Article
Lorenzo Scalese · Déc 7, 2022 9m de lecture

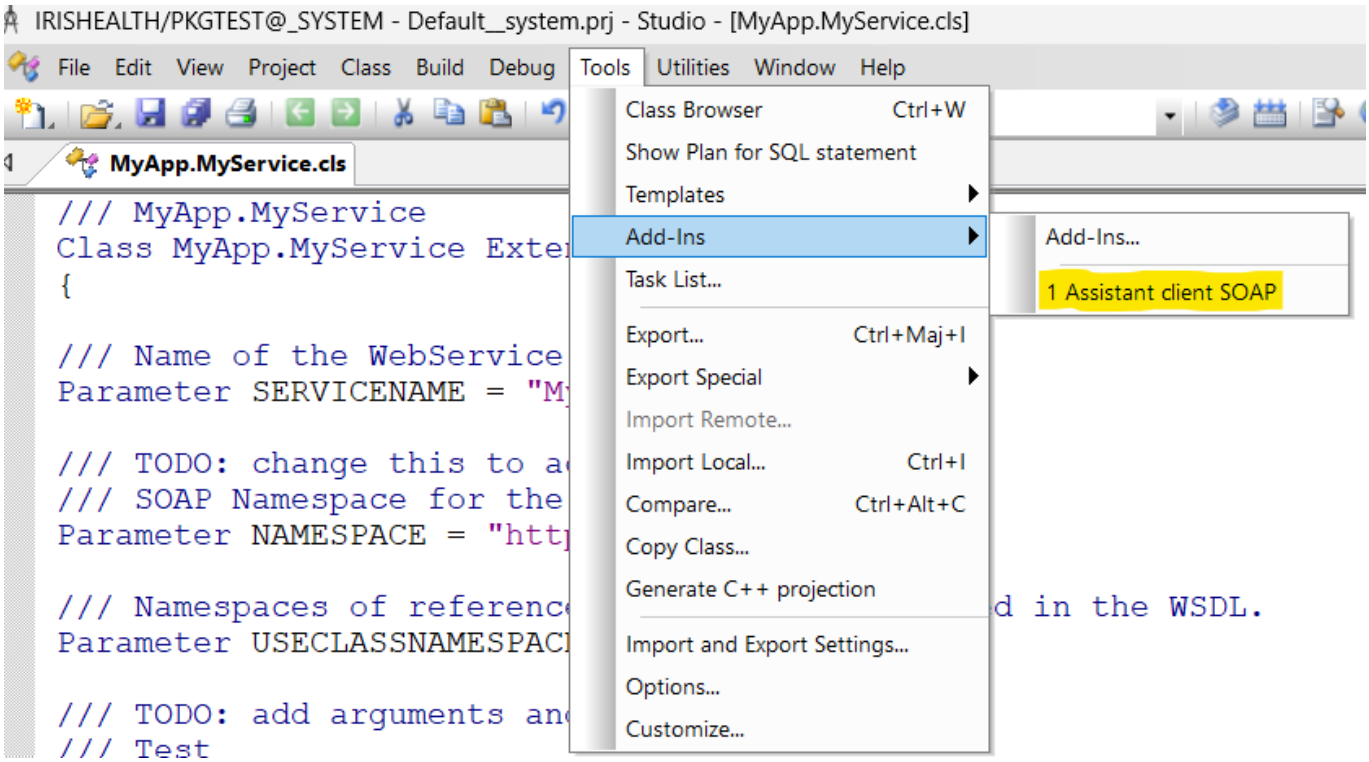
Génération de client SOAP et REST

Salut la communauté,

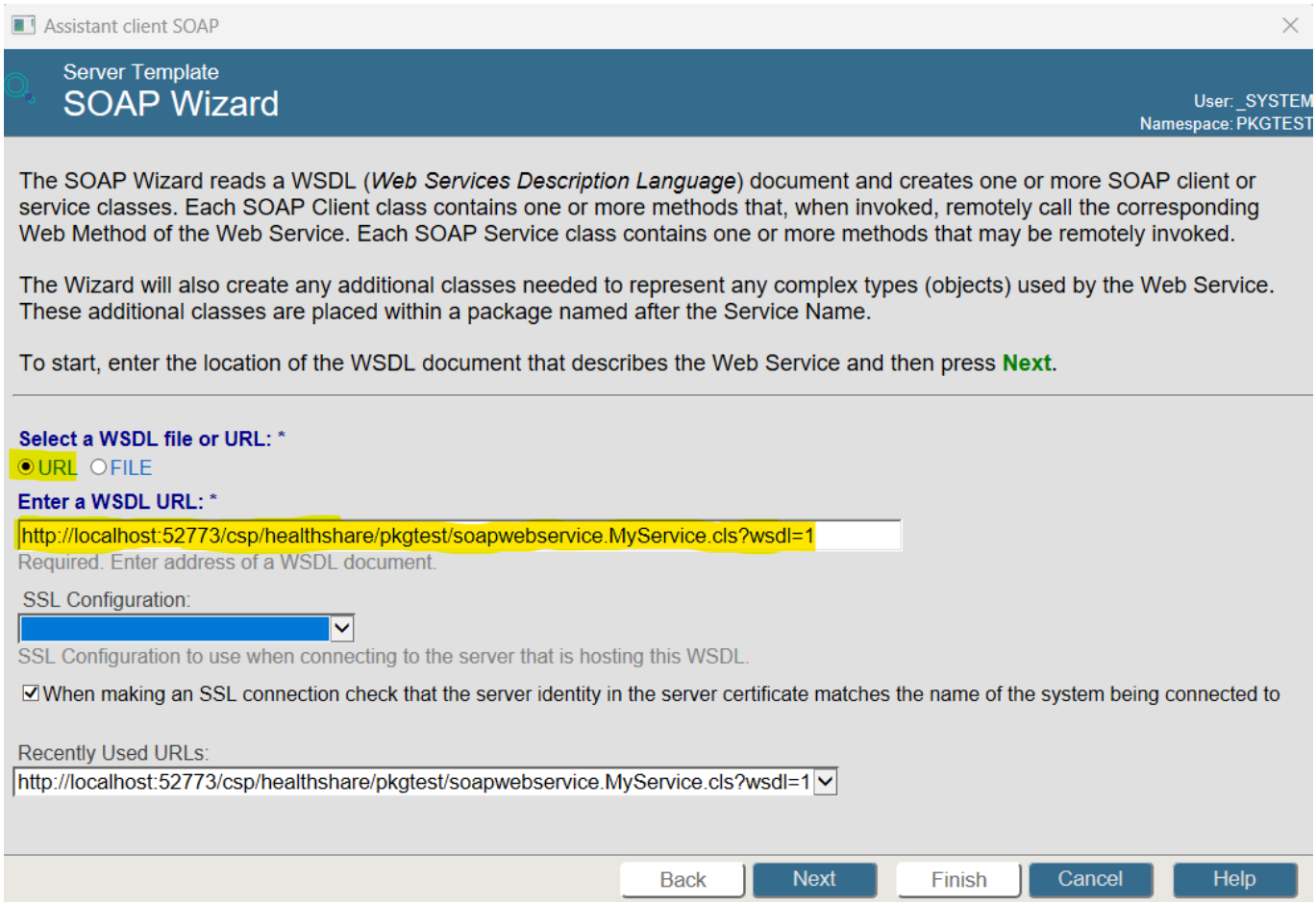
J ' aimerais profiter de notre sujet sur la capture pour les Entrepôts de Données de Santé (EDS) pour vous présenter comment créer rapidement des clients HTTP SOAP et REST. IRIS ainsi que des applications disponibles sur Open Exchange proposent des solutions permettant de les générer à partir d ' un WSDL ou d ' une spécification swagger.

