**RESTful Web Service - JAX-RS Annotations - Contents:**

|  |  |
| --- | --- |
| **Annotation** | **Package Detail/Import statement** |
| [@GET](http://www.techferry.com/articles/RESTful-web-services-JAX-RS-annotations.html#GET) | import javax.ws.rs.GET; |
| [@Produces](http://www.techferry.com/articles/RESTful-web-services-JAX-RS-annotations.html#Produces) | import javax.ws.rs.Produces; |
| [@Path](http://www.techferry.com/articles/RESTful-web-services-JAX-RS-annotations.html#Path) | import javax.ws.rs.Path; |
| [@PathParam](http://www.techferry.com/articles/RESTful-web-services-JAX-RS-annotations.html#PathParam) | import javax.ws.rs.PathParam; |
| [@QueryParam](http://www.techferry.com/articles/RESTful-web-services-JAX-RS-annotations.html#QueryParam) | import javax.ws.rs.QueryParam; |
| [@POST](http://www.techferry.com/articles/RESTful-web-services-JAX-RS-annotations.html#POST) | import javax.ws.rs.POST; |
| [@Consumes](http://www.techferry.com/articles/RESTful-web-services-JAX-RS-annotations.html#Consumes) | import javax.ws.rs.Consumes; |
| [@FormParam](http://www.techferry.com/articles/RESTful-web-services-JAX-RS-annotations.html#FormParam) | import javax.ws.rs.FormParam; |
| [@PUT](http://www.techferry.com/articles/RESTful-web-services-JAX-RS-annotations.html#PUT) | import javax.ws.rs.PUT; |
| [@DELETE](http://www.techferry.com/articles/RESTful-web-services-JAX-RS-annotations.html#DELETE) | import javax.ws.rs.DELETE; |

As stated earlier in [Example Application](http://www.techferry.com/articles/JEE-annotations.html#exampleApp), we are using Jersey for RESTful Web services and JAX-RS annotations. 

|  |  |
| --- | --- |
| Jersey Annotation Tip | REST follows one-to-one mapping between create, read, update, and delete (CRUD) operations and HTTP methods.   * To create a resource on the server, use POST. * To retrieve a resource, use GET. * To change the state of a resource or to update it, use PUT. * To remove or delete a resource, use DELETE. |

**@GET**

Annotate your Get request methods with @GET.

|  |  |
| --- | --- |
| 1  2  3  4 | @GET  public String getHTML() {    ...  } |

**@Produces**

@Produces annotation specifies the type of output this method (or web service) will produce.

|  |  |
| --- | --- |
| 1  2  3  4  5 | @GET  @Produces("application/xml")  public Contact getXML() {    ...  } |
| 1  2  3  4  5 | @GET  @Produces("application/json")  public Contact getJSON() {    ...  } |

**@Path**

@Path annotation specify the URL path on which this method will be invoked.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @GET  @Produces("application/xml")  @Path("xml/{firstName}")  public Contact getXML() {    ...  } |

**@PathParam**

We can bind REST-style URL parameters to method arguments using @PathParam annotation as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @GET  @Produces("application/xml")  @Path("xml/{firstName}")  public Contact getXML(@PathParam("firstName") String firstName) {    Contact contact = contactService.findByFirstName(firstName);    return contact;  } |
| 1  2  3  4  5  6  7 | @GET  @Produces("application/json")  @Path("json/{firstName}")  public Contact getJSON(@PathParam("firstName") String firstName) {    Contact contact = contactService.findByFirstName(firstName);    return contact;  } |

**@QueryParam**

Request parameters in query string can be accessed using @QueryParam annotation as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @GET  @Produces("application/json")  @Path("json/companyList")  public CompanyList getJSON(@QueryParam("start") int start, @QueryParam("limit") int limit) {    CompanyList list = new CompanyList(companyService.listCompanies(start, limit));    return list;  } |

The example above returns a list of companies (with server side pagination) which can be displayed with rich clients implemented using Ext-js or jQuery. You can read more more about setting up [ExtJS grid panel with remote sorting and pagination using Hibernate](http://blog.techferry.com/2012/01/25/extjs-grid-panel-with-remote-sorting-and-pagination-using-hibernate/).

**@POST**

Annotate POST request methods with @POST.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @POST  @Consumes("application/json")  @Produces("application/json")  public RestResponse<Contact> create(Contact contact) {  ...  } |

**@Consumes**

The @Consumes annotation is used to specify the MIME media types a REST resource can consume.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @PUT  @Consumes("application/json")  @Produces("application/json")  @Path("{contactId}")  public RestResponse<Contact> update(Contact contact) {  ...  } |

**@FormParam**

The REST resources will usually consume XML/JSON for the complete Entity Bean. Sometimes, you may want to read parameters sent in POST requests directly and you can do that using @FormParam annotation. GET Request query parameters can be accessed using[@QueryParam](http://www.techferry.com/articles/RESTful-web-services-JAX-RS-annotations.html#QueryParam) annotation.

|  |  |
| --- | --- |
| 1  2  3  4  5 | @POST  public String save(@FormParam("firstName") String firstName,      @FormParam("lastName") String lastName) {        ...    } |

**@PUT**

Annotate PUT request methods with @PUT.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @PUT  @Consumes("application/json")  @Produces("application/json")  @Path("{contactId}")  public RestResponse<Contact> update(Contact contact) {  ...  } |

**@DELETE**

Annotate DELETE request methods with @DELETE.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @DELETE  @Produces("application/json")  @Path("{contactId}")  public RestResponse<Contact> delete(@PathParam("contactId") int contactId) {  ...  } |

-------------------=====================

In the first [RESTEasy tutorial](http://www.mastertheboss.com/jboss-frameworks/resteasy/resteasy-tutorial) [RESTEasy tutorial](http://www.mastertheboss.com/web-interfaces/273-resteasy-tutorial-.html" \t "_blank) we have learnt the basics about REST Web services and we have tested a simple RESTful Web service. In this tutorial we will show how to inject web application elements (form parameters, query parameters and more) into a RESTful Web service.

You can use the following annotations to bind HTTP requests to a RESTful web service:

@FormParam

@PathParam

@QueryParam

@HeaderParam

@CookieParam

@MatrixParam

Let's explore all the possible interactions.

**@FormParam**

The annotation **@FormParam** can be used to inject the parameters of a Web form into a RESTful Web service.

Here's an example:



Here we are submitting a POST request containing two parameters email and password which are translated into the parameters "e" and "p" of the login method.

Here's the full example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <form method="POST" action="login">      Email Address: <input type="text" name="email">  <br>  Password: <input type="text" name="password">  <input type="submit">      </form> |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @Path("/")  public class LoginService  {        @Path("login")    @POST    public String login(@FormParam("email") String e, @FormParam("password") String p) {     return "Logged with " + e + " " + p;    }    } | |

As an alternative, you can bind the parameters email and password at class level, which can be useful if you need to re-use the same parameters across different methods of the service.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class User {      @FormParam("email")    private String email;    @FormParam("password")    private String password;    } |

You would need to modify the REST method accordingly:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @POST   @Path("login")   public String login(@Form User form) {   return "Logged with " + form.email + " " + form.password;   } |

**@PathParam**

The @PathParam annotation binds the value of a path segment to a resource method parameter. For example, the following method would intercept an HTTP GET like **http://server:port/login/12345** and convert the PathParam "12345" into the String "id"

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @Path("/")  public class LoginService  {        @GET    @Path("login/{zip}")    public String login(@PathParam("zip") String id) {     return "Id is " +id;    }    } |

As for @FormParam, you can embed the @PathParam declaration at class level, if you prefer.

**@QueryParam**

The @QueryParam annotation binds the value of a path segment to a resource method parameter. For example, the following method would intercept an HTTP GET like [http://server:port/login?zip=12345](http://serverport/) and inject the query parameter "zip" into the method parameter "zip"

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @Path("/")  public class LoginService  {     @GET   @Path("login/{zip}")    public String login(@QueryParam("zip") String zip) {     return "Id is " +id;    }    } |

QueryParam can be convenientely used with the DefaultValue annotation so that you can avoid a null pointer exception if no query parameter is passed.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @GET   @Path("login/{zip}")    public String login(@DefaultValue("11111") @QueryParam("zip") String zip) {   return "Id is " +id;   } |

As for @FormParam, you can embed the @PathParam declaration at class level, if you prefer.

