**ADVANCE JAX-RS**

**Advanced JAX-RS 04 - Setting up a JAX RS application , Advanced JAX-RS 05 - The Application class**

package com.tcs.kunj.rest;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.Produces;

import javax.ws.rs.core.MediaType;

@Path("test")

public class MyResource {

@GET

@Produces(MediaType.TEXT\_PLAIN)

public String testMethod() {

return "It works!";

}

}

package com.tcs.kunj.rest;

import java.util.HashSet;

import java.util.Set;

import javax.ws.rs.ApplicationPath;

import javax.ws.rs.core.Application;

// this class is meant to configure this application as a JAX-RS application

@ApplicationPath("webapi") // Maps this application to the root URL (/webapi)

public class MyApp extends Application{ //This tells jersey that this is a JAX-RS application

// This MyApp class searches for all the resource available in classpath to look for path annotation

/\* However if you want to restrict some of the classes (or to include certain resources) you can override the default implementation of getClasses() method

\*/

// if you do not implement the below method, then all the classes in the calsspath with @Path annotation will be considered.

public Set<Class<?>> getClasses(){

// Here you can add your classes

return new HashSet<Class<?>>();

}

}

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

**Advanced JAX-RS 06 - Resource Life Cycle**

package com.tcs.kunj.rest;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.Produces;

import javax.ws.rs.core.MediaType;

@Path("test")

public class MyResource {

private int count;

@GET

@Produces(MediaType.TEXT\_PLAIN)

public String testMethod() {

count++;

return "It works! This method was called " + count + " time(s)";

}

}

[**http://localhost:8080/advanced-jaxrs-02/webapi/test**](http://localhost:8080/advanced-jaxrs-02/webapi/test)

It works! This method was called 1 time(s)

No matter how many times you enter the above url, you get the same result. count does not increase. WHY?

It is happening because count is reset to zero every time. Every time you are making a request, JAX-RS is creating a new instance of Myresource class. Every request results in a new response. This happens to be the default behavior of the resource in JAX-RS. They are set to be request scoped.

Using @singleton annotation for class you can create just one instance for a class and JAX-RS just holds on that instance no matter how many times you make a request. This instance gets destroyed once you stop the server.

@ singleton

public class MyResource {

---------

}

[**http://localhost:8080/advanced-jaxrs-02/webapi/test**](http://localhost:8080/advanced-jaxrs-02/webapi/test)

It works! This method was called 2 time(s)

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

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

**Advanced JAX-RS 07 - Param Annotations and member variables**

The {pathParam} in the path annotation lets you map a resource to a variable path param.

@PathParam and @QueryParam applies to member variables as well.

package com.tcs.kunj.rest;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.PathParam;

import javax.ws.rs.Produces;

import javax.ws.rs.QueryParam;

import javax.ws.rs.core.MediaType;

@Path("{pathParam}/test")

public class MyResource {

@PathParam("pathParam") private String pathParamExample;

@QueryParam("query") private String queryParamExample;

@GET

@Produces(MediaType.TEXT\_PLAIN)

public String testMethod() {

return "It works! Path param used " + pathParamExample + " Query param used "+ queryParamExample;

}

}

<http://localhost:8080/advanced-jaxrs-03/webapi/value/test?query=queryParam>

It works! Path param used value Query param used queryParam

Why does the above code work?

This works because the instance of MyResource is created after you make the above request. So there is new instance per request. When JAX-RS creates e new instance of the MyResource , the member variables are going to be initialized to blank values. JAX-RS is going to set the variables in the url to the member variables of the instance of MessageResource class. Since instance is created after the request, JAX-RS has all the values to populate and it is accessible at the method level. This applies to all the values that you can gather from a url.

Q. Why to annotate member variables in the class in place of annotating the method arguments?

ANS: If other methods also want to use the same member variables.

Q. What happens if I make the above class as singleton?

ANS: The instance of the MyResource class gets created before the request. We get the below error:

org.glassfish.jersey.server.model.ModelValidationException: Validation of the application resource model has failed during application initialization.

[[FATAL] Parameter pathParamExample of private java.lang.String com.tcs.kunj.rest.MyResource.pathParamExample cannot be injected into singleton resource.; source='private java.lang.String com.tcs.kunj.rest.MyResource.pathParamExample', [FATAL] Parameter queryParamExample of private java.lang.String com.tcs.kunj.rest.MyResource.queryParamExample cannot be injected into singleton resource.; source='private java.lang.String com.tcs.kunj.rest.MyResource.queryParamExample']

**NOTE :** Singleton resources are instantiated during application startup. So, you cannot inject request-specific information to their member variables.

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

**Advanced JAX-RS 08 – ParamConverters**

All the params in url (sent over http) are passed as String. Jersey makes conversion of these params into int, double etc data types. Jersey comes with set of bundled param converters that know how to convert from String to the right data type.

The above conversion happens well for native data types. But what if jersey has to convert String to a user defined data type that jersey is not aware of. The solution is to create your own param converters.

**package** com.tcs.kunj.rest;

**public** **class** MyDate { // Creating my own data type

**private** **int** date;

**private** **int** month;

**private** **int** year;

**public** **int** getDate() {

**return** date;

}

**public** **void** setDate(**int** date) {

**this**.date = date;

}

**public** **int** getMonth() {

**return** month;

}

**public** **void** setMonth(**int** month) {

**this**.month = month;

}

**public** **int** getYear() {

**return** year;

}

**public** **void** setYear(**int** year) {

**this**.year = year;

}

}

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

package com.tcs.kunj.rest;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.PathParam;

import javax.ws.rs.Produces;

import javax.ws.rs.core.MediaType;

@Path("date/{dateString}")

public class DateResource {

@GET

@Produces(MediaType.TEXT\_PLAIN)

public String getRequestedDate(@PathParam("dateString") MyDate myDate) {

return "This is " + myDate.toString();

}

}

While loading the application we get the following error:

org.glassfish.jersey.server.model.ModelValidationException: Validation of the application resource model has failed during application initialization.

[[FATAL] No injection source found for a parameter of type public java.lang.String com.tcs.kunj.rest.DateResource.getRequestedDate(com.tcs.kunj.rest.MyDate) at index 0.; source='ResourceMethod{httpMethod=GET, consumedTypes=[], producedTypes=[text/plain], suspended=false, suspendTimeout=0, suspendTimeoutUnit=MILLISECONDS, invocable=Invocable{handler=ClassBasedMethodHandler{handlerClass=class com.tcs.kunj.rest.DateResource, handlerConstructors=[org.glassfish.jersey.server.model.HandlerConstructor@4212c8d]}, definitionMethod=public java.lang.String com.tcs.kunj.rest.DateResource.getRequestedDate(com.tcs.kunj.rest.MyDate), parameters=[Parameter [type=class com.tcs.kunj.rest.MyDate, source=dateString, defaultValue=null]], responseType=class java.lang.String}, nameBindings=[]}']

The above error says that jersey does not have a converter that can convert from String(in url) to MyDate. It does not even load the application. It throws the error upfront.

