier using Spring. You can see here to learn

[](http://javarevisited.blogspot.sg/2016/12/top-5-spring-and-hibernate-training-courses-java-jee-programmers.html)

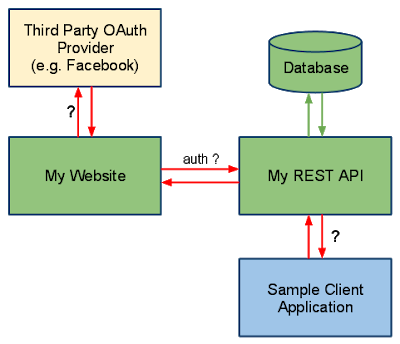
**When do you need @ResponseBody annotation in Spring MVC?**([answer](http://javarevisited.blogspot.sg/2018/01/7-reasons-for-using-spring-to-develop-RESTful-web-service.html#axzz55a8rTeu7))  
The @ResponseBody annotation can be put on a method to indicates that the return type should be written directly to the HTTP response body (and not placed in a Model, or interpreted as a view name).  
  
For example:  
  
@RequestMapping(path = "/hello", method = RequestMethod.PUT)  
@ResponseBody  
public String helloWorld() {  
   return "Hello World";  
}  
  
Alternatively, you can also use @RestController annotation instead of @Controller annotation. This will remove the need for using @ResponseBody because as discussed in the previous answer, it comes automatically with @RestController annotation.  
  
  
  
**What does @PathVariable do in Spring MVC? Why it's useful in REST with Spring?**([answer](http://javarevisited.blogspot.sg/2017/10/differences-between-requestparam-and-pathvariable-annotations-spring-mvc.html))  
It's one of the useful annotations from Spring MVC which allows you to read values from URI like query parameter. It's particularly useful in case of creating RESTful web service using Spring because in REST resource identifiers are part of URI.This questions is normally asked to experienced Spring MVC developers e.g. 4 to 6 years of experience.  
  
For example, in the URL http://myapp.com/books/101 if you want to extract 101 the id, then you can use @PathVariable annotation of Spring MVC.  If you are not familiar with Spring MVC annotations then [Spring MVC For Beginners: Build Java Web App in 25 Steps](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https%3A%2F%2Fwww.udemy.com%2Fspring-mvc-tutorial-for-beginners-step-by-step%2F) is a good place to start with.

[](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https://www.udemy.com/spring-mvc-tutorial-for-beginners-step-by-step/)

**What is the HTTP status return code for a successful DELETE statement?**([answer](http://www.java67.com/2015/09/top-10-restful-web-service-interview-questions-answers.html))  
There is no strict rule with respect to what status code your REST API should return after a successful DELETE i.e it can return 200 Ok or 204 No Content. In general, if the DELETE operation is successful and the response body is empty return 204. If the DELETE request is successful and the response body is NOT empty, return 200  
  
  
**What does CRUD mean?**(answer)  
CRUD is a short form of Create, Read, Update and Delete. In REST API, the POST is used to create a resource, GET is used to read a resource, [PUT](http://javarevisited.blogspot.sg/2016/10/difference-between-put-and-post-in-restful-web-service.html) is used to updated a resource and DELETE is used to remove a resource from the server. This one is another beginner level Spring MVC questions for 1 to 3 years experienced programmers  
  
  
**Where do you need @EnableWebMVC?**(answer)  
The @EnableWebMvc annotation is required to enable Spring MVC when Java configuration is used to configure Spring MVC instead of XML. It is equivalent to <mvc: annotation-driven>  in XML configuration.  
  
It enables support for @Controller-annotated classes that use @RequestMapping to map incoming requests to handler methods not already familiar with Spring's support for Java configuration, [Spring Master Class](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https%3A%2F%2Fwww.udemy.com%2Fspring-tutorial-for-beginners%2F) on Udemy is a good place to start.

**When do you need @ResponseStatus annotation in Spring MVC?**([answer](http://javarevisited.blogspot.sg/2018/01/7-reasons-for-using-spring-to-develop-RESTful-web-service.html#axzz55a8rTeu7))

A good questions for 3 to 5 years experienced spring developers. The @ResponseStatus annotation is required during error handling in Spring MVC and REST. Normally when an error or exception is thrown at server side, web server return a blanket HTTP status code 500 - Internal server error.  
  
This may work for a human user but not for REST clients. You need to send them proper status code e.g. 404 if the resource is not found. That's where you can use @ResponseStatus annotation, which allows you to send custom HTTP status code along with proper error message in case of Exception.  
  
In order to use it, you can create custom exceptions and annotated them using @ResponseStatus annotation and proper HTTP status code and reason.  
  
When such exceptions are thrown from controller's handler methods and not handled anywhere else, then appropriate HTTP response with the proper HTTP status code, which you have set is sent to the client.  
  
For example, if you are writing a RESTful Web Service for a library which provides book information then you can use @ResponseStatus to create Exception which returns HTTP response code 404 when a book is not found instead of Internal Server Error (500), as shown below:  
  
 @ResponseStatus(value=HttpStatus.NOT\_FOUND, reason="No such Book")  // 404  
 public class BookNotFoundException extends RuntimeException {  
     // ...  
 }  
  
If this Exception is thrown from any handler method then HTTP error code 404 with reason "No such Book" will be returned to the client.  
  
  
**Is REST secure? What can you do to secure it?**([answer](http://www.java67.com/2017/04/3-great-books-to-learn-java-web-services-soap-and-restful.html))  
This question is mostly asked with experienced Java programmers e.g. 2 to 5 years experience with both REST and Spring. Security is a broad term, it could mean security of message which is provided by encryption or access restriction which is provided using authentication and authorization. REST is normally not secure but you can secure it by using Spring security.  
  
At the very least you can enable HTTP basic authentication by using HTTP in your Spring security configuration file. Similarly, you can expose your REST API using[HTTPS](http://javarevisited.blogspot.sg/2013/07/how-ssl-https-and-certificates-works-in-java-web-application.html) if the underlying server supports HTTPS.

[](http://javarevisited.blogspot.sg/2018/01/how-to-enable-http-basic-authentication-spring-security-java-xml-configuration.html)

**Does REST work with transport layer security (TLS)?**([answer](http://javarevisited.blogspot.sg/2012/01/rest-web-services-framework-interview.html))  
TLS or Transport Layer Security is used for secure communication between client and server. It is the successor of SSL (Secure Socket Layer). Since HTTPS can work with both SSL and TLS, REST can also work with TLS.  
  
Actually, REST says anything about Security, it's up to the server which implements that. Same RESTful Web Service can be accessed using HTTP and HTTPS if the server supports [SSL](http://javarevisited.blogspot.sg/2013/07/how-to-configure-https-ssl-in-tomcat-6-7-web-server-java.html#axzz56WXxxAC0).  
  
If you are using Tomcat, you can see here to learn more about how to enable SSL in Tomcat.  
  
  
**Do you need Spring MVC in your classpath for developing RESTful Web Service?**([answer](http://javarevisited.blogspot.sg/2017/01/where-and-how-to-download-spring-JAR-Files-Spring4-without-Maven-Gradle.html#axzz4pp42TeHu))  
This question is often asked to Java programmers with 1 to 2 years of experience in Spring. Short answer is Yes, you need Spring MVC in your Java application's classpath to develop RESTful web services using Spring framework. It's actually Spring MVC which provides all useful annotations e.g. @RestController, @ResponseCode, @ResponseBody, @RequestBody, and @PathVariable, hence you must spring-mvc.jar or appropriate Maven entry in your pom.xml  
  
  
That's all about some **frequently asked Spring REST Interview questions** for beginners and experienced Java JEE developers. These questions are also very useful to brush up your knowledge about Spring REST if you are going to take Spring Certification. If you need more questions from Spring certification perspective, you will find a lot of question on this topic on David Mayer's [Core Spring Simulator](https://certification-questions.com/spring-free-mock-exams/spring-core-v4.2-practice-test.html?affiliateCode=fcff36fd-557a-4713-abf6-973e9924770f&utm_source=Javin&utm_medium=affiliate&utm_campaign=affiliate), one of the best simulator to pass Spring certification at the moment.