Client SOAP

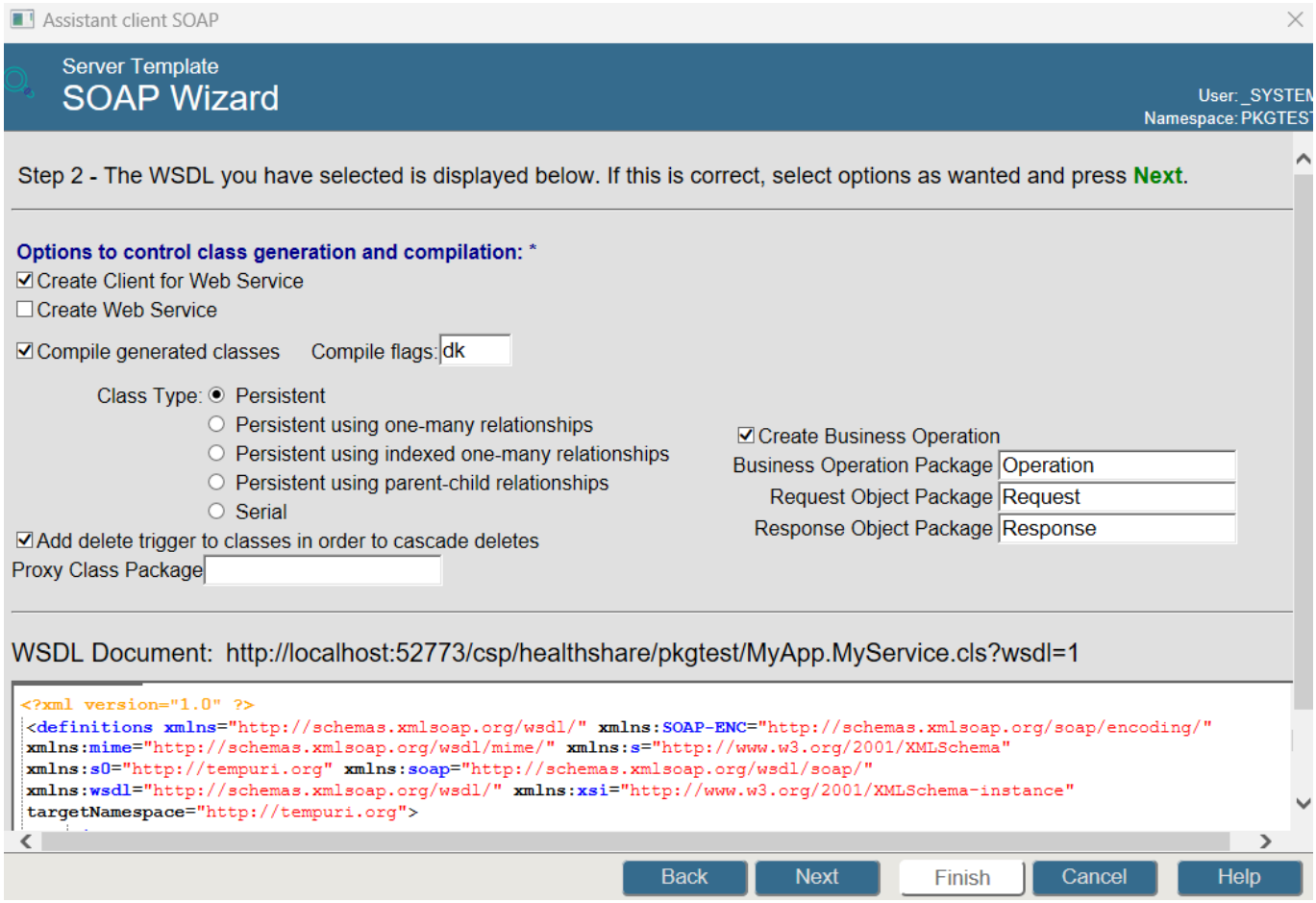
Pour créer un client SOAP, rien de plus simple, vous avez juste besoin du WSDL. Un assistant est disponible depuis le Studio IRIS, il permet de générer vos classes pour un client web service, mais également les business services et business operations si vous souhaitez le consommer avec le framework d'interopérabilité.



