**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

RESTLet - Restlet is leading RESTful web framework for Java applications is used to build RESTFul web services it has two part Restlet API and a Restlet implementation much like Servlet specification.

## 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

## 16.1. Securing server

### 16.1.1. SecurityContext

Security information of a request is available by injecting a JAX-RS [SecurityContext](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/SecurityContext.html" \t "_top) instance using [@Context](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html) annotation. The injected security context instance provides the equivalent of the functionality available on [HttpServletRequest](http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletRequest.html" \t "_top) API. The injected security context depends on the actual Jersey application deployment. For example, for a Jersey application deployed in a Servlet container, the Jersey SecurityContext will encapsulate information from a security context retrieved from the Servlet request. In case of a Jersey application deployed on a Grizzly server, the SecurityContext will return information retrieved from the Grizzly request.

**Example 16.1. Using SecurityContext for a Resource Selection**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Path("basket")  public ShoppingBasketResource get(@Context SecurityContext sc) {      if (sc.isUserInRole("PreferredCustomer") {          return new PreferredCustomerShoppingBasketResource();      } else {          return new ShoppingBasketResource();      }  } |

SecurityContext is inherently request-scoped, yet can be also injected into fields of singleton resources and JAX-RS providers. In such case the proxy of the request-scopedSecurityContext will be injected.

**Example 16.2. Injecting SecurityContext into a singleton resource**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @Path("resource")  @Singleton  public static class MyResource {      // Jersey will inject proxy of Security Context      @Context      SecurityContext securityContext;        @GET      public String getUserPrincipal() {          return securityContext.getUserPrincipal().getName();      }  } |

#### 16.1.1.1. Initializing Security Context with Servlets

As described above, the SecurityContext by default (if not overwritten by a request filter) only exposes security information from the underlying container. In the case you deploy a Jersey application in a Servlet container, you need to configure the Servlet container security aspects (<security-constraint>, <auth-constraint> and user to roles mappings) in order to be able to secure requests via calls to to the JAX-RS SecurityContext.

### 16.1.2. Authorization - securing resources

#### 16.1.2.1. Security resources with web.xml

In cases where a Jersey application is deployed in a Servlet container you can rely only on the standard Java EE Web application security mechanisms offered by the Servlet container and configurable via application's web.xml descriptor. You need to define the <security-constraint> elements in the web.xml and assign roles which are able to access these resources. You can also define HTTP methods that are allowed to be executed. See the following example.

**Example 16.3. Securing resources using web.xml**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | <security-constraint>      <web-resource-collection>          <url-pattern>/rest/admin/\*</url-pattern>      </web-resource-collection>      <auth-constraint>          <role-name>admin</role-name>      </auth-constraint>  </security-constraint>  <security-constraint>      <web-resource-collection>          <url-pattern>/rest/orders/\*</url-pattern>      </web-resource-collection>      <auth-constraint>          <role-name>customer</role-name>      </auth-constraint>  </security-constraint>  <login-config>      <auth-method>BASIC</auth-method>      <realm-name>my-default-realm</realm-name>  </login-config> |

The example secures two kinds of URI namespaces using the HTTP Basic Authentication. rest/admin/\* will be accessible only for user group "admin" and rest/orders/\* will be accessible for "customer" user group. This security configuration does not use JAX-RS or Jersey features at all as it is enforced by the Servlet container even before a request reaches the Jersey application. Keeping these security constrains up to date with your JAX-RS application might not be easy as whenever you change the [@Path](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html) annotations on your resource classes you may need to update also the web.xml security configurations to reflect the changed JAX-RS resource paths. Therefore Jersey offers a [more flexible solution](https://jersey.java.net/documentation/latest/user-guide.html#annotation-based-security) based on placing standard Java EE security annotations directly on JAX-RS resource classes and methods.

#### 16.1.2.2. Securing JAX-RS resources with standard javax.annotation.security annotations

With Jersey you can define the access to resources based on the user group using annotations. You can, for example, define that only a user group "admin" can execute specific resource method. To do that you firstly need to register [RolesAllowedDynamicFeature](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/server/filter/RolesAllowedDynamicFeature.html" \t "_top) as a provider. The following example shows how to register the feature if your deployment is based on a [ResourceConfig](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/server/ResourceConfig.html" \t "_top).

**Example 16.4. Registering RolesAllowedDynamicFeature using ResourceConfig**

|  |  |
| --- | --- |
| 1  2 | final ResourceConfig resourceConfig = new ResourceConfig(MyResource.class);  resourceConfig.register(RolesAllowedDynamicFeature.class); |

Alternatively, typically when deploying your application to a Servlet container, you can implement your JAX-RS [Application](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html) subclass by extending from the JerseyResourceConfig and registering he RolesAllowedDynamicFeature in the constructor:

**Example 16.5. Registering RolesAllowedDynamicFeature by extending ResourceConfig**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class MyApplication extends ResourceConfig {      public MyApplication() {          super(MyResource.class);          register(RolesAllowedDynamicFeature.class);      }  } |

Once the feature is registered, you can use annotations from package javax.annotation.security defined by JSR-250. See the following example.

**Example 16.6. Applying javax.annotation.security to JAX-RS resource methods.**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | @Path("/")  @PermitAll  public class Resource {      @RolesAllowed("user")      @GET      public String get() { return "GET"; }        @RolesAllowed("admin")      @POST      public String post(String content) { return content; }        @Path("sub")      public SubResource getSubResource() {          return new SubResource();      }  } |

The resource class Resource defined in the example is annotated with a [@PermitAll](http://docs.oracle.com/javaee/7/api/javax/annotation/security/PermitAll.html) annotation. This means that all methods in the class which do not override this annotation will be permitted for all user groups (no restrictions are defined). In our example, the annotation will only apply to the getSubResource() method as it is the only method that does not override the annotation by defining custom role-based security settings using the [@RolesAllowed](http://docs.oracle.com/javaee/7/api/javax/annotation/security/RolesAllowed.html) annotation. @RolesAllowed annotations present on the other methods define a role or a set of roles that are allowed to execute a particular method.