1. [What is a Web Service?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#what-is-web-service)
2. [What are the advantages of Web Services?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#advantages-of-web-services)
3. [What are different types of Web Services?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#types-of-web-services)
4. [What is SOAP?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#soap-definition)
5. [What are advantages of SOAP Web Services?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#soap-advantages)
6. [What are disadvantages of SOAP Web Services?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#soap-disadvantages)
7. [What is WSDL?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#wsdl-definition)
8. [What are different components of WSDL?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#wsdl-components)
9. [What is UDDI?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#uddi-definition)
10. [What is difference between Top Down and Bottom Up approach in SOAP Web Services?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#soap-ws-approach)
11. [What is REST Web Services?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#rest-web-services)
12. [What are advantages of REST web services?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#rest-advantages)
13. [What are disadvantages of REST web services?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#rest-disadvantages)
14. [What is a Resource in Restful web services?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#restful-resource)
15. [What are different HTTP Methods supported in Restful Web Services?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#http-methods)
16. [Compare SOAP and REST web services?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#soap-vs-rest)
17. [What are different ways to test web services?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#web-services-testing)
18. [Can we maintain user session in web services?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#web-services-seesion)
19. [What is difference between SOA and Web Services?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#soa-vs-web-services)
20. [What is the use of Accept and Content-Type Headers in HTTP Request?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#http-request-headers)
21. [How would you choose between SOAP and REST web services?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#soap-rest-choice)
22. [What is JAX-WS API?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#jax-ws-api)
23. [Name some frameworks in Java to implement SOAP web services?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#java-soap-frameworks)
24. [Name important annotations used in JAX-WS API?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#jax-ws-annotations)
25. [What is use of javax.xml.ws.Endpoint class?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#jax-ws-endpoint)
26. [What is the difference between RPC Style and Document Style SOAP web Services?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#soap-rpc-vs-document)
27. [How to get WSDL file of a SOAP web service?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#soap-wsdl-url)
28. [What is sun-jaxws.xml file?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#soap-config-file)
29. [What is JAX-RS API?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#jax-rs-api)
30. [Name some implementations of JAX-RS API?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#jax-rs-implementations)
31. [What is wsimport utility?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#wsimport-utility)
32. [Name important annotations used in JAX-RS API?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#jax-rs-annotations)
33. [What is the use of @XmlRootElement annotation?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#xmlrootelement-jaxb)
34. [How to set different status code in HTTP response?](https://www.journaldev.com/9193/web-services-interview-questions-soap-restful#http-response-code)
35. **What is a Web Service?**

Web Services work on client-server model where client applications can access web services over the network. Web services provide endpoint URLs and expose methods that can be accessed over network through client programs written in java, shell script or any other different technologies.  
Web services are stateless and doesn’t maintain user session like web applications.

1. **What are the advantages of Web Services?**

Some of the advantages of web services are:

* + Interoperability: Web services are accessible over network and runs on HTTP/SOAP protocol and uses XML/JSON to transport data, hence it can be developed in any programming language. Web service can be written in java programming and client can be PHP and vice versa.
  + Reusability: One web service can be used by many client applications at the same time.
  + Loose Coupling: Web services client code is totally independent with server code, so we have achieved loose coupling in our application.
  + Easy to deploy and integrate, just like web applications.
  + Multiple service versions can be running at same time.



1. **What are different types of Web Services?**

There are two types of web services:

* + SOAP Web Services: Runs on SOAP protocol and uses XML technology for sending data.
  + Restful Web Services: It’s an architectural style and runs on HTTP/HTTPS protocol almost all the time. REST is a stateless client-server architecture where web services are resources and can be identified by their URIs. Client applications can use HTTP GET/POST methods to invoke Restful web services.

1. **What is 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.

1. **What are advantages of SOAP Web Services?**

SOAP web services have all the advantages that web services has, some of the additional advantages are:

* + WSDL document provides contract and technical details of the web services for client applications without exposing the underlying implementation technologies.
  + SOAP uses XML data for payload as well as contract, so it can be easily read by any technology.
  + SOAP protocol is universally accepted, so it’s an industry standard approach with many easily available open source implementations.

1. **What are disadvantages of SOAP Web Services?**

Some of the disadvantages of SOAP protocol are:

* + Only XML can be used, JSON and other lightweight formats are not supported.
  + SOAP is based on the contract, so there is a tight coupling between client and server applications.
  + SOAP is slow because payload is large for a simple string message, since it uses XML format.
  + Anytime there is change in the server side contract, client stub classes need to be generated again.
  + Can’t be tested easily in browser

1. **What is 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.

1. **What are different components of WSDL?**

Some of the different tags in WSDL xml are:

* + xsd:import namespace and schemaLocation: provides WSDL URL and unique namespace for web service.
  + message: for method arguments
  + part: for method argument name and type
  + portType: service name, there can be multiple services in a wsdl document.
  + operation: contains method name
  + soap:address for endpoint URL.

1. **What is 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.

1. **What is difference between Top Down and Bottom Up approach in SOAP Web Services?**

In Top Down approach first WSDL document is created to establish the contract between web service and client and then code is written, it’s also termed as contract first approach. This is hard to implement because classes need to be written to confirm the contract established in WSDL. Benefit of this approach is that both client and server code can be written in parallel.

In Bottom Up approach, first web service code is written and then WSDL is generated. It’s also termed as contract last approach. This approach is easy to implement because WSDL is generated based on code. In this approach client code have to wait for WSDL from server side to start their work.

1. **What is REST Web Services?**

REST is the acronym for REpresentational State Transfer. REST is an architectural style for developing applications that can be accessed over the network. REST architectural style was brought in light by Roy Fielding in his doctoral thesis in 2000.

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

1. **What are advantages of REST web services?**

Some of the advantages of REST web services are:

* + Learning curve is easy since it works on HTTP protocol
  + Supports multiple technologies for data transfer such as text, xml, json, image etc.
  + No contract defined between server and client, so loosely coupled implementation.
  + REST is a lightweight protocol
  + REST methods can be tested easily over browser.

1. **What are disadvantages of REST web services?**

Some of the disadvantages of REST are:

* + Since there is no contract defined between service and client, it has to be communicated through other means such as documentation or emails.
  + Since it works on HTTP, there can’t be asynchronous calls.
  + Sessions can’t be maintained.

1. **What is a Resource in Restful web services?**

Resource is the fundamental concept of Restful architecture. A resource is an object with a type, relationship with other resources and methods that operate on it. Resources are identified with their URI, HTTP methods they support and request/response data type and format of data.

1. **What are different HTTP Methods supported in Restful Web Services?**

Restful web services supported HTTP methods are – GET, POST, PUT, DELETE and HEAD.

1. **Compare SOAP and REST web services?**

|  |  |
| --- | --- |
| 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. |

## SOAP vs. REST

Let' have a quick overview of SOAP and REST before we do a deep dive into the key differences between them.

**SOAP** – SOAP is a protocol which was designed before REST and came into the picture. The main idea behind designing SOAP was to ensure that programs built on different platforms and programming languages could exchange data in an easy manner.

**REST** – This was designed specifically for working with components such as media components, files, or even objects on a particular hardware device. Any web service that is defined on the principles of REST can be called a RestFul web service. A Restful service would use the normal HTTP verbs of GET, POST, PUT and DELETE for working with the required components.