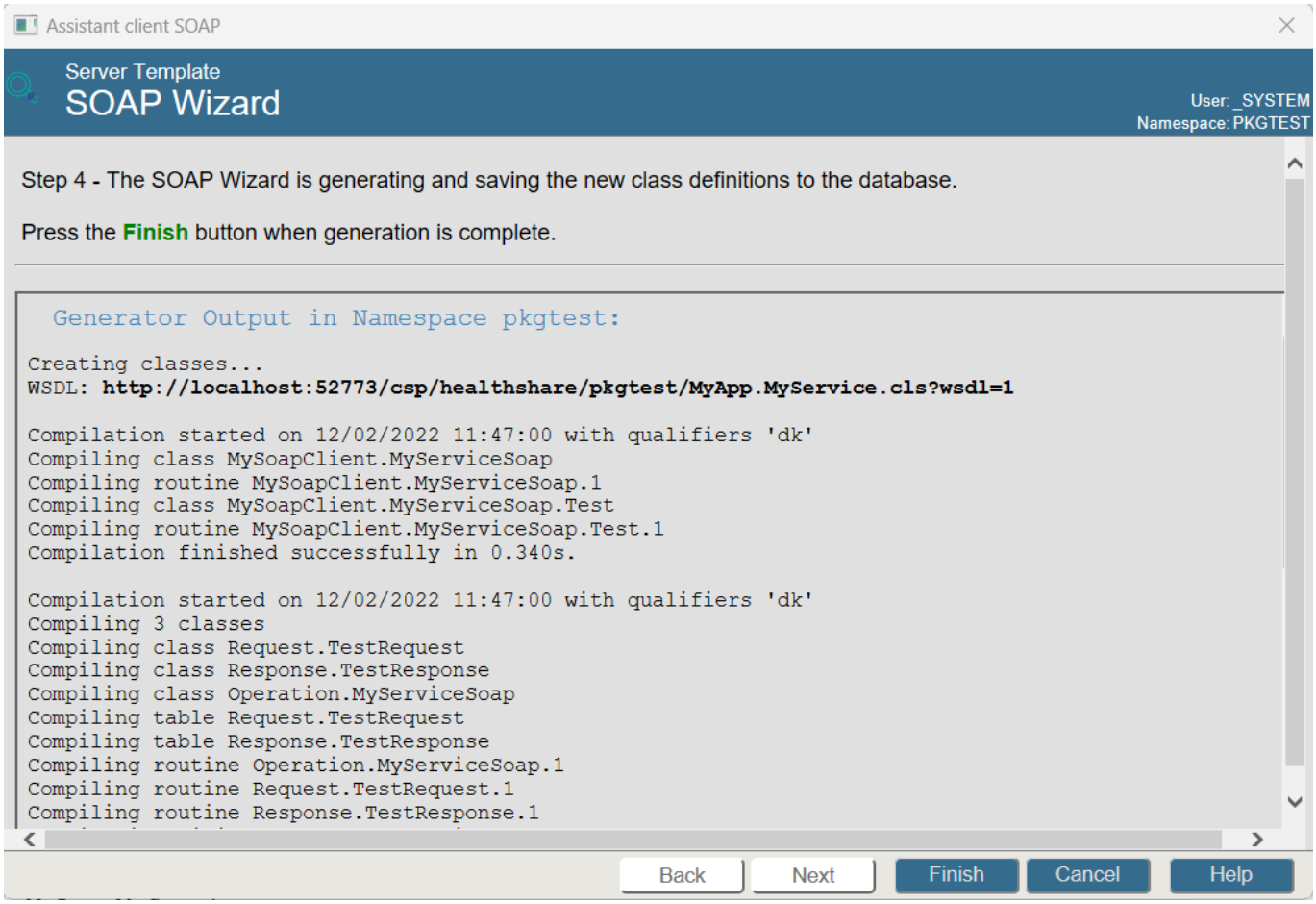
Indiquez simplement l ' url (ou le chemin d ' accès vers le fichier) pointant sur le WSDL.



Dans la fenêtre suivante, vous aurez le choix de générer un simple client web service, mais aussi les classes dédiées au framework d'interopérabilité. Pour cela, cochez " Business operation " , vous pourrez ensuite les utiliser dans vos productions.



Cliquez sur “ Next ” et voilà, votre client est généré.



Dans cet exemple, le service ne contient qu ' une méthode de test. Il peut être invoqué à l ' aide de la classe MySoapClient.MyServiceSoap de cette manière :

```
Set soapClient = ##class(MySoapClient.MyServiceSoap).%New()
Set soapClient.HttpUsername = "_SYSTEM"
Set soapClient.HttpPassword = "SYS"
Set value = soapClient.Test()
Write value
```

Production REST cliente

Si vous disposez de la spécification swagger 2.0 correspondant aux services REST que vous souhaitez consommer, il est possible de générer une production cliente avec le package openapi-client-gen disponible sur OpenExchange.

Vous pouvez installer ce package via cette commande ZPM.

```
zpm "install openapi-client-gen"
```

Dans le cas où ZPM ne serait pas encore disponible sur votre instance, voici comment l ' installer en une ligne de commande :

```
set $namespace="%SYS", name="DefaultSSL" do:'##class
(Security.SSLConfigs).Exists(name) ##class
(Security.SSLConfigs).Create(name) set url=
"https://pm.comm
unity.intersystems.com/packages/
zpm/latest/installer" Do ##class(%Net.URLParser).Parse(url,.comp) set ht =
##class(%Net.HttpRequest).%New(), ht.Server = comp("host"), ht.Port = 443
, ht.Https=1
, ht.SSLConfiguration=name, st=ht.Get(comp
("path")) quit:'st $System.Status.GetErrorText(st) set xml=##class(%File
).TempFilename("xml"), tFile = ##class(%Stream.FileBinary).%New
(), tFile.Filename = xml do
 tFile.CopyFromAndSave(ht.HttpResponse.D
ata) do ht.%Close(), $system.OBJ.Load(xml,"ck") do ##class(%File).Delete(xml)
```

Ensuite, il faut définir une configuration SSL par défaut.