These Java EE security annotations are processed internally in the request filter registered using the Jersey RolesAllowedDynamicFeature. The roles defined in the annotations are tested against current roles set in the SecurityContext using the SecurityContext.isUserInRole(String role) method. In case the caller is not in the role specified by the annotation, the HTTP 403 (Forbidden) error response is returned.

## 16.2. Client Security

The JAX-RS client API can be utilized to consume any Web service exposed on top of a HTTP protocol or it's extension (e.g. WebDAV), and is not restricted to services implemented using JAX-RS.

**Example 5.1. POST request with form parameters**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Client client = ClientBuilder.newClient();  WebTarget target = client.target("[http://localhost:9998](http://localhost:9998/)").path("resource");    Form form = new Form();  form.param("x", "foo");  form.param("y", "bar");    MyJAXBBean bean =  target.request(MediaType.APPLICATION\_JSON\_TYPE)      .post(Entity.entity(form,MediaType.APPLICATION\_FORM\_URLENCODED\_TYPE),          MyJAXBBean.class); |

### 5.3.6. Example summary

The following code puts together the pieces used in the earlier examples.

**Example 5.2. Using JAX-RS Client API**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | ClientConfig clientConfig = new ClientConfig();  clientConfig.register(MyClientResponseFilter.class);  clientConfig.register(new AnotherClientFilter());    Client client = ClientBuilder.newClient(clientConfig);  client.register(ThirdClientFilter.class);    WebTarget webTarget = client.target("<http://example.com/rest>");  webTarget.register(FilterForExampleCom.class);  WebTarget resourceWebTarget = webTarget.path("resource");  WebTarget helloworldWebTarget = resourceWebTarget.path("helloworld");  WebTarget helloworldWebTargetWithQueryParam =          helloworldWebTarget.queryParam("greeting", "Hi World!");    Invocation.Builder invocationBuilder =          helloworldWebTargetWithQueryParam.request(MediaType.TEXT\_PLAIN\_TYPE);  invocationBuilder.header("some-header", "true");    Response response = invocationBuilder.get();  System.out.println(response.getStatus());  System.out.println(response.readEntity(String.class)); |

## 5.7. Closing connections

The underlying connections are opened for each request and closed after the response is received and entity is processed (entity is read). See the following example:

**Example 5.5. Closing connections**

|  |  |
| --- | --- |
| 1  2  3  4  5 | final WebTarget target = ... some web target  Response response = target.path("resource").request().get();  System.out.println("Connection is still open.");  System.out.println("string response: " + response.readEntity(String.class));  System.out.println("Now the connection is closed."); |

If you don't read the entity, then you need to close the response manually by response.close(). Also if the entity is read into an [InputStream](http://docs.oracle.com/javase/6/docs/api/java/io/InputStream.html" \t "_top) (byresponse.readEntity(InputStream.class)), the connection stays open until you finish reading from the InputStream. In that case, the InputStream or the Response should be closed manually at the end of reading from InputStream.

## 5.9. Securing a Client

This section describes how to setup SSL configuration on Jersey client (using JAX-RS API). The SSL configuration is setup in [ClientBuilder](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/ClientBuilder.html" \t "_top). The client builder contains methods for definition of [KeyStore](http://docs.oracle.com/javase/6/docs/api/java/security/KeyStore.html" \t "_top), [TrustStore](http://docs.oracle.com/javase/6/docs/api/java/security/TrustStore.html" \t "_top) or entire [SslContext](http://docs.oracle.com/javase/6/docs/api/javax/net/ssl/SslContext.html" \t "_top). See the following example:

|  |  |
| --- | --- |
| 1  2  3 | SSLContext ssl = ... your configured SSL context;  Client client = ClientBuilder.newBuilder().sslContext(ssl).build();  Response response = client.target("<https://example.com/resource>").request().get(); |

The example above shows how to setup a custom SslContext to the ClientBuilder. Creating a SslContext can be more difficult as you might need to init instance properly with the protocol, KeyStore, TrustStore, etc. Jersey offers a utility [SslConfigurator](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/SslConfigurator.html" \t "_top) class that can be used to setup the SslContext. The SslConfigurator can be configured based on standardized system properties for SSL configuration, so for example you can configure the KeyStore file name using a environment variablejavax.net.ssl.keyStore and SslConfigurator will use such a variable to setup the SslContext. See javadoc of [SslConfigurator](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/SslConfigurator.html" \t "_top) for more details. The following code shows how a SslConfigurator can be used to create a custom SSL context.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | SslConfigurator sslConfig = SslConfigurator.newInstance()          .trustStoreFile("./truststore\_client")          .trustStorePassword("secret-password-for-truststore")          .keyStoreFile("./keystore\_client")          .keyPassword("secret-password-for-keystore");    SSLContext sslContext = sslConfig.createSSLContext();  Client client = ClientBuilder.newBuilder().sslContext(sslContext).build(); |

Note that you can also setup KeyStore and TrustStore directly on a ClientBuilder instance without wrapping them into the SslContext. However, if you setup aSslContext it will override any previously defined KeyStore and TrustStore settings. ClientBuilder also offers a method for defining a custom [HostnameVerifier](http://docs.oracle.com/javase/6/docs/api/javax/net/ssl/HostnameVerifier.html)implementation. HostnameVerifier implementations are invoked when default host URL verification fails.

### Important

Note that to utilize HTTP with SSL it is necessary to utilize the “https” scheme.

Currently the default connector provider [HttpUrlConnectorProvider](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/client/HttpUrlConnector.html" \t "_top) provides connectors based on HttpUrlConnection which implement support for SSL defined by JAX-RS configuration discussed in this example.