Below are the main differences between SOAP and REST

|  |  |
| --- | --- |
| **SOAP** | **REST** |
| * SOAP stands for Simple Object Access Protocol | * REST stands for Representational State Transfer |
| * SOAP is a protocol. SOAP was designed with a specification. It includes a WSDL file which has the required information on what the web service does in addition to the location of the web service. | * REST is an Architectural style in which a web service can only be treated as a RESTful service if it follows the constraints of being   1. Client Server   2. Stateless   3. Cacheable   4. Layered System   5. Uniform Interface |
| * SOAP cannot make use of REST since SOAP is a protocol and REST is an architectural pattern. | * REST can make use of SOAP as the underlying protocol for web services, because in the end it is just an architectural pattern. |
| * SOAP uses service interfaces to expose its functionality to client applications. In SOAP, the WSDL file provides the client with the necessary information which can be used to understand what services the web service can offer. | * REST use Uniform Service locators to access to the components on the hardware device. For example, if there is an object which represents the data of an employee hosted on a URL as http://demo.guru99 , the below are some of URI that can exist to access them   http://demo.guru99.com/Employee  http://demo.guru99.com/Employee/1 |
| * SOAP requires more bandwidth for its usage. Since SOAP Messages contain a lot of information inside of it, the amount of data transfer using SOAP is generally a lot.   <?xml version="1.0"?>  <SOAP-ENV:Envelope  xmlns:SOAP-ENV  ="http://www.w3.org/2001/12/soap-envelope"  SOAP-ENV:encodingStyle  =" http://www.w3.org/2001/12/soap-encoding">  <soap:Body>  <Demo.guru99WebService  xmlns="http://tempuri.org/">  <EmployeeID>int</EmployeeID>  </Demo.guru99WebService>  </soap:Body>  </SOAP-ENV:Envelope> | * REST does not need much bandwidth when requests are sent to the server. REST messages mostly just consist of JSON messages. Below is an example of a JSON message passed to a web server. You can see that the size of the message is comparatively smaller to SOAP.   {"city":"Mumbai","state":"Maharastra"} |
| * SOAP can only work with XML format. As seen from SOAP messages, all data passed is in XML format. | * REST permits different data format such as Plain text, HTML, XML, JSON, etc. But the most preferred format for transferring data is JSON. |

## When to use REST and when to use SOAP

One of the most highly debatable topics is when REST should be used or when to use SOAP while designing web services.

Below are some of the key factors that determine when each technology should be used for web services **REST services should be used in the following instances**

* **Limited resources and bandwidth** – Since SOAP messages are heavier in content and consume a far greater bandwidth, REST should be used in instances where network bandwidth is a constraint.
* **Statelessness** – If there is no need to maintain a state of information from one request to another then REST should be used. If you need a proper information flow wherein some information from one request needs to flow into another then SOAP is more suited for that purpose. We can take the example of any online purchasing site. These sites normally need the user first to add items which need to be purchased to a cart. All of the cart items are then transferred to the payment page in order to complete the purchase. This is an example of an application which needs the state feature. The state of the cart items needs to be transferred to the payment page for further processing.
* **Caching**– If there is a need to cache a lot of requests then REST is the perfect solution. At times, clients could request for the same resource multiple times. This can increase the number of requests which are sent to the server. By implementing a cache, the most frequent queries results can be stored in an intermediate location. So whenever the client requests for a resource, it will first check the cache. If the resources exist then, it will not proceed to the server. So caching can help in minimizing the amount of trips which are made to the web server.
* **Ease of coding**– Coding REST Services and subsequent implementation is far easier than SOAP. So if a quick win solution is required for web services, then REST is the way to go.

SOAP should be used in the following instances

1. **Asynchronous processing and subsequent invocation** – if there is a requirement that the client needs a guaranteed level of reliability and security then the new SOAP standard of SOAP 1.2 provides a lot of additional features, especially when it comes to security.
2. **A Formal means of communication** – if both the client and server have an agreement on the exchange format then SOAP 1.2 gives the rigid specifications for this type of interaction. An example is an online purchasing site in which users add items to a cart before the payment is made. Let's assume we have a web service that does the final payment. There can be a firm agreement that the web service will only accept the cart item name, unit price, and quantity. If such a scenario exists then, it's always better to use the SOAP protocol.
3. **Stateful operations –**ifthe application has a requirement that state needs to be maintained from one request to another, then the SOAP 1.2 standard provides the WS\* structure to support such requirements.

## SOAP vs. REST API challenges

API is known as the **Application Programming Interface** and is offered by both the client and the server. In the client world, this is offered by the browser whereas in the server world it's what is provided by the web service which can either be SOAP or REST.

**Challenges with the SOAP API**

1. WSDL file - One of the key challenges of the SOAP API is the WSDL document itself. The WSDL document is what tells the client of all the operations that can be performed by the web service. The WSDL document will contain all information such as the data types being used in the SOAP messages and what all operations are available via the web service. The below code snippet is just part of a sample WSDL file.

<?xml version="1.0"?>

<definitions name="Tutorial"

targetNamespace=http://demo.guru99.com/Tutorial.wsdl

xmlns:tns=http://demo.guru99.com/Tutorial.wsdl

xmlns:xsd1=http://demo.guru99.com/Tutorial.xsd

xmlns:soap=http://schemas.xmlsoap.org/wsdl/soap/

xmlns="http://schemas.xmlsoap.org/wsdl/">

<types>

<schema targetNamespace=http://Demo.guru99.com/Tutorial.xsd

xmlns="http://www.w3.org/2000/10/XMLSchema">

<element name="TutorialNameRequest">

<complexType>

<all>

<element name="TutorialName" type="string"/>

</all>

</complexType>

</element>

<element name="TutorialIDRequest">

<complexType>

<all>

<element name="TutorialID" type="number"/>

</all>

</complexType>

</element>

</schema>

</types>

As per the above WSDL file, we have an element called "TutorialName" which is of the type String which is part of the element TutorialNameRequest.

Now, suppose if the WSDL file were to change as per the business requirements and the TutorialName has to become TutorialDescription. This would mean that all the clients who are currently connecting to this web service would then need to make this corresponding change in their code to accommodate the change in the WSDL file.

This shows the biggest challenge of the WSDL file which is the tight contract between the client and the server and that one change could cause a large impact, on the whole, client applications.

1. Document size – The other key challenge is the size of the SOAP messages which get transferred from the client to the server. Because of the large messages, using SOAP in places where bandwidth is a constraint can be a big issue.

**Challenges with the REST API**

1. **Lack of Security**– REST does not impose any sort of security like SOAP. This is why REST is very appropriate for public available URL's, but when it comes down to confidential data being passed between the client and the server, REST is the worst mechanism to be used for web services.
2. **Lack of state**– Most web applications require a stateful mechanism. For example, if you had a purchasing site which had the mechanism of having a shopping cart, it is required to know the number of items in the shopping cart before the actual purchase is made. Unfortunately, the burden of maintaining this state lies with the client, which just makes the client application heavier and difficult to maintain.

## Difference between SOAP Vs CORBA Vs DCOM Vs Java RMI

Remote access techniques such as the RPC (Remote Procedure calls) methods were in common use before SOAP and REST came along. The various remote access techniques which were available are mentioned below.

1. **CORBA**– This was known as **C**ommon **O**bject **R**equest **B**roker **A**rchitecture. This system was put in place to ensure that applications built on various platforms could talk to each other. CORBA was based on an object-oriented architecture, but it was not necessary for the calling application to be based on this architecture. The major disadvantage of this technique was that it has to be developed in a separate language called the Interface Definition Language, and it just presented an additional language that had to be learned by developers to make use of the CORBA system.
2. **DCOM** – This is the **D**istributed **C**omponent **O**bject **M**odel, which is a proprietary Microsoft technology for clients to access remote components. The biggest issue with this mechanism was it was up to the client application to free up resources when no longer required.

Secondly, when the client sent the request, it was up to the client to ensure that the request was wrapped or marshaled in a correct way so that the web service could understand the request sent. Another issue was if the client application was a[Java](https://www.guru99.com/java-tutorial.html)based application which had to work DCOM (Microsoft Technology) additional coding was required to ensure that applications built in other programming languages could work with DCOM based web services.