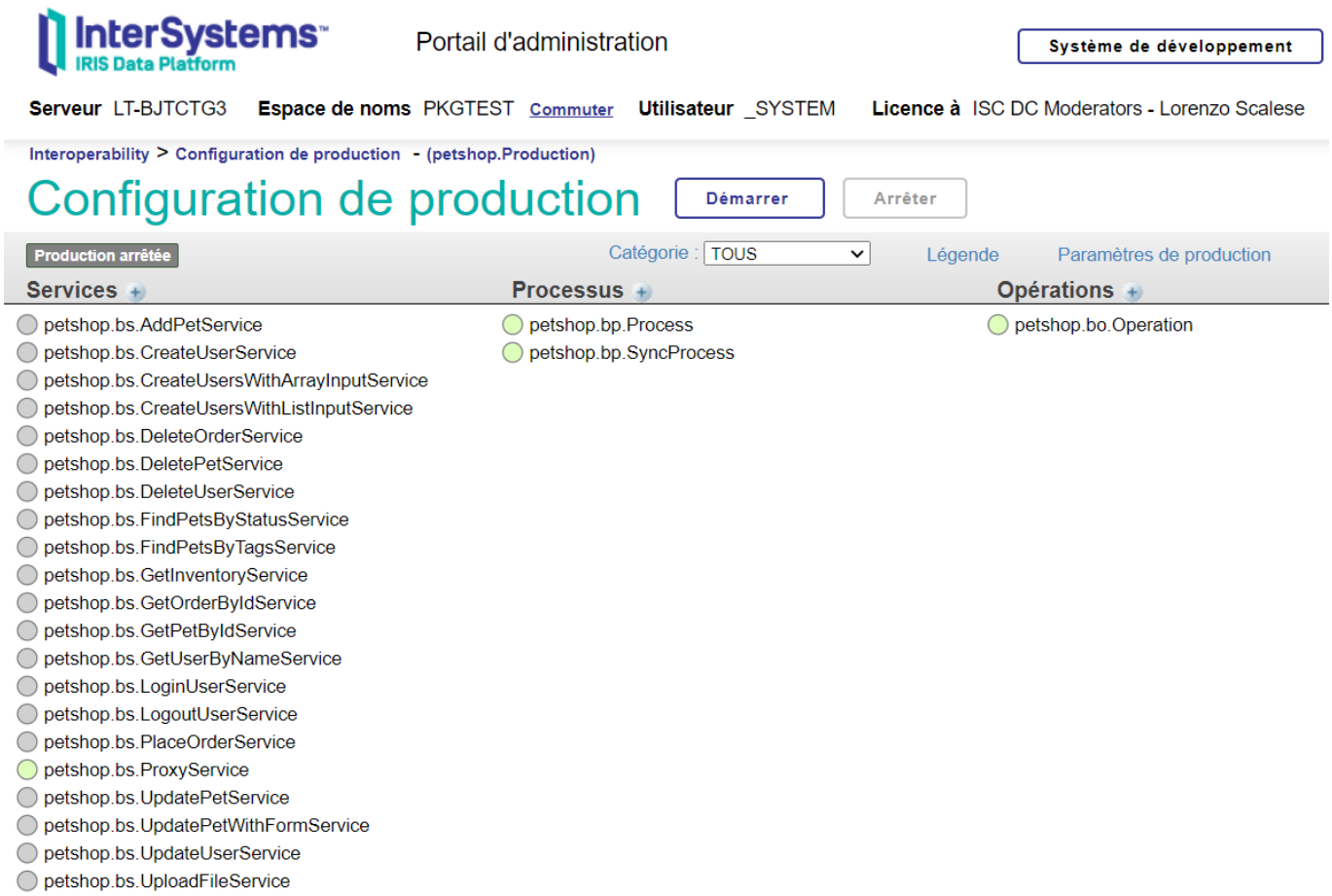
Dans les exemples présentés dans cet article, les requêtes seront toujours effectuées en HTTPS. La configuration SSL est donc obligatoire.

```
set $namespace="%SYS", name="default" do:'##class
(Security.SSLConfigs).Exists(name) ##class(Security.SSLConfigs).Create(name)
```