In order to enable http authentication support in Jersey client register the [HttpAuthenticationFeature](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/client/authentication/HttpAuthenticationFeature.html" \t "_top). This feature can provide both authentication methods, digest and basic. Feature can work in the following modes:

* BASIC: Basic preemptive authentication. In preemptive mode the authentication information is send always with each HTTP request. This mode is more usual than the following non-preemptive mode (if you require BASIC authentication you will probably use this preemptive mode). This mode must be combined with usage of SSL/TLS as the password is send only BASE64 encoded.
* BASIC NON-PREEMPTIVE:Basic non-preemptive authentication. In non-preemptive mode the authentication information is added only when server refuses the request with 401 status code and then the request is repeated with authentication information. This mode has negative impact on the performance. The advantage is that it does not send credentials when they are not needed. This mode must be combined with usage of SSL/TLS as the password is send only BASE64 encoded.
* DIGEST: Http digest authentication. Does not require usage of SSL/TLS.
* UNIVERSAL: Combination of basic and digest authentication. The feature works in non-preemptive mode which means that it sends requests without authentication information. If 401 status code is returned, the request is repeated and an appropriate authentication is used based on the authentication requested in the response (defined in WWW-Authenticate HTTP header. The feature remembers which authentication requests were successful for given URI and next time tries to preemptively authenticate against this URI with latest successful authentication method.

To initialize the feature use static methods and builder of this feature. Example of building the feature in Basic authentication mode:

|  |  |
| --- | --- |
| 1 | HttpAuthenticationFeature feature = HttpAuthenticationFeature.basic("user", "superSecretPassword"); |

Example of building the feature in basic non-preemptive mode:

|  |  |
| --- | --- |
| 1  2 | HttpAuthenticationFeature feature = HttpAuthenticationFeature.basicBuilder()      .nonPreemptive().credentials("user", "superSecretPassword").build(); |

You can also build the feature without any default credentials:

|  |  |
| --- | --- |
| 1 | HttpAuthenticationFeature feature = HttpAuthenticationFeature.basicBuilder().build(); |

In this case you need to supply username and password for each request using request properties:

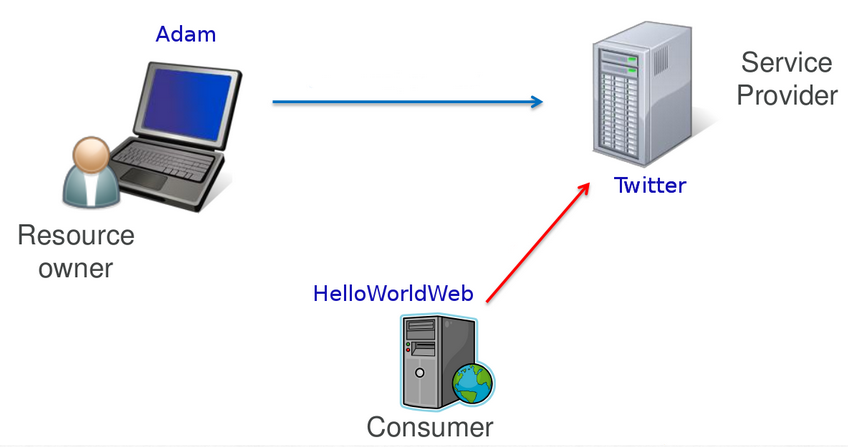
|  |  |
| --- | --- |
| 1  2  3 | Response response = client.target("<http://localhost:8080/rest/homer/contact>").request()      .property(HTTP\_AUTHENTICATION\_BASIC\_USERNAME, "homer")      .property(HTTP\_AUTHENTICATION\_BASIC\_PASSWORD, "p1swd745").get(); |

This allows you to reuse the same client for authenticating with many different credentials.

## 16.3. OAuth Support

OAuth is a specification that defines secure authentication model on behalf of another user. Two versions of OAuth exists at the moment - OAuth 1 defined by [OAuth 1.0 specification](http://tools.ietf.org/html/rfc5849) and OAuth 2 defined by [OAuth 2.0 specification](http://tools.ietf.org/html/rfc6749). OAuth 2 is the latest version and it is not backward compatible with OAuth 1 specification. OAuth in general is widely used in popular social Web sites in order to grant access to a user account and associated resources for a third party consumer (application). The consumer then usually uses RESTful Web Services to access the user data. The following example describes a use case of the OAuth (similar for OAuth 1 and OAuth 2). The example is simple and probably obvious for many developers but introduces terms that are used in this documentation as well as in Jersey OAuth API documentation.

Three parties act in an OAuth scenario.



The first party represents a user, in our case Adam, who is called in the OAuth terminology a Resource Owner. Adam has an account on Twitter. Twitter represents the second party. This party is called a Service Provider. Twitter offers a web interface that Adam uses to create new tweets, read tweets of others etc. Now, Adam uses our new web site, HelloWorldWeb, which is a very simple web site that says Hello World but it additionally displays the last tweet of the logged in user. To do so, our web site needs to have access to the Twitter account of Adam. Our web site is a 3rd party application that wants to connect to Twitter and get Adam's tweets. In OAuth, such party is called Consumer. Our Consumer would like to use Twitter's RESTful APIs to get some data associated with Adam's Twitter account. In order to solve this situation Adam could directly give his Twitter password to the HelloWorldWeb. This would however be rather unsafe, especially if we do not know much about the authors of the application. If Adam would give his password to HelloWorldWeb, he would have to deal with the associated security risks. First of all, Adam would have to fully trust HelloWorldWeb that it will not misuse the full access to his Twitter account. Next, if Adam would change his password, he would need to remember to give the new password also to the HelloWorldWeb application. And at last, if Adam would like to revoke the HelloWorldWeb's access to his Twitter account, he would need to change his password again. The OAuth protocol has been devised to address all these challenges.

