**REST Web Services**

# REST(Representation state transfer) webservice

* 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 REST architectural style constrains an architecture to a client/server architecture and is designed to use a stateless communication protocol, typically HTTP.
* Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUTcreates a new resource, which can be then deleted by using DELETE. GETretrieves the current state of a resource in some representation. POSTtransfers a new state onto a resource
* 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.
* 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.
* A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia. By contrast, in a [service-oriented architecture](https://en.wikipedia.org/wiki/Service-oriented_architecture) (SOA), clients and servers interact through a fixed [interface](https://en.wikipedia.org/wiki/Interface_(computing)) shared through documentation or an [interface description language](https://en.wikipedia.org/wiki/Interface_description_language)(IDL).

## JAX-RS

JAX-RS: Java API for RESTful Web Services (JAX-RS) is a [Java programming language](https://en.wikipedia.org/wiki/Java_(programming_language)) [API](https://en.wikipedia.org/wiki/Application_programming_interface) that provides support in creating [web services](https://en.wikipedia.org/wiki/Web_service) according to the [Representational State Transfer](https://en.wikipedia.org/wiki/Representational_State_Transfer) (REST) architectural pattern. JAX-RS uses [annotations](https://en.wikipedia.org/wiki/Java_annotation), introduced in[Java SE 5](https://en.wikipedia.org/wiki/Java_Platform,_Standard_Edition), to simplify the development and deployment of web service clients and endpoints

## JAX-RS implementations

Apache CXF - an open source [Web service](https://en.wikipedia.org/wiki/Web_service) framework

Jersey - the [reference implementation](https://en.wikipedia.org/wiki/Reference_implementation) from [Sun](https://en.wikipedia.org/wiki/Sun_Microsystems) (now [Oracle](https://en.wikipedia.org/wiki/Oracle_Corporation))

RESTeasy - [JBoss](https://en.wikipedia.org/wiki/JBoss)'s implementation

## 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 method is identical to GET except that the server MUST NOT return a message-body in the response. The metainformation contained in the HTTP headers in response to a HEAD request SHOULD be identical to the information sent in response to a GET request. This method can be used for obtaining metainformation about the entity implied by the request without transferring the entity-body itself. This method is often used for testing hypertext links for validity, accessibility, and recent modification.  For HEAD, the runtime will invoke the implemented GET method, if present, and ignore the response entity, if set |
| @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. |
| @OPTIONS | This method allows the client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action or initiating a resource retrieval.  The server should return response like  200 OK  Allow: HEAD,GET,PUT,DELETE,OPTIONS  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 |
| @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 | Providers are a simply a way of extending and customizing the JAX-RS runtime. Providers are *not* the same as resources classes, they exist, conceptually, at a level in-between resources classes and the JAX-RS implementation.  There are three classes of providers defined by the current JAX-RS specification. The commonality between them is that all providers must be identified by the @Provider annotation and follow certain rules for constructor declaration. Apart from that, different provider types may have additional annotations, and will implement different interfaces.  Entity Providers  These providers control the mapping of data representations (like XML, JSON, CSV) to their Java object equivalents.  Context Providers  These providers control the context that resources can access via @Context annotations.  Exception Providers  These providers control the mapping of Java exceptions to a JAX-RS Response instance.  The @Provider annotation is used for anything that is of interest to the JAX-RS runtime, such asMessageBodyReader and MessageBodyWriter. For HTTP requests, theMessageBodyReader 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 aMessageBodyWriter. 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 using Response.ResponseBuilder. |

# Request method designators:

@GET,@POST,@DELETE,@PUT,@HEAD are called request method designators.

## When should we use PUT and when should we use POST?

The HTTP methods POST and PUT aren't the HTTP equivalent of the CRUD's create and update. They both serve a different purpose. It's quite possible, valid and even preferred in some occasions, to use POST to create resources, or use PUT to update resources.

