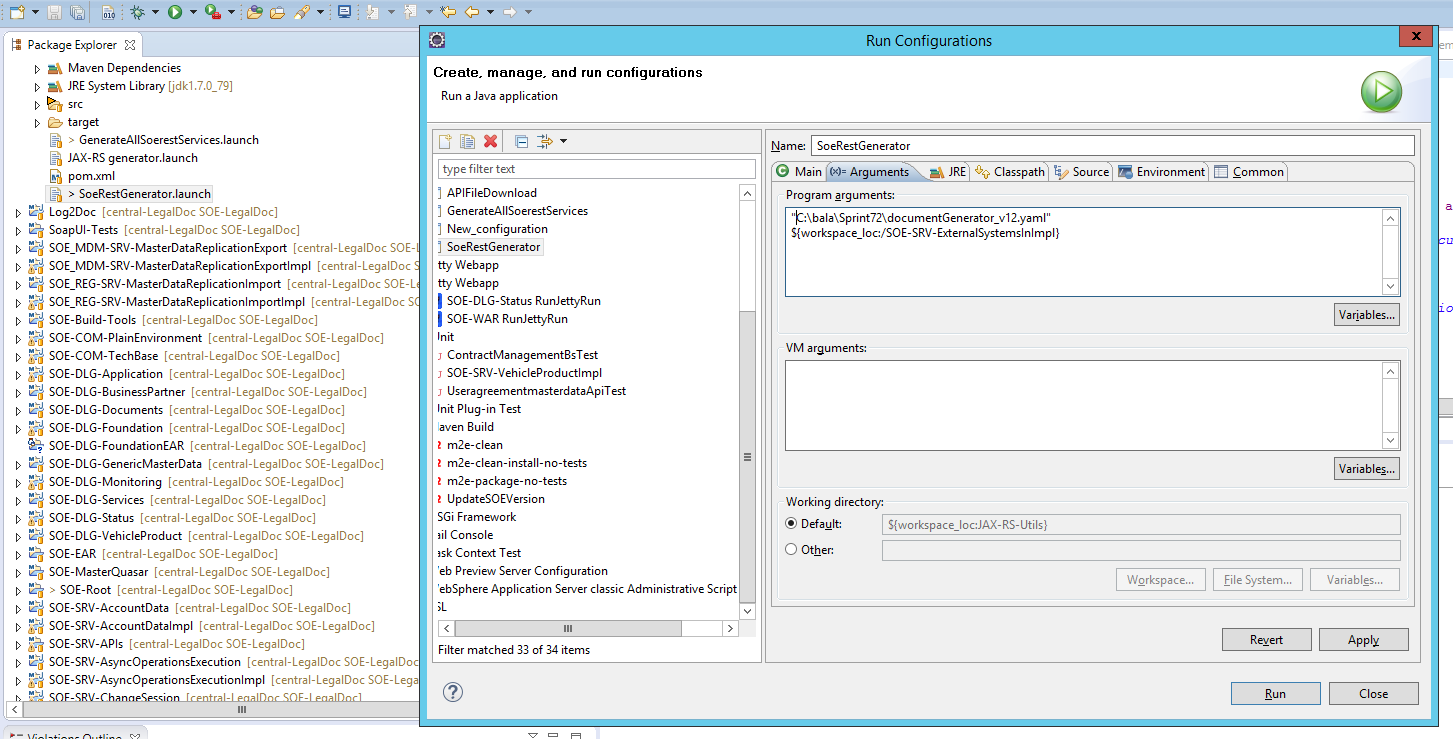
**Provide a REST services (SI)**

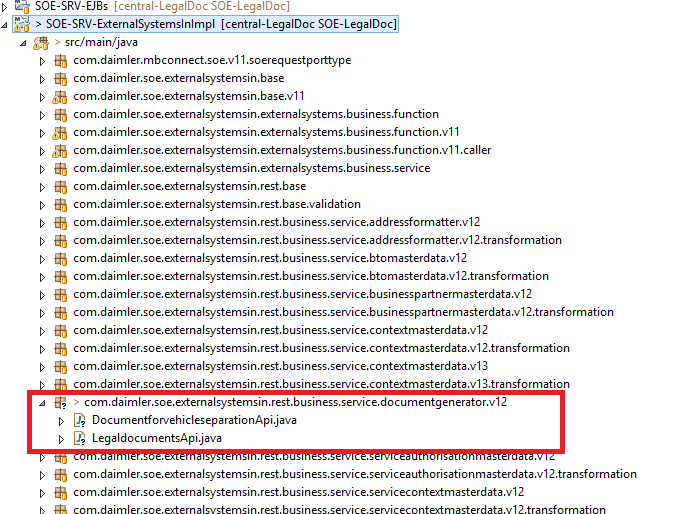
The interface descriptions a REST of services within of Mercedes is defined connect ME by YAML files. These serve service implementations as basis for the production the REST.  
The following steps are to do around a REST service in the SOE to make available:

The YAML definitions should have to be found 100 MBconnect in the Confluence on the page SOE SST (offered) or in the SVN under the path \ 30\_REA \ 01 SOE \ 20 interfaces \ REST.

### Generieren der Implementierung

1. Import project JAX-RS-Utils into SOE Workspace
2. Under Run configurations > Java Application > SoeRestGenerator > Arguments give two parameters:
   1. source YAML-File
   2. destination path for generated files (default: "${SOE\_WORKSPACE\_PATH}\SOE-SRV-ExternalSystemsInImpl")
   3. Select Java1.7 in the JRE tab.
3. Click Run





