Upgrading from Resteasy 2 to Resteasy 3

3.1.0-Final

A number of API classes in Resteasy 2, which is based on the JAX-RS 1.1 specification (*https://jcp.org/en/jsr/detail?id=311*), have been deprecated in, and eventually removed from, Resteasy 3, which is based on JAX-RS 2.0 (*https://jcp.org/aboutJava/communityprocess/final/jsr339/index.html*). In particular, those classes are deprecated by the end of the 3.0.x series of releases, and removed as of the 3.1.0.Final release. For the most part, these changes are due to the fact that a number of facilities specific to Resteasy were introduced in Resteasy 2 and then formalized, in somewhat different form, in JAX-RS 2.0. A few other facilities in Resteasy simply were not carried over to Resteasy 3.

This short document describes the principal changes from Resteasy 2 to Resteasy 3 and gives some hints about upgrading code from the Resteasy 2 API to Resteasy 3. Additional information can be found in the Resteasy Users Guides (*http://resteasy.jboss.org/docs.html*). A more extensive treatment may be found in the O'Reilly book *RESTful Java with JAX-RS 2.0, 2nd Edition*, by Bill Burke.

Chapter 1. Client Framework

1.1. Fluent interface

The two principal client side classes in Resteasy 2 are `ClientRequest` and `ClientResponse`:

```
ClientRequest request = new ClientRequest("http://localhost:8081/test");
request.body("text/plain", "hello world");
ClientResponse<?> response = request.post();
String result = response.getEntity(String.class);
```

`ClientRequest` holds the target URL and entity, if any. `ClientResponse` holds the response entity, which can be extracted by the getEntity() method.

In JAX-RS 2.0, these classes are replaced by four classes that support a fluent call pattern: `Client`, `WebTarget`, `Invocation.Builder`, and `Response`:

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target("http://localhost:8081/test");
Invocation.Builder builder = target.request();
Entity<String> entity = Entity.entity("hello world", "text/plain");
Response response = builder.post(entity);
String result = response.readEntity(String.class);
```

The invocation process begins with `Client`, whose primary responsibility is to create a `WebTarget`. `Client`s are somewhat expensive to build, so it often makes sense to reuse a `Client` to create multiple `WebTarget`s.

Resteasy extends `ClientBuilder` and `Client` with methods that allow the registration of providers:

```
static class TestWriter implements MessageBodyWriter<String>
{
   @Override
    public boolean isWriteable(Class<?> type, Type genericType, Annotation[]
 annotations, MediaType mediaType)
   {
      return false;
```

```
   }

   @Override
   public long getSize(String t, Class<?> type, Type genericType, Annotation[]
 annotations, MediaType mediaType)
   {
      return 0;
   }

   @Override
   public void writeTo(String t, Class<?> type, Type genericType, Annotation[]
 annotations, MediaType mediaType,
         MultivaluedMap<String, Object> httpHeaders, OutputStream entityStream)
         throws IOException, WebApplicationException
   {
      //
   }
}

ResteasyClientBuilder clientBuilder = new ResteasyClientBuilder();
Client client = clientBuilder.register(TestWriter.class).build();
```

All `Clients` created by that `ResteasyClientBuilder`, and all invocations on all `WebTargets` created by those `Clients`, will have `TestWriter` available.

`WebTarget`, as its name implies, constructs and holds a URL which targets a server side resource. It has various options for extending and manipulating URIs:

```
WebTarget target = client.target("http://localhost:8081/test/{index}");
WebTarget target1 = target.resolveTemplate("index", "1");
WebTarget target2 = target.resolveTemplate("index", "2");
```

Here, two new `WebTargets` are created from the original target, each with a different ending path segment. Query and matrix parameters can also be appended:

```
WebTarget target3 = target2.queryParam("x", "y");
```

Here, target3 targets "http://localhost:8081/test/2?x=y".

Resteasy also extends `WebTarget` with the ability to register providers:

```
Client client = ClientBuilder.newClient();
String url = "http://localhost:8081/test/{index}";
WebTarget target = client.target(url).register(TestWriter.class);
WebTarget target1 = target.resolveTemplate("index", "1");
WebTarget target2 = target.resolveTemplate("index", "2");
WebTarget target3 = target2.queryParam("x", "y");
```

Here, `TestWriter` is available to all invocations on target1, target2, and target3.

`Invocation.Builder` plays a role similar to the old `ClientRequest`:

```
Response response = builder.header("User-Agent", "Mozilla/5.0").get();
```

or

```
String s = builder.header("User-Agent", "Mozilla/5.0").get(String.class);
```

Finally, note that `Response`, unlike the old `ClientResponse<T>`, is not a generic type, so it is necessary to give a type when extracting a response entity:

```
String result = response.readEntity(String.class);
```

Note. `Response.getEntity()` still exists, but it plays a different role, which could easily lead to bugs. It is necessary to call `readEntity()` to extract the response entity. If `getEntity()` is called instead, it will return null.

Note. Unlike the old `getEntity()`, `readEntity()` is not idempotent. Once it is called, the response is closed, and subsequent calls will throw an `IllegalStateException`. This behavior can be circumvented by calling `Response.bufferEntity()` before calling `readEntity()`. I.e., this will work:

```
response.bufferEntity();
```

```
System.out.println(response.readEntity(String.class));
System.out.println(response.readEntity(String.class));
```

1.2. Client proxies

The client framework in Resteasy 2 included a facility for interacting with JAX-RS resources through client side POJOs, not unlike the Java RMI facility:

```
@Path("/test")
public static interface TestResource
{
   @GET
   @Produces("text/plain")
   public String test();
}

public void testProxy() throws Exception
{
   String url = "http://localhost:8081";
   TestResource pojo = ProxyFactory.create(TestResource.class, url);
   String result = pojo.test();
```

This technique avoids a lot of complications, but, perhaps because it is perceived as not being in the RESTful spirit, it is not part of the JAX-RS 2.0 client framework. It still exists in Resteasy 3, but in a re-worked form that fits into the official client framework. Now, a proxy is created by a call to `ResteasyWebTarget.proxy()`:

```
Client client = ClientBuilder.newClient();
String url = "http://localhost:8081";
ResteasyWebTarget target = (ResteasyWebTarget) client.target(url);
TestResource pojo = target.proxy(TestResource.class);
String result = pojo.test();
```

1.3. Client side error handling

Resteasy 2 had two facilities for handling errors on the client side.

An instance of an `org.jboss.resteasy.client.core.ClientErrorInterceptor` could be registered to handle exceptions thrown during a proxied call. Also, an

instance of an `org.jboss.resteasy.client.exception.mapper.ClientExceptionMapper` could be registered to map exceptions thrown during a proxied call. A default `ClientExceptionMapper` was installed that mapped exceptions thrown by the HttpClient transport layer to Resteasy specific analogs. For example, an `org.apache.http.client.ClientProtocolException` would be mapped to an `org.jboss.resteasy.client.exception.ResteasyClientProtocolException`.

These two facilities do not exist in Resteasy 3. Instead, the JAX-RS 2.0 specification mandates the use of `javax.ws.rs.ProcessingException` and `javax.ws.rs.client.ResponseProcessingException`. In particular, exceptions thrown while processing a request should be mapped to a `ProcessingException`, and exceptions thrown while processing a response should be mapped to a `ResponseProcessingException`.

For example, the `ProcessingException` javadoc lists possible conditions leading to a `ProcessingException`:

- failures in filter or interceptor chain execution
- errors caused by missing message body readers or writers for the particular Java type and media type combinations
- propagated `java.io.IOException`s thrown by `javax.ws.rs.ext.MessageBodyReader`s and `javax.ws.rs.ext.MessageBodyWriter`s during entity serialization and de-serialization

Note that `ProcessingException` and `ResponseProcessingException` represent internal problems. If the client side receives a response with status codes 3xx, 4xx or 5xx, it will map the response to an instance of `javax.ws.rs.WebApplicationException` or one of its subclasses.

1.4. Maven considerations

In Resteasy 2, the client framework lives in the resteasy-jaxrs module. In Resteasy 3, it has its own module, resteasy-client:

```
<dependency>
   <groupId>org.jboss.resteasy</groupId>
   <artifactId>resteasy-client</artifactId>
   <version>${project.version}</version>
</dependency>
```

Chapter 2. Filters and Interceptors

Interceptors are another facility from Resteasy 2 that now appear in JAX-RS 2.0 but in a rather different form. There were four kinds of interceptors in Resteasy 2:

1. reader/writer interceptors
2. server side `PreProcessInterceptor`
3. server side `PostProcessInterceptor`
4. `ClientExecutionInterceptor`

Of these, reader/writer interceptors, which wrap around the reading or writing of entities, carry over essentially unchanged, except for class and method names. `javax.ws.rs.ext.ReaderInterceptor` and `javax.ws.rs.ext.WriterInterceptor` replace the old `MessageBodyReaderInterceptor` and `MessageBodyWriterInterceptor`.

The two kinds of server side interceptors are replaced by *filters*, which behave similarly. There are four kinds of filters:

1. `ContainerRequestFilter`
2. `ContainerResponseFilter`
3. `ClientRequestFilter`
4. `ClientResponseFilter`

Like the old `PreProcessInterceptors`, `ContainerRequestFilters` can access requests. A