**@HeaderParam**

The @HeaderParam annotation extracts information from the HTTP header and binds it to a method parameter. Example:

|  |  |
| --- | --- |
| 1  2  3  4 | @GET  public String callService(@HeaderParam("User-Agent") String whichBrowser) {  ...  } |

**@CookieParam**

The @CookieParam annotation reads an information stored as a cookie and binds it to a method parameter. Example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @GET    public String callService(@CookieParam("sessionid") String sessionid) {    ...   } |

**@MatrixParam**

The @MatrixParam annotation can be used to bind an expression containing several property=value to a method parameter. For example, supposing you were to invoke an URL like[http://server:port/login](http://serverport/);name=francesco;surname=marchioni

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @GET    public String callService(@MatrixParam("name") String name,                                    @MatrixParam("surname") String surname) {    ...   } |

For Attachment

<http://cxf.apache.org/docs/jax-rs-multiparts.html#JAX-RSMultiparts-MultipartannotationandOptionalattachments>

Request & Response Details of any RestWebservice :

Below details will print only if you enable logging as feature

ID: 2

Address: http://localhost:8080/RestDemo/services/rest/UserManager/hello

Http-Method: GET

Content-Type:

Headers: {Accept=[text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8], accept-encoding=[gzip, deflate, sdch], accept-language=[en-US,en;q=0.8], cache-control=[max-age=0], connection=[keep-alive], Content-Type=[null], cookie=[JSESSIONID=9BD7FED01ACF82F0519F6A681F746DB9], host=[localhost:8080], user-agent=[Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/43.0.2357.134 Safari/537.36]}

--------------------------------------

Nov 24, 2015 9:35:16 PM org.apache.cxf.interceptor.LoggingOutInterceptor

INFO: Outbound Message

---------------------------

ID: 2

Response-Code: 200

Content-Type: application/xml

Headers: {Content-Type=[application/xml], Date=[Tue, 24 Nov 2015 16:05:16 GMT]}

Payload: <?xml version="1.0" encoding="UTF-8" standalone="yes"?><UserResponse><users><id>10</id></users><errorMessage>Hello</errorMessage><success>true</success></UserResponse>

From HTTP 1.1 :

**ExMPLE uri** :

"http://www.example.com/hello.txt":

Client request:

GET /hello.txt HTTP/1.1

User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3

Host: www.example.com

Accept-Language: en, mi

Server response:

HTTP/1.1 200 OK

Date: Mon, 27 Jul 2009 12:28:53 GMT

Server: Apache

Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT

ETag: "34aa387-d-1568eb00"

Accept-Ranges: bytes

Content-Length: 51

Vary: Accept-Encoding

Content-Type: text/plain

Hello World! My payload includes a trailing CRLF

----------------------

JAX-RS provides the ***@Context*** annotation to inject a variety of resources in your RESTful services. Some of the most commonly injected components are HTTP headers, HTTP URI related information. Here is a complete list (in no specific order)

* HTTP headers
* HTTP URI details
* Security Context
* Resource Context
* Request
* Configuration
* Application
* Providers

UriInfo, SecurityContext, HttpHeaders, Providers, Request, ContextResolver, Servlet types (HttpServletRequest, HttpServletResponse, ServletContext, ServletConfig) can be injected.

Lets look at these one by one with the help of examples

**HTTP headers**

Although HTTP headers can be injected using the @HeaderParam annotation, JAX-RS also provides the facility of injecting an instance of the ***HttpHeaders*** interface (as an instance variable or method parameter). This is useful when you want to iterate over all possible headers rather than injecting a specific header value by name

@Path("testinject")

public class InjectURIDetails{

//localhost:8080/<root-context>/testinject/httpheaders

@GET

@Path("httpheaders")

public void test(@Context HttpHeaders headers){

System.out.println("ALL headers -- "+ headers.getRequestHeaders().toString());

System.out.println("'Accept' header -- "+ headers.getHeaderString("Accept"));

System.out.println("'TestCookie' value -- "+ headers.getCookies().get("TestCookie").getValue());

}

}

**HTTP URI details**

UriInfo is another interface whose instance can be injected by JAX-RS (as an instance variable or method parameter). Use this instance to fetch additional details related to the request URI and its parameters (query, path)

@Path("testinject")

public class InjectURIDetails{

//localhost:8080/<root-context>/testinject/uriinfo

@GET

@Path("uriinfo")

public void test(@Context UriInfo uriDetails){

System.out.println("ALL query parameters -- "+ uriDetails.getQueryParameters().toString());

System.out.println("'id' query parameter -- "+ uriDetails.getQueryParameters.get("id"));

System.out.println("Complete URI -- "+ uriDetails.getRequestUri());

}

}

**Providers**

An instance of the Providers interface can be injected using @Context. One needs to be aware of the fact that this is only valid within an existing provider. A Providers instance enables the current Provider to search for other registered providers in the current JAX-RS container.

**Note**: Please do not get confused between Provider and Providers.

**Provider**

* A JAX-RS Provider is a is generic term for any class which supplements/extends the JAX-RS features by implementing standard interfaces exposed by the JAX-RS specification
* It is annotated using the @Provider annotation for automatic discovery by the run time
* Examples of JAX-RS providers are – Message Body Reader, Message Body Writer, Exception Mapper and Context Providers.

**Providers**

Refers to the (injectable) javax.ws.rs.ext.Providers interface which was discussed in this sub section

**Security Context**

Inject an instance of the javax.ws.rs.core.SecurityContext interface (as an instance variable or method parameter) if you want to gain more insight into identity of the entity invoking your RESTful service. This interface exposes the following information

* Instance of java.security.Principal representing the caller
* Whether or not the user if a part of a specific role
* Which authentication scheme is being used (BASIC/FORM/DIGEST/CERT)
* Whether or not the request invoked over HTTPS

@Path("testinject")

public class InjectSecurityContext{

//localhost:8080/<root-context>/testinject/securitycontext

@GET

@Path("securitycontext")

public void test(@Context SecurityContext secContext){

System.out.println("Caller -- "+ secContext.getUserPrincipal()getName());

System.out.println("Authentication Scheme -- "+ secContext.getAuthenticationScheme());

System.out.println("Over HTTPS ? -- "+ secContext.isSecure());

System.out.println("Belongs to 'admin' role? -- "+ secContext.isUserInRole("admin");

}

}

# Context annotations

A number of context types can be injected as parameters, in fields or through dedicated methods.  
UriInfo, SecurityContext, HttpHeaders, Providers, Request, ContextResolver, Servlet types (HttpServletRequest, HttpServletResponse, ServletContext, ServletConfig) can be injected.

A CXF-specific composite context interface, [MessageContext](http://svn.apache.org/repos/asf/cxf/trunk/rt/frontend/jaxrs/src/main/java/org/apache/cxf/jaxrs/ext/MessageContext.java) is also supported which makes it easier to deal with all the supported JAX-RS contexts (and indeed with future ones) and also lets us check the current message's properties.

Example:

|  |
| --- |
| @Path("/customer")  public class CustomerService {        @Context      private org.apache.cxf.jaxrs.ext.MessageContext mc;      @Context      private ServletContext sc;      private UriInfo ui;        @Context      public void setUriInfo(UriInfo ui) {          this.ui = ui;      }        @PUT      public Response updateCustomer(@Context HttpHeaders h, Customer c) {          mc.getHttpHeaders();      }  } |

Note that all types of supported JAX-RS providers such as MessageBodyWriter, MessageBodyReader, ExceptionMapper and ContextResolver, as well as the list of body providers which can be provided by Providers can have contexts injected too. The only exception is that no parameter level injection is supported for providers due to methods of JAXRS providers being fixed.

Note that Providers and ContextResolver are likely to be of interest to message body providers rather than to the actual application code. You can also inject all the context types into @Resource annotated fields.

## Custom Contexts

Registering a custom [ContextProvider](http://svn.apache.org/repos/asf/cxf/trunk/rt/frontend/jaxrs/src/main/java/org/apache/cxf/jaxrs/ext/ContextProvider.java) implementation such as [SearchContextProvider](http://svn.apache.org/repos/asf/cxf/trunk/rt/rs/extensions/search/src/main/java/org/apache/cxf/jaxrs/ext/search/SearchContextProvider.java) lets attach Context annotations to arbitrary classes which can be helpful when some of the information representing the current request needs to be optimized or specialized, example:

|  |
| --- |
| package resources;  import org.apache.cxf.jaxrs.ext.search.SearchContext;  @Path("/")  public class RootResource {      @Context      private  SearchContext sc;      // the rest of the code  } |

and

|  |
| --- |
| <jaxrs:server>    <serviceBeans>      <bean class="resources.RootResource"/>    </serviceBeans>    <jaxrs:providers>      <bean class="org.apache.cxf.jaxrs.ext.search.SearchContextProvider"/>    </jaxrs:providers>  </jaxrs:server> |

Custom Context implementations may get all the information about the HTTP request from the current CXF message.