Use PUT when you can update a resource completely through a specific resource. For instance, if you know that an article resides at http://example.org/article/1234, you can PUT a new resource representation of this article directly through a PUT on this URL.

If you do not know the actual resource location, for instance, when you add a new article, but do not have any idea where to store it, you can POST it to an URL, and let the server decide the actual URL.

PUT /article/1234 HTTP/1.1

<article>

<title>red stapler</title>

<price currency="eur">12.50</price>

</article>

POST /articles HTTP/1.1

<article>

<title>blue stapler</title>

<price currency="eur">7.50</price>

</article>

HTTP/1.1 201 Created

Location: /articles/63636

As soon as you know the new resource location, you can use PUT again to do updates to the blue stapler article. But as said before: you CAN add new resources through PUT as well. The next example is perfectly valid if your API provides this functionality:

PUT /articles/green-stapler HTTP/1.1

<article>

<title>green stapler</title>

<price currency="eur">9.95</price>

</article>

HTTP/1.1 201 Created

Location: /articles/green-stapler

- See more at: <http://restcookbook.com/HTTP%20Methods/put-vs-post/#sthash.USfwiRrS.dpuf>

## @PathParam Example:

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

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

**import** javax.ws.rs.PathParam;

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

@Path("/users")

**public** **class** UserRestService {

@GET

@Path("{year}/{month}/{day}")

**public** Response getUserHistory(

@PathParam("year") **int** year,

@PathParam("month") **int** month,

@PathParam("day") **int** day) {

String date = year + "/" + month + "/" + day;

**return** Response.status(200)

.entity("getUserHistory is called, year/month/day : " + date)

.build();

}

}

URI Pattern : “**/users/2011/06/30**”

getUserHistory is called, year**/**month**/**day : 2011**/**6**/**30

## @QueryParam Example

**import** java.util.List;

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

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

**import** javax.ws.rs.QueryParam;

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

@Path("/users")

**public** **class** UserService {

@GET

@Path("/query")

**public** Response getUsers(

@QueryParam("from") **int** from,

@QueryParam("to") **int** to,

@QueryParam("orderBy") List<String> orderBy) {

**return** Response

.status(200)

.entity("getUsers is called, from : " + from + ", to : " + to

+ ", orderBy" + orderBy.toString()).build();

}

}

URI Pattern : “**users/query?from=100&to=200&orderBy=age&orderBy=name**”

getUsers is called, from : 100, to : 200, orderBy**[**age, name**]**

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 @PathParam annotation 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.

# [Example of producing and consuming JSON/XML](https://www.nabisoft.com/tutorials/java-ee/producing-and-consuming-json-or-xml-in-java-rest-services-with-jersey-and-jackson)

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

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 ResponseBuilderclass 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 POSTrequests.

## 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. ThedoGetAsHtml method’s @Produces annotation overrides the class-level @Produces setting and specifies that the method can produce HTML rather than plain text.

If a resource class is capable of producing 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 doGetAsPlainTextmethod 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 doGetAsHtmlmethod 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 andapplication/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.

# 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 @Consumes annotations 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 doPost2method overrides the class level @Consumes annotation 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

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 thejavax.ws.rs.QueryParam annotation in the method parameter arguments. The following example, from thesparklines sample application, demonstrates using @QueryParam to extract query parameters from the Querycomponent 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.

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.

# @CookieParam

Cookie parameters, indicated by decorating the parameter with javax.ws.rs.CookieParam, extract information from the cookies declared in cookie-related HTTP headers.

Example:

 @GET

    @Produces({MediaType.APPLICATION\_JSON})

    public Response get(

            @CookieParam(value = "User-Agent") String userAgent,

            @CookieParam(value = "Last-Accessed") String lastAccessed,

            @HeaderParam(value = "User-Agent") String userAgentHeader) {

# @HeaderParam

Header parameters, indicated by decorating the parameter withjavax.ws.rs.HeaderParam, extract information from the HTTP headers.

