JSPIDER
BASAVANGUDI
BANGALORE

WEB SERVICES
JERSEY SERVER API
(PRODUCER)

| Praveen D

Table of Contents

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. **<u>Resource Identification through URI:-</u>**
   Resources in RESTful web services are identified by URIs

2. **<u>Uniform Interface:-</u>**
   - 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)

3. **<u>Self-descriptive messages:-</u>**
   - 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.

javax.ws.rs.core.Response

- Many times we may need to send "additional information OR Metadata information" along with the response like Cookies, Last Modified Date, Language, Caching Information, Location URI of the Newly created Resource, etc.,

- In such cases, resource methods can return instances of javax.ws.rs.core.Response

- "javax.ws.rs.core.Response" is an Abstract Class and "javax.ws.rs.core.Response.ResponseBuilder" is an inner abstract class of Response. Both are part of JAX-RS API and both classes have couple of abstract methods.

- The Classes which extends above abstract classes are in Jersey Framework
  1. org.glassfish.jersey.message.internal.OutboundJaxrsResponse extends Response
  2. org.glassfish.jersey.message.internal.OutboundJaxrsResponse$Builder extends ResponseBuilder

- Hence implementations of abstract methods present in Response & ResponseBuilder classes are provided in Jersey Framework.

- Response object cannot be created directly; instead,
  1. First we need to get the javax.ws.rs.core.Response.ResponseBuilder Object by invoking one of the static helper methods (depending on our need) of javax.ws.rs.core.Response

  2. ResponseBuilder object has many non-static methods which would help us to set the Metadata information to Response. Hence we need to invoke one/more non-static methods (depending on our need) on ResponseBuilder

  3. Finally invoke build() method to get the Response objects

  Psuedo-Code
  ResponseBuilder builder = Response.one-of-the-Static_Methods();
  builder.one-or-more-Non_Static_Methods();
  Response resp = builder.build();

- The ResponseBuilder class is a factory that helps to create one individual Response object

- For example, below is the code which returns 201 (Created) status code and a Location header whose value is the URI to the newly created resource

```
@POST
@Consumes("application/xml")
public Response createResource(String content)
{
  //Code to Create the Resource using content
  URI createdUri = ...
  return Response.created(createdUri).build();
}
```

Handling Cookies in RESTFul Web Services

- W.K.T Cookies are little piece of information in the form of "name=value String pair" exchanged between Client & Server
- Servers can store state information in cookies on the client, and can retrieve that information when the client makes its next request
- Many web applications use cookies to set up a session between the client and the server.
- They also use cookies to remember identity and user preferences between requests. These cookie values are transmitted back and forth between the client and server

1. Sending Cookie as part of the Response:-

- javax.ws.rs.core.NewCookie is a Concreate Class part of JAX-RS API which extends javax.ws.rs.core.Cookie
- It's used to create a new HTTP cookie and transferred as part of a response.
- To set cookies as part of response, create instances of NewCookie and pass them to the method ResponseBuilder.cookie()

Example:

```
@Path("/myservice")
public class MyService {

   @GET
   public Response createCookie()  {
    NewCookie cookie = new NewCookie("key", "value");
    ResponseBuilder builder = Response.ok("hello", "text/plain");
    return builder.cookie(cookie).build();
   }

}
```

2. Getting Cookie from the Request:-

- javax.ws.rs.core.Cookie Represents the value of a HTTP cookie, transferred in a request
- The @javax.ws.rs.CookieParam annotation allows us to inject cookies sent by a client request into your JAX-RS resource methods
- For example, let's say if web service sends a customerId cookie to clients so that we can track users as they invoke and interact with our web services, code to pull in this information might look like this:

```
@Path("/myservice")
public class MyService
{
   @GET
   @Produces("text/html")
   public String getCookie(@CookieParam("customerId")int custId){
         ...
   }
}
```

- @CookieParam makes JAX-RS to search all cookies with the name "customerId". It then converts the corresponding value into an int and injects it into the custId parameter

HTTP "Content-Disposition" Header

- It's optional in HTTP can be present either in HTTP Request or Response

- Content-Disposition in HTTP Response indicates
  - if the content is to be displayed in the browser or
  - as an attachment (i.e. downloaded and saved locally)

- Content-Disposition will be present ONLY in "multipart/form-data" HTTP Request Body. It's subpart of a multipart body to give information about the field it applies to. The subpart is delimited by the boundary defined in the Content-Type header.