# Exception handling

One can either throw an unchecked WebApplicationException or return Response with a proper error code set.  
The former option may be a better one when no JAX-RS types can be added to method signatures.

For example :

|  |
| --- |
| @Path("/customerservice/")  public class CustomerService {       @PUT      @Path("/customers/{id}")      public Response updateCustomer(@PathParam("id") Long id, Customer customer) {          return Response.status(errorCode).build();      }        @POST      @Path("/customers")      public Customer addCustomer(Customer customer) {          throw new WebApplicationException(errorCode);      }    } |

Yet another option is to register an ExceptionMapper provider. Ex :

|  |
| --- |
| public BookExceptionMapper implements ExceptionMapper<BookException> {      public Response toResponse(BookException ex) {          // convert to Response      }  } |

This allows for throwing a checked or runtime exception from an application code and map it to an HTTP response in a registered provider.

Have a look please at [this exception mapper](http://svn.apache.org/repos/asf/cxf/trunk/systests/jaxrs/src/test/java/org/apache/cxf/systest/jaxrs/security/SecurityExceptionMapper.java) which converts Spring Security exceptions into HTTP 403 error code for another example.

Note that when no mappers are found for custom exceptions, they are propagated to the underlying container as required by the specification where they will typically be wrapped in ServlerException, eventually resulting in HTTP 500 status being returned by default. Thus one option for intercepting the exceptions is to register a custom servlet filter which will catch ServletExceptions and handle the causes.

This propagation can be disabled by registering a boolean jaxrs property 'org.apache.cxf.propagate.exception' with a false value. If such property is set and no exception mapper can be found for a given exception then it will be wrapped into an xml error response by the CXF [XMLFaultOutInterceptor](http://svn.apache.org/repos/asf/cxf/trunk/rt/bindings/xml/src/main/java/org/apache/cxf/binding/xml/interceptor/XMLFaultOutInterceptor.java).

One can also register a custom CXF out fault interceptor which can handle all the exceptions by writing directly to the HttpServletResponse stream or XMLStreamWriter (as XMLFaultOutInterceptor does). For example, see this [test interceptor](http://svn.apache.org/repos/asf/cxf/trunk/systests/jaxrs/src/test/java/org/apache/cxf/systest/jaxrs/CustomOutFaultInterceptor.java).

The CXF Exception Handler class lives in the client.

|  |  |
| --- | --- |
| ExceptionHandler.java | |
| 01  02  03  04  05  06  07  08  09  10  11 | import org.apache.cxf.jaxrs.client.ResponseExceptionMapper;    import javax.ws.rs.core.Response;    public class ExceptionHandler implements ResponseExceptionMapper {        @Override      public Throwable fromResponse(Response response) {          throw new InvalidUserException(response.getHeaderString("exception"));      }  } |

This implements ResponseExceptionMapper which will give access to the Response object. Once this handler is added to the provider list it will be invoked for any 500 response for the service call.

|  |  |
| --- | --- |
| applicationContext-cxf.xml | |
| 01  02  03  04  05  06  07  08  09 | <!-- rest container -->  <jaxrs:client id="sampleServiceREST"                serviceClass="com.luckyryan.sample.ws.SampleServiceREST"                address="http://localhost:8090/services/rest">      <jaxrs:providers>          <bean class="org.codehaus.jackson.jaxrs.JacksonJsonProvider"/>          <bean class="com.luckyryan.sample.exception.ExceptionHandler"/>      </jaxrs:providers>  </jaxrs:client> |

That’s it! Now the application

# Dealing with Parameters

PathParam annotation is used to map a given Path template variable to a method parameter.  
For example :

|  |
| --- |
| @Path("/customer/{id}")  public class CustomerService {          @PUT      @Path("{name}")      public Response updateCustomer(@PathParam("id") Long id, @PathParam("name") String name) {          ...      }  } |

In this case a template variable id available from a root class annotation is mapped to a parameter of type Long, while a name variable is mapped to a parameter of type String.

@QueryParam, @HttpHeader, @MatrixParam, @FormParam and @CookieParam annotations are also supported.

Note that the parameters, marked with @FormParam annotation, can take the values from the query parameters in case, if request body is already consumed. This is defined in JAX-RS specification due to the filters (Spring security, etc) consuming the body and thus JAX-RS form parameters becoming empty. User can optionally deactivate standard behavior through setting "set.form.parameters.from.http.parameters" message property to false.

Parameters can be of type String or of any type that have constructors accepting a String parameter or stat ic valueOf(String s) methods.   
Additionally CXF JAXRS checks for static fromString(String s) method, so types with no valueOf(String) factory methods can also be dealt with:

|  |
| --- |
| public enum Gender {     MALE,     FEMALE;       public static Gender fromString(String s) {         if ("1".equals(s)) {             return FEMALE;         } else if ("1".equals(s)) {             return MALE;         }         return valueOf(s);     }  }    @Path("/{g}")  public class Service {          @PUT      @Path("{id}")      public Response update(@PathParam("g") Gender g, @PathParam("id") UUID u) {          ...      }  } |

Note that on the trunk enums with fromValue() factory methods are also supported.

JAX-RS PathSegment is also supported. A sequence of identically named parameters (queries, headers, etc) can be mapped to List or Set or SortedSet.

CXF JAXRS supports ParameterHandler extensions which can be used to deal with method parameters annotated with one of the JAXRS parameter annotations :

|  |
| --- |
| public class MapHandler implements ParameterHandler<Map> {      public Map fromString(String s) {...}  }    @Path("/map")  public class Service {          @PUT      @Path("/{mapvalue:(.)+}")      public Response update(@PathParam("mapvalue") Map m, byte[] bytes) {          ...      }  } |

Note that ParameterHandlers can not be used to deal with parameters representing a message body, "byte[] byte" in this example. MessageBodyReaders have to deal with this task. That said, a given MessageBodyReader implementation can also implement ParameterHandler.

ParameterHandlers can be registered as providers either from Spring or programmatically.  
Note that by default the handlers are checked last after all the other options recommended by the JAX-RS specification have been tried.  
Starting from CXF 2.5.3 the handlers will always be checked first for java.util.Date and java.util.Locale parameters. Additionally, a "check.parameter.handlers.first" contextual property can be used to get the handlers checked first when the parameters of other types are processed.

All the parameters are automatically decoded. This can be disabled by using @Encoded annotation.  
Parameters can have a default value set using a DefaultValue annotation :

|  |
| --- |
| public Response updateCustomer(@DefaultValue("123") @QueryParam("id") Long id, @PathParam("name") String name) { ... } |

JAX-RS mandates that only a single method parameter which is not annotated with JAXRS annotations applicable to method parameters is allowed in a resource method. For example :

|  |
| --- |
| public Response do(@PathParam("id") String id, String body) {  } |

Parameters like 'String body' are expected to represent the request body/input stream. It's the job of JAX-RS MessageBodyReaders to deserialize the request body into an object of the expected type.

It's also possible to inject all types of parameters into fields or through dedicated setters. For example, the first code fragment in this section can be rewritten like this:

|  |
| --- |
| @Path("/customer/{id}")  public class CustomerService {        @PathParam("id")      private Long id;        private String name;        @PathParam("name")      public setName(String name) {          this.name = name;      }        @PUT      @Path("{name}")      public Response updateCustomer() {          // use id and name      }  } |

## Parameter beans

There's a CXF extension which makes it possible to inject a sequence of @PathParam, @QueryParam, @FormParam or @MatrixParam parameters into a bean. For example:

|  |
| --- |
| @Path("/customer/{id}")  public class CustomerService {          @PUT      @Path("{name}")      public Response updateCustomer(@PathParam("") Customer customer) {          ...      }        @GET      @Path("/order")      public Response getCustomerOrder(@PathParam("id") int customerId,                                       @QueryParam("") OrderBean bean,                                       @MatrixParam("") OrderBean bean) {          ...      }        @POST      public Response addCustomerOrder(@PathParam("id") int customerId,                                       @FormParam("") OrderBean bean) {          ...      }  }    public class Customer {     public void setId(Long id) {...}     public void setName(String s) {...}  }    public class OrderBean {     public void setId(Long id) {...}     public void setWeight(int w) {...}  } |

Note that there's a single @PathParam with an empty value in updateCustomer() - this is an extension bit. The value for a template variable 'id' is injected into Customer.setId(Long id), while the value for 'name' is injected into Customer.setName(String s). The setter methods should have a single parameter, the conversion from the actual value to the parameter instance follows the same procedure as outlined above.