Below step is not clear, because in the above step it was mentioned to run using SOEgenerator and here we are seeing the JAX-RS generator.

While Selecting the JAX-RS generator from the Run configurations menu we are not seeing the configuration options as mentioned in the below step.

The steps in the below box are not clear.

Below Steps are same as above but using the JAX-RS generator instead of SOERestGenerator

When starting the generator by Run configurations ...> JAX-RS generator man has more configuration options and can e.g.

• generation of the already known types as Skip enumerations from SOE SRV Foundation

• Avoiding overwriting the Servicestubs

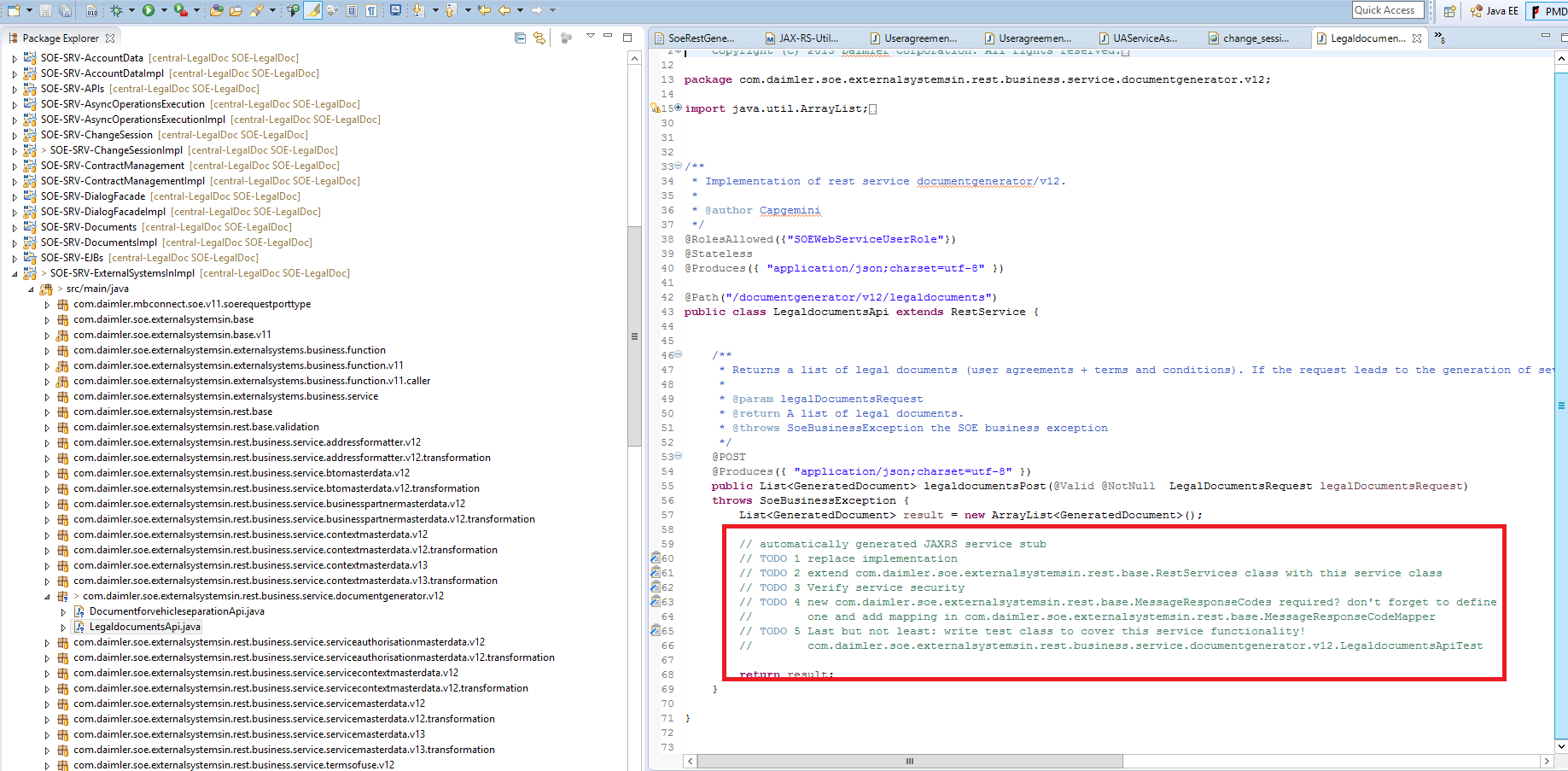
Overwriting the Servicestubs can be configured with the JVM Property. If Property overwrite service is available the Servicestub is overwritten, otherwise not. So with changes only in jerk gift guy who Servicestub need not be adjusted again.