Pour l'exemple, les services REST disponibles sur petstore.swagger.io sont utilisés. La commande suivante permet de générer la production REST en passant l ' URL de la spécification swagger 2.0 en second paramètre. Il n ' est pas obligatoire d ' utiliser une URL, le second paramètre peut aussi être un chemin d ' accès vers un fichier contenant la spécification au format JSON ou YAML.

```
Set sc = ##class(dc.openapi.client.Spec).generateApp("petshop", "https://petstore.swagger.io/v2/swagger.json")
```

La production " petshop " est maintenant générée et accessible dans le portail d ' administration, on y retrouve les business services, processus et le business opération.



Une fois la production démarrée, les services peuvent être utilisés. Par exemple, pour récupérer la liste des animaux il faut utiliser le service “ FindPetsByStatusService ” .

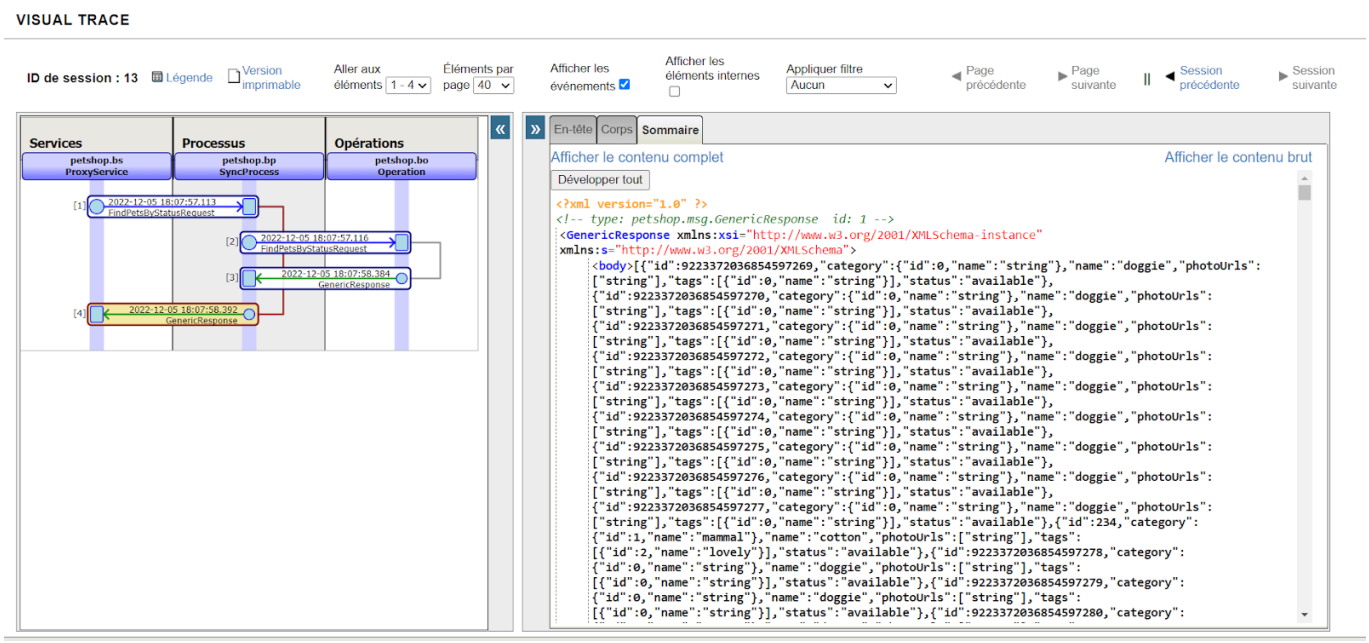
Voici le code permettant de la récupérer sur le “ petstore ” de façon synchrone:

```
Set ensRequest = ##class(petshop.msg.FindPetsByStatusRequest).%New()
Set ensRequest.accept = "application/json"
Do ensRequest.querystatus.Insert("available")
Set sc = ##class(petshop.Utils).invokeHostSync(
"petshop.bp.SyncProcess", ensRequest, "petshop.bs.ProxyService", , .pResponse)
```

Pour afficher la réponse JSON indentée dans le terminal, cette ligne de code peut être utilisée :

```
Do ##class(%JSON.Formatter).%New().Format({}.%FromJSON(pResponse.body))
```

Dans le “ Visual Trace ” (disponible via le portail administration), il est possible d ’ observer tous les messages échangés entre les différentes couches services, processus et opérations.