With OAuth, a resource owner (Adam) grants an access to a consumer (HelloWorldWeb) without giving it his password. This access grant is achieved by a procedure calledauthorization flow. Authorization flow is out of the scope of this documentation and its description can be found in the OAuth specification linked above. The result of the authorization flow is an Access Token which is later used by the consumer to authenticate against the service provider. While this brief description applies to both OAuth 1 and 2, note that there are some differences in details between these two specifications.

Jersey OAuth is currently supported for the following use cases and OAuth versions:

* OAuth 1: Client (consumer) and server (service provider)
* OAuth 2: Client (consumer)

With client and server support there are two supported scenarios:

* Authorization flow
* Authentication with Access Token (support for authenticated requests)

### 16.3.1. OAuth 1

OAuth 1 protocol is based on message signatures that are calculated using specific signature methods. Signatures are quite complex and therefore are implemented in a separate module. The OAuth 1 Jersey modules are (groupId:artifactId:description):

* org.glassfish.jersey.security:oauth1-client: provides client OAuth 1 support for authorization flow and authentication
* org.glassfish.jersey.security:oauth1-server: provides server OAuth 1 support for authorization flow, SPI for token management including authentication filter.
* org.glassfish.jersey.security:oauth1-signature : provides support for OAuth1 request signatures. This module is a dependency of previous two modules and as such it will be implicitly included in your maven project. The module can be used as a standalone module but this will not be needed in most of the use cases. You would do that if you wanted to implement your own OAuth support and would not want to deal with implementing the complex signature algorithms.

#### 16.3.1.1. Server

To add support for OAuth into your server-side application, add the following dependency to your pom.xml:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>      <groupId>org.glassfish.jersey.security</groupId>      <artifactId>oauth1-server</artifactId>      <version>2.19</version>  </dependency> |

Again, there is no need to add a direct dependency to the signature module, it will be transitively included.

Let's now briefly go over the most important server Jersey OAuth APIs and SPIs:

* [OAuth1ServerFeature](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/server/oauth1/OAuth1ServerFeature.html): The feature which enables the OAuth 1 support on the server and registers OAuth1Provider explained in the following point.
* [OAuth1Provider](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/server/oauth1/OAuth1Provider.html): Implementation of this SPI must be registered to the server runtime as a standard provider. The implementation will be used to create request and access token, get consumer by consumer key, etc. You can either implement your provider or use the default implementation provided by Jersey by[DefaultOAuth1Provider](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/server/oauth1/DefaultOAuth1Provider.html).
* [OAuth1ServerProperties](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/server/oauth1/OAuth1ServerProperties.html): properties that can be used to configure the OAuth 1 support.
* [OAuth1Consumer](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/server/oauth1/OAuth1Consumer.html), [OAuth1Token](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/server/oauth1/OAuth1Token.html): classes that contain consumer key, request and access tokens. You need to implement them only if you also implement the interfaceOAuth1Provider.

First step in enabling Jersey OAuth 1 support is to register a OAuth1ServerFeature instance initialized with an instance of OAuth1Provider. Additionally, you may configure the Request Token URI and Access Token URI - the endpoints accessible on the OAuth server that issue Request and Access Tokens. These endpoints are defined in the OAuth 1 specification and are contacted as part of the OAuth authorization flow.

Next, when a client initiates the OAuth authorization flow, the provided implementation of OAuth1Provider will be invoked as to create new tokens, get tokens and finally to store the issued Access Token. If a consumer already has a valid Access Token and makes Authenticated Requests (with OAuth 1 Authorization information in the HTTP header), the provider will be invoked to provide the OAuth1Token for the Access Token information in the header.

**Difference between OAuth1 and OAuth2**

**More OAuth Flows to allow better support for non-browser based applications.** This is a main criticism against OAuth from client applications that were not browser based. For example, in OAuth 1.0, desktop applications or mobile phone applications had to direct the user to open their browser to the desired service, authenticate with the service, and copy the token from the service back to the application. The main criticism here is against the user experience. With OAuth 2.0, there are now new ways for an application to get authorization for a user.

**OAuth 2.0 no longer requires client applications to have cryptography.** This hearkens back to the old Twitter Auth API, which didn't require the application to HMAC hash tokens and request strings. With OAuth 2.0, the application can make a request using only the issued token over HTTPS.

**OAuth 2.0 signatures are much less complicated.** No more special parsing, sorting, or encoding.

**Finally, OAuth 2.0 is meant to have a clean separation of roles between the server responsible for handling OAuth requests and the server handling user authorization.**

#### 16.3.1.2. Client

### Note

OAuth client support in Jersey is almost identical for OAuth 1 and OAuth 2. As such, this chapter provides useful information even for users that use OAuth 2 client support.

To add support for OAuth into your Jersey client application, add the following dependency to your pom.xml:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>      <groupId>org.glassfish.jersey.security</groupId>      <artifactId>oauth1-client</artifactId>      <version>2.19</version>  </dependency> |

As mentioned earlier, there is no need to add a direct dependency to the signature module, it will be transitively included.

OAuth 1 client support initially started as a code migration from Jersey 1.x. During the migration however the API of was significantly revised. The high level difference compared to Jersey 1.x OAuth client API is that the authorization flow is no longer part of a client OAuth filter. Authorization flow is now a standalone utility and can be used without a support for subsequent authenticated requests. The support for authenticated requests stays in the ClientRequestFilter but is not part of a public API anymore and is registered by a Jersey OAuth [Feature](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Feature.html).

The most important parts of the Jersey client OAuth API and SPI are explained here:

* [OAuth1ClientSupport](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/client/oauth1/OAuth1ClientSupport.html): The main class which contains builder methods to build features that enable the OAuth 1 support. Start with this class every time you need to add any OAuth 1 support to the Jersey Client (build an Authorization flow or initialize client to perform authenticated requests). The class contains a static method that returns an instance of [OAuth1Builder](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/client/oauth1/OAuth1Builder.html) and also the class defines request properties to influence behaviour of the authenticated request support.
* [OAuth1AuthorizationFlow](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/client/oauth1/OAuth1AuthorizationFlow.html): API that allows to perform the Authorization flow against service provider. Implementation of this interface is a class that is used as a standalone utility and is not part of the JAX-RS client. In other words, this is not a feature that should be registered into the client.
* [AccessToken](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/client/oauth1/AccessToken.html), [ConsumerCredentials](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/client/oauth1/ConsumerCredentials.html" \t "_top): Interfaces that define Access Token classes and Consumer Credentials. Interfaces contain getters for public keys and secret keys of token and credentials.