Generation of the already known types, can be avoided by specifying the import mappings to existing classes as --import-mappings Error = java.util.Error.

### Einbinden der generierten JAVA-Dateien

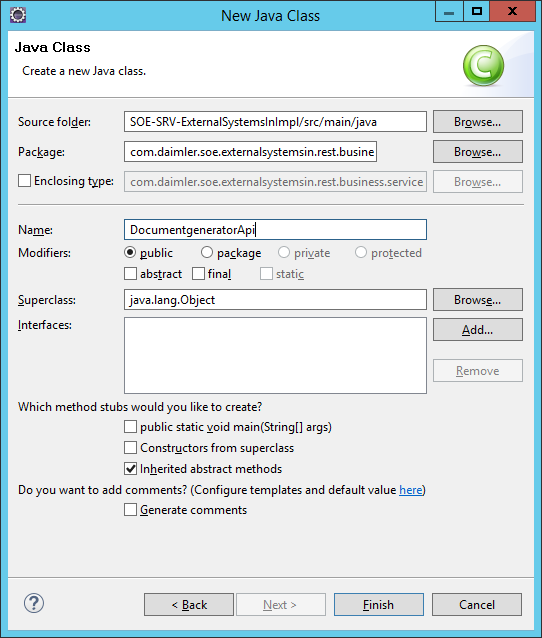
1. Check out the TODOs in generated service stub class.

Open the API class and follow the TODO list



1. When you want to use **two or more** services under **the same basic path** (for example basic path /api/vehiclevalidation/v12 and two services /checkvehicle and /availableuseragreementsforvehicle) then you have to put all the services into one file, because a path can only be registered for just one class! For the above example it would look like this:

Do not create the API class, please use the API classes whatever the SOERestgenerator has created and create the \*Impl,\*Convertor,\*ApiTest, \*ConvertorTest,\*TestSet classes for these API classes

