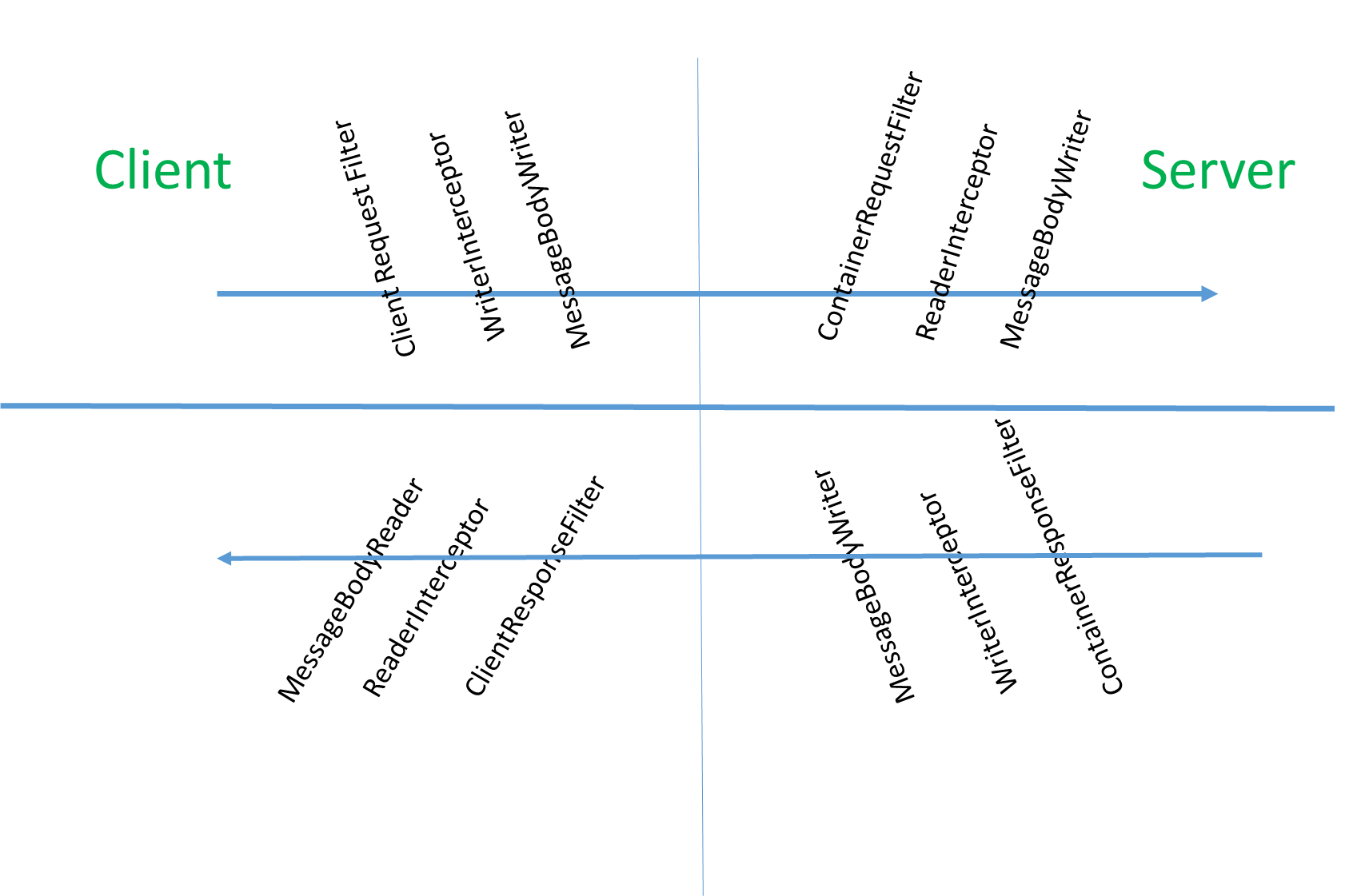
<https://jersey.github.io/documentation/latest/modules-and-dependencies.html>



All above are of type provider so while implementing these interfaces we need to place @Provider annotation above implemented class

**public** **interface** ContainerRequestFilter {

**public** **void** filter(ContainerRequestContext requestContext) **throws** IOException;

}

**public** **interface** ContainerResponseFilter {

**public** **void** filter(ContainerRequestContext requestContext, ContainerResponseContext responseContext)**throws** IOException;

}

ContainerRequestFilter is call before request reaches rest service and ContainerResponseFilter is called before the response is sent to the client.