An example of how Jersey OAuth 1 client API is used can be found in the [OAuth 1 Twitter Client Example](https://github.com/jersey/jersey/tree/2.19/examples/oauth-client-twitter). The following code snippets are extracted from the example and explain how to use the Jersey OAuth client API.

Before we start with any interaction with Twitter, we need to register our application on Twitter. See the example README.TXT file for the instructions. As a result of the registration, we get the consumer credentials that identify our application. Consumer credentials consist of consumer key and consumer secret.

As a first step in our code, we need to perform the authorization flow, where the user grants us an access to his/her Twitter client.

**Example 16.7. Build the authorization flow utility**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | ConsumerCredentials consumerCredentials = new ConsumerCredentials(                  "a846d84e68421b321a32d, "f13aed84190bc");  OAuth1AuthorizationFlow authFlow = OAuth1ClientSupport.builder(consumerCredentials)      .authorizationFlow(          "<http://api.twitter.com/oauth/request_token>",          "<http://api.twitter.com/oauth/access_token>",          "<http://api.twitter.com/oauth/authorize>")      .build(); |

Here we have built a OAuth1AuthorizationFlow utility component representing the OAuth 1 authorization flow, using OAuth1ClientSupport and OAuth1Builder API. The static builder method accepts mandatory parameter with ConsumerCredentials. These are credentials earlier issued by Twitter for our application. We have specified the Twitter OAuth endpoints where Request Token, Access Token will be retrieved and Authorization URI to which we will redirect the user in order to grant user's consent. Twitter will present an HTML page on this URI and it will ask the user whether he/she would like us to access his/her account.

Now we can proceed with the OAuth authorization flow.

**Example 16.8. Perform the OAuth Authorization Flow**

|  |  |
| --- | --- |
| 1  2  3  4 | String authorizationUri = authFlow.start();  // here we must direct the user to authorization uri to approve  // our application. The result will be verifier code (String).  AccessToken accessToken = authFlow.finish(verifier); |

In the first line, we start the authorization flow. The method internally makes a request to the http://api.twitter.com/oauth/request\_token URL and retrieves a Request Token. Details of this request can be found in the OAuth 1 specification. It then constructs a URI to which we must redirect the user. The URI is based on Twitter's authorization URI (http://api.twitter.com/oauth/authorize) and contains a Request Token as a query parameter. In the Twitter example, we have a simple console application therefore we print the URL to the console and ask the user to open the URL in a browser to approve the authorization of our application. Then the user gets a verifier and enters it back to the console. However, if our application would be a web application, we would need to return a redirection response to the user in order to redirect the user automatically to the authorizationUri. For more information about server deployment, check our [OAuth 2 Google Client Web Application Example](https://github.com/jersey/jersey/tree/2.19/examples/oauth2-client-google-webapp), where the client is part of the web application (the client API for OAuth 2 is similar to OAuth 1).

Once we have a verifier, we invoke the method finish() on our OAuth1AuthorizationFlow instance, which internally sends a request to an access token service URI (http://api.twitter.com/oauth/access\_token) and exchanges the supplied verifier for a new valid Access Token. At this point the authorization flow is finished and we can start using the retrieved AccessToken to make authenticated requests. We can now create an instance of an OAuth 1 client [Feature](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Feature.html) using OAuth1ClientSupport and pass it our accessToken. Another way is to use authFlow that already contains the information about access token to create the feature instance for us:

**Example 16.9. Authenticated requests**

|  |  |
| --- | --- |
| 1  2  3  4 | Feature feature = authFlow.getOAuth1Feature();  Client client = ClientBuilder.newBuilder()      .register(feature)      .build(); |

Once the feature is configured in the JAX-RS [Client](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/Client.html) (or [WebTarget](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/WebTarget.html" \t "_top)), all requests invoked from such Client (or WebTarget) instance will automatically include an OAuth Authorization HTTP header (that contains also the OAuth signature).

Note that if you already have a valid Access Token (for example stored in the database for each of your users), then you can skip the authorization flow steps and directly create the OAuth Feature configured to use your Access Token.

**Example 16.10. Build feature from Access Token**

|  |  |
| --- | --- |
| 1  2  3  4  5 | AccessToken storedToken = ...;  Feature filterFeature = OAuth1ClientSupport.builder(consumerCredentials)      .feature()      .accessToken(storedToken)      .build(); |

Here, the storedToken represents an [AccessToken](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/client/oauth1/AccessToken.html" \t "_top) that your client application keeps stored e.g. in a database.

Note that the OAuth feature builder API does not require the access token to be set. The reason for it is that you might want to build a feature which would register the internal Jersey OAuth ClientRequestFilter and other related providers but which would not initialize the OAuth providers with a single fixed [AccessToken](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/client/oauth1/AccessToken.html" \t "_top) instance. In such case you would need to specify a token for every single request in the request properties. Key names and API documentation of these properties can be found inOAuth1ClientSupport. Using this approach, you can have a single, OAuth enabled instance of a JAX-RS Client (or [WebTarget](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/WebTarget.html" \t "_top)) and use it to make authenticated requests on behalf of multiple users. Note that you can use the aforementioned request properties even if the feature has been initialized with an AccessToken to override the default access token information for particular requests, even though it is probably not a common use case.

The following code shows how to set an access token on a single request using the Jersey OAuth properties.