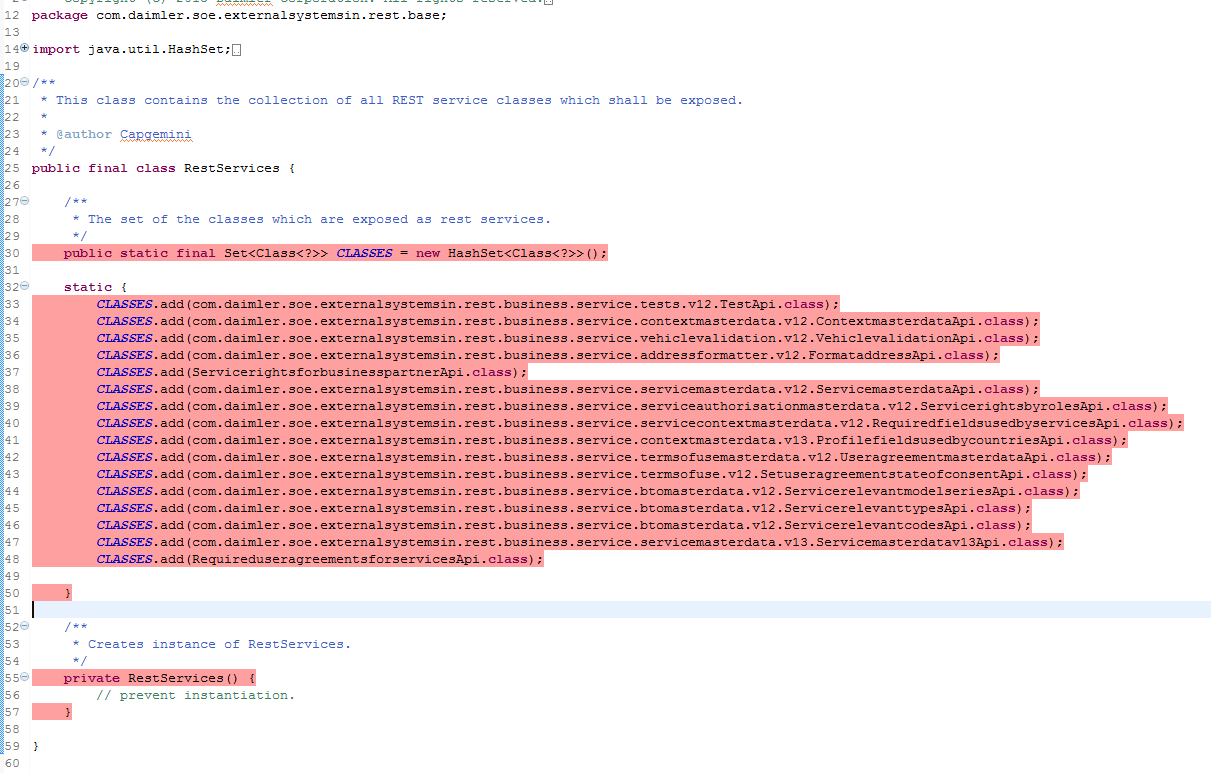


|  |
| --- |
| @Path("/vehiclevalidation/v12")  public class VehiclevalidationApi extends RestService {        @Path("/checkvehicle")      public CheckVehicleResponse checkvehiclePost()      {          //logic      }     @Path("/availableuseragreementsforvehicle")      public AvailableUserAgreementsForVehicleResponse availableuseragreementsforvehiclePost()      {          //logic      }  } |

1. **Wichtig**: Öffne die Datei com.daimler.soe.externalsystemsin.rest.base.RestServices und füge im static Block den Klassennamen des neuen Services zur Liste der Klassennamen hinzu.

Below is the translated text of the above line

Open the com.daimler.soe.externalsystemsin.rest.base.RestServices file and add the newly created Api Class to the SET of classes under static block.