Voilà, comment en seulement quelques lignes de code il est possible de disposer d ' une production REST cliente prête à l ' emploi.

Client REST

Bien sûr, il est également possible de générer un client HTTP sans utilisation du framework d'interopérabilité. Le package openapi-client-gen le permet aussi, il faut pour cela ajouter un paramètre au moment de la génération :

```
Set features("simpleHttpClientOnly") = 1
Set sc = ##class
(dc.openapi.client.Spec).generateApp("petshopclient",
"https://petstore.swagger.io/v2/swagger.json", .features)
```

Une fois l ' opération terminée, la classe " petshopclient.HttpClient " est disponible.

Celle-ci contient toutes les méthodes permettant d'effectuer les requêtes HTTP pour tous les services définis dans la spécification. Voici un extrait de cette classe:

```
Class petshopclient.HttpClient Extends %RegisteredObject [ ProcedureBlock ]
{
...
/// Multiple status values can be provided with comma separated strings
Method GETFindPetsByStatus(pRequest As
 petshopclient.msg.FindP
etsByStatusRequest, Output pResponse As
 petshopclient.msg.GenericResponse
, pHttpRequestIn As %Net.HttpRequest = {..GetRequest
(pRequest)}, Output pHttpResponse As %Net.HttpResponse) As %Status
{
```

```
Set sc = $$$OK, pURL = "/v2/pet/findByStatus"
Set pHttpRequestIn.ContentType = pRequest.consume
Set key = ""
For  {
Set tmpValue = pRequest.querystatus.GetNext(.key)
Quit:key=""
Do pHttpRequestIn.SetParam("status", tmpValue, key)
}
$$$QuitOnError(pHttpRequestIn.Send("GET", pURL))
Set pHttpResponse=pHttpRequestIn.HttpResponse
Set pResponse = ##class(petshopclient.msg.GenericResponse).%New()
Set sc = ..genericProcessResponse(pRequest, pResponse,
"GETFindPetsByStatus", sc, $Get(pHttpResponse),
"petshopclient.msg.FindPetsByStatusResponse")
Return sc
}
...
Method GetRequest(pRequest As %RegisteredObject) As %Net.HttpRequest
{
#dim pHttpRequestIn As %Net.HttpRequest = ##class(%Net.HttpRequest).%New()
Do:##class(%Dictionary.CompiledProperty).%ExistsId(pRequest.%ClassName(1)_
"||accept") pHttpRequestIn.SetHeader("accept", pRequest.accept)
Set pHttpRequestIn.Server = "petstore.swagger.io"
Set pHttpRequestIn.Port = ""
Set pHttpRequestIn.Https = 1
Set:pHttpRequestIn.Https pHttpRequestIn.SSLConfiguration = "default"
Return pHttpRequestIn
}
}
```

Dans ce cas-ci la récupération de la liste des animaux disponibles se fera de manière un peu différente, c ' est un peu comme utiliser le business operation en direct :

```
Set requestMsg = ##class(petshopclient.msg.FindPetsByStatusRequest).%New()
Do requestMsg.querystatus.Insert("available")
Set httpClient = ##class(petshopclient.HttpClient).%New()
Set sc = httpClient.GETFindPetsByStatus(requestMsg, .responseMsg)
; pour afficher le résultat :
Do ##class(%JSON.Formatter).%New().Format({}.%FromJSON(responseMsg.body))
```

Tout comme dans l ' exemple avec le framework d'interopérabilité, le message de réponse contient une propriété body qui est le contenu brut de la réponse http.

En observant l ' objet " responseMsg " , on peut remarquer qu ' il contient une propriété " parsedResponse " qui dans ce cas-ci est de type " petshopclient.msg.FindPetsByStatusResponse " , celle-ci contient la réponse sérialisée dans un modèle objet persistant. Cela peut être utile pour le stockage en vue d ' une future transformation.

Requête HTTP

Jusqu ' à présent, le cas présenté est idéal, car une spécification swagger est disponible et permet de générer beaucoup de code. C ' est évidemment un grand avantage, mais pour certains services, il se peut qu ' aucune spécification ne soit disponible (connexion à une ancienne application pour laquelle il n ' y a plus de maintenance par exemple).