Similarly, in getCustomerOrder(), OrderBean can be injected with corresponding values from a query string like ?id=1&weight=2 or from matrix parameters set as part of one of the path segments : /customer/1/order;id=1;weight=2. Likewise, in addCustomerOrder(), FormParam("") can capture all the values submitted from an HTML form and inject them into OrderBean.

Nested beans are also supported, which among other things, makes it possible to formulate advanced search queries. For example, given the following bean definitions:

|  |
| --- |
| class Name {      String first;      String last;  }    class Address {      String city;      String state;  }    class Person {      Name legalName;      Address homeAddr;      String race;      String sex;      Date birthDate;  }    class MyService  {      @GET      @Path("/getPerson")      Person getPerson(@QueryParam("") Person person);  } |

a query like

> /getPerson?sex=M&legalName.first=John&legalName.last=Doe&homeAddr.city=Reno&homeAddr.state=NV

will result in a Person bean being properly initialized and all the search criteria being captured and easily accessible. Note more enhancements are being planned in this area.

# Resource lifecycles

The scopes which are supported by default are Singleton and Prototype(per-request).  
Note that JAXRS MessageBodyWriter and MessageBodyReader providers are always singletons.

Classes with prototype scopes can get JAXRS contexts or parameters injected at construction time:

|  |
| --- |
| @Path("/")  public class PerRequestResourceClass {       public PerRequestResourceClass(@Context HttpHeaders headers, @QueryParam("id") Long id) {}  } |

Classes with singleton scopes can only have contexts injected at the construction time and it is only a CXFNonSpringJaxrsServlet which can do it. In most cases you can have contexts injected as bean properties right after construction time.

See the "Lifecycle management" section for more details.

# JAX-RS Endpoint Lifecycle Explained

### Default JAX-RS Bean Lifecycle

The JAX-RS specification dictates that the default lifecycle of an endpoint class is once-per-request. Take the following endpoint class, for example:

@Path ("/mypath")

public class MyEndpoint {

@GET

@Path("/hello")

public String helloWorld() {

return "hello world!";

}

}

According to the JAX-RS specification, there will be one instance of MyEndpoint for every request to /mypath/hello.

There are advantages to this approach. For example, the JAX-RS can manage the entire lifecycle of the endpoint bean and do things like inject request-specific parameters and context into fields on the endpoint bean. For example:

@Path ("/mypath")

public class MyEndpoint {

@Context

HttpServletRequest request;

@GET

@Path("/hello")

public String helloWorld() {

return "hello world!";

}

}

Since the JAX-RS implementation is managing the lifecycle of this bean, it can safely inject the HttpServletRequest into the request field without having to worry about disrupting the state of the endpoint bean.

### JAX-RS Bean Lifecycle and Other Containers

The problem with the one-bean-per-request lifecycle strategy comes when you try to integrate with other DI containers. Spring, for example, defaults the lifecycle of its beans to be one-per-application (a.k.a. "singleton"). Developers who want to leverage the features of Spring to manage the lifecycle of their endpoint beans have to somehow reconcile the difference between the default JAX-RS behavior (one-per-request) and the behavior of Spring (one-per-application).

To account for this reconciliation, the JAX-RS specification relaxed its restriction to allow different JAX-RS implementations to define their own lifecycle management capabilities. Jersey, for example, tends to hold to the one-per-request lifecycle mapping, but provides some support for Spring integration by allowing developers to have fields on the endpoint bean whose instances are provided by Spring (see the com.sun.jersey.spi.inject.Injectannotation). CXF, on the other hand, uses the one-per-request lifecycle only when there is no endpoint bean defined in the Spring container. Otherwise, it assumes one-per-application.

Custom HttpVerbs :

### PATCH methods on JAX-RS

We added PATCH semantics for [Virgil](http://code.google.com/a/apache-extras.org/p/virgil/).

This was fairly straight forward, except we need to add support for a @PATCH annotation and PatchMethod for HttpClient.

To do this, we created a PATCH annotation. Take a look at [PATCH.java](http://code.google.com/a/apache-extras.org/p/virgil/source/browse/trunk/src/main/java/org/apache/cassandra/http/ext/PATCH.java). The contents of which are shown below:

@Target({ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@HttpMethod("PATCH")  
public @interface PATCH {  
}

This then allows us to use @PATCH on an annotation on a [REST service.](http://code.google.com/a/apache-extras.org/p/virgil/source/browse/trunk/src/main/java/org/apache/cassandra/http/CassandraRestService.java)

@PATCH  
@Path("/data/{keyspace}/{columnFamily}/{key}")  
@Produces({ "application/json" })  
public void patchRow(@PathParam("keyspace") String keyspace,  
@PathParam("columnFamily") String columnFamily, @PathParam("key") String key,  
@QueryParam("index") boolean index, String body) throws Exception

That worked like a charm. Then we needed to call it using HttpClient. To that, we created a PatchMethod class that extended PostMethod. You can see that [here](http://code.google.com/a/apache-extras.org/p/virgil/source/browse/trunk/src/main/java/org/apache/cassandra/http/ext/PatchMethod.java).

Then we could use that just like any other HTTP method.

PatchMethod patch = new PatchMethod(BASE\_URL + KEYSPACE + "/" + COLUMN\_FAMILY + "/" + KEY);  
requestEntity = new StringRequestEntity("{\"ADDR1\":\"1235 Fun St.\",\"COUNTY\":\"Montgomery\"}",  
"appication/json", "UTF8");  
patch.setRequestEntity(requestEntity);

Hope that helps people.

# Delayed Invocations / Request Building

The javax.ws.rs.client.Invocation class allows us to build up a request for later use – either synchronous or asynchronous.

In the following example we’re building up two invocations for later use – one to persist a book and the other to fetch all available books and we’re using both afterwards to persist a book and fetch all available books.

Use invoke() for a synchronous request and submit() for an asynchronous request – either methods may return either a javax.ws.rs.core.Response or the desired entity if you give the entity’s class as method parameter.

# Exception handling

One can either throw an unchecked WebApplicationException or return Response with a proper error code set.  
The former option may be a better one when no JAX-RS types can be added to method signatures.

For example :

|  |
| --- |
| @Path("/customerservice/")  public class CustomerService {          @PUT      @Path("/customers/{id}")      public Response updateCustomer(@PathParam("id") Long id, Customer customer) {          return Response.status(errorCode).build();      }        @POST      @Path("/customers")      public Customer addCustomer(Customer customer) {          throw new WebApplicationException(errorCode);      }    } |

Yet another option is to register an ExceptionMapper provider. Ex :

|  |
| --- |
| public BookExceptionMapper implements ExceptionMapper<BookException> {      public Response toResponse(BookException ex) {          // convert to Response      }  } |

This allows for throwing a checked or runtime exception from an application code and map it to an HTTP response in a registered provider.

Have a look please at [this exception mapper](http://svn.apache.org/repos/asf/cxf/trunk/systests/jaxrs/src/test/java/org/apache/cxf/systest/jaxrs/security/SecurityExceptionMapper.java) which converts Spring Security exceptions into HTTP 403 error code for another example.

Note that when no mappers are found for custom exceptions, they are propagated to the underlying container as required by the specification where they will typically be wrapped in ServlerException, eventually resulting in HTTP 500 status being returned by default. Thus one option for intercepting the exceptions is to register a custom servlet filter which will catch ServletExceptions and handle the causes.

This propagation can be disabled by registering a boolean jaxrs property 'org.apache.cxf.propagate.exception' with a false value. If such property is set and no exception mapper can be found for a given exception then it will be wrapped into an xml error response by the CXF [XMLFaultOutInterceptor](http://svn.apache.org/repos/asf/cxf/trunk/rt/bindings/xml/src/main/java/org/apache/cxf/binding/xml/interceptor/XMLFaultOutInterceptor.java).

One can also register a custom CXF out fault interceptor which can handle all the exceptions by writing directly to the HttpServletResponse stream or XMLStreamWriter (as XMLFaultOutInterceptor does). For example, see this [test interceptor](http://svn.apache.org/repos/asf/cxf/trunk/systests/jaxrs/src/test/java/org/apache/cxf/systest/jaxrs/CustomOutFaultInterceptor.java).