1. **Java RMI** – Known as Java **R**emote **M**ethod **I**nvocation, this was Java implementation on how remote objects could be called through remote procedure calls. The biggest restriction of this technology was that Java RMI could only be run on a Java Virtual Machine. This meant that the calling application also has to be run on the Java framework in order to make use of Java RMI.

The main differences between SOAP and these techniques are as follows

1. **Working over HTTP** – All of the RPC techniques have one big limitation, and it is that they don't work by the HTTP protocol. Since all applications on the web had to work on this protocol, this used to be a major roadblock for clients which had to access these RPC-style web services.
2. **Working with non-standard ports** – Since the RPC style web services did not work by the HTTP protocol, separate ports had to be open for them for clients to access the functionality from these web services.

**Summary**

* One of the key differences between SOAP and REST is that SOAP is a protocol and REST is an architectural pattern.
* Other key differences between the SOAP and REST protocol is that the requests sent via REST tend to be much lighter than SOAP. Because of this, applications don't require much bandwidth to use REST web services over SOAP.
* Security is another major concern with Web services and SOAP. REST is good when working with web services open to the public, but if security is required, then the SOAP API has the necessary implementation for the same.
* REST has the ability to have a caching solution which will help save responses which have been received from the server. In such cases, the client does not need to make the same request to the server and can make use of the cache to get the desired response.

### What are different ways to test web services?

SOAP web services can be tested programmatically by generating client stubs from WSDL or through software such as Soap UI.

REST web services can be tested easily with program, curl commands and through browser extensions. Resources supporting GET method can be tested with browser itself, without any program.

### Can we maintain user session in web services?

Web services are stateless so we can’t maintain user sessions in web services.

### What is difference between SOA and Web Services?

Service Oriented Architecture (SOA) is an architectural pattern where applications are designed in terms of services that can be accessed through communication protocol over network. SOA is a [design pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial) and doesn’t go into implementation.

Web Services can be thought of as Services in SOAP architecture and providing means to implement SOA pattern.

### What is the use of Accept and Content-Type Headers in HTTP Request?

These are important headers in Restful web services. Accept headers tells web service what kind of response client is accepting, so if a web service is capable of sending response in XML and JSON format and client sends Accept header as “application/xml” then XML response will be sent. For Accept header “application/json”, server will send the JSON response.

Content-Type header is used to tell server what is the format of data being sent in the request. If Content-Type header is “application/xml” then server will try to parse it as XML data. This header is useful in HTTP Post and Put requests.

### 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.

### What is JAX-WS API?

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. It’s part of standard Java API, so we don’t need to include anything else which working with it. Refer to [JAX-WS Tutorial](https://www.journaldev.com/9123/jax-ws-tutorial) for a complete example.

### Name some frameworks in Java to implement SOAP web services?

We can create SOAP web services using JAX-WS API, however some of the other frameworks that can be used are Apache Axis and Apache CXF. Note that they are not implementations of JAX-WS API, they are totally different framework that work on Servlet model to expose your business logic classes as SOAP web services. Read more at [Java SOAP Web Service Eclipse](https://www.journaldev.com/9131/soap-webservices-in-java-example-eclipse) example.

### Name important annotations used in JAX-WS API?

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

* + @WebService
  + @SOAPBinding
  + @WebMethod

### What is use of javax.xml.ws.Endpoint class?

Endpoint class provides useful methods to create endpoint and publish existing implementation as web service. This comes handy in testing web services before making further changes to deploy it on actual server.

### What is the difference between RPC Style and Document Style SOAP web Services?

RPC style generate WSDL document based on the method name and it’s parameters. No type definitions are present in WSDL document.  
Document style contains type and can be validated against predefined schema. Let’s look at these with a simple program. Below is a simple test program where I am using Endpoint to publish my simple SOAP web service.

TestService.java

package com.journaldev.jaxws.service;

import javax.jws.WebMethod;

import javax.jws.WebService;

import javax.jws.soap.SOAPBinding;

import javax.xml.ws.Endpoint;

@WebService

@SOAPBinding(style = SOAPBinding.Style.RPC)

public class TestService {

@WebMethod

public String sayHello(String msg){

return "Hello "+msg;

}

public static void main(String[] args){

Endpoint.publish("http://localhost:8888/testWS", new TestService());

}

}

When I run above program and then access the WSDL, it gives me below XML.

rpc.xml

<?xml version='1.0' encoding='UTF-8'?>

<!-- Published by JAX-WS RI (http://jax-ws.java.net). RI's version is JAX-WS RI 2.2.10 svn-revision#919b322c92f13ad085a933e8dd6dd35d4947364b. --><!-- Generated by JAX-WS RI (http://jax-ws.java.net). RI's version is JAX-WS RI 2.2.10 svn-revision#919b322c92f13ad085a933e8dd6dd35d4947364b. -->

<definitions xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" xmlns:wsp="http://www.w3.org/ns/ws-policy" xmlns:wsp1\_2="http://schemas.xmlsoap.org/ws/2004/09/policy" xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://service.jaxws.journaldev.com/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace="http://service.jaxws.journaldev.com/" name="TestServiceService">

<types/>

<message name="sayHello">

<part name="arg0" type="xsd:string"/>

</message>

<message name="sayHelloResponse">

<part name="return" type="xsd:string"/>

</message>

<portType name="TestService">

<operation name="sayHello">

<input wsam:Action="http://service.jaxws.journaldev.com/TestService/sayHelloRequest" message="tns:sayHello"/>

<output wsam:Action="http://service.jaxws.journaldev.com/TestService/sayHelloResponse" message="tns:sayHelloResponse"/>

</operation>

</portType>

<binding name="TestServicePortBinding" type="tns:TestService">

<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>

<operation name="sayHello">

<soap:operation soapAction=""/>

<input>

<soap:body use="literal" namespace="http://service.jaxws.journaldev.com/"/>

</input>

<output>

<soap:body use="literal" namespace="http://service.jaxws.journaldev.com/"/>

</output>

</operation>

</binding>

<service name="TestServiceService">

<port name="TestServicePort" binding="tns:TestServicePortBinding">

<soap:address location="http://localhost:8888/testWS"/>

</port>

</service>

</definitions>

Notice that **types** element is empty and we can’t validate it against any schema. Now just change the SOAPBinding.Style.RPC to SOAPBinding.Style.DOCUMENT and you will get below WSDL.

document.xml

<?xml version='1.0' encoding='UTF-8'?>

<!-- Published by JAX-WS RI (http://jax-ws.java.net). RI's version is JAX-WS RI 2.2.10 svn-revision#919b322c92f13ad085a933e8dd6dd35d4947364b. --><!-- Generated by JAX-WS RI (http://jax-ws.java.net). RI's version is JAX-WS RI 2.2.10 svn-revision#919b322c92f13ad085a933e8dd6dd35d4947364b. -->

<definitions xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" xmlns:wsp="http://www.w3.org/ns/ws-policy" xmlns:wsp1\_2="http://schemas.xmlsoap.org/ws/2004/09/policy" xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://service.jaxws.journaldev.com/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace="http://service.jaxws.journaldev.com/" name="TestServiceService">

<types>

<xsd:schema>

<xsd:import namespace="http://service.jaxws.journaldev.com/" schemaLocation="http://localhost:8888/testWS?xsd=1"/>

</xsd:schema>

</types>

<message name="sayHello">

<part name="parameters" element="tns:sayHello"/>

</message>

<message name="sayHelloResponse">

<part name="parameters" element="tns:sayHelloResponse"/>

</message>

<portType name="TestService">

<operation name="sayHello">

<input wsam:Action="http://service.jaxws.journaldev.com/TestService/sayHelloRequest" message="tns:sayHello"/>