Q. How do we let Jersey know how to do this conversion?

ANS: Using ParamConverter and ParamConverterProvider

ParamConverter

Whenever you need to convert from String to custom data type, you need to implement a ParamConverter. It is an interface that has methods to convert from String to a custom data type.

Data type instance 🡨 T fromString(String value) 🡨 String

Once you implement ParamConverter, you also need to implement ParamConverterProvider. ParamConverterProvider is a provider class which tells jersey to use this ParamConverter class given the type.

ParamConverterProvider

ParamConverter 🡨 ParamConverter getConverter(Class rawType, …) 🡨 Type

Jersey will ask all ParamConverterProvider if it does not know how to convert from String to a particular data type.

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

**Advanced JAX-RS 09 - Implementing Custom ParamConverters**

The param annotations maps incoming request parameters, so we are just interested in String-to-Datatype conversions here.

package org.koushik.javabrains.rest;

import java.lang.annotation.Annotation;

import java.lang.reflect.Type;

import java.util.Calendar;

import javax.ws.rs.ext.ParamConverter;

import javax.ws.rs.ext.ParamConverterProvider;

import javax.ws.rs.ext.Provider;

@Provider

public class MyDateConverterProvider implements ParamConverterProvider {

@Override

public <T> ParamConverter<T> getConverter(final Class<T> rawType, Type genericType, Annotation[] annotations) {

if (rawType.getName().equals(MyDate.class.getName())) {

// This ParamConverter converts from String to what we want

return new ParamConverter<T>() {

@Override

public T fromString(String value) {

Calendar requestedDate = Calendar.getInstance();

if ("tomorrow".equalsIgnoreCase(value)) {

requestedDate.add(Calendar.DATE, 1);

}

else if ("yesterday".equalsIgnoreCase(value)) {

requestedDate.add(Calendar.DATE, -1);

}

MyDate myDate = new MyDate();

myDate.setDate(requestedDate.get(Calendar.DATE));

myDate.setMonth(requestedDate.get(Calendar.MONTH));

myDate.setYear(requestedDate.get(Calendar.YEAR));

return rawType.cast(myDate); // returning raw type

}

@Override

public String toString(T myBean) {

if (myBean == null) {

return null;

}

return myBean.toString();

}

};

}

return null;

}

}

<http://localhost:8080/advanced-jaxrs-04/webapi/date/today>

Got MyDate [date=9, month=4, year=2018]

<http://localhost:8080/advanced-jaxrs-04/webapi/date/tomorrow>

Got MyDate [date=10, month=4, year=2018]

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

**Advanced JAX-RS 10 - MessageBodyReaders and MessageBodyWriters**

MessageBodyReaders and MessageBodyWriters are what converts from the raw content (that gets sent over the request and response) to java types and vice versa.

For ex, conversion from java object ot JSON and vice versa. When json (coming through a request) gets converted to java object, it with the help of MessageBodyWriter. The vice versa is accomplished with the help of MessageBodyWriter. These two classes are very helpful in doing marshalling and unmarshalling in java.

**MessageBodyReader and MessageBodyWriter VS ParamCoverter**

**MessageBody vs param**

MessageBody deals with the content of the message(request or response). Param are the parameters to the request but not the request itself.

For ex:

PUT /webapi/users/kunj 🡪 Here ‘kunj’ is a Path param so if you need to deal with it, you will need a param converter.

{

“data” : “……”

} 🡪 Message body (content of the request). You will need Message BodyReader and MessageBodyWriter to deal with it.

-------------------------------------------------------------------------------------------------------------------------------**Advanced JAX-RS 12 - Custom Media Types**

You can also create your own media type and implement your MessagBodyReader and MessageBodyWriter.

MyResource.java

package org.koushik.javabrains.rest;

import java.util.Calendar;

import java.util.Date;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.Produces;

import javax.ws.rs.core.MediaType;

@Path("test")

public class MyResource {

@GET

@Produces(value = "text/shortdate" ) // We will have to write custom MessageBedyWriter for this Date to TEXT\_PLAIn conversion as Jersey does not know how to convert from Date to TEST\_PLAIN.

public Date testMethod() {

return Calendar.getInstance().getTime();

}

}

ShortDateMessageBodyWriter.java

package org.koushik.javabrains.rest;

import java.io.IOException;

import java.io.OutputStream;

import java.lang.annotation.Annotation;

import java.lang.reflect.Type;

import java.util.Date;

import javax.ws.rs.Produces;

import javax.ws.rs.WebApplicationException;

import javax.ws.rs.core.MediaType;

import javax.ws.rs.core.MultivaluedMap;

import javax.ws.rs.ext.MessageBodyWriter;

import javax.ws.rs.ext.Provider;

@Provider

@Produces("text/shortdate")

public class ShortDateMessageBodyWriter implements MessageBodyWriter<Date> {

@Override

public long getSize(Date arg0, Class<?> arg1, Type arg2, Annotation[] arg3,

MediaType arg4) {

return -1;

}

@Override

public boolean isWriteable(Class<?> type, Type arg1, Annotation[] arg2,

MediaType arg3) {

return Date.class.isAssignableFrom(type);

}

@Override

public void writeTo(Date date,

Class<?> type,

Type type1,

Annotation[] antns,

MediaType mt,

MultivaluedMap<String, Object> mm,

OutputStream out) throws IOException, WebApplicationException {

String shortDate = date.getDate() + "-" + date.getMonth() + "-" + date.getYear();

out.write(shortDate.getBytes());

}

}

Q.) What if we make these changes to MyResource.java.

Produces(value = { MediaType.TEXT\_PLAIN, "text/shortdate" })

Ans. Based on the Accept header in the request, jersey will decide the below:

1. If the accept header is nothing or text/plain, it will call the ‘DateMessageBodyWriter.java’ . If the Accept header is ‘text/shortdate’ jersey will call ShortDateMessageBodyWriter.java.

Note: Some people use ‘Custom media type ’ concept for versioning.