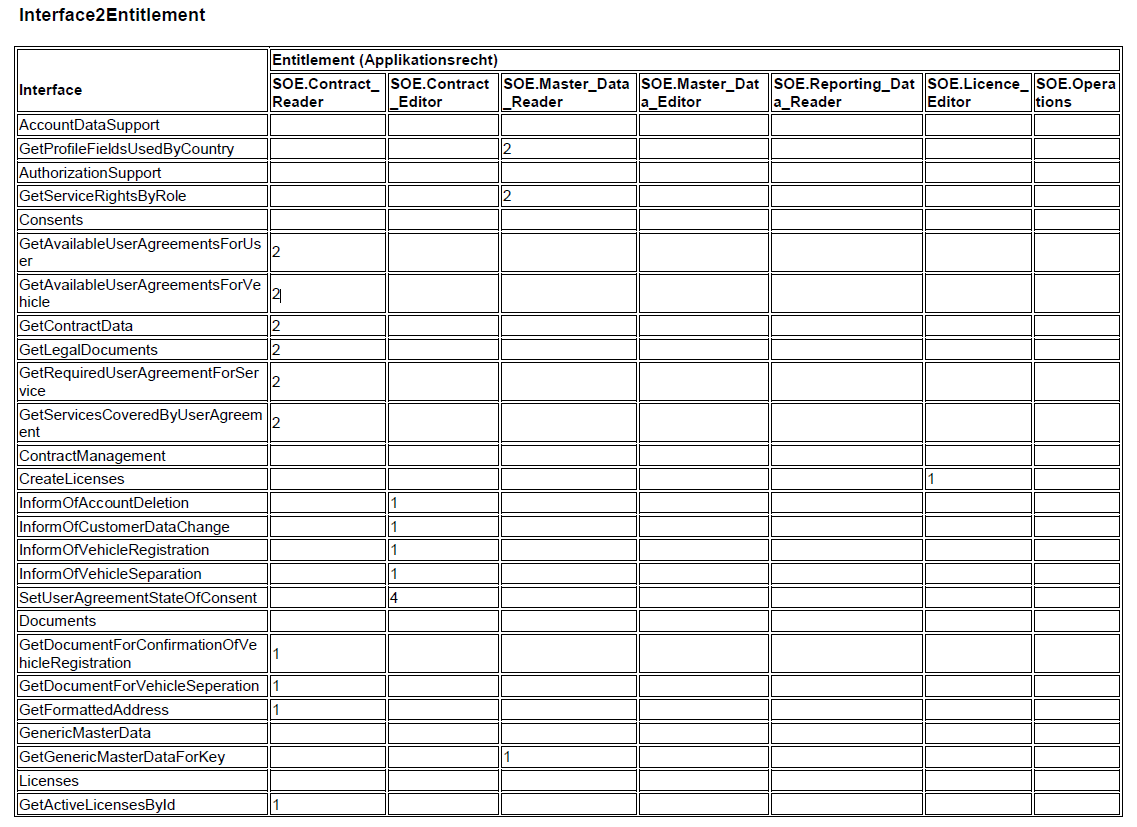


### Service Security

* Verify generated roles i.e. @RolesAllowed({"SOEWebServiceUserRole"})
* check the table in <https://s415vmmt060.detss.corpintra.net/confluence/display/MBC/SOE+Service+Security> and add allowed roles for this REST Serviced, which is defined in the table, into @RolesAllowed list

If you refer the below screenshot on roles it is not clear for us how to consider the roles for the services based on the number.

During the translation cross marks were replaced with numbers. Please read the numbers as cross mark

****

Need to Analyse further to have more clarity on the below steps.

It is better to follow related classes of VehicleValidationApi for information on the below steps.

**Implementation of the REST of the services**

The implementation of the rest services contains:

* the conversion of the parameters and/or JSON objects in DTOs and following delivery on \*IfBs classes.

Should we need to follow the above step as it is or can we use the MasterDataBs as there is no respective \*IfBs class for ServiceCategoriesApi?

Yes we can use MasterDataBs or IfBs whichever existing in the Caller class .

* the conversion of the answer DTOs into JSON objects.

use the corresponding DTOs( RDTO, CDTO, DTO) and set them into JSON response objects.

* Catch from *SoeBusinessExceptions* and Wrappen in *SoeExternalInterfaceFunctionalExceptions*. The errors are then prepared by technical Dealers/Mappers, so that a valid answer is returned.

Catch the SOEBusinessExceptions and throw the *SoeExternalInterfaceFunctionalExceptions*

Also need to analyse understand below steps

If new error codes are defined, must be defined this in the class com.daimler.soe.externalsystemsin.rest.base.MessageResponseCodes and the Mapping in more com.daimler.soe.externalsystemsin.rest.base.MessageResponseCodeMapper fixed

New ResponseCodes needs to be added to the MessageResponseCodes class and respective MessageIds needs to be added to the MessageResponseCodeMapper class

**Provide a test class**

Provide a test class com.daimler.soe.externalsystemsin.rest.business.service.<RESOURCENAME>.v<VERSION>.<SUBRESOURCE>ApiTest the test class tests the call the REST of the services, is one guarantee that

* the parameters to the IfBs method to be handed over correctly.
* SoeBusinesExceptions correctly in *SoeExternalInterfaceFunctionalExceptions* to be wrapped.