<output wsam:Action="http://service.jaxws.journaldev.com/TestService/sayHelloResponse" message="tns:sayHelloResponse"/>

</operation>

</portType>

<binding name="TestServicePortBinding" type="tns:TestService">

<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>

<operation name="sayHello">

<soap:operation soapAction=""/>

<input>

<soap:body use="literal"/>

</input>

<output>

<soap:body use="literal"/>

</output>

</operation>

</binding>

<service name="TestServiceService">

<port name="TestServicePort" binding="tns:TestServicePortBinding">

<soap:address location="http://localhost:8888/testWS"/>

</port>

</service>

</definitions>

Open schemaLocation URL in browser and you will get below XML.

schemaLocation.xml

<?xml version='1.0' encoding='UTF-8'?>

<!-- Published by JAX-WS RI (http://jax-ws.java.net). RI's version is JAX-WS RI 2.2.10 svn-revision#919b322c92f13ad085a933e8dd6dd35d4947364b. -->

<xs:schema xmlns:tns="http://service.jaxws.journaldev.com/" xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0" targetNamespace="http://service.jaxws.journaldev.com/">

<xs:element name="sayHello" type="tns:sayHello"/>

<xs:element name="sayHelloResponse" type="tns:sayHelloResponse"/>

<xs:complexType name="sayHello">

<xs:sequence>

<xs:element name="arg0" type="xs:string" minOccurs="0"/>

</xs:sequence>

</xs:complexType>

<xs:complexType name="sayHelloResponse">

<xs:sequence>

<xs:element name="return" type="xs:string" minOccurs="0"/>

</xs:sequence>

</xs:complexType>

</xs:schema>

So here WSDL document can be validated against the schema definintion.

### How to get WSDL file of a SOAP web service?

WSDL document can be accessed by appending ?wsdl to the SOAP endoint URL. In above example, we can access it at http://localhost:8888/testWS?wsdl location.

### What is sun-jaxws.xml file?

This file is used to provide endpoints details when JAX-WS web services are deployed in servlet container such as Tomcat. This file is present in WEB-INF directory and contains endpoint name, implementation class and URL pattern. For example;

sun-jaxws.xml

<?xml version="1.0" encoding="UTF-8"?>

<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">

<endpoint

name="PersonServiceImpl"

implementation="com.journaldev.jaxws.service.PersonServiceImpl"

url-pattern="/personWS"/>

</endpoints>

### What is JAX-RS 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.

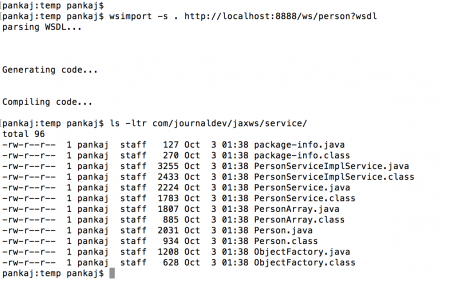
### Name some implementations of JAX-RS API?

There are two major implementations of JAX-RS API.

* + 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.
  + RESTEasy: RESTEasy is the JBoss project that provides JAX-RS implementation.

### What is wsimport utility?

We can use wsimport utility to generate the client stubs. This utility comes with standard installation of JDK. Below image shows an example execution of this utility for one of JAX-WS project.

[](https://cdn.journaldev.com/wp-content/uploads/2015/10/wsimport-utility-parse-wsdl.png)

### Name important annotations used in JAX-RS API?

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.

### What is the use of @XmlRootElement annotation?

XmlRootElement annotation is used by JAXB to transform java object to XML and vice versa. So we have to annotate model classes with this annotation.

### How to set different status code in HTTP response?

For setting HTTP status code other than 200, we have to use javax.ws.rs.core.Response class for response. Below are some of the sample return statements showing it’s usage.

return Response.status(422).entity(exception).build();

return Response.ok(response).build(); //200

# HTTP Methods

RESTful APIs enable you to develop any kind of web application having all possible CRUD (create, retrieve, update, delete) operations. REST guidelines suggest using a specific HTTP method on a specific type of call made to the server (though technically it is possible to violate this guideline, yet it is highly discouraged).

Use below-given information to find suitable HTTP method for the action performed by API.

Table of Contents

[HTTP GET](https://restfulapi.net/http-methods/#get)

[HTTP POST](https://restfulapi.net/http-methods/#post)

[HTTP PUT](https://restfulapi.net/http-methods/#put)

[HTTP DELETE](https://restfulapi.net/http-methods/#delete)

[HTTP PATCH](https://restfulapi.net/http-methods/#patch)

[Summary](https://restfulapi.net/http-methods/#summary)

[Glossary](https://restfulapi.net/http-methods/#glossary)

## HTTP GET

Use GET requests **to retrieve resource representation/information only** – and not to modify it in any way. As GET requests do not change the state of the resource, these are said to be **safe methods**. Additionally, GET APIs should be **idempotent**, which means that making multiple identical requests must produce the same result every time until another API (POST or PUT) has changed the state of the resource on the server.

If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

For any given HTTP GET API, if the resource is found on the server then it must return HTTP response code 200 (OK) – along with response body which is usually either XML or JSON content (due to their platform independent nature).

In case resource is NOT found on server then it must return HTTP response code 404 (NOT FOUND). Similarly, if it is determined that GET request itself is not correctly formed then server will return HTTP response code 400 (BAD REQUEST).

#### Example request URIs

* HTTP GET http://www.appdomain.com/users
* HTTP GET http://www.appdomain.com/users?size=20&page=5
* HTTP GET http://www.appdomain.com/users/123
* HTTP GET http://www.appdomain.com/users/123/address

## HTTP POST

Use POST APIs **to create new subordinate resources**, e.g. a file is subordinate to a directory containing it or a row is subordinate to a database table. Talking strictly in terms of REST, POST methods are used to create a new resource into the collection of resources.

Ideally, if a resource has been created on the origin server, the response SHOULD be HTTP response code 201 (Created) and contain an entity which describes the status of the request and refers to the new resource, and a [Location](https://en.wikipedia.org/wiki/HTTP_location) header.

Many times, the action performed by the POST method might not result in a resource that can be identified by a URI. In this case, either HTTP response code 200 (OK) or 204 (No Content) is the appropriate response status.

Responses to this method are **not cacheable**, unless the response includes appropriate [Cache-Control](https://en.wikipedia.org/wiki/Web_cache#Cache_control)or [Expires](https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html) header fields.

Please note that POST is **neither safe nor idempotent** and invoking two identical POST requests will result in two different resources containing the same information (except resource ids).

#### Example request URIs

* HTTP POST http://www.appdomain.com/users
* HTTP POST http://www.appdomain.com/users/123/accounts

## HTTP PUT

Use PUT APIs primarily **to update existing resource** (if the resource does not exist then API may decide to create a new resource or not). If a new resource has been created by the PUT API, the origin server MUST inform the user agent via the HTTP response code 201 (Created) response and if an existing resource is modified, either the 200 (OK) or 204 (No Content) response codes SHOULD be sent to indicate successful completion of the request.

If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries SHOULD be treated as stale. Responses to this method are **not cacheable**.

*The difference between the POST and PUT APIs can be observed in request URIs. POST requests are made on resource collections whereas PUT requests are made on an individual resource.*

#### Example request URIs

* HTTP PUT http://www.appdomain.com/users/123
* HTTP PUT http://www.appdomain.com/users/123/accounts/456

## HTTP DELETE

As the name applies, DELETE APIs are used **to delete resources** (identified by the Request-URI).

A successful response of DELETE requests SHOULD be HTTP response code 200 (OK) if the response includes an entity describing the status, 202 (Accepted) if the action has been queued, or 204 (No Content) if the action has been performed but the response does not include an entity.

DELETE operations are **idempotent**. If you DELETE a resource, it’s removed from the collection of resource. Repeatedly calling DELETE API on that resource will not change the outcome – however calling DELETE on a resource a second time will return a 404 (NOT FOUND) since it was already removed. Some may argue that it makes DELETE method non-idempotent. It’s a matter of discussion and personal opinion.