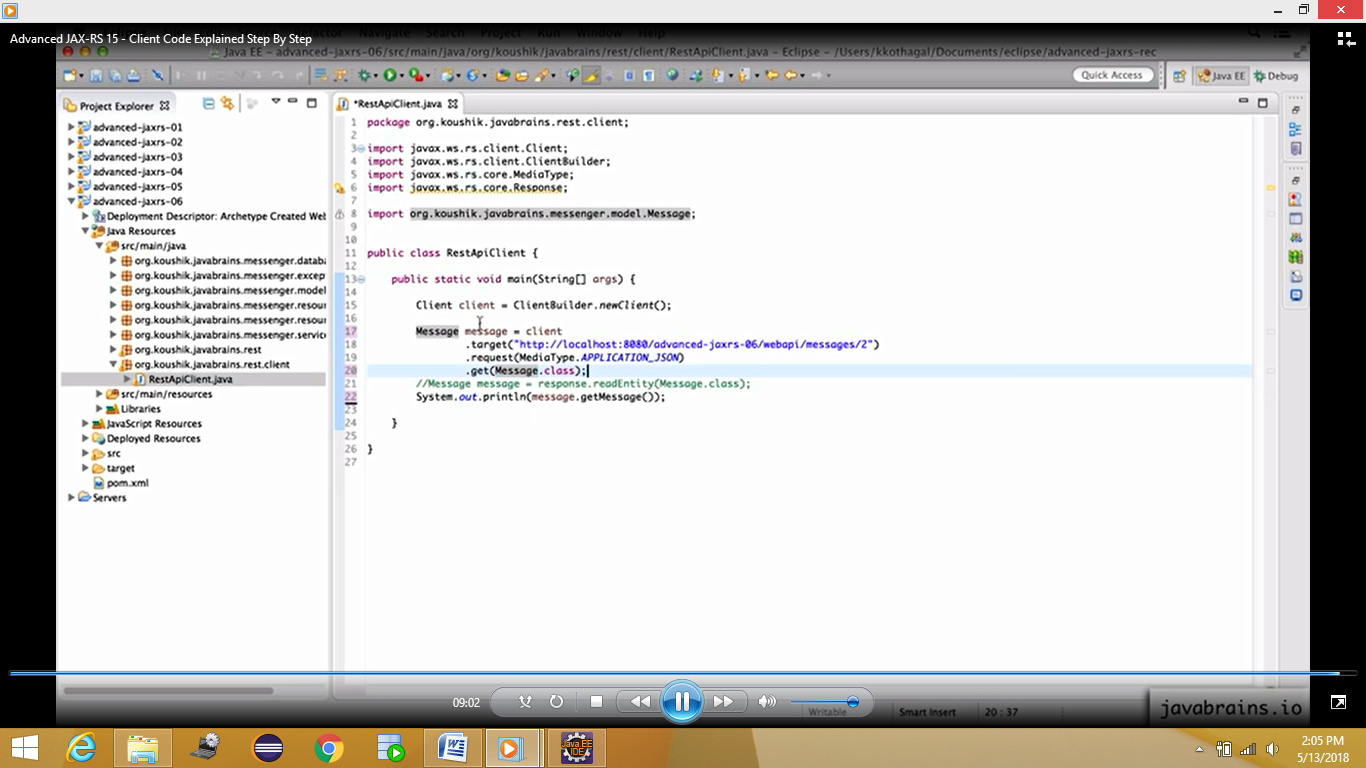
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

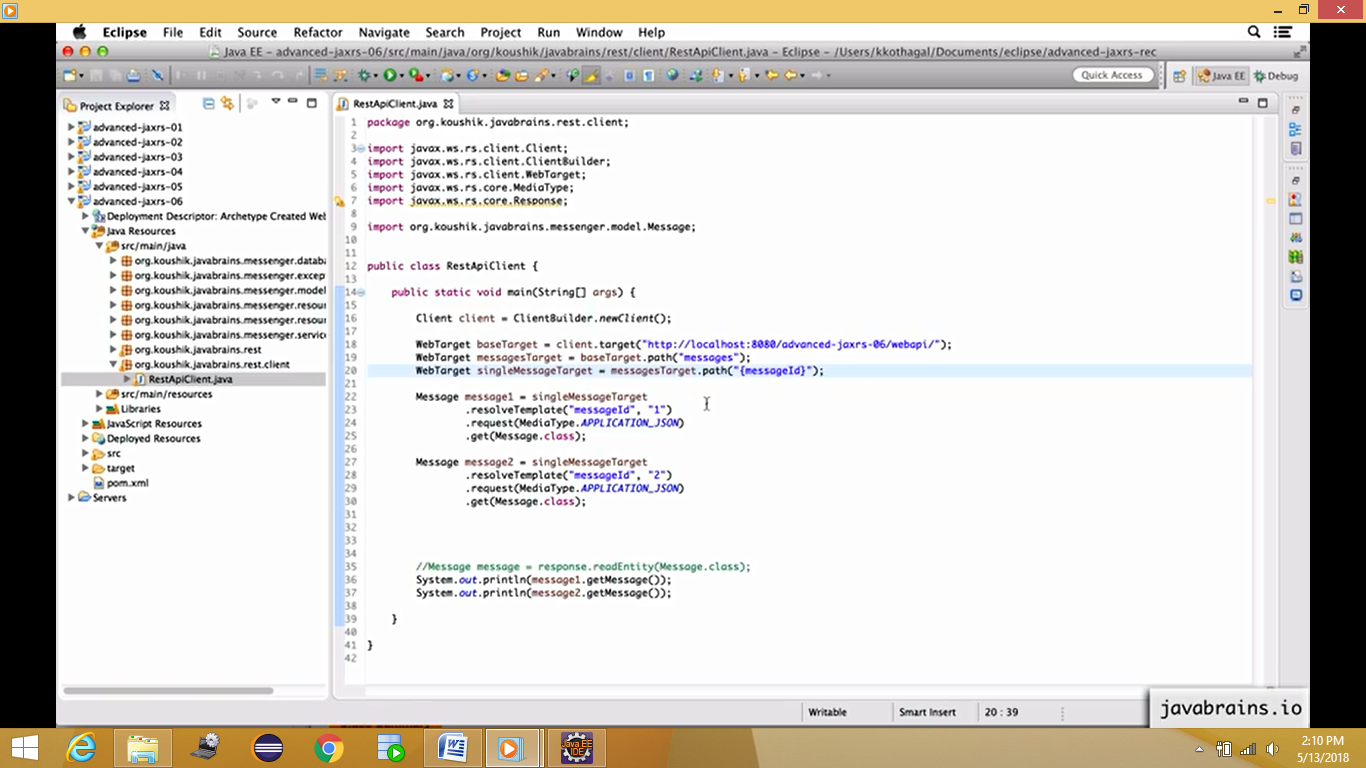
**Advanced JAX-RS 13 - JAX RS Client**

**Advanced JAX-RS 14 - Writing a JAX-RS client**

**Advanced JAX-RS 15 - Client Code Explained Step By Step**

**Advanced JAX-RS 16 - Some Best Practices**





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

**Advanced JAX-RS 17 - Making a POST request**

package org.koushik.javabrains.rest.client;

import javax.ws.rs.client.Client;

import javax.ws.rs.client.ClientBuilder;

import javax.ws.rs.client.Entity;

import javax.ws.rs.client.WebTarget;

import javax.ws.rs.core.MediaType;

import javax.ws.rs.core.Response;

import org.koushik.javabrains.messenger.model.Message;

public class RestApiClient {

public static void main(String[] args) {

Client client = ClientBuilder.newClient();