We have searched for VehicleValidationApiTest class and it is not available in the latest workspace which we have downloaded from Git.

And we found ApiTest class name fors VehicleValidation with the names CheckVehicleApiTest , AvailableUserAgreementForVehicleApiTest

**Is it mandatory to create two Api Test classes for a single Api which has two methods under one path?**

**For each API , it is mandatory to create the ApiTest, TestSet and ConvertorTest Classes**

**Provide a SOAP UI test**

|  |  |  |
| --- | --- | --- |
| **Setup** | **Description** | **Example** |
| **Method** | The HTTP method, which should be used to implement an operation.  According to the Annotation in the implementation | POST OFFICE, GET |
| **End POINT** | The end point results from the host address and the base Path for REST of services. | <http://soe-dev-maint.es.corpintra.net:9080/ws/soe/api> |
| **Resource** | The path under that the resource is attainable. | /contextmasterdata/v12/profilefieldsusedbycountries |
| **Parameter** | QueryParameter to be handed over are.  SoapUI currently has a known issue ("defect"?) in that it will not allow you to supply same named parameter multiple times, such as ids=111&ids=222 in your example.  Solution:  In your endpoint, where the methods is defined, select the parameter and turn on "Disable Encoding". Then in your call, for the parameter ids, you would provide the *literal* value: 111&ids=222. In order to make this dynamic, you would probably have to resort to Groovy scripting.  (see http://stackoverflow.com/questions/24167874/how-to-populate-a-query-parameter-list-in-soapui/24168328#24168328) | ? countryCode=DE&locale=de\_DE |
| **Request** | At POST OFFICE it is possible to send a Parameterobject. The Request in JSON notation.  As Media type must be selected for it *application/json*. |  |
| **Auth** | Attitude like the SOAP client authentifizieren themselves is. | |  |  | | --- | --- | | **Characteristic** | **Value** | | Authorization | Basic | | Username | SOE\_tec\_00 | | Password | \*\*\*\*\*\*\*\* | | Pre emptive Auth | Authenticate pre-emptively | |
| **Header** | The headers parameters are given in the HTTP header and likewise used with the processing of a question. | |  |  | | --- | --- | | **Header parameter name** | **Value** | | X-FinOrVin | FIN of the vehicle if available | | X-ApplicationName | SOAPUI | | X-TrackingId | SOAPUI $ {=java.util.UUID.randomUUID ()} | | Accept | application/json | |

**Assertions**

* That the HTTP status code should be always checked *ADD Assertion - > Compliance, status and standards - > Valid HTTP status code*
* With cases of an error the X-error code HTTP header of the answer must be evaluated. This can take place via a Script Assertion.  
  *ADD Assertion - > Script - > Script Assertion*

def expectedErrorCode = ‘99’

assert messageExchange.responseHeaders [‘X-error code’] .get (0) .equals (expectedErrorCode)

* News can be evaluated with JSON Path. Under http://goessner.net/articles/JsonPath/ it gives a few references to the notation.  
  *ADD Assertion - > Property content - > JSON Path match*

**Do we need to follow the below step of using Swagger?**

 Do not follow the swagger steps as we are creating the APi classes using SoeRestGenerator.

**Swagger version**

 Before creating a REST service, please check the Swagger Dependancy in pom.xml under SOR SRV ExternalSystemsInImpl whether the version is newer than 1.5.2-M2.

Note: Version 1.5.2-M2 is newer than 1.5.3-M1.

Swagger before version 1.5.2-M2 (e.g. 1.5.3-M1) does not support responseContainer within @ApiResponse.

see http://grokbase.com/t/gg/swagger-swaggersocket/1563838a6d/swagger-2-0-annotations-for-apiresponse-responsecontainer

**Open points**

Validation:

* The generated entities do not contain Bean validation Annotation, so that no validation can be made.
* To accomplish it still no suitable tools found around a validation on news level - > Request against YAML definition.

LOGGING of the messages

* Still no way could be found (approximately over the Handler) to capture the incoming and outgoing messages, so that these can be out-logged.