- Syntax for HTTP Request Header
  Content-Disposition: form-data
  Content-Disposition: form-data; name="fieldName"
  Content-Disposition: form-data; name="fieldName"; filename="filename.jpg"

- Syntax for HTTP Response Header
  Content-Disposition: inline
  Content-Disposition: attachment
  Content-Disposition: attachment; filename="filename.jpg"

- The first parameter of "Content-Disposition" in "HTTP Response Header" is either
  - "inline" (default value, indicating it can be display inside the Web page, or as the Web page) or
  - "attachment" (indicating it should be downloaded; most browsers presenting a 'Save as' dialog, prefilled with the value of the filename parameters if present

- The first parameter of "Content-Disposition" in "HTTP Request Header" is always "form-data". Additional parameters are case-insensitive and have arguments, that use quoted-string syntax after the '=' sign. Multiple parameters are separated by a semi-colon (';')

Dealing with Large Binary Objects (i.e. Uploading File) using Jersey

- While developing web services we may need to create web services which needs to deal with large binary Objects such as images, documents and various types of media files

- When "Web Service Consumer" wants to upload any files then
  - Request SHOULD have method="post" and
  - Request should have encoded POST body

- For this Consumer have to use, enctype="multipart/form-data" encoding to deal with a large binary object uploaded via <input type="file">

- Unfortunately, JAX-RS API does not have standardized Annotation to deal with large files

- to overcome this problem Jersey provides "@FormDataParam" Annotation

@FormDataParam Annotation

- This annotation binds the named body part(s) of a "multipart/form-data" request entity body to a
  - resource method parameter or
  - a class member variable (@BeanParam) as appropriate

- This annotation should be used in conjunction with the media type "multipart/form-data" should be used for submitting and consuming forms that contain files, non-ASCII data, and binary data

- Jersey allows to inject @FormDataParam onto following parameter types
  - Any type of parameter for which a message body reader is available
  - "org.glassfish.jersey.media.multipart.FormDataContentDisposition" this represents the form data content disposition header

- "FormDataContentDisposition" is a concrete class of Jersey which consist of Metadata information of the "multipart/form-data" request like file name, size, creation date, etc.,

javax.ws.rs.core.Application

- It's a Concreate Class present in JAX-RS API

- The Application class is the only portable way of telling container that which web services (@Path annotated classes) we want to be deployed. This Class list "Classes and Objects" that JAX-RS is supposed to deploy.

- Application Class has below 3 methods & hence subclass of this class can override any/all/none of these methods
  1. public Set<Class<?>> getClasses()
  2. public Set<Object> getSingletons()
  3. public Map<String, Object> getProperties()

- The getClasses() method returns a list of JAX-RS web service and provider classes. These Classes will get instantiated once request comes & garbage collected once response is given (non-singleton in nature). These Classes SHOULD have public default constructor. In this case, it's a Container responsibility to create an instance of resource class

- The getSingletons() method returns a list of pre allocated JAX-RS web services and providers. These Classes will get instantiated during the server start-up & garbage collected during the server shut down. Hence they are "singleton" in nature. These Classes need not to have public default constructor. In this case, we as a developer are responsible for creating these objects.

- The getProperties() method returns a map of custom application-wide properties

- The JAX-RS runtime will iterate through the list of objects and register them internally.

- We can fully leverage "servlet class scanning abilities" of application server in case if we don't override getClasses() and getSingletons() OR both return an empty set

  For Example:-

```
@ApplicationPath("/services")
public class MyRestApplication extends Application
{
    //Empty Class
}
```

- When scanning, the application server will look within WEB-INF/classes and any JAR file within the WEB-INF/lib directory
- It will add any class annotated with @Path to the list of things that need to be deployed and registered with the JAX-RS runtime
- You can also deploy as many Application classes as you want in one WAR
- The scanner will also ignore any Application classes not annotated with @ApplicationPath