|  |
| --- |
| | Praveen D |

|  |  |
| --- | --- |
| JSpider Basavangudi Bangalore | Web Services Jersey Server API ( Producer) |

Table of Contents

[Representational State Transfer (REST) Web Services 2](#_Toc500054042)

[JAX-RS API 3](#_Toc500054043)

[Root Resource Classes 3](#_Toc500054044)

[Resource Methods 3](#_Toc500054045)

[JAX-RS Annotations 3](#_Toc500054046)

[@Path 3](#_Toc500054047)

[@<Resource\_Method\_Designators> 4](#_Toc500054048)

[@<\*>Param (Parameter Annotations) 5](#_Toc500054049)

[@PathParam 5](#_Toc500054050)

[@QueryParam 5](#_Toc500054051)

[@FormParam 5](#_Toc500054052)

[@HeaderParam 5](#_Toc500054053)

[@CookieParam 5](#_Toc500054054)

[@MatrixParam 5](#_Toc500054055)

[@BeanParam 6](#_Toc500054056)

[@DefaultValue 6](#_Toc500054057)

[@Produces 6](#_Toc500054058)

[@Consumes 7](#_Toc500054059)

[javax.ws.rs.core.MediaType 8](#_Toc500054060)

[JAXB and JSON JAX-RS Handlers 8](#_Toc500054061)

# Representational State Transfer (REST) Web Services

* It's an "architectural style" of client-server application, centred on the "transfer" of "representations" of "resources" through requests and responses
* In the REST architectural style, data and functionality (i.e. Web Service Methods) are considered as resources and are accessed using Uniform Resource Identifiers (URIs), typically hyperlinks on the Web
* The representation of that resource might be
* an XML document
* a JSON File
* a Simple Text
* an image file
* an HTML page, etc.,

* A client application might
* retrieve a particular representation
* modify the resource by updating its data or
* delete the resource entirely
* The REST architectural style is designed to use a stateless communication protocol, typically HTTP

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

1. **Resource Identification through URI:-**

Resources in RESTful web services are identified by URIs

1. **Uniform Interface:-**

* Resources are manipulated using a fixed set of 4 operations

1. Create
2. Read /Get
3. Update
4. Delete

* These operations can be perform using below HTTP Methods respectively

1. PUT (creates a new resource)
2. GET (retrieves the current state of a resource in some representation)
3. POST (transfers a new state onto a resource OR Update the existing resource)
4. DELETE (Delete an existing resource)
5. **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 other formats.
* Hence RESTful web services are loosely coupled, lightweight web services they are well suited for creating APIs for clients spread across the internet

# JAX-RS API

* JAX-RS stands for JAVA API for RESTful Web Services
* JAX-RS makes it easy for developers to build RESTful web services using the Java programming language as compared to SOAP/XML Web Services
* "javax.ws.rs.\*" is the package representation of JAX-RS API
* The JAX-RS API uses "annotations" to simplify the development of RESTful web services. So Developers
* can decorate Java Beans with JAX-RS annotations to define resources and
* the actions that can be performed on those resources

# Root Resource Classes

* They are Java Classes that are
* annotated with @Path
* have at least "one method"
* annotated with @Path OR

a "resource method designator" annotation (such as @GET, @PUT, @POST, @DELETE)

* They MUST BE "public" in nature & They MUST have "public default constructor" (ONLY in case of JAXRS–unaware servlet containers)

# Resource Methods

* They are methods of a "resource class" annotated with a "resource method designator"
* They SHOULD be public methods
* They May / May-Not return value i.e. the resource method may returns "void". This means "No Representation" is returned and response with a status code of 204 (No Content) will be returned to the client.

# JAX-RS Annotations

## @Path

* It identifies a particular "Resource Method" in a “Root Resource Class"
* It can be specified at "Class" or "Method" level
* Java classes that you want to be recognized as JAX-RS services must have this annotation.
* Declaration at Class Level is Mandatory. However declaration at Method Level is Optional
* If it’s not present at Method Level then always First Method gets executed
* Avoid using spaces in Path Name.
* Instead uses underscore (\_) or hyphen (-) while using a long resource name.
* For example, use "/create\_employee" instead of "/create employee"
* Use lowercase letters in Path Name
* @Path value may or may not begin with a '/', it makes no difference
* Likewise, @Path value may or may not end in a '/', it makes no difference
* Thus request URLs that end or do not end in a '/' will both be matched
* Few Examples:

@Path("customers/{firstname}-{lastname}")

@Path("/") ===> Can be used with Resource Class

## @<Resource\_Method\_Designators>

* This annotations are used with Java Methods & they are called as "Resource Method Designator Annotations"
* The JAX-RS spec disallows multiple method designators on a single Java method
* Its Mandatory Information & every Resource Method should have ONLY ONE Resource Method Designator
* The Java method annotated with

@GET will process HTTP GET requests

@POST will process HTTP POST requests

@PUT will process HTTP PUT requests

@DELETE will process HTTP DELETE requests

@HEAD will process HTTP HEAD requests

@OPTIONS will process HTTP OPTIONS requests

* NOTE:
* There is NO @TRACE and @CONNECT annotation
* For Resource Methods @Path is Optional, however,