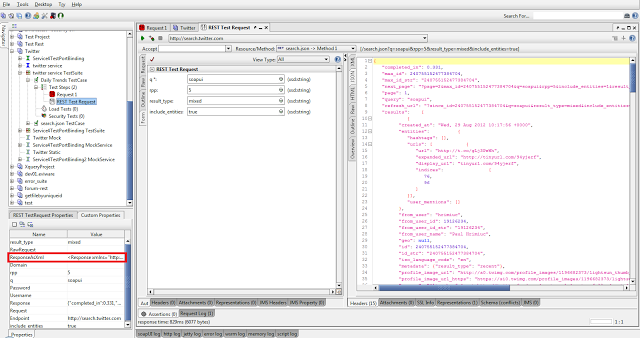
**Asserting JSON in soapUI (Copied from http://www.robert-nemet.com/2012/08/json-asserting-in-soapui.html)**

   JSON stands for JavaScript Object Notation and it is lightweight data presentation format. It should be easy understandable for humans. Basically if you do not want and do not need to go in details it is same thing as XML just different packaging. As you already know or guessed when soapUI was created SOAP services were in team focus.

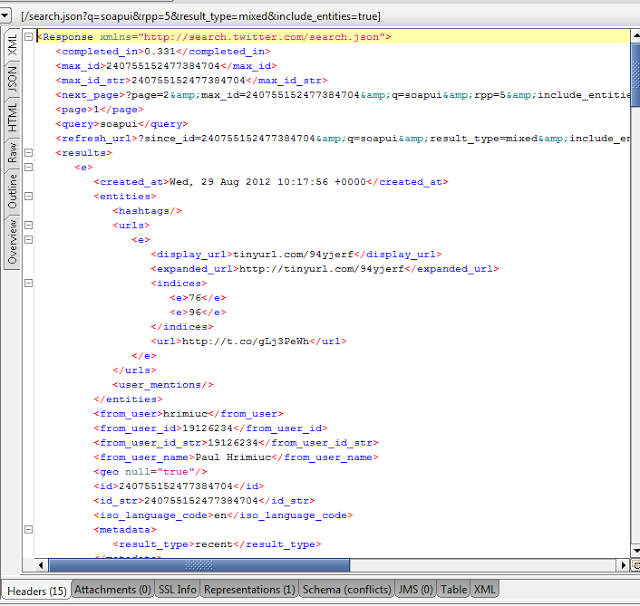
    So, in soapUI you have good support for testing SOAP services and manipulating XML. What about REST and JSON?

**JSON is soapUI**

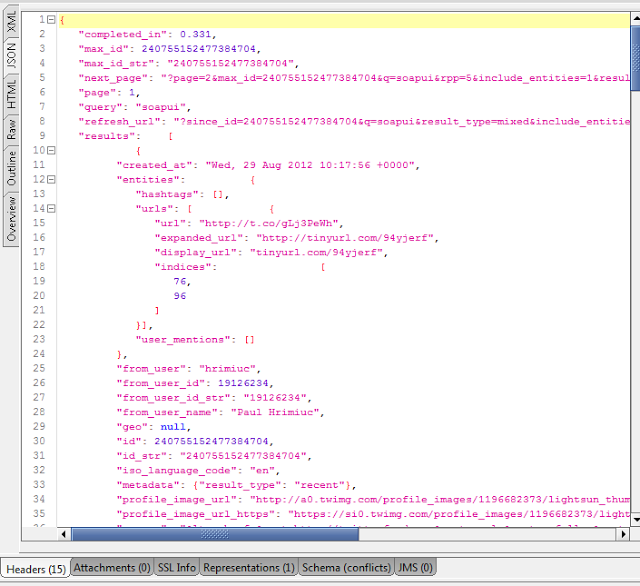
   Good news is that there is no much difference when working with XML and JSON in soapUI. soapUI have number of features that uses XPath to manipulate XML. Of course, you can not use XPath  expressions on JSON. Or maybe you can? In soapUI you can thanks to the fact that soapUI converts JSON to XML. That way what ever you did to assert, validate, check on etc..on XML response same thing you can do with JSON response. Only difference is that instead using  **Response** property you have to use  **ResponseAsXml** property.  **ResponseAsXml**is soapUI's attempt to convert JSON to XML and all tools that you already used for XML use on JSON.