```
public interface PreProcessInterceptor
{
   ServerResponse preProcess(HttpRequest request, ResourceMethod method) throws
 Failure, WebApplicationException;
}
```

can modify the `HttpRequest` and then return a response or null. If it returns a response, then the execution process is interrupted and that response is returned. Similarly, a new

```
public interface ContainerRequestFilter
{
```

```
        public void filter(ContainerRequestContext requestContext) throws
 IOException;
}
```

can access and modify a JAX-RS `Request` by calling `ContainerRequestContext.getRequest()`, and it can supply a response by calling `ContainerRequestContext.abortWith(Response)`.

An old

```
public interface PostProcessInterceptor
{
   void postProcess(ServerResponse response);
}
```

can modify the response, as can a new

```
public interface ContainerResponseFilter
{
            public    void    filter(ContainerRequestContext    requestContext,
 ContainerResponseContext responseContext) throws IOException;
}
```

by calling, for example, `ContainerResponseContext.setStatus()` or `ContainerResponseContext.setEntity()`.

The situation is somewhat different with the old `ClientExecutionInterceptor`. Unlike `PreProcessInterceptor` and `PreProcessInterceptor`, this one really wraps the invocation process on the client side. That is, it can examine and/or modify the request and return by calling `ClientRequestContext.abortWith(Response)`, or proceed with the invocation and examine and/or modify the response. Two client side filters, `ClientRequestFilter` and `ClientResponseFilter`, are required to replace the functionality of `ClientExecutionInterceptor`. The former can access the request, and the latter can access both the request and response.

Chapter 3. Asynchronous HTTP Request Processing

Asynchronous request processing is another case in which a facility from Resteasy 2 has been formalized in JAX-RS 2.0. The result in Resteasy is quite similar to the old version. For example,

```
@Path("/")
public static class TestResource
{
   @GET
   @Produces("text/plain")
   public void test(@Suspend(2000) AsynchronousResponse response)
   {
      Thread t = new Thread()
      {
         @Override
         public void run()
         {
            try
            {
               System.out.println("STARTED!!!!");
               Thread.sleep(100);
               Response jaxrs = Response.ok().type("text/plain").build();
               response.setResponse(jaxrs);
            }
            catch (Exception e)
            {
               e.printStackTrace();
            }
         }
      };
      t.start();
   }
}
```

would be turned into

```
@Path("/")
public static class TestResource
{
```

```
   @GET
   @Produces("text/plain")
   public void get(@Suspended final AsyncResponse response)
   {
      response.setTimeout(2000, TimeUnit.MILLISECONDS);
      Thread t = new Thread()
      {
         @Override
         public void run()
         {
            try
            {
               System.out.println("STARTED!!!!");
               Thread.sleep(100);
               Response jaxrs = Response.ok().type("text/plain").build();
               response.setResponse(jaxrs);
            }
            catch (Exception e)
            {
               e.printStackTrace();
            }
         }
      };
      t.start();
   }
}
```

Other than the name changes, the one change to note is that the `@Suspended` annotation does not have a timeout field. Instead, the timeout can be set on the `AsyncResponse`.

Chapter 4. Validation

Validation is yet another facility that appeared as an ad hoc extension in Resteasy 2 and was later formalized in JAX-RS 2.0. In Resteasy 2, validation was implemented in the resteasy-hibernatevalidator-provider module, and it was necessary to annotate classes and/or methods with `@ValidateRequest` to enable validation.

In Resteasy, validation is implemented in the resteasy-validator-provider-11 module, and `@ValidateRequest` is no longer relevant. In fact, validation is enabled by default, as long as resteasy-validator-provider-11 is on the classpath.

Chapter 5. Resteasy Caching Features

Client and server side caching facilities are Resteasy specific extensions of JAX-RS, and they each work differently in Resteasy 2 and Resteasy 3.

5.1. Client side

Resteasy 3 offers the same client side cache facility as Resteasy 2, but it is enabled differently, by way of `org.jboss.resteasy.client.jaxrs.cache.BrowserCacheFeature`, which implements the JAX-RS 2.0 class `javax.ws.rs.core.Feature`:

```
Client client = ClientBuilder.newClient();
String url = "http://localhost:8081/orders/{id}";
ResteasyWebTarget target = (ResteasyWebTarget) client.target(url);
BrowserCacheFeature cacheFeature = new BrowserCacheFeature();
target.register(cacheFeature);
String rtn = target.resolveTemplate("id", "1").request().get(String.class);
```

Client side caching also works for proxies:

```
@Path("/orders")
public interface OrderServiceClient
{
   @Path("{id}")
   @GET
   @Produces("application/xml")
   public Order getOrder(@PathParam("id") String id);
}
...
Client client = ClientBuilder.newClient();
String "http://localhost:8081";
ResteasyWebTarget target = (ResteasyWebTarget) client.target(url);
BrowserCacheFeature cacheFeature = new BrowserCacheFeature();
target.register(cacheFeature);
OrderServiceClient orderService = target.proxy(OrderServiceClient.class);
```

5.2. Server side

As in Resteasy 2, a server side caching facility that sits in front of JAX-RS resources is made available in Resteasy 3, but the default underlying cache in Resteasy 3 is Infinispan, which supercedes the JBoss Cache project. It is highly configurable, and the documentation should be consulted for additional information: *http://infinispan.org/documentation/*.

Server side caching is also enabled differently. Resteasy 3 uses the JAX-RS 2.0 `javax.ws.rs.core.Feature` facility, in the form of `org.jboss.resteasy.plugins.cache.server.ServerCacheFeature`, which should be registered via the `javax.ws.rs.core.Application`.

Chapter 6. Miscellaneous changes

In addition to the various updated frameworks discussed in previous sections, a few individual classes have been updated or discarded.

6.1. Link

`org.jboss.resteasy.spi.Link` has been replaced by the abstract class `javax.ws.rs.core.Link` and its implementation `org.jboss.resteasy.specimpl.LinkImpl`. They both represent links as described in *RFC 5988* [https://tools.ietf.org/html/rfc5988], with slight variations. For example, there is now `javax.ws.rs.core.Link.getRel()` instead of `org.jboss.resteasy.spi.Link.getRelationship()`. Also, they are constructed differently. For example,

```
@GET
@Path("/link-header")
public Response getWithHeader(@Context UriInfo uri)
{
   URI subUri = uri.getAbsolutePathBuilder().path("next-link").build();
   Link link = new Link();
   link.setHref(subUri.toASCIIString());
   link.setRelationship("nextLink");
   return Response.noContent().header("Link", link.toString()).build();
}
```

would now be written

```
@GET
@Path("/link-header")
public Response getWithHeader(@Context UriInfo uri)
{
   URI subUri = uri.getAbsolutePathBuilder().path("next-link").build();
   Link link = new LinkBuilderImpl().uri(subUri).rel("nextLink").build();
   return Response.noContent().header("Link", link.toString()).build();
}
```

6.2. GenericType

`org.jboss.resteasy.util.GenericType`, which allows the creation of parameterized type objects at runtime, is now replaced by `javax.ws.rs.core.GenericType`. They are essentially the same class, with minor method name changes. In particular, `getGenericType()` becomes `getType()` and `getType()` becomes `getRawType()`.

6.3. StringConverter

Implementations of the `org.jboss.resteasy.spi.StringConverter` interface in Resteasy 2 are providers that can marshal and unmarshal string-based parameters labelled with @HeaderParam, @MatrixParam, @QueryParam, or @PathParam. JAX-RS 2.0 introduces a similar interface, `javax.ws.rs.ext.ParamConverter`, but implementations of `ParamConverter` are not recognized as providers. Rather, a provider that implements `javax.ws.rs.ext.ParamConverterProvider`, which produces a `ParamConverter`, may be registered. For example,

```
public static class POJO { ... }

public static class POJOConverter implements ParamConverter<POJO>
{
   public POJO fromString(String str)
   {
      POJO pojo = new POJO();
      return pojo;
   }

   public String toString(POJO value)
   {
      return value.getName();
   }
}

public static class POJOConverterProvider implements ParamConverterProvider
{
   @Override
   public <T> ParamConverter<T> getConverter(Class<T> rawType, Type genericType,
 Annotation[] annotations)
   {
      if (!POJO.class.equals(rawType)) return null;
      return (ParamConverter<T>)new POJOConverter();
   }
}

...
ResteasyProviderFactory.getInstance().registerProvider(POJOConverterProvider.class);
```

```
...
```

6.4. Logger

Resteasy 2 comes with a logging abstraction called `org.jboss.resteasy.logging.Logger`, extensions of which delegate to logging frameworks such as log4j and slf4j. Resteasy 3 no longer uses its own logging abstraction but rather adopts the JBoss Logging framework, a brief description of which can be found at *http://docs.jboss.org/hibernate/orm/4.3/topical/html/logging/Logging.html*. JBoss Logging was chosen for its internationalization and localization support.