WebTarget baseTarget = client.target("http://localhost:8080/advanced-jaxrs-06/webapi/");

WebTarget messagesTarget = baseTarget.path("messages");

WebTarget singleMessageTarget = messagesTarget.path("{messageId}");

Message message1 = singleMessageTarget

.resolveTemplate("messageId", "1")

.request(MediaType.APPLICATION\_JSON)

.get(Message.class);

Message message2 = singleMessageTarget

.resolveTemplate("messageId", "2")

.request(MediaType.APPLICATION\_JSON)

.get(Message.class);

Message newMessage = new Message(4, "My New message from JAX-RS client", "koushik");

Response postResponse = messagesTarget

.request()

.post(Entity.json(newMessage)); // post method expects a type of Entity instance -- Sending a java object as json

if (postResponse.getStatus() != 201) {

System.out.println("Error");

}

Message createdMessage = postResponse.readEntity(Message.class);

System.out.println(createdMessage.getMessage());

}

}

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

**Advanced JAX-RS 18 - Creating Invocations**

The idea behind invocation is that you can prepare a requesnd you can put all the request that you want that request to do such as ; request target, request format, body of the request, method. Some other code will invoke this request. So you create an object called Invocation which creates all the details about the request and then you invoke it somewhere else.

package org.koushik.javabrains.rest.client;

import javax.ws.rs.client.Client;

import javax.ws.rs.client.ClientBuilder;

import javax.ws.rs.client.Invocation;

import javax.ws.rs.client.WebTarget;

import javax.ws.rs.core.MediaType;

import javax.ws.rs.core.Response;

public class InvocationDemo {

public static void main(String[] args) {

InvocationDemo demo = new InvocationDemo();

Invocation invocation = demo.prepareRequestForMessagesByYear(2015);

Response response = invocation.invoke();

System.out.println(response.getStatus());

}

public Invocation prepareRequestForMessagesByYear(int year) {

Client client = ClientBuilder.newClient();

return client.target("http://localhost:8080/advanced-jaxrs-06/webapi/")

.path("messages")

.queryParam("year", year) .request(MediaType.APPLICATION\_JSON)

.buildGet();

}

}

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

Advanced JAX-RS 19 - Handling Generic Types

JAX-RS will not be able to provide the response in the form of generic type (for ex: List.class).

GenericDemo.java

package org.koushik.javabrains.rest.client;

import java.util.List;

import javax.ws.rs.client.Client;

import javax.ws.rs.client.ClientBuilder;

import javax.ws.rs.core.GenericType;

import javax.ws.rs.core.MediaType;

import javax.ws.rs.core.Response;

import org.koushik.javabrains.messenger.model.Message;

public class GenericDemo {

public static void main(String[] args) {

Client client = ClientBuilder.newClient();

List response= client.target("http://localhost:8080/advanced-jaxrs-06/webapi/")

.path("messages")

.queryParam("year", 2015)

.request(MediaType.APPLICATION\_JSON)

.get(List.class); // This line will throw exception as jax-rs does not have MessagBodyWriter to convert from java object to generic type (List).

System.out.println(response);

}

}

Exception in thread "main" org.glassfish.jersey.message.internal.MessageBodyProviderNotFoundException: MessageBodyReader not found for media type=application/json, type=interface java.util.List, genericType=interface java.util.List.

The correct code for GenericDemo.java is:

package org.koushik.javabrains.rest.client;

import java.util.List;

import javax.ws.rs.client.Client;

import javax.ws.rs.client.ClientBuilder;

import javax.ws.rs.core.GenericType;

import javax.ws.rs.core.MediaType;

import javax.ws.rs.core.Response;

import org.koushik.javabrains.messenger.model.Message;

public class GenericDemo {

public static void main(String[] args) {

Client client = ClientBuilder.newClient();

List<Message> messages = client.target("http://localhost:8080/advanced-jaxrs-06/webapi/")

.path("messages")

.queryParam("year", 2015)

.request(MediaType.APPLICATION\_JSON)

.get(new GenericType<List<Message>>() { });

System.out.println(messages);