[](http://4.bp.blogspot.com/-4JzLfiJWe-U/UD4BAwnNHQI/AAAAAAAAA_s/ZsmvonyCTUs/s1600/Screenshot-2012-08-29_13.44.29.png)

And this is how converted XML looks like:

[](http://4.bp.blogspot.com/-u2YOMQiywOY/UD4BtfmDIzI/AAAAAAAAA_8/coAWz0_xhs8/s1600/Screenshot-2012-08-29_13.48.54.png)

and original JSON like this:

[](http://4.bp.blogspot.com/-dVAQm0TzKPc/UD4BhWNuTOI/AAAAAAAAA_0/Ysw11rPY5mw/s1600/Screenshot-2012-08-29_13.48.07.png)

**Nice and peachy but...**

There is always one but. Problem with converting JSON to XML is that XML forces some rules that JSON do not have:

* Names can contain letters, numbers, and other characters
* Names cannot start with a number or punctuation character
* Names cannot start with the letters xml (or XML, or Xml, etc)
* Names cannot contain spaces

Here you have an issue in soapUI, since soapUI can not convert JSON to XML if any of those rules will be broken. For example look at Twitters daily trends JSON response:

{  
   "trends":    {  
      "2012-08-29 08:20":       [   
                  {  
            "events": null,  
            "name": "Bizim İçin Saldır Fenerbahçe",  
            "query": "\"Bizim İçin Saldır Fenerbahçe\"",  
            "promoted\_content": null  
         },  
                  {  
            "events": null,  
            "name": "#RemajaIndonesiaSeptemberWish",  
            "query": "#RemajaIndonesiaSeptemberWish",  
            "promoted\_content": null  
         },  
                  {  
            "events": null,  
            "name": "Most Share-Worthy Video",  
            "query": "\"Most Share-Worthy Video\"",  
            "promoted\_content": null

As you can notice "2012-08-29 08:20" should be node value if converted to XML but that breaks XML naming rules. So you are stuck here since there is no XML and you need to do asserting.

**Asserting JSON**

   Problem is soapUI cannot convert JSON to XML because it will break XML naming rules. You cannot use XPath and XQuery assertions because of that. You can use  **Contains**and  **Not Contains** assertions but they have very limited usage. What to do?

     You should use your favourite assertion  **Script Assertion** with Groovy flavour. It is easy and fun, look:

import groovy.json.JsonSlurper

def response = messageExchange.response.responseContent  
def slurper = new JsonSlurper()  
def json = slurper.parseText response

Script above get response content as String and pass it to  [JsonSlurper](http://groovy.codehaus.org/gapi/groovy/json/JsonSlurper.html) which parse this string and produce one big [HashMap](http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html). Now working with  [HashMap is very easy in Groovy](http://groovy.codehaus.org/JN1035-Maps).

     In this example (  [Twitter Daily Trends](https://dev.twitter.com/docs/api/1/get/trends/daily) ) JSON response have trends for number of dates, so if I want check if it contains trends for some particular date and time:

assert json.trends.'2012-08-29 08:20' != null

Notice that I had to put date under quotes since date contains white space, but  **trends** are good. And now I want to be sure that I have eight trends for that date and time:

assert json.trends.'2012-08-29 08:20'.size == 8

 and 7th element  **name** is Toby:

assert json.trends.'2012-08-29 08:20'[7].name == "Toby"

and so on...Whole script can look like this:

import groovy.json.JsonSlurper

def response = messageExchange.response.responseContent  
def slurper = new JsonSlurper()  
def json = slurper.parseText response

assert json.trends.'2012-08-29 08:20' != null  
assert json.trends.'2012-08-29 08:20'.size == 8  
assert json.trends.'2012-08-29 08:20'[7].name == "Toby"

Notice that is you have several assertions one by one in  **Script Assertion** like on example above first assertion that fails will stop executing script, which means assertions under it ( in script ) will not be executed.

That is it..have fun...

Assertions of Testcases in SOAPUI are not required

**Note:-**

In the APi or Implementation classes when there is a parameter which is of List type it needs to validated like below to avoid the empty List.

Eg:-

service(@NotNull @size(min=1) List<String. ids)

SoeArguementChecker.checkIsNotEmpty(ids);