Dans ce cas, il est nécessaire d ' écrire le code permettant d'effectuer les requêtes http manuellement. Voici comment récupérer la liste des animaux disponible sur le " petstore " en écrivant manuellement le code pour la requête http :

```
Set httprequest=##class(%Net.HttpRequest).%New()
Set httprequest.Https=1
Set httprequest.SSLConfiguration="default"
Set httprequest.Server="petstore.swagger.io"
Do httprequest.SetHeader("HTTP_ACCEPT","application/json")
Do httprequest.InsertParam("status","available")
Do httprequest.Get("/v2/pet/findByStatus")
Set availablePets = {}.%FromJSON(httprequest.HttpResponse.Data)

Set jsonObject = {}.%FromJSON(httprequest.HttpResponse.Data)
Do ##class(%JSON.Formatter).%New().Format(jsonObject)
```

Bien sûr, cette fois il n ' y a pas la sérialisation de la réponse dans un modèle objet persistant. Toutefois, le langage ObjectScript supporte nativement le JSON grâce au %DynamicObject ce qui permettra de manipuler aisément les réponses.

Dans l ' exemple précédent, il s ' agissait d ' une simple requête HTTP GET, cela peut être plus complexe lors d ' un POST json ou upload de fichier. Petite astuce, n ' hésitez pas à générer un client depuis une spécification publique comme petstore afin de disposer du code généré en exemple. Cela peut vous faire gagner du temps dans la recherche des classes clés et leurs utilisations (%Net.MIMEPart, %Net.MIMEWriter) .

Ci-dessous, vous trouverez deux exemples à titre informatif.

Exemple de requête POST avec un message JSON dans le body :

```
Set sc = $$$OK, pURL = "/v2/pet"
Set pHttpRequestIn=##class(%Net.HttpRequest).%New()
Set pHttpRequestIn.Https=1
Set pHttpRequestIn.SSLConfiguration="default"
Set pHttpRequestIn.Server="petstore.swagger.io"
Set pHttpRequestIn.ContentType = pRequest.consume
Do pHttpRequestIn.SetHeader("HTTP_ACCEPT", "application/json")
Set pet = {
  "id": 0,
  "category": {
"id": 0,
"name": "string"
  },
  "name": "doggie",
  "photoUrls": [
"string"
  ],
```

```
  "tags": [
{
  "id": 0,
  "name": "string"
}
  ],
  "status": "available"
}
Do pHttpRequestIn.EntityBody.Write(pet.%ToJSON())
Set sc = pHttpRequestIn.Send("POST", pURL)
```

Exemple de requête POST pour uploader un fichier :

```
Set sc = $$$OK, pURL = "/v2/pet/{petId}/uploadImage"
Set pHttpRequestIn=##class(%Net.HttpRequest).%New()
Set pHttpRequestIn.Https=1
Set pHttpRequestIn.SSLConfiguration="default"
Set pHttpRequestIn.Server="petstore.swagger.io"
Set petId = 1
Set pHttpRequestIn.ContentType = "multipart/form-data"
Set pURL = $Replace(pURL, "{petId}", petId)
Set fileToUpload = ##class(%Stream.FileBinary).%New()
Set sc=fileToUpload.LinkToFile("/file/path/to_upload.png")
Set valueStream = ##class(%Stream.GlobalBinary).%New()
Do valueStream.CopyFrom(fileToUpload)
Set mParts = ##class(%Net.MIMEPart).%New()
Set mimePart = ##class(%Net.MIMEPart).%New(valueStream)
Do mimePart.SetHeader("Content-Disposition", "form-
data; name=""file""; filename=""file""")
Do mParts.Parts.Insert(mimePart)
Set mimeWriter = ##class(%Net.MIMEWriter).%New()
Do mimeWriter.OutputToStream(.stream)
Do mimeWriter.WriteMIMEBody(mParts)
Set pHttpRequestIn.EntityBody = stream
Set pHttpRequestIn.ContentType = "multipart/form-data; boundary=" _ mParts.Boundary
Set sc = pHttpRequestIn.Send("POST", pURL)
```

Voilà, c ’ est tout pour cet article. J ’ espère qu ’ il vous permettra de prendre en main facilement les captures de données via HTTP.

#Interopérabilité #REST API #SOAP #InterSystems IRIS

URL de la source:https://fr.community.intersystems.com/post/g%C3%A9n%C3%A9ration-de-client-soap-et-rest