}

}

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

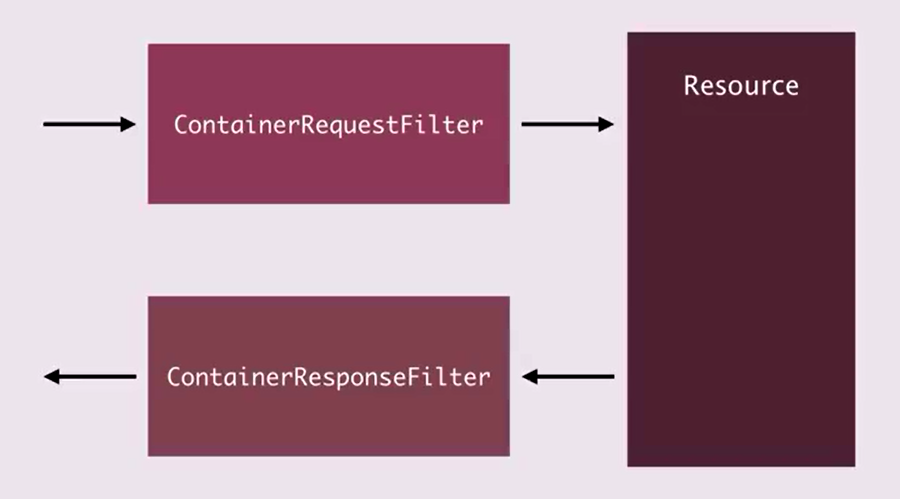
Advanced JAX-RS 20 - Unit Introduction

Advanced JAX-RS 21 - Implementing Filters

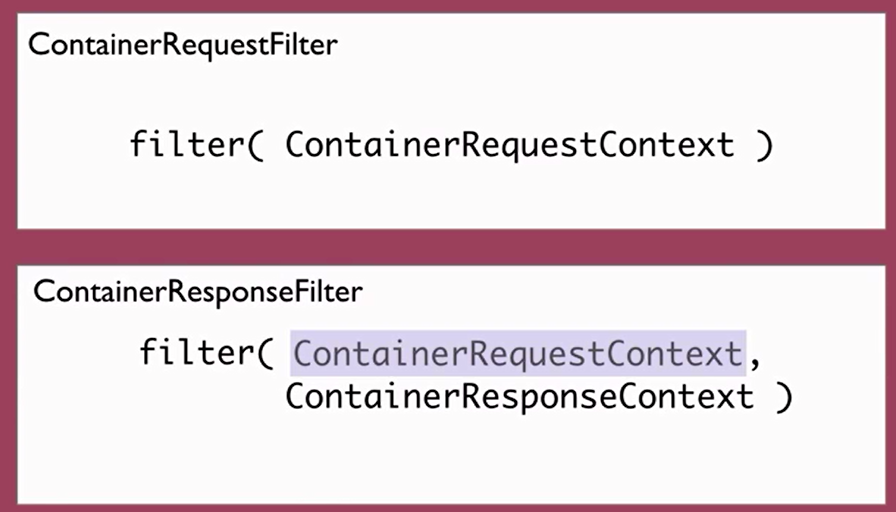
The best way to implement authentication in JAX-RS api is using filters of JAX-RS.

Filters are a way for you to take some of the cross cutting concerns and cross cutting logics out from your individual resource into a common filter class. The idea is, if you have to apply some logic to different api in different resources , rather than make a copy of each of these resources , you take that logic out and create one common class which contains that logic and then you apply it to different requests.

There are 2 types of filters in jax-rs, RequestFilter and responseFilter. Requestfilter is a filter that gets executed before a request is served. ResponseFilter is executed after a response is prepared and is about to be sent to the client.



ContainerRequestContext – Contains details about the request.



PoweredByResponseFilter.java

package org.koushik.javabrains.rest;

import java.io.IOException;

import javax.ws.rs.container.ContainerRequestContext;

import javax.ws.rs.container.ContainerResponseContext;

import javax.ws.rs.container.ContainerResponseFilter;

import javax.ws.rs.ext.Provider;

@Provider

public class PoweredByResponseFilter implements ContainerResponseFilter {

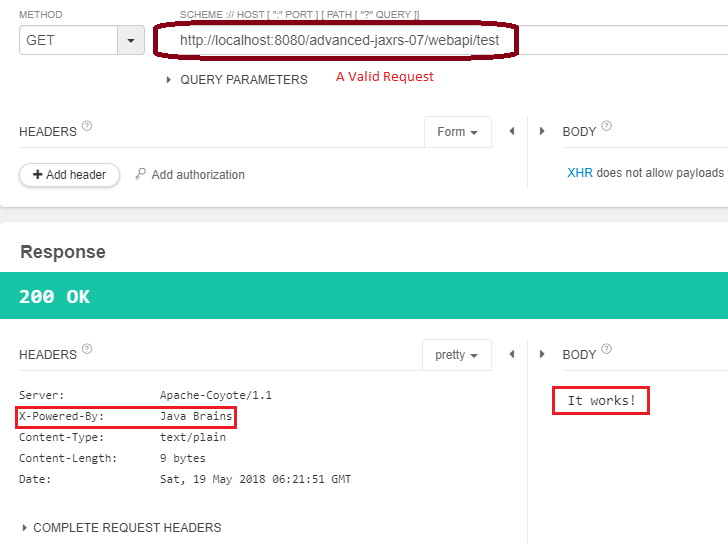
@Override

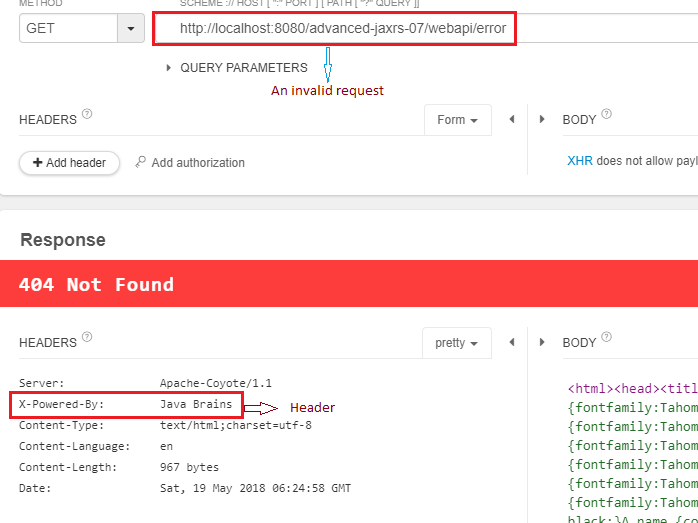
public void filter(ContainerRequestContext requestContext, ContainerResponseContext responseContext) throws IOException {

responseContext.getHeaders().add("X-Powered-By", "Java Brains");

}

}





The filter is getting executed irrespective of whether the respo nse is successful or not.

Another implemetation of filter. To log the header value of the request as well as response.

LoggingFilter.java

package org.koushik.javabrains.rest;

import java.io.IOException;

import javax.ws.rs.container.ContainerRequestContext;

import javax.ws.rs.container.ContainerRequestFilter;

import javax.ws.rs.container.ContainerResponseContext;

import javax.ws.rs.container.ContainerResponseFilter;

import javax.ws.rs.ext.Provider;

@Provider

public class LoggingFilter implements ContainerRequestFilter, ContainerResponseFilter {

@Override

public void filter(ContainerRequestContext requestContext) throws IOException {

System.out.println("Request filter");

System.out.println("Headers: " + requestContext.getHeaders());

}

@Override

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

System.out.println("Response filter");

System.out.println("Headers: " + responseContext.getHeaders());

}

}

Output:

Request filter

Headers: {accept=[image/jpeg, application/x-ms-application, image/gif, application/xaml+xml, image/pjpeg, application/x-ms-xbap, \*/\*], accept-language=[en-US], ua-cpu=[AMD64], accept-encoding=[gzip, deflate], user-agent=[Mozilla/5.0 (Windows NT 6.2; Win64; x64; Trident/7.0; MASPJS; rv:11.0) like Gecko], host=[localhost:8080], connection=[Keep-Alive]}