@<Resource\_Method\_Designators> is Mandatory (ONLY ONE)

## @<\*>Param (Parameter Annotations)

* Parameters of a "resource method" may be annotated with parameter-based annotations to extract information from a request
* Usually, these annotations are used on the input arguments of a "Resource Methods"

### @PathParam

* It represents the parameter of the URI path

**Syntax:** {variable\_name}

**Ex:** @Path("/users/**{username}**")

* This annotation allows us to extract values from extract a path parameter from the path component of the request URL

* It can be used with Regular Expressions

**Syntax:** {variable\_name : regular\_expression}

**Ex:** @Path("**{id : \\d+}**") //It supports digit only

### @QueryParam

* This annotation allows us to extract values from URL Query Parameters

### @FormParam

* This annotation allows us to extract values from "posted" form data
* This annotation is used to access "application/x-www-form-urlencoded" request bodies.
* In other words, whenever we submit the form which has method="post" then request header will have "Content-Type: application/x-www-form-urlencoded" information
* It should not be used with @GET

### @HeaderParam

* This annotation allows us to extract values from HTTP request headers

### @CookieParam

* This annotation allows us to extract values from HTTP request cookies

### @MatrixParam

* Matrix parameters are a set of “name=value” in URI path

**For Ex:** /users/praveen;userid=abcd

* URI can consist of N number of Matrix parameters but they should be separate by a semi colon “;“
* They can be present anywhere in URI
* This annotation allows us to extract values from URI matrix parameters

**NOTE:-**

* All these Parameter Annotations refer various parts of an HTTP request. These parts are represented as a string of characters within the HTTP request.
* So we can get them as a String values or else JAX-RS can convert this string data into any Java type that we want, provided that it matches one of the following criteria:

1. Be a primitive type (byte, short, int, long, float, double, char & boolean)
2. Have a Class Name which has constructor that accepts a single String argument
3. Be a List<T>, Set<T> or SortedSet<T> resulting collection is read-only

### @BeanParam

* The @BeanParam annotation is something new added in the JAX-RS 2.0 specification.
* It allows you to inject an application-specific class whose property methods or fields are annotated with "Parameter Annotations"
* The JAX-RS runtime will introspect the @BeanParam parameter’s type for injection annotations and then set them as appropriate.

## @DefaultValue

* Assigns a default value to a parameters (Parameter Annotations)
* If the @DefaultValue is not used in conjunction with "Parameter Annotations" and if any parameter is not present in the request then value will be
* an "empty collection" for List, Set or SortedSet
* "null" for other object types and
* "default values" for primitive types

## @Produces

* This 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",
  + "application/json",
  + "application/xml", etc.,
* @Produces can be applied at both the class as well as at method levels
* If @Produces is applied at the class level, all the methods in a resource can produce the specified MIME types by default
* If it is applied at the method level, it overrides any @Produces annotations applied at the class level
* For Example:

@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 type of the @Produces annotation at the class level
* The doGetAsHtml method's @Produces annotation overrides the class-level @Produces setting, and specifies that the method can produce HTML rather than plain text
* The value of @Produces is an array of String of MIME types. For example:

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

* Hence more than one media type may be declared in the same @Produces declaration.

**Ex:**

@GET

@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 are acceptable
* If both are equally acceptable (i.e. Request with Accept Header value as "\*/\*"), then the former will be chosen because it occurs first
* If no methods in a resource are able to produce the MIME type in a client request, the Jersey runtime sends back an HTTP “406 Not Acceptable” error

## @Consumes

* This annotation is used to specify the MIME media types of representations a resource can consume from the client
* @Consumes can be applied at both the class and the method levels
* If it is applied at the class level, all the methods in a resource can consume the specified MIME types by default
* If it is applied at the method level, it overrides any @Consumes annotations applied at the class level
* The value of @Consumes is an array of String of acceptable MIME types. For example:
* @Consumes({"text/plain", "text/html"})
* If a resource is unable to consume the MIME type of a client request, the Jersey runtime sends back an HTTP “415 Unsupported Media Type” error

# javax.ws.rs.core.MediaType

* It's a Concrete Class part of JAX-RS API which has lot of Constants with most popular MIME Types
* Rather than typing MIME media types, It is possible to refer to constant values, which may reduce typographical errors

**EX:-**

Rather than typing

@Produces("application/xml")

We can use

@Produces(MediaType.APPLICATION\_XML)

# JAXB and JSON JAX-RS Handlers

* Once we apply JAXB annotations to Java classes, with JAX-RS API it is very easy to exchange XML/JSON data between client and web services
* The built-in JAXB and JSON (Jettison, Jackson, etc.,) handlers will automatically takes care of Marshalling & Unmarshalling of these Java Classes to XML/JSON
* Also, by default, JAX-RS API will take care of the creation and initialization of JAXBContext instances
* Because the creation of JAXBContext instances can be expensive, JAX-RS implementations usually cache them after they are first initialized.