# @MatrixParam

Matrix parameters, indicated by decorating the parameter with javax.ws.rs.MatrixParam, extract information from URL path segments

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

**/**books**/**2011;author=mkyong

In above URI, the matrix parameter is “**author=mkyong**“, separates by a semi colon “**;**“.

* More readable
* No encoding and decoding of "&" in XML documents is required
* URLs with "?" are not cached in many cases; URLs with matrix params are cached
* Matrix parameters can appear everywhere in the path and are not limited to its end. Matrix parameters can have more than one value: paramA=val1,val2.
* The important difference is that matrix parameters apply to a particular path element while query parameters apply to the request as a whole.

Example:

@GET

@Path("{year}")

**public** Response getBooks(@PathParam("year") String year,

@MatrixParam("author") String author,

@MatrixParam("country") String country) {

# Form Parameters

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.

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

}

# MultiValuedMap

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();

}

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

}

# 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.

In addition to supporting static content negotiation, JAX-RS also supports runtime content negotiation using the javax.ws.rs.core.Variant class and Request objects. The Variantclass specifies the resource representation of content negotiation. Each instance of the Variantclass 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.VariantListBuilderclass 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();

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);

}

}

//TODO

## Using JAX-RS With JAXB

REST security

# HTTP Status codes:

**200 OK**

**201 Created**

The request has been fulfilled and resulted in a new resource being created.

**202 Accepted**

The request has been accepted for processing, but the processing has not been completed. The request might or might not eventually be acted upon, as it might be disallowed when processing actually takes place.

**204 No Content**

The server successfully processed the request, but is not returning any content. Usually used as a response to a successful delete request.

**400 Bad Request**

The server cannot or will not process the request due to something that is perceived to be a client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).

**401 Unauthorized**

[**403 Forbidden**](https://en.wikipedia.org/wiki/HTTP_403)

[**404 Not Found**](https://en.wikipedia.org/wiki/HTTP_404)

The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

**405 Method Not Allowed**

A request was made of a resource using a request method not supported by that resource; for example, using GET on a form which requires data to be presented via [POST](https://en.wikipedia.org/wiki/POST_(HTTP)), or using PUT on a read-only resource.

**406 Not Acceptable**

The requested resource is only capable of generating content not acceptable according to the Accept headers sent in the request.

**408 Request Timeout**

The server timed out waiting for the request. According to HTTP specifications: "The client did not produce a request within the time that the server was prepared to wait. The client MAY repeat the request without modifications at any later time."

**409 Conflict**

Indicates that the request could not be processed because of conflict in the request, such as an [edit conflict](https://en.wikipedia.org/wiki/Edit_conflict) in the case of multiple updates.

**412 Precondition Failed (**[**RFC 7232**](https://tools.ietf.org/html/rfc7232)**)**

The server does not meet one of the preconditions that the requester put on the request.

**415 Unsupported Media Type**

The request entity has a [media type](https://en.wikipedia.org/wiki/Internet_media_type) which the server or resource does not support. For example, the client uploads an image as [image/svg+xml](https://en.wikipedia.org/wiki/Scalable_Vector_Graphics), but the server requires that images use a different format.

**500 Internal Server Error**

A 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 fulfil the request. Usually this implies future availability (e.g., a new feature of a web-service API).

**502 Bad Gateway**

The server was acting as a [gateway](https://en.wikipedia.org/wiki/Gateway_(telecommunications)) or proxy and received an invalid response from the upstream server.

**503 Service Unavailable**

The server is currently unavailable (because it is overloaded or down for maintenance). Generally, this is a temporary state.

**504 Gateway Timeout**

The server was acting as a gateway or proxy and did not receive a timely response from the upstream server.

**505 HTTP Version Not Supported**

The server does not support the HTTP protocol version used in the request.

# Links

<https://www.nabisoft.com/tutorials/java-ee/producing-and-consuming-json-or-xml-in-java-rest-services-with-jersey-and-jackson>