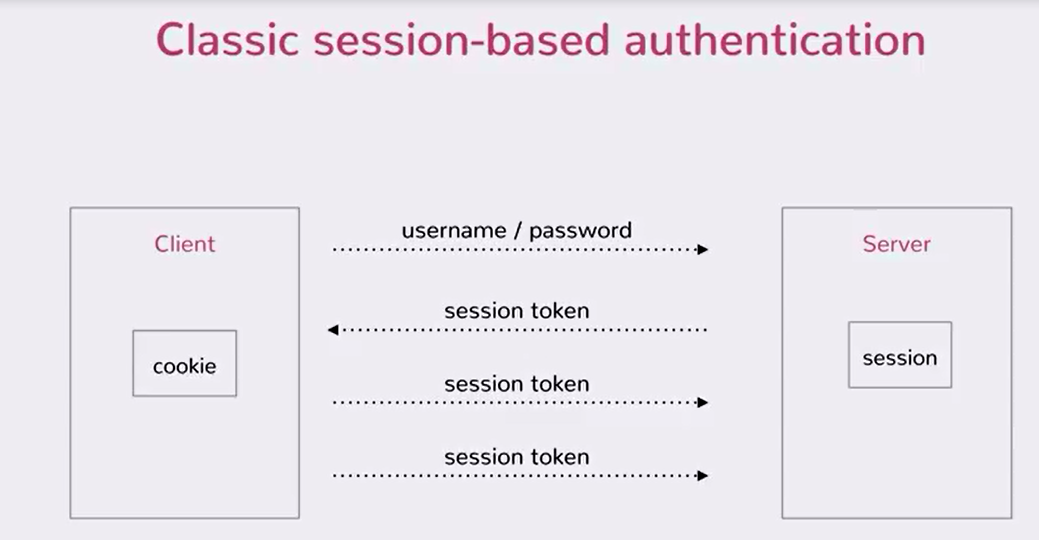
Response filter

Headers: {Content-Type=[text/plain]}

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

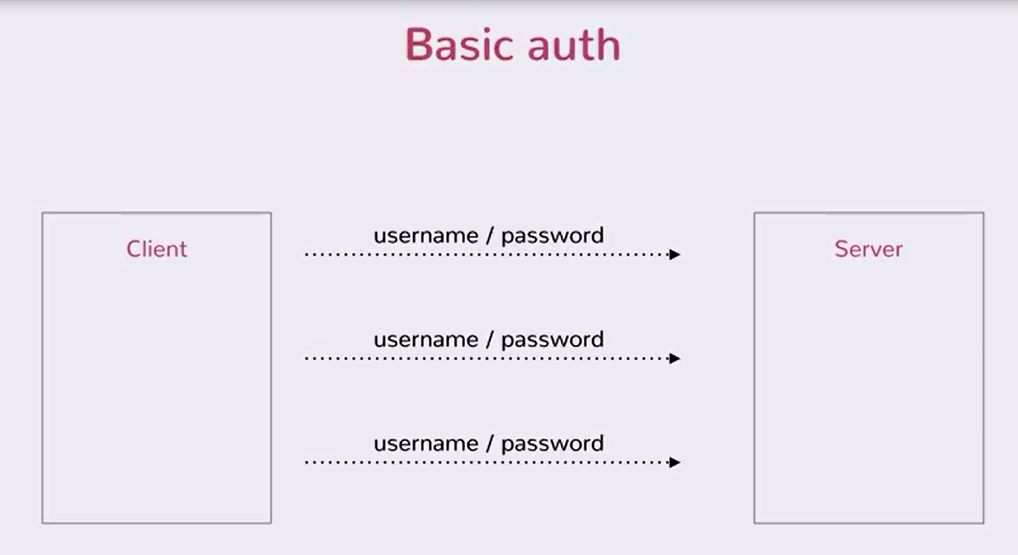
Advanced JAX-RS 22 - REST API Authentication Mechanisms

A typical session based authentication is explained in the below diagram:

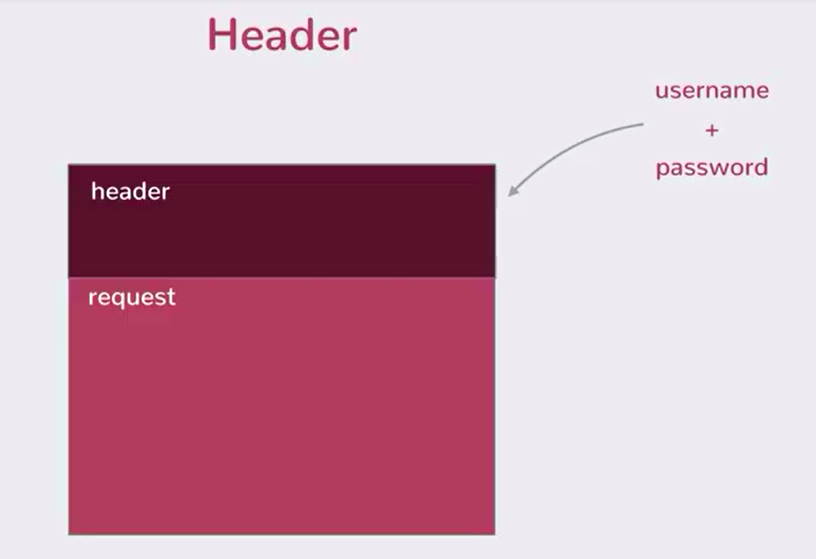


The above is going to be a problem in rest api. Bcz REST APIs are stateless. Stateless means that the application running on the server does not maintain any state.

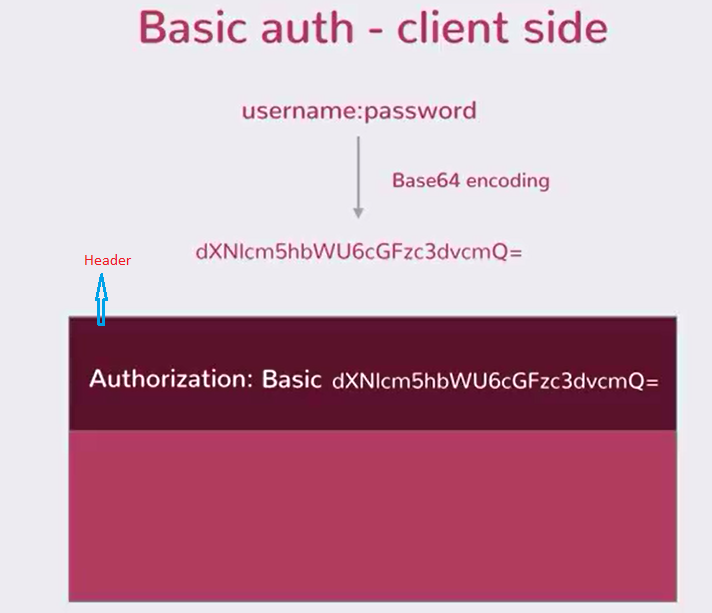
We have many ways to implement stateless authentication. The most basic among them is **Basic Auth (**Basic Access Authentication**)**.