If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries SHOULD be treated as stale. Responses to this method are **not cacheable**.

#### Example request URIs

* HTTP DELETE http://www.appdomain.com/users/123
* HTTP DELETE http://www.appdomain.com/users/123/accounts/456

## HTTP PATCH

HTTP PATCH requests are **to make partial update on a resource**. If you see PUT requests also modify a resource entity so to make more clear – PATCH method is the correct choice for partially updating an existing resource and PUT should only be used if you’re replacing a resource in its entirety.

Please note that there are some challenges if you decide to use PATCH APIs in your application:

* Support for PATCH in browsers, servers, and web application frameworks is not universal. IE8, PHP, Tomcat, Django, and lots of other software has missing or broken support for it.
* Request payload of PATCH request is not straightforward as it is for PUT request. e.g.

HTTP GET /users/1

produces below response:

{id: 1, username: 'admin', email: 'email@example.org'}

A sample patch request to update the email will be like this:

HTTP PATCH /users/1

[  
{ “op”: “replace”, “path”: “/email”, “value”: “new.email@example.org” }  
]

There may be following possible operations are per HTTP specification.

[  
{ "op": "test", "path": "/a/b/c", "value": "foo" },  
{ "op": "remove", "path": "/a/b/c" },  
{ "op": "add", "path": "/a/b/c", "value": [ "foo", "bar" ] },  
{ "op": "replace", "path": "/a/b/c", "value": 42 },  
{ "op": "move", "from": "/a/b/c", "path": "/a/b/d" },  
{ "op": "copy", "from": "/a/b/d", "path": "/a/b/e" }  
]

PATCH method is not a replacement for the POST or PUT methods. It applies a delta (diff) rather than replacing the entire resource.

## Summary of HTTP Methods for RESTful APIs

Below table summarises the use of HTTP methods discussed above.

| **HTTP METHOD** | **CRUD** | **ENTIRE COLLECTION (E.G. /USERS)** | **SPECIFIC ITEM (E.G. /USERS/123)** |
| --- | --- | --- | --- |
| POST | Create | 201 (Created), ‘Location’ header with link to /users/{id} containing new ID. | Avoid using POST on single resource |
| GET | Read | 200 (OK), list of users. Use pagination, sorting and filtering to navigate big lists. | 200 (OK), single user. 404 (Not Found), if ID not found or invalid. |
| PUT | Update/Replace | 404 (Not Found), unless you want to update every resource in the entire collection of resource. | 200 (OK) or 204 (No Content). Use 404 (Not Found), if ID not found or invalid. |
| PATCH | Partial Update/Modify | 404 (Not Found), unless you want to modify the collection itself. | 200 (OK) or 204 (No Content). Use 404 (Not Found), if ID not found or invalid. |
| DELETE | Delete | 404 (Not Found), unless you want to delete the whole collection — use with caution. | 200 (OK). 404 (Not Found), if ID not found or invalid. |

## Glossary

# HTTP Status Codes

This page is created from HTTP status code information found at [ietf.org](https://www.ietf.org/assignments/http-status-codes/http-status-codes.xml) and [Wikipedia](https://en.wikipedia.org/wiki/HTTP_status_code). Click on the **category heading** or the **status code** link to read more.

## 1xx Informational

100 Continue

[101 Switching Protocols](https://www.restapitutorial.com/httpstatuscodes.html)

[102 Processing (WebDAV)](https://www.restapitutorial.com/httpstatuscodes.html)

## 2xx Success

 200 OK

 201 Created

202 Accepted

203 Non-Authoritative Information

 204 No Content

205 Reset Content

206 Partial Content

207 Multi-Status (WebDAV)

208 Already Reported (WebDAV)

226 IM Used

## 3xx Redirection

300 Multiple Choices

301 Moved Permanently

302 Found

303 See Other

 304 Not Modified

305 Use Proxy

306 (Unused)

307 Temporary Redirect

308 Permanent Redirect (experimental)

## [4xx Client Error](https://www.restapitutorial.com/httpstatuscodes.html)

 400 Bad Request

 401 Unauthorized

402 Payment Required

 403 Forbidden

 404 Not Found

405 Method Not Allowed

406 Not Acceptable

407 Proxy Authentication Required

408 Request Timeout

 409 Conflict

410 Gone

411 Length Required

412 Precondition Failed

413 Request Entity Too Large

414 Request-URI Too Long

415 Unsupported Media Type

416 Requested Range Not Satisfiable

417 Expectation Failed

418 I'm a teapot (RFC 2324)

420 Enhance Your Calm (Twitter)

422 Unprocessable Entity (WebDAV)

423 Locked (WebDAV)

424 Failed Dependency (WebDAV)

425 Reserved for WebDAV

426 Upgrade Required

428 Precondition Required

429 Too Many Requests

431 Request Header Fields Too Large

444 No Response (Nginx)

449 Retry With (Microsoft)

450 Blocked by Windows Parental Controls (Microsoft)

451 Unavailable For Legal Reasons

499 Client Closed Request (Nginx)

## [5xx Server Error](https://www.restapitutorial.com/httpstatuscodes.html)

 500 Internal Server Error

501 Not Implemented

502 Bad Gateway

503 Service Unavailable

504 Gateway Timeout

505 HTTP Version Not Supported

506 Variant Also Negotiates (Experimental)

507 Insufficient Storage (WebDAV)

508 Loop Detected (WebDAV)

509 Bandwidth Limit Exceeded (Apache)

510 Not Extended

511 Network Authentication Required

598 Network read timeout error

599 Network connect timeout error

# HTTP Status Codes

REST APIs use the **Status-Line** part of an HTTP response message to inform clients of their request’s overarching result. [RFC 2616](https://www.ietf.org/rfc/rfc2616.txt) defines the [Status-Line syntax](https://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html#sec6.1) as shown below:

Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF

HTTP defines forty standard status codes that can be used to convey the results of a client’s request. The status codes are divided into the five categories presented below.

|  |  |
| --- | --- |
| **CATEGORY** | **DESCRIPTION** |
| **1xx: Informational** | Communicates transfer protocol-level information. |
| **2xx: Success** | Indicates that the client’s request was accepted successfully. |
| **3xx: Redirection** | Indicates that the client must take some additional action in order to complete their request. |
| **4xx: Client Error** | This category of error status codes points the finger at clients. |
| **5xx: Server Error** | The server takes responsibility for these error status codes. |

Now look at subset of codes that specially apply to the design of a REST APIs – in some more detail.

#### [200 (OK)](https://restfulapi.net/http-status-200-ok/" \t "_blank)

It indicates that the REST API successfully carried out whatever action the client requested, and that no more specific code in the 2xx series is appropriate.

Unlike the 204 status code, a 200 response should include a response body.The information returned with the response is dependent on the method used in the request, for example:

* GET an entity corresponding to the requested resource is sent in the response;
* HEAD the entity-header fields corresponding to the requested resource are sent in the response without any message-body;
* POST an entity describing or containing the result of the action;
* TRACE an entity containing the request message as received by the end server.

#### [201 (Created)](https://restfulapi.net/http-status-201-created/" \t "_blank)

A REST API responds with the 201 status code whenever a resource is created inside a collection. There may also be times when a new resource is created as a result of some controller action, in which case 201 would also be an appropriate response.

The newly created resource can be referenced by the URI(s) returned in the entity of the response, with the most specific URI for the resource given by a Location header field.

The origin server MUST create the resource before returning the 201 status code. If the action cannot be carried out immediately, the server SHOULD respond with 202 (Accepted) response instead.

#### [202 (Accepted)](https://restfulapi.net/http-status-202-accepted/" \t "_blank)

A 202 response is typically used for actions that take a long while to process. It indicates that the request has been accepted for processing, but the processing has not been completed. The request might or might not be eventually acted upon, or even maybe disallowed when processing occurs.