**Example 16.11. Specifying Access Token on a Request.**

|  |  |
| --- | --- |
| 1  2  3  4  5 | Response resp =      client.target("<http://my-serviceprovider.org/rest/foo/bar>")          .request()          .property(OAuth1ClientSupport.OAUTH\_PROPERTY\_ACCESS\_TOKEN, storedToken)          .get(); |

OAuth1AuthorizationFlow internally uses a Client instance to communicate with the OAuth server. For this a new client instance is automatically created by default. You can supply your instance of a Client to be used for the authorization flow requests (for performance and/or resource management reasons) using [OAuth1Builder](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/client/oauth1/OAuth1Builder.html) methods.

# Chapter 25. Jersey Test Framework

Jersey Test Framework originated as an internal tool used for verifying the correct implementation of server-side components.

Current implementation of Jersey Test Framework supports the following set of features:

* pre-configured client to access deployed application
* support for multiple containers - grizzly, in-memory, jdk, simple, jetty
* able to run against any external container
* automated configurable traffic logging

## 25.1. Basics

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class SimpleTest extends JerseyTest {        @Path("hello")      public static class HelloResource {          @GET          public String getHello() {              return "Hello World!";          }      }        @Override      protected Application configure() {          return new ResourceConfig(HelloResource.class);      }        @Test      public void test() {          final String hello = target("hello").request().get(String.class);          assertEquals("Hello World!", hello);      }  } |

If you want to develop a test using Jersey Test Framework, you need to subclass [JerseyTest](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/test/JerseyTest.html" \t "_top) and configure the set of resources and/or providers that will be deployed as part of the test application. This short code snippet shows basic resource class HelloResource used in tests defined as part of the SimpleTest class. The overridden configuremethod returns a [ResourceConfig](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/server/ResourceConfig.html" \t "_top) of the test application,that contains only the HelloResource resource class. ResourceConfig is a sub-class of JAX-RS [Application](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html). It is a Jersey convenience class for configuring JAX-RS applications. ResourceConfig also implements JAX-RS [Configurable](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Configurable.html) interface to make the application configuration more flexible.

## 25.2. Supported Containers

Following is a brief description of all container factories supported in Jersey Test Framework.

* Jersey provides 2 different test container factories based on Grizzly. The GrizzlyTestContainerFactory creates a container that can run as a light-weight, plain HTTP container. Almost all Jersey tests are using Grizzly HTTP test container factory. Second factory is GrizzlyWebTestContainerFactory that is Servlet-based and supports Servlet deployment context for tested applications. This factory can be useful when testing more complex Servlet-based application deployments.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>      <groupId>org.glassfish.jersey.test-framework.providers</groupId>      <artifactId>jersey-test-framework-provider-grizzly2</artifactId>      <version>2.19</version>  </dependency> |

* In-Memory container is not a real container. It starts Jersey application and directly calls internal APIs to handle request created by client provided by test framework. There is no network communication involved. This containers does not support servlet and other container dependent features, but it is a perfect choice for simple unit tests.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>      <groupId>org.glassfish.jersey.test-framework.providers</groupId>      <artifactId>jersey-test-framework-provider-inmemory</artifactId>      <version>2.19</version>  </dependency> |

* HttpServer from Oracle JDK is another supported test container.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>      <groupId>org.glassfish.jersey.test-framework.providers</groupId>      <artifactId>jersey-test-framework-provider-jdk-http</artifactId>      <version>2.19</version>  </dependency> |

* Simple container (org.simpleframework.http) is another light-weight HTTP container that integrates with Jersey and is supported by Jersey Test Framework.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>      <groupId>org.glassfish.jersey.test-framework.providers</groupId>      <artifactId>jersey-test-framework-provider-simple</artifactId>      <version>2.19</version>  </dependency> |

* Jetty container (org.eclipse.jetty) is another high-performance, light-weight HTTP server that integrates with Jersey and is supported by Jersey Test Framework.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>      <groupId>org.glassfish.jersey.test-framework.providers</groupId>      <artifactId>jersey-test-framework-provider-jetty</artifactId>      <version>2.19</version>  </dependency> |

## 25.3. Running TestNG Tests

It is possible to run not only JUnit tests but also tests based on TestNG. In order to do this you need to make sure the following 2 steps are fulfilled:

* Extend [JerseyTestNg](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/test/JerseyTestNg.html" \t "_top), or one of it's inner classes [JerseyTestNg.ContainerPerClassTest](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/test/JerseyTestNg.ContainerPerClassTest.html) / [JerseyTestNg.ContainerPerMethodTest](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/test/JerseyTestNg.ContainerPerMethodTest.html), instead of [JerseyTest](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/test/JerseyTest.html" \t "_top).
* Add TestNG to your class-patch, i.e.:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <dependency>      <groupId>org.glassfish.jersey.test-framework</groupId>      <artifactId>jersey-test-framework-core</artifactId>      <version>2.19</version>  </dependency>  <dependency>      <groupId>org.testng</groupId>      <artifactId>testng</artifactId>      <version>...</version>  </dependency> |

JUnit creates a new instance of a test class for every test present in that class which, from the point of view of Jersey Test Framework, means that new test container and client is created for each test of a test class. However, TestNG creates only one instance of a test class and the initialization of the test container depends more on setup/teardown methods (driven by @BeforeXXX and @AfterXXXannotations) than in JUnit. This means that, basically, you can start one instance of test container for all tests present in a test class or separate test container for each and every test. For this reason a separate subclasses of [JerseyTestNg](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/test/JerseyTestNg.html" \t "_top) have been created:

* [JerseyTestNg.ContainerPerClassTest](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/test/JerseyTestNg.ContainerPerClassTest.html) creates one container to run all the tests in. Setup method is annotated with @BeforeClass, teardown method with @AfterClass.