With request filters we can change the request parameters and with response filters we can change the response header parameters.

**public** **interface** MessageBodyReader<T> {

**public** **boolean** isReadable(Class<?> type, Type genericType,

Annotation[] annotations, MediaType mediaType);

**public** T readFrom(Class<T> type, Type genericType,

Annotation[] annotations, MediaType mediaType,

MultivaluedMap<String, String> httpHeaders,

InputStream entityStream) **throws** java.io.IOException, javax.ws.rs.WebApplicationException;

}

**public** **interface** MessageBodyWriter<T> {

**public** **boolean** isWriteable(Class<?> type, Type genericType,

Annotation[] annotations, MediaType mediaType);

**public** **long** getSize(T t, Class<?> type, Type genericType, Annotation[] annotations,

MediaType mediaType);

**public** **void** writeTo(T t, Class<?> type, Type genericType, Annotation[] annotations,

MediaType mediaType,

MultivaluedMap<String, Object> httpHeaders,

OutputStream entityStream)

**throws** java.io.IOException, javax.ws.rs.WebApplicationException;

}

**public** **interface** ReaderInterceptor {

**public** Object aroundReadFrom(ReaderInterceptorContext context)

**throws** java.io.IOException, javax.ws.rs.WebApplicationException;

}

**public** **interface** WriterInterceptor {

**void** aroundWriteTo(WriterInterceptorContext context)

**throws** java.io.IOException, javax.ws.rs.WebApplicationException;

}

**public** **interface** ClientRequestFilter {

**public** **void** filter(ClientRequestContext requestContext) **throws** IOException;

}

**public** **interface** ClientResponseFilter {

**public** **void** filter(ClientRequestContext requestContext, ClientResponseContext responseContext)

**throws** IOException;

}

**Filters are one of the important features provided by the JAX-RS framework, and they are used in various contexts when RESTful web services are developed.**

Filters are used to modify request and response entities, headers, and other parameters. In this article, we will explore different types of filters and different ways to use them. All the code examples are based on the Jersey framework.

**Note**: The source code for all the examples in this article is available on [GitHub](https://github.com/cloudskol/restskol).

**Types of Filters**

Filters are classified under two major categories: server (container) filters and client filters. These two categories are further divided into request filters and response filters, as shown in Figure 1.

**Figure 1. Types of filters**

**Container Filters**

The following sections describe container filters and an available annotation.

**Container Response Filters**

When we want to change the response header or modify some parameters after the resource method is executed, we can use a container response filter. For that, we have to provide an implementation of the ContainerResponseFilter interface, as shown in Listing 1.

1. public class RestSkolResponseFilter implements ContainerResponseFilter {
2. @Override
3. public void filter(ContainerRequestContext containerRequestContext,
4. ContainerResponseContext containerResponseContext) throws IOException {
5. containerResponseContext.getHeaders().add("X-Powered-By", "RESTSkol");
6. }
7. }

**Listing 1. Example container response filter**

This has to be registered as a provider in our resource configuration. In the example in Listing 1, we have injected a parameter called X-Powered-By in the response. (This example is inspired from the Jersey website). In this way, we can easily inject any header value that is common across all our responses.

**Container Request Filters**

A container request filter is useful when we want to enforce some kind of validation before executing the resource method. So a container request filter will act as a shield for our resource methods. We can enforce any kind of validation, and if the validation fails we can abort the resource execution at that point.

In the example in Listing 2, we are enforcing some kind of validation for the API key in our resource method call. If the expected API key is not available in the request header, then further call execution is stopped and a response is generated from this layer itself.

1. public class APIKeyCheckRequestFilter implements
2. ContainerRequestFilter { private static final String API\_KEY = "X-API-KEY";
3. @Override
4. public void filter(ContainerRequestContext containerRequestContext) throws
5. IOException { final String apiKey =
6. containerRequestContext.getHeaders().getFirst(API\_KEY); System.out.println("API KEY: " + apiKey);
8. if (apiKey == null ||
9. apiKey.isEmpty()) {
10. containerRequestContext
11. .abortWith(
12. Response
13. .status(Response.Status.UNAUTHORIZED)
14. .entity("Please provide a valid API Key")
15. .build());
16. }
17. }
18. }

**Listing 2. Example container request filter**

**@PreMatching Annotation**

By default, the container request filter implementation is executed after the appropriate request execution method is found. Here we will *not* have control over finding or choosing the required request method.