# Filters :

Filters can be used when you want to modify any request or response parameters like headers. For example you would like to add a response header "X-Powered-By" to each generated response. Instead of adding this header in each resource method you would use a response filter to add this header.

There are filters on the server side and the client side.

Server filters:

|  |
| --- |
| [ContainerRequestFilter](https://jersey.java.net/apidocs-javax.jax-rs/2.0.1/javax/ws/rs/container/ContainerRequestFilter.html) |
| [ContainerResponseFilter](https://jersey.java.net/apidocs-javax.jax-rs/2.0.1/javax/ws/rs/container/ContainerResponseFilter.html) |

Client filters:

|  |
| --- |
| [ClientRequestFilter](https://jersey.java.net/apidocs-javax.jax-rs/2.0.1/javax/ws/rs/client/ClientRequestFilter.html) |
| [ClientResponseFilter](https://jersey.java.net/apidocs-javax.jax-rs/2.0.1/javax/ws/rs/client/ClientResponseFilter.html) |

#### Pre-matching and post-matching filters

All the request filters shown above was implemented as post-matching filters. It means that the filters would be applied only after a suitable resource method has been selected to process the actual request i.e. after request matching happens. Request matching is the process of finding a resource method that should be executed based on the request path and other request parameters. Since post-matching request filters are invoked when a particular resource method has already been selected, such filters can not influence the resource method matching process.

To overcome the above described limitation, there is a possibility to mark a server request filter as a pre-matching filter, i.e. to annotate the filter class with the [@PreMatching](https://jersey.java.net/apidocs-javax.jax-rs/2.0.1/javax/ws/rs/container/PreMatching.html)annotation. Pre-matching filters are request filters that are executed before the request matching is started. Thanks to this, pre-matching request filters have the possibility to influence which method will be matched. Such a pre-matching request filter example is shown here:

**Example 10.3. Pre-matching request filter**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | ...  import javax.ws.rs.container.ContainerRequestContext;  import javax.ws.rs.container.ContainerRequestFilter;  import javax.ws.rs.container.PreMatching;  ...    @PreMatching  public class PreMatchingFilter implements ContainerRequestFilter {        @Override      public void filter(ContainerRequestContext requestContext)                          throws IOException {          // change all PUT methods to POST          if (requestContext.getMethod().equals("PUT")) {              requestContext.setMethod("POST");          }      }  } |

The PreMatchingFilter is a simple pre-matching filter which changes all PUT HTTP methods to POST. This might be useful when you want to always handle these PUT and POST HTTP methods with the same Java code. After the PreMatchingFilter has been invoked, the rest of the request processing will behave as if the POST HTTP method was originally used. You cannot do this in post-matching filters (standard filters without @PreMatching annotation) as the resource method is already matched (selected). An attempt to tweak the original HTTP method in a post-matching filter would cause an IllegalArgumentException.

As written above, pre-matching filters can fully influence the request matching process, which means you can even modify request URI in a pre-matching filter by invoking the setRequestUri(URI) method of ContainerRequestFilter so that a different resource would be matched.

Like in post-matching filters you can abort a response in pre-matching filters too.

### Client filters

Client filters are similar to container filters. The response can also be aborted in the [ClientRequestFilter](https://jersey.java.net/apidocs-javax.jax-rs/2.0.1/javax/ws/rs/client/ClientRequestFilter.html" \t "_top) which would cause that no request will actually be sent to the server at all. A new response is passed to the abort method. This response will be used and delivered as a result of the request invocation. Such a response goes through the client response filters. This is similar to what happens on the server side. The process is shown in the following example:

**Example 10.4. Client request filter**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class CheckRequestFilter implements ClientRequestFilter {        @Override      public void filter(ClientRequestContext requestContext)                          throws IOException {          if (requestContext.getHeaders(                          ).get("Client-Name") == null) {              requestContext.abortWith(                          Response.status(Response.Status.BAD\_REQUEST)                  .entity("Client-Name header must be defined.")                          .build());           }      }  } |

The CheckRequestFilter validates the outgoing request. It is checked for presence of a Client-Name header. If the header is not present the request will be aborted with a made up response with an appropriate code and message in the entity body. This will cause that the original request will not be effectively sent to the server but the actual invocation will still end up with a response as if it would be generated by the server side. If there would be any client response filter it would be executed on this response.

To summarize the workflow, for any client request invoked from the client API the client request filters ([ClientRequestFilter](https://jersey.java.net/apidocs-javax.jax-rs/2.0.1/javax/ws/rs/client/ClientRequestFilter.html" \t "_top)) are executed that could manipulate the request. If not aborted, the outcoming request is then physically sent over to the server side and once a response is received back from the server the client response filters ([ClientResponseFilter](https://jersey.java.net/apidocs-javax.jax-rs/2.0.1/javax/ws/rs/client/ClientResponseFilter.html" \t "_top)) are executed that might again manipulate the returned response. Finally the response is passed back to the code that invoked the request. If the request was aborted in any client request filter then the client/server communication is skipped and the aborted response is used in the response filters.

# Interceptors

Interceptors share a common API for the server and the client side. Whereas filters are primarily intended to manipulate request and response parameters like HTTP headers, URIs and/or HTTP methods, interceptors are intended to manipulate entities, via manipulating entity input/output streams. If you for example need to encode entity body of a client request then you could implement an interceptor to do the work for you.

There are two kinds of interceptors, [ReaderInterceptor](https://jersey.java.net/apidocs-javax.jax-rs/2.0.1/javax/ws/rs/ext/ReaderInterceptor.html" \t "_top) and [WriterInterceptor](https://jersey.java.net/apidocs-javax.jax-rs/2.0.1/javax/ws/rs/ext/WriterInterceptor.html" \t "_top). Reader interceptors are used to manipulate inbound entity streams. These are the streams coming from the "wire". So, using a reader interceptor you can manipulate request entity stream on the server side (where an entity is read from the client request) and response entity stream on the client side (where an entity is read from the server response). Writer interceptors are used for cases where entity is written to the "wire" which on the server means when writing out a response entity and on the client side when writing request entity for a request to be sent out to the server. Writer and reader interceptors are executed before message body readers or writers are executed and their primary intention is to wrap the entity streams that will be used in message body reader and writers.

The following example shows a writer interceptor that enables GZIP compression of the whole entity body.

**Example 10.5. GZIP writer interceptor**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class GZIPWriterInterceptor implements WriterInterceptor {        @Override      public void aroundWriteTo(WriterInterceptorContext context)                      throws IOException, WebApplicationException {          final OutputStream outputStream = context.getOutputStream();          context.setOutputStream(new GZIPOutputStream(outputStream));          context.proceed();      }  } |

The interceptor gets a output stream from the [WriterInterceptorContext](https://jersey.java.net/apidocs-javax.jax-rs/2.0.1/javax/ws/rs/ext/WriterInterceptorContext.html" \t "_top) and sets a new one which is a GZIP wrapper of the original output stream. After all interceptors are executed the output stream lastly set to the WriterInterceptorContext will be used for serialization of the entity. In the example above the entity bytes will be written to the GZIPOutputStream which will compress the stream data and write them to the original output stream. The original stream is always the stream which writes the data to the "wire". When the interceptor is used on the server, the original output stream is the stream into which writes data to the underlying server container stream that sends the response to the client.

The interceptors wrap the streams and they itself work as wrappers. This means that each interceptor is a wrapper of another interceptor and it is responsibility of each interceptor implementation to call the wrapped interceptor. This is achieved by calling the proceed() method on the WriterInterceptorContext. This method will call the next registered interceptor in the chain, so effectivelly this will call all remaining registered interceptors. Calling proceed() from the last interceptor in the chain will call the appropriate message body reader. Therefore every interceptor must call the proceed() method otherwise the entity would not be written. The wrapping principle is reflected also in the method name, aroundWriteTo, which says that the method is wrapping the writing of the entity.

The method aroundWriteTo() gets WriterInterceptorContext as a parameter. This context contains getters and setters for header parameters, request properties, entity, entity stream and other properties. These are the properties which will be passed to the final MessageBodyWriter<T>. Interceptors are allowed to modify all these properties. This could influence writing of an entity by MessageBodyWriter<T> and even selection of such a writer. By changing media type (WriterInterceptorContext.setMediaType()) the interceptor can cause that different message body writer will be chosen. The interceptor can also completely replace the entity if it is needed. However, for modification of headers, request properties and such, the filters are usually more preferable choice. Interceptors are executed only when there is any entity and when the entity is to be written. So, when you always want to add a new header to a response no matter what, use filters as interceptors might not be executed when no entity is present. Interceptors should modify properties only for entity serialization and deserialization purposes.

