JAX-RS Filters

JAX-RS Filters

Filters

Often it's necessary to pre- or post-process some requests according to a number of requirements.
For example, a request like

GET /resource?_type=xml is supported by a CXF specific RequestPreprocessor code which modifies an input message
by updating one of its headers.

A standard mechanism for updating the request or response properties is to use JAX-RS 2.0 ContainerRequestFilter or ContainerResponseFilter.

ContainerRequestFilters with a @PreMatching annotation are run before the JAX-RS resource selection process starts. PreMatching filters should be used to modify request URI or headers or input stream. Post-matching filters are run just before a selected resource method is executed.

Multiple ContainerRequestFilter and ContainerResponseFilter filters can be ordered using a @Priority annotation.

The implementations can be registered like any other type of providers :

```
<beans>
<jaxrs:server id="customerService" address="/">
    <jaxrs:serviceBeans>
      <bean class="org.CustomerService" />
    </jaxrs:serviceBeans>

    <jaxrs:providers>
      <ref bean="authorizationFilter" />
    </jaxrs:providers>
    <bean id="authorizationFilter" class="com.bar.providers.
AuthorizationContainerRequestFilter">
        <!-- authorization bean properties -->
    </bean>
</jaxrs:server>
</beans>
```

Difference between JAXRS filters and CXF interceptors

JAXRS runtime flow is mainly implemented by a pair of 'classical' CXF interceptors. JAXRSInInterceptor is currently at Phase.UNMARSHAL (was at Phase.PRE_STREAM before CXF 2.2.2) phase while JAXRSOutInterceptor is currently at Phase.MARSHAL phase.

JAXRS filters can be thought of as additional handlers. JAXRSInInterceptor deals with a chain of Pre and Post Match ContainerRequestFilters, just before the invocation. JAXRSOutInterceptor deals with a chain of ContainerResponseFilters, just after the invocation but before message body writers get their chance.

Sometimes you may want to use CXF interceptors rather than writing JAXRS filters. For example, suppose you combine JAXWS and JAXRS and you need to log only inbound or outbound messages. You can reuse the existing CXF interceptors :

```
<beans>
<bean id="logInbound" class="org.apache.cxf.interceptor.
LoggingInInterceptor"/>
<bean id="logOutbound" class="org.apache.cxf.interceptor.
LoggingOutInterceptor"/>

<jaxrs:server>
 <jaxrs:inInterceptors>
     <ref bean="logInbound"/>
 </jaxrs:inInterceptors>
 <jaxrs:outInterceptors>
    <ref bean="logOutbound"/>
 </jaxrs:outInterceptors>
</jaxrs:server>

<jaxws:endpoint>
 <jaxws:inInterceptors>
     <ref bean="logInbound"/>
 </jaxws:inInterceptors>
 <jaxws:outInterceptors>
    <ref bean="logOutbound"/>
 </jaxws:outInterceptors>
</jaxws:endpoint>

</beans>
```

Reusing other CXF interceptors/features such as GZIP handlers can be useful too.

Overriding request and response properties

Now and then one needs to overwrite various request and response properties like HTTP method or request URI,
response headers or status codes and even the request or response body. JAX-RS Response may be used to specify custom status and response headers but it might be intrusive to add it to method signatures.

Using filters and interceptors makes it possible to override all the needed request/response properties.

Overriding HTTP method

Use @PreMatching ContainerRequestFilter or register a custom CXF in intrerceptor filter which will replace the current method value keyed by Message.HTTP_REQUEST_METHOD in a given Message.

Overriding request URI, query and headers

One can do it either from @PreMatching ContainerRequestFilter or CXF input interceptor (registered at the early phase like USER_STREAM), for example :

```
String s = m.get(Message.REQUEST_URI);
s += "/data/";
m.put(Message.REQUEST_URI, s);
```

If the updated Request URI has a new query string, then you also need to update a Message.QUERY_STRING property.

Similarly, one can update request HTTP headers, by modifying a Message.REQUEST_HEADERS Message object which is a Map containing String and List of Strings entries.

Overriding response status code and headers

It is assumed here a user prefers not to use explicit Response objects in the application code.
This can be done either from ContainerResponseFilter or CXF output interceptor (phase like MARSHALL will do), for example this code will work for both JAXRS and JAXWS :

```
import java.util.Map;
import java.util.TreeMap;

import org.apache.cxf.message.Message;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;


public class CustomOutInterceptor extends
AbstractOutDatabindingInterceptor {

    public CustomOutInterceptor() {
        super(Phase.MARSHAL);
    }

    public void handleMessage(Message outMessage) {
        Map<String, List<String>> headers = (Map<String, List<String>>)
outMessage.get(Message.PROTOCOL_HEADERS);
        if (headers == null) {
            headers = new TreeMap<String, List<String>>(String.
CASE_INSENSITIVE_ORDER);
            message.put(Message.PROTOCOL_HEADERS, headers);
        }
        // modify headers
    }
```

At the moment it is not possible to override a response status code from a CXF interceptor running before JAXRSOutInterceptor, like CustomOutInterceptor above, which will be fixed.
The only option at the moment is to use a custom ResponseHandler which will replace the current Response object with another one containing the required status.

Ignoring JAXRS MessageBodyWriters

In some cases you may want to have a JAXRS Response entity which a given filter has produced to be directly written to the output stream. For example, a CXF JAXRS WADLGenerator filter produces an XML content which does not have to be serialized by JAXRS MessageBodyWriters. If you do need to have the writers ignored then set the following property on the current exchange in the custom handler :

```
message.getExchange().put("ignore.message.writers", true);
```

Custom invokers

Note This feature is available starting from CXF 2.2.2

Using custom JAXR-RS invokers is yet another way to pre or post process a given invocation. For example, this invoker does a security check before delegating to the default JAXRS invoker. A custom invoker, like a request filter, has the access to all the information accumulated during the processing of a given call, but additionally, it can also check the actual method parameter values.

Custom invokers can be registered like this :

```
<beans>

<jaxrs:server address="/">
 <jaxrs:invoker>
   <bean class="org.apache.cxf.systest.jaxrs.CustomJAXRSInvoker"/>
 </jaxrs:invoker>
</jaxrs:server>

</beans>
```