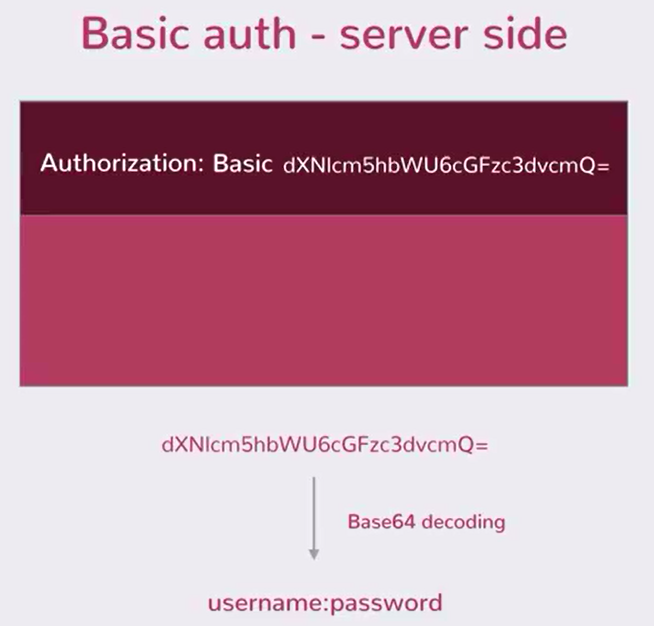
The way username and password is sent in every request is in the header of the request.



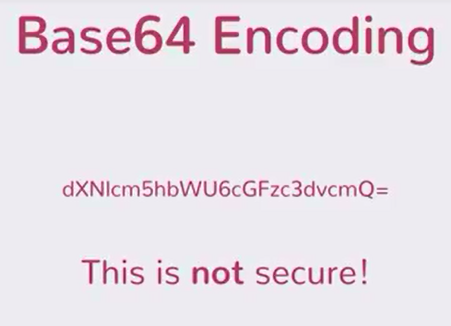
However you do not just add the username and password as it is. There is a special protocol for this. Below image explains how we pass the username and password.



Now, below diagram shows how does it work on the server side?



**NOTE : En**coding is not security. This is just a way to convert one string form to other.

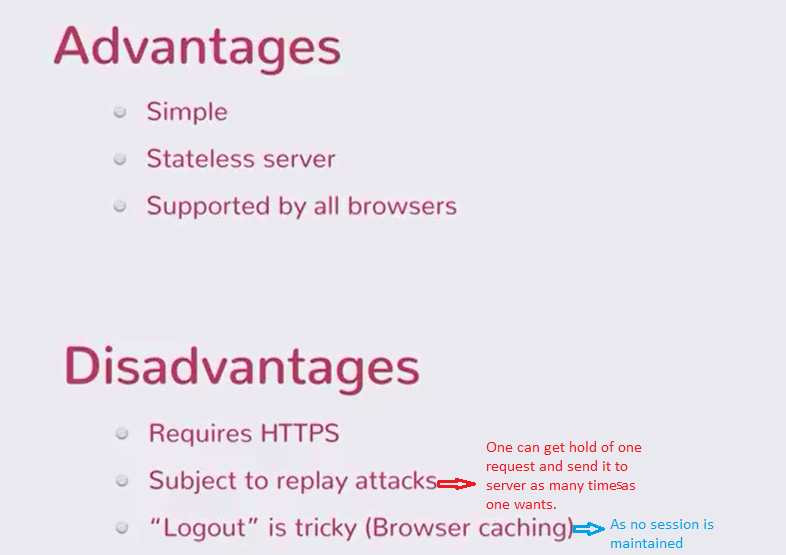


If somebody gets to know the header value, it is easy to get the username and password. PLEASE ensure to send this request over HTTPS, so that the header value is protected.

So if anybody can encode this Base64 encoding, then why are we doing it? Why not just send the username and password?

Ans: The reason we encode it so that we encode all non HTTP characters.







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

Advanced JAX-RS 23 - Sending Basic Auth Requests

Advanced JAX-RS 24 - Implementing REST API Authorization

SecuredResource.java

package org.koushik.javabrains.rest;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.Produces;

import javax.ws.rs.core.MediaType;

@Path("secured")

public class SecuredResource {

@GET

@Path("message")

@Produces(MediaType.TEXT\_PLAIN)

public String securedMethod() {

return "This API is secured";

}

}

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

SecurityFilter.java

package org.koushik.javabrains.rest;

import java.io.IOException;

import java.util.List;

import java.util.StringTokenizer;

import javax.ws.rs.container.ContainerRequestContext;

import javax.ws.rs.container.ContainerRequestFilter;

import javax.ws.rs.core.Response;

import javax.ws.rs.ext.Provider;

import org.glassfish.jersey.internal.util.Base64;

@Provider

public class SecurityFilter implements ContainerRequestFilter {

private static final String AUTHORIZATION\_HEADER\_KEY = "Authorization";

private static final String AUTHORIZATION\_HEADER\_PREFIX = "Basic ";

private static final String SECURED\_URL\_PREFIX = "secured";

@Override

public void filter(ContainerRequestContext requestContext) throws IOException {

if (requestContext.getUriInfo().getPath().contains(SECURED\_URL\_PREFIX)) { // this is here just to secure the request that contains ‘/secured’. If we remove this if block then all the request will go through this authorization.

List<String> authHeader = requestContext.getHeaders().get(AUTHORIZATION\_HEADER\_KEY);

if (authHeader != null && authHeader.size() > 0) {

String authToken = authHeader.get(0);

authToken = authToken.replaceFirst(AUTHORIZATION\_HEADER\_PREFIX, "");

String decodedString = Base64.decodeAsString(authToken);

StringTokenizer tokenizer = new StringTokenizer(decodedString, ":");

String username = tokenizer.nextToken();

String password = tokenizer.nextToken();

if ("user".equals(username) && "password".equals(password)) {

return;

}

}

Response unauthorizedStatus = Response

.status(Response.Status.UNAUTHORIZED)

.entity("User cannot access the resource.")