For example take a look at ContainerPerClassTest test. It contains two test methods (first and second), one singleton resource that returns an increasing sequence of number. Since we spawn only one instance of a test container for the whole class the value expected in the first test is 1 and in the second it's 2.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | public class ContainerPerClassTest extends JerseyTestNg.ContainerPerClassTest {        @Path("/")      @Singleton      @Produces("text/plain")      public static class Resource {            private int i = 1;            @GET          public int get() {              return i++;          }      }        @Override      protected Application configure() {          return new ResourceConfig(Resource.class);      }        @Test(priority = 1)      public void first() throws Exception {          test(1);      }        @Test(priority = 2)      public void second() throws Exception {          test(2);      }        private void test(final Integer expected) {          final Response response = target().request().get();            assertEquals(response.getStatus(), 200);          assertEquals(response.readEntity(Integer.class), expected);      }  } |

* [JerseyTestNg.ContainerPerMethodTest](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/test/JerseyTestNg.ContainerPerMethodTest.html) creates separate container for each test. Setup method is annotated with @BeforeMethod, teardown method with@AfterMethod.

We can create a similar test to the previous one. Take a look at ContainerPerMethodTest test. It looks the same except the expected values and extending class: it contains two test methods (first and second), one singleton resource that returns an increasing sequence of number. In this case we create a separate test container for each test so value expected in the first test is 1 and in the second it's also 1.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | public class ContainerPerMethodTest extends JerseyTestNg.ContainerPerMethodTest {        @Path("/")      @Singleton      @Produces("text/plain")      public static class Resource {            private int i = 1;            @GET          public int get() {              return i++;          }      }        @Override      protected Application configure() {          return new ResourceConfig(Resource.class);      }        @Test      public void first() throws Exception {          test(1);      }        @Test      public void second() throws Exception {          test(1);      }        private void test(final Integer expected) {          final Response response = target().request().get();            assertEquals(response.getStatus(), 200);          assertEquals(response.readEntity(Integer.class), expected);      }  } |

If you need more complex setup of your test you can achieve this by directly extending the [JerseyTestNg](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/test/JerseyTestNg.html" \t "_top) class create setup/teardown methods suited to your needs and provide a strategy for storing and handling a test container / client instance (see JerseyTestNg.configureStrategy(TestNgStrategy) method).

## 25.4. Advanced features

### 25.4.1. JerseyTest Features

JerseyTest provide [enable(...)](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/test/JerseyTest.html#enable(java.lang.String)), [forceEnable(...)](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/test/JerseyTest.html#forceEnable(java.lang.String)) and [disable(...)](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/test/JerseyTest.html#disable(java.lang.String)) methods, that give you control over configuring values of the properties defined and described in theTestProperties class. A typical code that overrides the default property values is listed below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class SimpleTest extends JerseyTest {      // ...        @Override      protected Application configure() {          enable(TestProperties.LOG\_TRAFFIC);          enable(TestProperties.DUMP\_ENTITY);            // ...        }  } |

The code in the example above enables test traffic logging (inbound and outbound headers) as well as dumping the HTTP message entity as part of the traffic logging.

### 25.4.2. External container

Complicated test scenarios may require fully started containers with complex setup configuration, that is not easily doable with current Jersey container support. To address these use cases, Jersey Test Framework providers general fallback mechanism - an External Test Container Factory. Support of this external container "wrapper" is provided as the following module:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>      <groupId>org.glassfish.jersey.test-framework.providers</groupId>      <artifactId>jersey-test-framework-provider-external</artifactId>      <version>2.19</version>  </dependency> |

As indicated, the "container" exposed by this module is just a wrapper or stub, that redirects all request to a configured host and port. Writing tests for this container is same as for any other but you have to provide the information about host and port during the test execution:

mvn test -Djersey.test.host=myhost.org -Djersey.config.test.container.port=8080

## 25.5. Parallel Testing with Jersey Test Framework

For a purpose of running multiple test containers in parallel you need to set the [TestProperties.CONTAINER\_PORT](https://jersey.java.net/apidocs/2.19/jersey/org/glassfish/jersey/test/TestProperties.CONTAINER_PORT.html" \t "_top) to 0 value. This will tell Jersey Test Framework (and the underlying test container) to use the first available port.

You can set the value as a system property (via command line option) or directly in the test (to not affect ports of other tests):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Override      protected Application configure() {          // Find first available port.          forceSet(TestProperties.CONTAINER\_PORT, "0");            return new ResourceConfig(Resource.class);      } |

The easiest way to setup your JUnit or TestNG tests to run in parallel is to configure Maven Surefire plugin. You can do this via configuration options parallel andthreadCount, i.e.:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | ...  <configuration>      <parallel>methods</parallel>      <threadCount>5</threadCount>      ...  </configuration>  ... |

## 7.1. Representations and Java Types

* All media types (\*/\*)
  + byte[]
  + java.lang.String
  + java.io.Reader (inbound only)
  + java.io.File
  + javax.activation.DataSource
  + javax.ws.rs.core.StreamingOutput (outbound only)
* XML media types (text/xml, application/xml and application/...+xml)
  + javax.xml.transform.Source
  + javax.xml.bind.JAXBElement
  + Application supplied JAXB classes (types annotated with [@XmlRootElement](http://docs.oracle.com/javase/6/docs/api/javax/xml/bind/annotation/XmlRootElement.html) or[@XmlType](http://docs.oracle.com/javase/6/docs/api/javax/xml/bind/annotation/XmlType.html))
* Form content (application/x-www-form-urlencoded)
  + MultivaluedMap<String,String>
* Plain text (text/plain)
  + java.lang.Boolean
  + java.lang.Character
  + java.lang.Number

**Example 7.1. Using File with a specific media type to produce a response**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | @GET  @Path("/images/{image}")  @Produces("image/\*")  public Response getImage(@PathParam("image") String image) {    File f = new File(image);      if (!f.exists()) {      throw new WebApplicationException(404);    }      String mt = new MimetypesFileTypeMap().getContentType(f);    return Response.ok(f, mt).build();  } |