Its purpose is to allow a server to accept a request for some other process (perhaps a batch-oriented process that is only run once per day) without requiring that the user agent’s connection to the server persist until the process is completed.

The entity returned with this response SHOULD include an indication of the request’s current status and either a pointer to a status monitor (job queue location) or some estimate of when the user can expect the request to be fulfilled.

#### [204 (No Content)](https://restfulapi.net/http-status-204-no-content/" \t "_blank)

The 204 status code is usually sent out in response to a PUT, POST, or DELETE request when the REST API declines to send back any status message or representation in the response message’s body.

An API may also send 204 in conjunction with a GET request to indicate that the requested resource exists, but has no state representation to include in the body.

If the client is a user agent, it SHOULD NOT change its document view from that which caused the request to be sent. This response is primarily intended to allow input for actions to take place without causing a change to the user agent’s active document view, although any new or updated metainformation SHOULD be applied to the document currently in the user agent’s active view.

The 204 response MUST NOT include a message-body and thus is always terminated by the first empty line after the header fields.

#### 301 (Moved Permanently)

The 301 status code indicates that the REST API’s resource model has been significantly redesigned and a new permanent URI has been assigned to the client’s requested resource. The REST API should specify the new URI in the response’s Location header and all future requests should be directed to the given URI.

You will hardly use this response code in your API as you can always use the API versioning for new API while retaining the old one.

#### 302 (Found)

The HTTP response status code 302 Found is a common way of performing URL redirection. An HTTP response with this status code will additionally provide a URL in the location header field. The user agent (e.g. a web browser) is invited by a response with this code to make a second, otherwise identical, request to the new URL specified in the location field.

Many web browsers implemented this code in a manner that violated this standard, changing the request type of the new request to GET, regardless of the type employed in the original request (e.g. POST). RFC 1945 and RFC 2068 specify that the client is not allowed to change the method on the redirected request. The status codes 303 and 307 have been added for servers that wish to make unambiguously clear which kind of reaction is expected of the client.

#### 303 (See Other)

A 303 response indicates that a controller resource has finished its work, but instead of sending a potentially unwanted response body, it sends the client the URI of a response resource. This can be the URI of a temporary status message, or the URI to some already existing, more permanent, resource.

Generally speaking, the 303 status code allows a REST API to send a reference to a resource without forcing the client to download its state. Instead, the client may send a GET request to the value of the Location header.

The 303 response MUST NOT be cached, but the response to the second (redirected) request might be cacheable.

#### 304 (Not Modified)

This status code is similar to 204 (“No Content”) in that the response body must be empty. The key distinction is that 204 is used when there is nothing to send in the body, whereas 304 is used when the resource has not been modified since the version specified by the request headers If-Modified-Since or If-None-Match.

In such case, there is no need to retransmit the resource since the client still has a previously-downloaded copy.

Using this saves bandwidth and reprocessing on both the server and client, as only the header data must be sent and received in comparison to the entirety of the page being re-processed by the server, then sent again using more bandwidth of the server and client.

#### 307 (Temporary Redirect)

A 307 response indicates that the REST API is not going to process the client’s request. Instead, the client should resubmit the request to the URI specified by the response message’s Location header. However, future requests should still use the original URI.

A REST API can use this status code to assign a temporary URI to the client’s requested resource. For example, a 307 response can be used to shift a client request over to another host.

The temporary URI SHOULD be given by the Location field in the response. Unless the request method was HEAD, the entity of the response SHOULD contain a short hypertext note with a hyperlink to the new URI(s). If the 307 status code is received in response to a request other than GET or HEAD, the user agent MUST NOT automatically redirect the request unless it can be confirmed by the user, since this might change the conditions under which the request was issued.

#### 400 (Bad Request)

400 is the generic client-side error status, used when no other 4xx error code is appropriate. Errors can be like malformed request syntax, invalid request message parameters, or deceptive request routing etc.

The client SHOULD NOT repeat the request without modifications.

#### 401 (Unauthorized)

A 401 error response indicates that the client tried to operate on a protected resource without providing the proper authorization. It may have provided the wrong credentials or none at all. The response must include a WWW-Authenticate header field containing a challenge applicable to the requested resource.

The client MAY repeat the request with a suitable Authorization header field. If the request already included Authorization credentials, then the 401 response indicates that authorization has been refused for those credentials. If the 401 response contains the same challenge as the prior response, and the user agent has already attempted authentication at least once, then the user SHOULD be presented the entity that was given in the response, since that entity might include relevant diagnostic information.

#### 403 (Forbidden)

A 403 error response indicates that the client’s request is formed correctly, but the REST API refuses to honor it i.e. the user does not have the necessary permissions for the resource. A 403 response is not a case of insufficient client credentials; that would be 401 (“Unauthorized”).

Authentication will not help and the request SHOULD NOT be repeated. Unlike a 401 Unauthorized response, authenticating will make no difference.

#### 404 (Not Found)

The 404 error status code indicates that the REST API can’t map the client’s URI to a resource but may be available in the future. Subsequent requests by the client are permissible.

No indication is given of whether the condition is temporary or permanent. The 410 (Gone) status code SHOULD be used if the server knows, through some internally configurable mechanism, that an old resource is permanently unavailable and has no forwarding address. This status code is commonly used when the server does not wish to reveal exactly why the request has been refused, or when no other response is applicable.

#### 405 (Method Not Allowed)

The API responds with a 405 error to indicate that the client tried to use an HTTP method that the resource does not allow. For instance, a read-only resource could support only GET and HEAD, while a controller resource might allow GET and POST, but not PUT or DELETE.

A 405 response must include the Allow header, which lists the HTTP methods that the resource supports. For example:

Allow: GET, POST

#### 406 (Not Acceptable)

The 406 error response indicates that the API is not able to generate any of the client’s preferred media types, as indicated by the Accept request header. For example, a client request for data formatted as application/xml will receive a 406 response if the API is only willing to format data as application/json.

If the response could be unacceptable, a user agent SHOULD temporarily stop receipt of more data and query the user for a decision on further actions.

#### 412 (Precondition Failed)

The 412 error response indicates that the client specified one or more preconditions in its request headers, effectively telling the REST API to carry out its request only if certain conditions were met. A 412 response indicates that those conditions were not met, so instead of carrying out the request, the API sends this status code.

#### 415 (Unsupported Media Type)

The 415 error response indicates that the API is not able to process the client’s supplied media type, as indicated by the Content-Type request header. For example, a client request including data formatted as application/xml will receive a 415 response if the API is only willing to process data formatted as application/json.

For example, the client uploads an image as image/svg+xml, but the server requires that images use a different format.

#### 500 (Internal Server Error)

500 is the generic REST API error response. Most web frameworks automatically respond with this response status code whenever they execute some request handler code that raises an exception.

A 500 error is never the client’s fault and therefore it is reasonable for the client to retry the exact same request that triggered this response, and hope to get a different response.

API response is the generic error message, given when an unexpected condition was encountered and no more specific message is suitable.

#### 501 (Not Implemented)

The server either does not recognize the request method, or it lacks the ability to fulfill the request. Usually, this implies future availability (e.g., a new feature of a web-service API).

### Safe Methods

As per HTTP specification, the **GET and HEAD methods should be used only for retrieval of resource representations** – and they do not update/delete the resource on the server. Both methods are said to be considered “**safe**“.

This allows user agents to represent other methods, such as **POST, PUT and DELETE**, in a special way, so that the user is made aware of the fact that a possibly unsafe action is being requested – and they can **update/delete the resource on server** and so should be used carefully.

### Idempotent Methods

The term idempotent is used more comprehensively to describe an **operation that will produce the same results if executed once or multiple times**. This is a very useful property in many situations, as it means that an operation can be repeated or retried as often as necessary without causing unintended effects. With non-idempotent operations, the algorithm may have to keep track of whether the operation was already performed or not.

