Microservices Oriented Software Architectures

TP4

In this practical work, I suggest you see how to manage exceptions on a REST web service, with the Jersey library.

Goals

Master the notion of HTTP errors for server-side REST web services.

Functioning

In REST, there is no automatic mechanism to automatically send exceptions to the client, contrary to SOAP/WSDL/XML web services.

We must therefore return the exceptions manually and there is a convention for this to implement yourself, using Jersey, based on two categories of exceptions:

1. The error comes from something incorrect sent by the customer
2. The error comes from a server side error

In either case, the server may or may not return additional data in the response body, in JSON, similar to how the client sends data to the server when making a request. It is strongly recommended to return a response to give details on the error encountered, in order to allow the customer to know where it can come from and his developer to adapt the behavior of his application.

The first thing to do to return an HTTP error is to determine the exception category, to return the correct HTTP error in the response:

- 4xx Client Error – These HTTP error codes are used to tell the client that an exception has occurred due to the items it sent. The client must not retransmit this request, which will systematically produce the same result, it must send a new, corrected request.
- 5xx Server Error – these HTTP error codes are used to tell the client that an exception has occurred while executing a seemingly correct client request (it is not possible to fully verify this since a error has occurred). The client can try to execute this request again later, when the server has corrected its problem (generally a production problem, a server or a database that is more accessible internally, for example).

There are also other HTTP return codes, here are the existing categories:

- 1xx: preliminary information related to the request
- 2xx: request success
- 3xx: request redirect

When an HTTP error must be returned (4xx and 5xx), it is recommended to provide details on the cause of the error, with a response containing in particular:

- **status** : redundantly contains the HTTP error code (to facilitate reading by the client-side developer who is testing, this avoids having to re-read the HTTP header to get the status)
- **code** : contains the internal error code related to the server-side API. Should allow the server-side developer to determine the error that occurred on the server-side
- **message** : a short description of the error that could possibly be displayed to the end user
- **reason** : a more detailed description intended for developers (client and/or server) to have more information on the cause of the error

Work to do

To start, you can create the ErrorMessage class containing the attributes we have just seen (status, code, message and reason).

Next, you can create a new Exception class that aims to be the parent class of all exceptions that will likely be thrown back to the client. For the example, I call this class RESTException. This class must take as an attribute an `errorMessage` from the ErrorMessage class that you created earlier. You also need to create a constructor that takes an instance of ErrorMessage as a parameter.

You can now modify your existing exceptions which are normally sent to the client (but for which you put a catch in your Resources as requested in the original lab statement) so that it no longer directly inherits from the Exception class, but of the RESTException class. You can also remove all try/catch so that the exception is thrown at the method level directly. Below is an illustration of what you have to do:

```
public class MyException extends Exception {
```

replaced by :

```
public class MyException extends RESTException {
```

and :

```
   @GET
   @Path ( "/hello" )
   @Produces (MediaType. APPLICATION_JSON )
   public Hello hello() {
      try {
         return MyClass.methodThatCanThrowMyException();
      } catch (MyException me) {
         return null ;
      }
   }
```

replaced by :

```
   @GET
   @Path ( "/hello" )
   @Produces (MediaType. APPLICATION_JSON )
   public Hello hello() throws MyException {
      return MyClass.methodThatCanTriggerMyException();
   }
```

Now we must create an ExceptionMapper class following the principle of this example class:

```
package alom.server.exception;

import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.ExceptionMapper;
import javax.ws.rs.ext.Provider;

@Provider
public class RESTExceptionMapper implements ExceptionMapper{

   public Response toResponse(RESTException ex) {
      ErrorMessage em = ex.getErrorMessage();
      return Response.status(em.getStatus())
            .entity(em)
            .type(MediaType. APPLICATION_JSON ).
           build();
   }

}
```

This class allows Jersey to map your Exception into the appropriate JSON message to call our web service. Here, the RESTExceptionMapper class allows us to map RESTExceptions into JSON responses containing what the errorMessage of our exception contains (status, code, message and reason).

Now, all you have to do is adapt your exception constructors so that they take an instance of ErrorMessage as a parameter (you can pass this parameter as a parameter of the `super()` method that you call at the very beginning of your constructor), then you can adapt the instantiation of your exceptions so that it takes this form: `throw new XXXException(new ErrorMessage(...))` . Don't forget that the HTTP error status that you will fill in must correspond to the convention of the HTTP protocol (4xx and 5xx) that we saw earlier in this statement.

For rendering, your project must only return HTTP error codes consistent with your application exceptions, without ever returning an HTTP 500 error for one of your exceptions (the HTTP 500 error corresponds to the default behavior when an Exception is thrown and not mapped, which displays a stack trace to the user; oo that's ugly).

Modified on: Tuesday 4 October 2022, 12:22

Rendering :
An archive containing:

All of the Jersey client and server source codes with the management of HTTP errors corresponding to exceptions.
A Readme explaining:
the difficulties/blockages encountered (whether you solved them or not)
any other information you deem useful in connection with this lab