Let's now look at an example of a ReaderInterceptor

**Example 10.6. GZIP reader interceptor**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class GZIPReaderInterceptor implements ReaderInterceptor {        @Override      public Object aroundReadFrom(ReaderInterceptorContext context)                      throws IOException, WebApplicationException {          final InputStream originalInputStream = context.getInputStream();          context.setInputStream(new GZIPInputStream(originalInputStream));          return context.proceed();      }  } |

The GZIPReaderInterceptor wraps the original input stream with the GZIPInputStream. All further reads from the entity stream will cause that data will be decompressed by this stream. The interceptor method aroundReadFrom() must return an entity. The entity is returned from the proceed method of the [ReaderInterceptorContext](https://jersey.java.net/apidocs-javax.jax-rs/2.0.1/javax/ws/rs/ext/ReaderInterceptorContext.html" \t "_top). Theproceed method internally calls the wrapped interceptor which must also return an entity. The proceed method invoked from the last interceptor in the chain calls message body reader which deserializes the entity end returns it. Every interceptor can change this entity if there is a need but in the most cases interceptors will just return the entity as returned from the proceed method.

As already mentioned above, interceptors should be primarily used to manipulate entity body. Similar to methods exposed by WriterInterceptorContext the ReaderInterceptorContext introduces a set of methods for modification of request/response properties like HTTP headers, URIs and/or HTTP methods (excluding getters and setters for entity as entity has not been read yet). Again the same rules as for WriterInterceptor applies for changing these properties (change only properties in order to influence reading of an entity).

## Filter and interceptor execution order

Let's look closer at the context of execution of filters and interceptors. The following steps describes scenario where a JAX-RS client makes a POST request to the server. The server receives an entity and sends a response back with the same entity. GZIP reader and writer interceptors are registered on the client and the server. Also filters are registered on client and server which change the headers of request and response.

1. Client request invoked: The POST request with attached entity is built on the client and invoked.
2. ClientRequestFilters: client request filters are executed on the client and they manipulate the request headers.
3. Client WriterInterceptor: As the request contains an entity, writer interceptor registered on the client is executed before a MessageBodyWriter is executed. It wraps the entity output stream with the GZipOutputStream.
4. Client MessageBodyWriter: message body writer is executed on the client which serializes the entity into the new GZipOutput stream. This stream zips the data and sends it to the "wire".
5. Server: server receives a request. Data of entity is compressed which means that pure read from the entity input stream would return compressed data.
6. Server pre-matching ContainerRequestFilters: ContainerRequestFilters are executed that can manipulate resource method matching process.
7. Server: matching: resource method matching is done.
8. Server: post-matching ContainerRequestFilters: ContainerRequestFilters post matching filters are executed. This include execution of all global filters (without name binding) and filters name-bound to the matched method.
9. Server ReaderInterceptor: reader interceptors are executed on the server. The GZIPReaderInterceptor wraps the input stream (the stream from the "wire") into the GZipInputStream and set it to context.
10. Server MessageBodyReader: server message body reader is executed and it deserializes the entity from new GZipInputStream (get from the context). This means the reader will read unzipped data and not the compressed data from the "wire".
11. Server resource method is executed: the deserialized entity object is passed to the matched resource method as a parameter. The method returns this entity as a response entity.
12. Server ContainerResponseFilters are executed: response filters are executed on the server and they manipulate the response headers. This include all global bound filters (without name binding) and all filters name-bound to the resource method.
13. Server WriterInterceptor: is executed on the server. It wraps the original output stream with a new GZIPOuptutStream. The original stream is the stream that "goes to the wire" (output stream for response from the underlying server container).
14. Server MessageBodyWriter: message body writer is executed on the server which serializes the entity into the GZIPOutputStream. This stream compresses the data and writes it to the original stream which sends this compressed data back to the client.
15. Client receives the response: the response contains compressed entity data.
16. Client ClientResponseFilters: client response filters are executed and they manipulate the response headers.
17. Client response is returned: the javax.ws.rs.core.Response is returned from the request invocation.
18. Client code calls response.readEntity(): read entity is executed on the client to extract the entity from the response.
19. Client ReaderInterceptor: the client reader interceptor is executed when readEntity(Class) is called. The interceptor wraps the entity input stream with GZIPInputStream. This will decompress the data from the original input stream.
20. Client MessageBodyReaders: client message body reader is invoked which reads decompressed data from GZIPInputStream and deserializes the entity.
21. Client: The entity is returned from the readEntity().

It is worth to mention that in the scenario above the reader and writer interceptors are invoked only if the entity is present (it does not make sense to wrap entity stream when no entity will be written). The same behaviour is there for message body readers and writers. As mentioned above, interceptors are executed before the message body reader/writer as a part of their execution and they can wrap the input/output stream before the entity is read/written. There are exceptions when interceptors are not run before message body reader/writers but this is not the case of simple scenario above. This happens for example when the entity is read many times from client response using internal buffering. Then the data are intercepted only once and kept 'decoded' in the buffer.

#### Priorities

A collection of built-in priority constants for the JAX-RS components that are supposed to be ordered based on their javax.annotation.Priority class-level annotation value when used or applied by JAX-RS runtime.

For example, JAX-RS filters and interceptors are grouped in chains for each of the message processing extension points: Pre, PreMatch, Post as well as ReadFrom and WriteTo. Each of these chains is sorted based on priorities which are represented as integer numbers. All chains, except Post, are sorted in ascending order; the lower the number the higher the priority. The Post filter chain is sorted in descending order to ensure that response filters are executed in *reverse order*.

JAX-RS components that belong to the same priority class (same integer value) are executed in an implementation-defined manner. By default, when the @Priority annotation is absent on a component, for which a priority should be applied, the [USER](https://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Priorities.html#USER) priority value is used.

|  |  |
| --- | --- |
| **Modifier and Type** | **Field and Description** |
| static int | [**AUTHENTICATION**](https://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Priorities.html#AUTHENTICATION) (1000)  Security authentication filter/interceptor priority. |
| static int | [**AUTHORIZATION**](https://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Priorities.html#AUTHORIZATION) (2000)  Security authorization filter/interceptor priority. |
| static int | [**ENTITY\_CODER**](https://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Priorities.html#ENTITY_CODER) (4000)  Message encoder or decoder filter/interceptor priority. |
| static int | [**HEADER\_DECORATOR**](https://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Priorities.html#HEADER_DECORATOR) (3000)  Header decorator filter/interceptor priority. |
| static int | [**USER**](https://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Priorities.html#USER) (5000)  User-level filter/interceptor priority. |

import javax.annotation.Priority;

import javax.ws.rs.Priorities;

@Priority(Priorities.HEADER\_DECORATOR)

public class ResponseFilter implements ContainerResponseFilter {

    @Override    public void filter(ContainerRequestContext requestContext,

                    ContainerResponseContext responseContext)

                    throws IOException {        responseContext.getHeaders().add("X-Powered-By", "Jersey :-)");

    }

}

As this is a response filter and response filters are executed in the reverse order, any other filter with priority lower than 3000 (Priorities.HEADER\_DECORATOR is 3000) will be executed after this filter. So, for example AUTHENTICATION filter (priority 1000) would be run after this filter.

# Exception Handling

## 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](https://jersey.java.net/apidocs-javax.jax-rs/2.0.1/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](https://jersey.java.net/apidocs-javax.jax-rs/2.0.1/javax/ws/rs/WebApplicationException.html" \t "_top), or classes that extend [WebApplicationException](https://jersey.java.net/apidocs-javax.jax-rs/2.0.1/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>](https://jersey.java.net/apidocs-javax.jax-rs/2.0.1/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](https://jersey.java.net/apidocs-javax.jax-rs/2.0.1/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](https://jersey.java.net/apidocs-javax.jax-rs/2.0.1/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.

Jersey supports extension of the exception mappers. These extended mappers must implement the org.glassfish.jersey.spi.ExtendedExceptionMapper interface. This interface additionally defines method isMappable(Throwable) which will be invoked by the Jersey runtime when exception is thrown and this provider is considered as mappable based on the exception type. Using this method the provider can reject mapping of the exception before the method toResponse is invoked. The provider can for example check the exception parameters and based on them return false and let other provider to be chosen for the exception mapping.

You need to annotate your exception mapper with [@Provider](http://docs.oracle.com/javaee/6/api/javax/ws/rs/ext/Provider.html), otherwise it will never get registered with the JAX-RS runtime.