.build();

requestContext.abortWith(unauthorizedStatus);

}

}

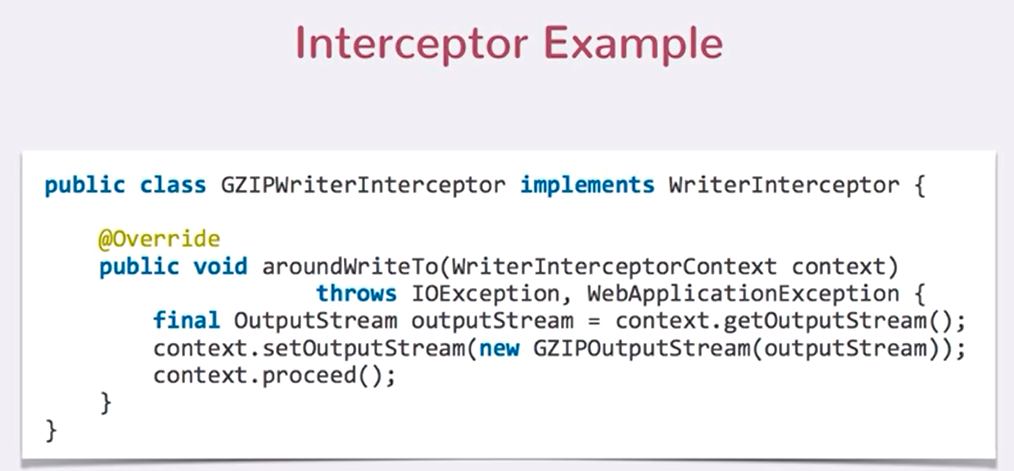
}

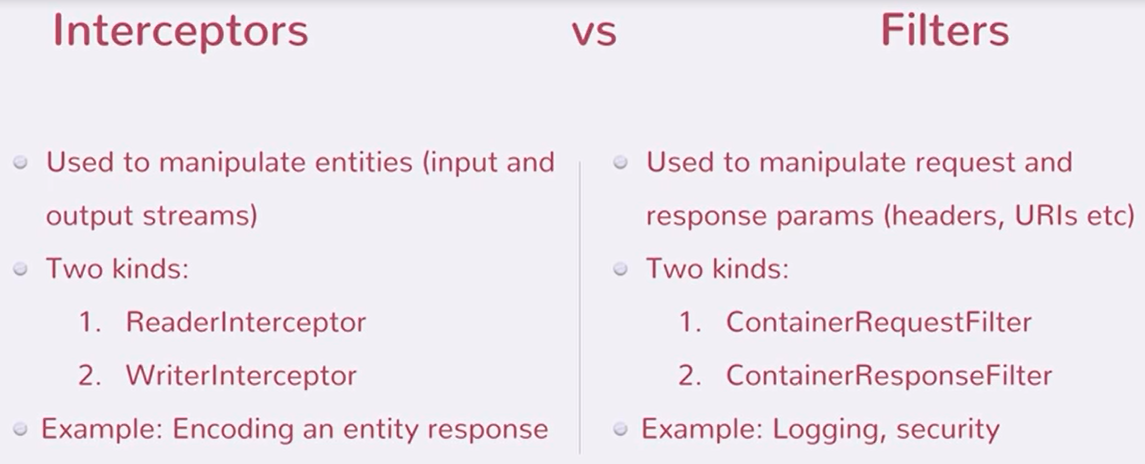
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Advanced JAX-RS 25 - Filters and Interceptors

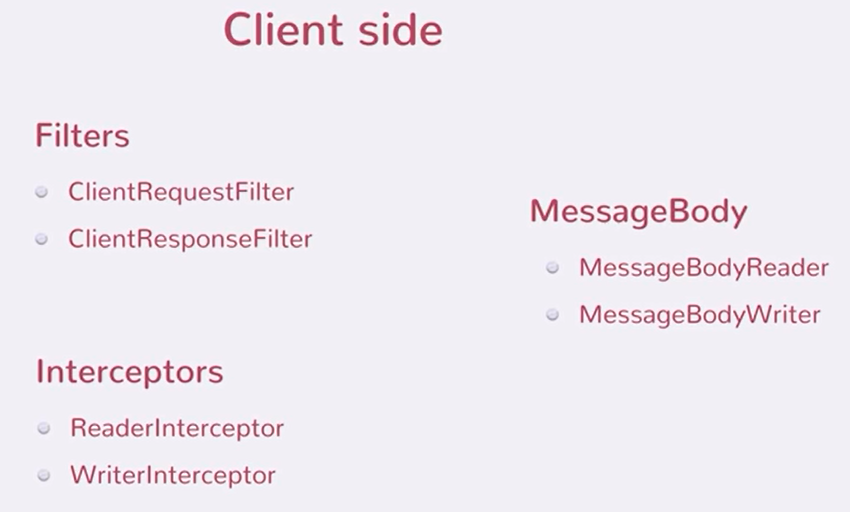
Interceptors:

1. Model similar to filters
2. Intercept request and response
3. Filter manipulates metadata information (for ex header information) while interceptor manipulates the actual body of the request and response. for ex: entity (input and output stream)
4. Two kinds:
5. ReaderInterceptor : In case of filter, request and response info is read while in case of Interceptor, it is request body and rsponse body
6. WriterInterceptor





Filters and Interceptors work on a client too. If you are writing a Jax-rs client using the JAX-RS api.



Sequence of various filters and interceptors in a typical client server communication:

1. ClientRequestFilter : When the client makes request, the first thing that gets called is ClientRequestFilter. You can change client request parameter before it goes to the server.
2. WriterInterceptor : Lets say you are making a post request. You have a java object that needs to be converted into a http payload. Before the conversion happens from instance to http payload, interceptor gets a chance to modify the payload.
3. MessageBodyWriter : Now you have your java instance written to http request using MesssageBodyWriter in the client side. With this the request now leaves the client and goes to the server.
4. ContainerRequestFilter : First thing to get executed on the server side. This modifies the request params.
5. ReaderInterceptor : Intercepts the read action and lets you modify the payload.
6. MessageBodyReader : Converts from payload to java instance. Then java instance is sent to your resource method on your server.

Now the resource method does what it needs to do and you have response to send back.

1. ContainerResponseFilter : Allows you to cahnge the response parameter.
2. WriterInterceptor : Intercepts the write.
3. MessageBodyWriter : Converts your instance on the server side to response payload.
4. ClientResponseFilter : ClientResponseFilter acts before the client gets hold of the response. It lets you modify the response parameters.
5. ReaderInterceptor : Intercepts the read.
6. MessageBodyWriter : Converts from payload to the java instance on the client.

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

Advanced JAX-RS 26 - JAX RS and EJBs

How you can wire in EJB resource into javaee JAX-RS application. 🡪 Using CDI (component Dependency Injection)