In HTTP specification, The methods **GET, HEAD, PUT and DELETE are declared idempotent methods**. Other methods OPTIONS and TRACE SHOULD NOT have side effects so both are also inherently idempotent.

Important Questions and Anseres

**Question 1 : What is REST?**  
Answer : REST is an architectural style of developing web services which take advantage of ubiquity of HTTP protocol and leverages HTTP method to define actions. REST stands for *REpresntational State Transfer*.  
  
  
**Question 2 : What is RESTFul Web Service?**  
Answer : There are two popular way to develop web services, using SOAP (Simple Object Access Protocol) which is XML based way to expose web services and second REST based web services which uses HTTP protocol. Web services developed using REST style is also known as [RESTful Web Services](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https%3A%2F%2Fwww.udemy.com%2Frest-web-services-using-java-ee%2F).  
  
  
Question 3 : What is HTTP Basic Authentication and how it works?  
  
  
**Question 4 : Can you tell me which API can be used to develop RESTFul web service in Java?**  
Answer : There are many framework and libraries out there which helps to develop RESTful web services in Java including JAX-RS which is standard way to develop REST web services. Jersey is one of the popular implementation of JAX-RS which also offers more than specification recommends. Then you also have RESTEasy, RESTlet and Apache CFX. If you like Scala then you can also use Play framework to develop RESTful web services.  
  
  
Question 5 : How do you configure RESTFul web service?  
  
Question 6 : How you apply security in RESTFul web services?  
  
Question 7 : Have you used securing RESTful APIs with HTTP Basic Authentication  
  
Question 8 : How you maintain session in RESTful services?  
  
  
**Question 9 : Have you used Jersey API to develop RESTful services in Java?**  
Answer : Jersey is one of the most popular framework and API to develop REST based web services in Java. Since many organization uses Jersey they check if candidate has used it before or not. It's simple to answer, say Yes if you have really used and No, if you have not. In case of No, you should also mention which framework you have used for developing RESTful web services e.g. Apache CFX, Play or Restlet.  
  
  
Question 10 : How you test RESTful web services?  
  
Question 11 : What is WADL in RESTFul?  
  
  
**Question 12 : What do you understand by payload in RESTFul?**  
Answer : Payload means data which passed inside request body also payload is not request parameters. So only you can do payload in POST  and not in GET and DELTE method  
  
  
**Question 13 : Can you do payload in GET method?**  
Answer : No, payload can only be passed using POST method.  
  
  
**Question 14 : Can you do payload in HTTP DELETE?**  
Answer : This is again similar to previous REST interview question, answer is No. You can only pass payload using HTTP POST method.  
  
  
**Question 15 : How much maximum pay load you could do in POST method?**  
Answer : If you remember [difference between GET and POST request](http://java67.blogspot.sg/2014/08/difference-between-post-and-get-request.html) then you know that unlike GET which passes data on URL and thus limited by maximum URL length, POST has no such limit. So, theoretically you can pass unlimited data as payload to POST method but you need to take practical things into account e.g. sending POST with large payload will consume more bandwidth, take more time and present performance challenge to your server.  
  
  
**Question 16 : What is difference between SOAP and RESTFul web services?**  
Answer : There are many difference between these two style of web services e.g. SOAP take more bandwidth because of heavy weight XML based protocol but REST takes less bandwidth because of popular use of JSON as message protocol and leveraging HTTP method to define action. This also means that REST is faster than SOAP based web services. You can derive many differences between SOAP and RESTful with the fact that its HTTP based e.g. REST URLs can be cached or bookmarked. Here are few more differences between them :

[](https://pluralsight.pxf.io/c/1193463/424552/7490?u=https://www.pluralsight.com/courses/restful-services-java-using-jersey)

**Question 17 : If you have to develop web services which one you will choose SOAP OR RESTful and why?**  
Answer : You can answer this question based upon your experience but the key here is if you know difference between them than you can answer this question in more detail. For example, its easy to develop RESTful web services than SOAP based web services but later comes with some in-built security features.  
  
  
**Question 18 :  What framework you had used to develop RESTFul services?**  
Answer : This is really experience based question. If you have used Jersey to develop RESTFul web services then answer as Jersey but expect some follow-up question on Jersey. Similarly if you have used Apache CFX or Restlet then answer them accordingly.  
  
  
That's all in this list of some good **RESTful web service interview questions for Java developers**. Though this list is meant for Java developer, you can use this questions to check any candidate's knowledge on REST style web services independent of programming language because REST doesn't say that you need to implement web service in Java only. Since it take advantage of ubiquitous HTTP protocol you can build backed with any web technology stack e.g. Java, .NET or any other.  
  
Read more: <http://www.java67.com/2015/09/top-10-restful-web-service-interview-questions-answers.html#ixzz5WEs0QwMp>

interesting *tricky or tough Java web service question* asked in Interviews then please share with us :

## **Frequently asked Java webservice interview questions**

[Java Web Service Interview Question and Answers SOAP REST](http://www.shareasale.com/m-pr.cfm?merchantID=53701&userID=880419&productID=546412145)Here is my list of frequently asked interview question on Java web service in any [Core Java Interview](http://java67.blogspot.sg/2012/08/10-java-coding-interview-questions-and.html). As SOAP is an standard way of making web service call which uses XML, good knowledge of XML and Java is expected from you and Interviewer may ask some*XML interview question*s as well.

What is Web Service ?

What is SOAP ?

What is REST Web Service ?

What is difference between REST Web Service and SOAP web service ?

Can a Java client can talk to C++ Server using Web Service ?

What is WSDL?

What is UDDI?

Does Web Service call is synchronous or asynchronous ?

How do you handle errors in Web Service call ?

What is JAX-RPC ?

Have you worked on Spring and Web-services ?

What is WebServiceTemplate etc

What is difference between RMI and Web Services

Read more: <http://www.java67.com/2012/09/top-10-java-web-service-interview-question-answer-soap-rest.html#ixzz5WEs5dwqx>

What is Web Service ?

What is SOAP ?

What is REST Web Service ?

What is difference between REST Web Service and SOAP web service ?

Can a Java client can talk to C++ Server using Web Service ?

What is WSDL?

What is UDDI?

Does Web Service call is synchronous or asynchronous ?

How do you handle errors in Web Service call ?

What is JAX-RPC ?

Have you worked on Spring and Web-services ?

What is WebServiceTemplate etc

What is difference between RMI and Web Services

2. What are the advantages of Web Services?

3. What are different types of Web Services?

5. What are advantages of SOAP Web Services?

6. What are disadvantages of SOAP Web Services?

7. What is WSDL?

8. What are different components of WSDL?

9. What is UDDI?

10. What is difference between Top Down and Bottom Up approach in SOAP Web Services?

11. What is REST Web Services?

12. What are advantages of REST web services?

13. What are disadvantages of REST web services?

14. What is a Resource in Restful web services?

15. What are different HTTP Methods supported in Restful Web Services?

16. Compare SOAP and REST web services?

17. What are different ways to test web services?

18. Can we maintain user session in web services?

19. What is difference between SOA and Web Services?

20. What is the use of Accept and Content-Type Headers in HTTP Request?

21. How would you choose between SOAP and REST web services?

22. What is JAX-WS API?

23. Name some frameworks in Java to implement SOAP web services?

24. Name important annotations used in JAX-WS API?

25. What is use of javax.xml.ws.Endpoint class?

26. What is the difference between RPC Style and Document Style SOAP web Services?

27. How to get WSDL file of a SOAP web service?

28. What is sun-jaxws.xml file?

29. What is JAX-RS API?

30. Name some implementations of JAX-RS API?

31. What is wsimport utility?

32. Name important annotations used in JAX-RS API?

33. What is the use of @XmlRootElement annotation?

34. How to set different status code in HTTP response?