@Provider

public class UserNotFoundMapper implements

ExceptionMapper<UserNotFoundException> {

@Override

public Response toResponse(UserNotFoundException ex) {

return Response.status(404).entity(ex.getMessage()).type("text/plain")

.build();

}

}

-======================------------------

# How token-based authentication works

In token-based authentication, the client exchanges hard credentials (such as username and password) for a piece of data called token. Instead of sending the hard credentials in every request, the client will send the token to the server to perform authentication and authorization.

In a few words, an authentication scheme based on tokens follow these steps:

1. The client sends their credentials (username and password) to the server.
2. The server authenticates the credentials and generates a token.
3. The server stores the previously generated token in some storage along with the user identifier and an expiration date.
4. The server sends the generated token to the client.
5. In every request, the client sends the token to the server.
6. The server, in each request, extracts the token from the incoming request. With the token, the server looks up the user details to perform authentication and authorization.
   1. If the token is valid, the server accepts the request.
   2. If the token is invalid, the server refuses the request.
7. The server can provide an endpoint to refresh expired tokens.

## What you can do with JAX-RS 2.0 (Jersey, RESTEasy and Apache CXF)

This solution uses only the JAX-RS 2.0 API, avoiding any vendor specific solution. So, it should work with the most popular JAX-RS 2.0 implementations, such as [Jersey](https://jersey.java.net/), [RESTEasy](http://resteasy.jboss.org/) and [Apache CXF](https://cxf.apache.org/).

It's important mention that if you are using a token-based authentication, you are not relying on the standard Java EE web application security mechanisms offered by the servlet container and configurable via application's web.xml descriptor.

Authenticate a user with their username and password and issue a token

Create a REST endpoint which receives and validates the credentials (username and password) and issue a token for the user:

@Path("/authentication")

public class AuthenticationEndpoint {

@POST

@Produces("application/json")

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

public Response authenticateUser(@FormParam("username") String username,

@FormParam("password") String password) {

try {

// Authenticate the user using the credentials provided

authenticate(username, password);

// Issue a token for the user

String token = issueToken(username);

// Return the token on the response

return Response.ok(token).build();

} catch (Exception e) {

return Response.status(Response.Status.UNAUTHORIZED).build();

}

}

private void authenticate(String username, String password) throws Exception {

// Authenticate against a database, LDAP, file or whatever

// Throw an Exception if the credentials are invalid

}

private String issueToken(String username) {

// Issue a token (can be a random String persisted to a database or a JWT token)

// The issued token must be associated to a user

// Return the issued token

}

}

If any exceptions happen when validating the credentials, a response with status 401 UNAUTHORIZEDwill be returned.

If the credentials are successfully validated, a response with status 200 OK will be returned and the issued token is sent to the client on the response. The client must send that token to the server in every request.

Using this approach, you expect your client will send the credentials in the following format in the body of the request:

username=admin&password=123456

Instead of form params, you can wrap the username and the password into a class:

public class Credentials implements Serializable {

private String username;

private String password;

// Getters and setters omitted

}

And consume it as JSON:

@POST

@Produces("application/json")

@Consumes("application/json")

public Response authenticateUser(Credentials credentials) {

String username = credentials.getUsername();

String password = credentials.getPassword();

// Authenticate the user, issue a token and return a response

}

Using this approach, you expect your client will send the credentials in the following format in the body of the request:

{

"username": "admin",

"password": "123456"

}

Extract the token from the request and validate it

The client should send the token on the standard HTTP Authorization header of the request. For example:

Authorization: Bearer <token-goes-here>

Note that the name of the standard HTTP header is unfortunate because it carries authentication information, not authorization.

JAX-RS provides [@NameBinding](http://docs.oracle.com/javaee/7/api/javax/ws/rs/NameBinding.html), a meta-annotation used to create name-binding annotations for filters and interceptors:

@NameBinding

@Retention(RUNTIME)

@Target({TYPE, METHOD})

public @interface Secured { }

The defined name-binding annotation @Secured will be used to decorate a filter class, which implements [ContainerRequestFilter](http://docs.oracle.com/javaee/7/api/javax/ws/rs/container/ContainerRequestFilter.html), allowing you to handle the request. The [ContainerRequestContext](http://docs.oracle.com/javaee/7/api/javax/ws/rs/container/ContainerRequestContext.html) helps you to extract the token from the HTTP request:

@Secured

@Provider

@Priority(Priorities.AUTHENTICATION)

public class AuthenticationFilter implements ContainerRequestFilter {

@Override

public void filter(ContainerRequestContext requestContext) throws IOException {

// Get the HTTP Authorization header from the request

String authorizationHeader =

requestContext.getHeaderString(HttpHeaders.AUTHORIZATION);

// Check if the HTTP Authorization header is present and formatted correctly

if (authorizationHeader == null || !authorizationHeader.startsWith("Bearer ")) {

throw new NotAuthorizedException("Authorization header must be provided");

}

// Extract the token from the HTTP Authorization header

String token = authorizationHeader.substring("Bearer".length()).trim();

try {

// Validate the token

validateToken(token);

} catch (Exception e) {

requestContext.abortWith(

Response.status(Response.Status.UNAUTHORIZED).build());

}

}

private void validateToken(String token) throws Exception {

// Check if it was issued by the server and if it's not expired

// Throw an Exception if the token is invalid

}

}

If any problems happen during the token validation, a response with status 401 UNAUTHORIZED will be returned.

Otherwise, the request will proceed to an endpoint.

Securing your REST endpoints

Bind the filter to your endpoints methods or classes by annotating them with the @Secured annotation created above. For the methods and/or classes which are annotated, the filter will be executed. It means that these endpoints only will be reached if the request is performed with a valid token.

If some methods or classes do not need authentication, simply do not annotate them.

@Path("/")

public class MyEndpoint {

@GET

@Path("{id}")

@Produces("application/json")

public Response myUnsecuredMethod(@PathParam("id") Long id) {

// This method is not annotated with @Secured

// The authentication filter won't be executed before invoking this method

...

}

@DELETE

@Secured

@Path("{id}")

@Produces("application/json")

public Response mySecuredMethod(@PathParam("id") Long id) {

// This method is annotated with @Secured

// The authentication filter will be executed before invoking this method

// The HTTP request must be performed with a valid token

...

}

}

In the example above, the filter will be executed only for mySecuredMethod(Long) because it's annotated with @Secured.

## Identifying the current user

It's very likely you will need to know the user who is performing the request within your REST endpoints. The following approaches can be useful to do it:

Overriding the**[SecurityContext](http://docs.oracle.com/javaee/7/api/javax/ws/rs/core/SecurityContext.html)**

Within your [ContainerRequestFilter.filter(ContainerRequestContext)](http://docs.oracle.com/javaee/7/api/javax/ws/rs/container/ContainerRequestFilter.html#filter-javax.ws.rs.container.ContainerRequestContext-) method, you can set a new security context information for the current request.

Override the [SecurityContext.getUserPrincipal()](http://docs.oracle.com/javaee/7/api/javax/ws/rs/core/SecurityContext.html" \l "getUserPrincipal--), returning a [Principal](http://docs.oracle.com/javase/7/docs/api/java/security/Principal.html) instance.

The [Principal](http://docs.oracle.com/javase/7/docs/api/java/security/Principal.html)'s name is the username of the user you issued the token for. You will have to know it when validating the token.

requestContext.setSecurityContext(new SecurityContext() {

@Override

public Principal getUserPrincipal() {

return new Principal() {

@Override

public String getName() {

return username;

}

};

}

@Override

public boolean isUserInRole(String role) {

return true;

}

@Override

public boolean isSecure() {

return false;

}

@Override

public String getAuthenticationScheme() {

return null;

}

});

Inject a proxy of the [SecurityContext](http://docs.oracle.com/javaee/7/api/javax/ws/rs/core/SecurityContext.html) in any REST endpoint class:

@Context

SecurityContext securityContext;

The same can be done in a method:

@GET

@Secured

@Path("{id}")

@Produces("application/json")

public Response myMethod(@PathParam("id") Long id,

@Context SecurityContext securityContext) {

...

}

And get the [Principal](http://docs.oracle.com/javase/7/docs/api/java/security/Principal.html):

Principal principal = securityContext.getUserPrincipal();

String username = principal.getName();

Using CDI (Context and Dependency Injection)

If, for some reason, you don't want to override the [SecurityContext](http://docs.oracle.com/javaee/7/api/javax/ws/rs/core/SecurityContext.html), you can use CDI, which provides useful features such as events and producers.