To achieve this behavior, we have the @PreMatching annotation for container request filters. If any container request filter is annotated with @PreMatching, then we can influence the request method. In the implementation shown in Listing 3, we are delegating the /books/v1 request to the /books/v2 resource. This might be useful when we want to deprecate any of the resource implementations.

1. @PreMatching
2. public class PreMatchingFilter implements ContainerRequestFilter {
3. @Override
4. public void filter(ContainerRequestContext containerRequestContext) throws
5. IOException { final URI absolutePath =
6. containerRequestContext.getUriInfo().getAbsolutePath(); String path =
7. absolutePath.getPath();
8. System.out.println("Path: " + path);
10. if (path.contains("/books/v1")) {
11. path = path.replace("/books/v1", "/books/v2");
12. }
13. System.out.println("After replaced Path: " + path);
14. try {
15. containerRequestContext.setRequestUri(new URI(path));
16. } catch (URISyntaxException e) {
17. e.printStackTrace();
18. }
19. }
20. }

**Listing 3. Example of using the @PreMatching annotation**

**Client Filters**

Two variants of client filters are available: client request filters and client response filters. These filters are applicable only at the client API layer. This means that when we use the Jersey (JAX-RS) client API to call the REST endpoints, we can add these filters as part of the Client object.

**Client Request Filters**

This filter is the starting point of the filter workflow. If the client request filter is registered with the Client object, then this will be triggered first before the request has been sent to the server. So we can use this filter when we want to enforce some kind of validation at the client layer itself.

The purpose of the client request filter shown in Listing 4 is to validate when something is calling the PATCH method. We can enforce a validation to prevent such unsupported methods. If this call is not validated successfully, we can abort the request and the complete workflow will come to an end. The request will be passed to the server level once this validation is satisfied.

1. public class RestSkolClientRequestFilter implements ClientRequestFilter {
2. private static final Logger logger =
3. LogManager.getLogger(RestSkolClientRequestFilter.class);
5. @Override
6. public void filter(ClientRequestContext clientRequestContext) throws IOException {
7. logger.info("Client request called");
8. final String methodName =
9. clientRequestContext.getMethod(); if
10. (methodName.equals("PATCH")) {
11. clientRequestContext.abortWith(          Response.status(Response.Status.METHOD\_NOT\_ALLOWED)
12. .entity("HTTP Patch is NOT supported")
13. .build());
14. }
15. }
16. }

**Listing 4. Example client request filter**

**Client Response Filters**

A client response filter is the final step in the filter workflow. Once the execution is completed at the server level, the server layer will generate either a successful response or a failure response. This response will be passed to the client layer. A client response filter is the one that will be executed once the response is received from the server. We can validate or inject further entity data into the response at the client response filter level. Then, the response will be passed back to the caller.

**How to Use Client Filters**

As I already mentioned, client filters (both request and response) should be registered with the Client object, as shown in Listing 5.

1. final Client client = ClientBuilder.newBuilder()
2. .register(RestSkolClientRequestFilter.class)
3. .register(RestSkolClientResponseFilter.class)
4. .build();

**Listing 5. Example of registering client filters**

Then these client filters will be enforced during the method call.

**Filter Execution Workflow and Output**

Figure 2 depicts the complete workflow or steps of the filter execution path.

**Figure 2. Steps in the filter execution path**

The following output shows the execution order that occurs if all of the above filter implementation is in place. Check out the source code and run the project locally to see the following output locally. The output should exact match the order depicted in Figure 2.

1. [INFO ] 2016-04-12 23:29:38.212 [main] RestSkolClientRequestFilter - Client request called
3. [INFO ] 2016-04-12 23:29:38.509 [http-bio-8080-exec-1] PreMatchingFilter - Pre matching container request filter
5. [INFO ] 2016-04-12 23:29:38.519 [http-bio-8080-exec-1] APIKeyCheckRequestFilter - Container request filter
7. [INFO ] 2016-04-12 23:29:38.530 [http-bio-8080-exec-1] RestSkolResponseFilter - Container response filter
9. [INFO ] 2016-04-12 23:29:38.935 [main] RestSkolClientResponseFilter - Client response filter