## 7.2. Building Responses

Sometimes it is necessary to return additional information in response to a HTTP request. Such information may be built and returned using [Response](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Response.html) and[Response.ResponseBuilder](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Response.ResponseBuilder.html). For example, a common RESTful pattern for the creation of a new resource is to support a POST request that returns a 201 (Created) status code and a Location header whose value is the URI to the newly created resource. This may be achieved as follows:

**Example 7.2. Returning 201 status code and adding Location header in response to POST request**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @POST  @Consumes("application/xml")  public Response post(String content) {    URI createdUri = ...    create(content);    return Response.created(createdUri).build();  } |

In the above no representation produced is returned, this can be achieved by building an entity as part of the response as follows:

**Example 7.3. Adding an entity body to a custom response**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @POST  @Consumes("application/xml")  public Response post(String content) {    URI createdUri = ...    String createdContent = create(content);    return Response.created(createdUri).entity(Entity.text(createdContent)).build();  } |

## 7.3. WebApplicationException and Mapping Exceptions to Responses

Previous section shows how to return HTTP responses, that are built up programmatically. It is possible to use the very same mechanism to return HTTP errors directly, e.g. when handling exceptions in a try-catch block. However, to better align with the Java programming model, JAX-RS allows to define direct mapping of Java exceptions to HTTP error responses.

The following example shows throwing CustomNotFoundException from a resource method in order to return an error HTTP response to the client:

**Example 7.4. Throwing exceptions to control response**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Path("items/{itemid}/")  public Item getItem(@PathParam("itemid") String itemid) {    Item i = getItems().get(itemid);    if (i == null) {      throw new CustomNotFoundException("Item, " + itemid + ", is not found");    }      return i;  } |

This exception is an application specific exception that extends [WebApplicationException](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/WebApplicationException.html" \t "_top) and builds a HTTP response with the 404 status code and an optional message as the body of the response:

**Example 7.5. Application specific exception implementation**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class CustomNotFoundException extends WebApplicationException {      /\*\*    \* Create a HTTP 404 (Not Found) exception.    \*/    public CustomNotFoundException() {      super(Responses.notFound().build());    }      /\*\*    \* Create a HTTP 404 (Not Found) exception.    \* @param message the String that is the entity of the 404 response.    \*/    public CustomNotFoundException(String message) {      super(Response.status(Responses.NOT\_FOUND).      entity(message).type("text/plain").build());    }  } |

In other cases it may not be appropriate to throw instances of [WebApplicationException](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/WebApplicationException.html" \t "_top), or classes that extend [WebApplicationException](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/WebApplicationException.html" \t "_top), and instead it may be preferable to map an existing exception to a response. For such cases it is possible to use a custom exception mapping provider. The provider must implement the [ExceptionMapper<E extends Throwable>](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/ExceptionMapper.html" \t "_top) interface. For example, the following maps the [EntityNotFoundException](http://docs.oracle.com/javaee/5/api/javax/persistence/EntityNotFoundException.html" \t "_top) to a HTTP 404 (Not Found) response:

**Example 7.6. Mapping generic exceptions to responses**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Provider  public class EntityNotFoundMapper implements ExceptionMapper<javax.persistence.EntityNotFoundException> {    public Response toResponse(javax.persistence.EntityNotFoundException ex) {      return Response.status(404).        entity(ex.getMessage()).        type("text/plain").        build();    }  } |

The above class is annotated with [@Provider](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/Provider.html), this declares that the class is of interest to the JAX-RS runtime. Such a class may be added to the set of classes of the [Application](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html" \t "_top)instance that is configured. When an application throws an [EntityNotFoundException](http://docs.oracle.com/javaee/6/api/javax/persistence/EntityNotFoundException.html" \t "_top) the toResponse method of the EntityNotFoundMapper instance will be invoked.

## 7.4. Conditional GETs and Returning 304 (Not Modified) Responses

Conditional GETs are a great way to reduce bandwidth, and potentially improve on the server-side performance, depending on how the information used to determine conditions is calculated. A well-designed web site may for example return 304 (Not Modified) responses for many of static images it serves.

JAX-RS provides support for conditional GETs using the contextual interface [Request](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Request.html).

The following example shows conditional GET support:

**Example 7.7. Conditional GET support**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public SparklinesResource(    @QueryParam("d") IntegerList data,    @DefaultValue("0,100") @QueryParam("limits") Interval limits,    @Context Request request,    @Context UriInfo ui) {    if (data == null) {      throw new WebApplicationException(400);    }      this.data = data;    this.limits = limits;      if (!limits.contains(data)) {      throw new WebApplicationException(400);    }      this.tag = computeEntityTag(ui.getRequestUri());      if (request.getMethod().equals("GET")) {      Response.ResponseBuilder rb = request.evaluatePreconditions(tag);      if (rb != null) {        throw new WebApplicationException(rb.build());      }    }  } |

The constructor of the SparklinesResouce root resource class computes an entity tag from the request URI and then calls the [request.evaluatePreconditions](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Request.html" \l "evaluatePreconditions(javax.ws.rs.core.EntityTag)" \t "_top) with that entity tag. If a client request contains an If-None-Match header with a value that contains the same entity tag that was calculated then the [evaluatePreconditions](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Request.html" \l "evaluatePreconditions(javax.ws.rs.core.EntityTag)" \t "_top) returns a pre-filled out response, with the 304 status code and entity tag set, that may be built and returned. Otherwise, [evaluatePreconditions](http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Request.html" \l "evaluatePreconditions(javax.ws.rs.core.EntityTag)" \t "_top) returns null and the normal response can be returned.

Notice that in this example the constructor of a resource class is used to perform actions that may otherwise have to be duplicated to invoked for each resource method. The life cycle of resource classes is per-request which means that the resource instance is created for each request and therefore can work with request parameters and for example make changes to the request processing by throwing an exception as it is shown in this example.

# 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>

<https://github.com/jersey/jersey/tree/2.19/examples/oauth-client-twitter>

<https://jersey.java.net/documentation/latest/user-guide.html>