Create a CDI qualifier which will be used when handling the authentication event and when injecting the authenticated user in your beans:

@Qualifier

@Retention(RUNTIME)

@Target({ METHOD, FIELD, PARAMETER })

public @interface AuthenticatedUser { }

In your AuthenticationFilter created above, inject an [Event](https://docs.oracle.com/javaee/7/api/javax/enterprise/event/Event.html):

@Inject

@AuthenticatedUser

Event<String> userAuthenticatedEvent;

When the user authenticates, fire the event passing the username as parameter (remember, your token must be associated to a user and you need to be able to retrieve the username from a token):

userAuthenticatedEvent.fire(username);

Probably you have a class which represents a user in your application. Let's call this class User.

The piece of code below handles the authentication event, finds a User instance with the correspondent username and assigns it to the field authenticatedUser:

@RequestScoped

public class AuthenticatedUserProducer {

@Produces

@RequestScoped

@AuthenticatedUser

private User authenticatedUser;

public void handleAuthenticationEvent(@Observes @AuthenticatedUser String username) {

this.authenticatedUser = findUser(username);

}

private User findUser(String username) {

// Hit the the database or a service to find a user by its username and return it

// Return the User instance

}

}

The authenticatedUser field produces a User instance which can be injected in your beans, such as JAX-RS services, CDI beans, servlets and EJBs:

@Inject

@AuthenticatedUser

User authenticatedUser;

Note that the CDI [@Produces](http://docs.oracle.com/javaee/7/api/javax/enterprise/inject/Produces.html) annotation is different from the JAX-RS [@Produces](http://docs.oracle.com/javaee/7/api/javax/ws/rs/Produces.html) annotation:

* CDI: [javax.enterprise.inject.Produces](http://docs.oracle.com/javaee/7/api/javax/enterprise/inject/Produces.html)
* JAX-RS: [javax.ws.rs.Produces](http://docs.oracle.com/javaee/7/api/javax/ws/rs/Produces.html)

## Supporting role-based authorization

Besides authentication you can also support role-based authorization in your REST endpoints.

Create an enumeration and define the roles according to your needs:

public enum Role {

ROLE\_1,

ROLE\_2,

ROLE\_3

}

Change the @Secured name binding annotation created above to support roles:

@NameBinding

@Retention(RUNTIME)

@Target({TYPE, METHOD})

public @interface Secured {

Role[] value() default {};

}

Annotate your endpoints to perform role-based authorization.

Note that the @Secured annotation can the used in classes and/or methods. So let's make the method annotations override the class annotations:

@Path("/example")

@Secured({Role.ROLE\_1})

public class MyEndpoint {

@GET

@Path("{id}")

@Produces("application/json")

public Response myMethod(@PathParam("id") Long id) {

// This method is not annotated with @Secured

// But it's declared within a class annotated with @Secured({Role.ROLE\_1})

// So it only can be executed by the users who have the ROLE\_1 role

...

}

@DELETE

@Path("{id}")

@Produces("application/json")

@Secured({Role.ROLE\_1, Role.ROLE\_2})

public Response myOtherMethod(@PathParam("id") Long id) {

// This method is annotated with @Secured({Role.ROLE\_1, Role.ROLE\_2})

// The method annotation overrides the class annotation

// So it only can be executed by the users who have the ROLE\_1 or ROLE\_2 roles

...

}

}

Create a filter with the [AUTHORIZATION](http://docs.oracle.com/javaee/7/api/javax/ws/rs/Priorities.html#AUTHENTICATION) priority, which is executed after the [AUTHENTICATION](http://docs.oracle.com/javaee/7/api/javax/ws/rs/Priorities.html#AUTHENTICATION) priority filter defined previously.

The [ResourceInfo](http://docs.oracle.com/javaee/7/api/javax/ws/rs/container/ResourceInfo.html) can be used to get the [Method](http://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Method.html) and [Class](http://docs.oracle.com/javase/8/docs/api/java/lang/Class.html) which match with the requested URL and extract the annotations from them:

@Secured

@Provider

@Priority(Priorities.AUTHORIZATION)

public class AuthorizationFilter implements ContainerRequestFilter {

@Context

private ResourceInfo resourceInfo;

@Override

public void filter(ContainerRequestContext requestContext) throws IOException {

// Get the resource class which matches with the requested URL

// Extract the roles declared by it

Class<?> resourceClass = resourceInfo.getResourceClass();

List<Role> classRoles = extractRoles(resourceClass);

// Get the resource method which matches with the requested URL

// Extract the roles declared by it

Method resourceMethod = resourceInfo.getResourceMethod();

List<Role> methodRoles = extractRoles(resourceMethod);

try {

// Check if the user is allowed to execute the method

// The method annotations override the class annotations

if (methodRoles.isEmpty()) {

checkPermissions(classRoles);

} else {

checkPermissions(methodRoles);

}

} catch (Exception e) {

requestContext.abortWith(

Response.status(Response.Status.FORBIDDEN).build());

}

}

// Extract the roles from the annotated element

private List<Role> extractRoles(AnnotatedElement annotatedElement) {

if (annotatedElement == null) {

return new ArrayList<Role>();

} else {

Secured secured = annotatedElement.getAnnotation(Secured.class);

if (secured == null) {

return new ArrayList<Role>();

} else {

Role[] allowedRoles = secured.value();

return Arrays.asList(allowedRoles);

}

}

}

private void checkPermissions(List<Role> allowedRoles) throws Exception {

// Check if the user contains one of the allowed roles

// Throw an Exception if the user has not permission to execute the method

}

}

If the user has no permission to execute the method, the request is aborted with a 403 FORBIDDEN.

To know the user who is performing the request, see the section above. You can get it from the [SecurityContext](http://docs.oracle.com/javaee/7/api/javax/ws/rs/core/SecurityContext.html) (which should be already set in the [ContainerRequestContext](http://docs.oracle.com/javaee/7/api/javax/ws/rs/container/ContainerRequestFilter.html" \l "filter-javax.ws.rs.container.ContainerRequestContext-)) or inject it using CDI, depending on the approach you are using.

If a @Secured annotation has no roles declared, you can assume all authenticated users can access that endpoint, independent the roles the users have.

## How to issue a token

A token can be opaque which reveals no details other than the value itself (like a random string) or can be self-contained (like JSON Web Token).

Random string

A token can be issued by generating a random string and persisting it to a database with an expiration date and with a user identifier associated to it. A good example of how to generate a random string in Java can be seen [here](http://stackoverflow.com/a/41156/1426227):

Random random = new SecureRandom();

String token = new BigInteger(130, random).toString(32);

JSON Web Token (JWT)

JSON Web Token (JWT) is a standard method for representing claims securely between two parties and is defined by the [RFC 7519](https://tools.ietf.org/html/rfc7519). It's a self-contained token and enables you to store a user identifier, an expiration date and whatever you want (but don't store passwords) in a payload, which is a JSON encoded as [Base64](https://en.wikipedia.org/wiki/Base64).

The payload can be read by the client and the integrity of the token can be easily checked by verifying its signature on the server.

You won't need to persist JWT tokens if you don't need to track them. Althought, by persisting the tokens, you will have the possibility of invalidating and revoking the access of them. To keep the track of JWT tokens, instead of persisting the whole token, you could persist the token identifier (the [jti](https://tools.ietf.org/html/rfc7519" \l "section-4.1.7)claim) and some metadata (the user you issued the token for, the expiration date, etc) if you need.

There are a few Java libraries to issue and validate JWT tokens (have a look [here](https://github.com/auth0/java-jwt) and [here](https://github.com/jwtk/jjwt)). To find some other great resources to work with JWT, have a look at [http://jwt.io](http://jwt.io/).

Your application can provide some functionality to revoke the tokens, but it's recommended revoking the tokens when the users change their password.

When persisting tokens, always consider removing the old ones in order to prevent your database from growing indefinitely.

## Additional information

* It doesn't matter which type of authentication you are using. Always use HTTPS to prevent the [man-in-the-middle attack](https://en.wikipedia.org/wiki/Man-in-the-middle_attack).
* Take a look at [this question](http://security.stackexchange.com/q/19676) from Information Security for more information about tokens.
* [In this article](https://stormpath.com/blog/token-auth-spa/) you will find some useful information about token-based authentication.
* [Apache DeltaSpike](https://deltaspike.apache.org/index.html) provides portable CDI extensions such as a [security module](https://deltaspike.apache.org/documentation/security.html), which can be used to secure REST applications.
* Interested in an OAuth 2.0 protocol implementation in Java? Check the [Apache Oltu project](https://oltu.apache.org/).