Article
Lorenzo Scalese · Feb 1, 2023 17m read

Open Exchange

OpenAPI Suite - Part 1

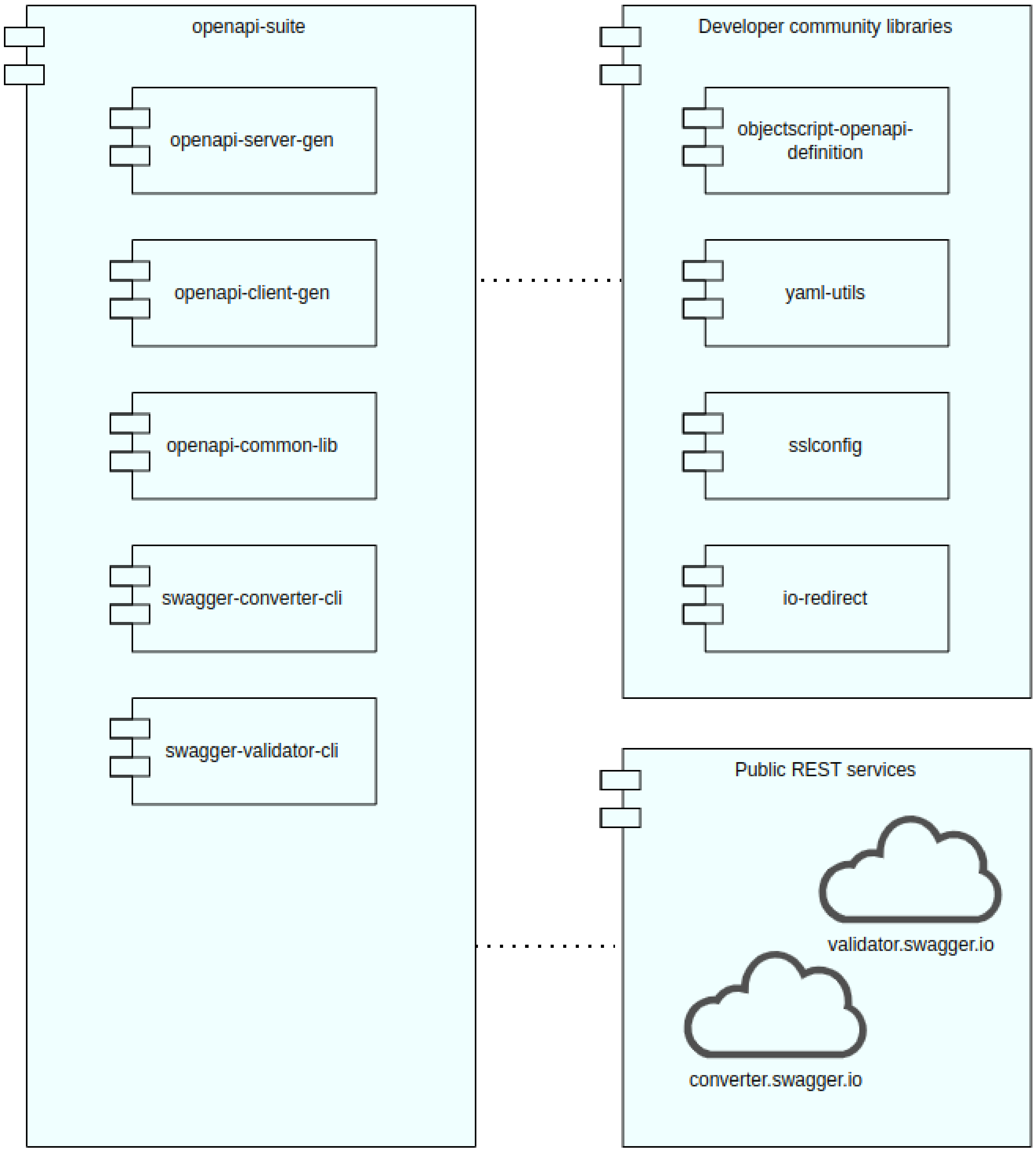
Hi Community,

I would like to present my last package OpenAPI-Suite, this is a set of tools to generate ObjectScript code from an OpenAPI specification version 3.0. In short, these packages allow to:

- Generate server-side class. It ’ s pretty similar to the generated code by %REST but the added value is the version 3.0 support.
- Generate HTTP client classes.
- Generate client production (business services, business operation, business process, Ens.Request, Ens.Response) classes.
- A web interface to generate and download the code or generate and compile directly on the server.
- Convert specification from version 1.x, 2.x to version 3.0.

Overview

OpenAPI Suite is split into many packages, and uses different developer community libraries and also public REST services. You can see on the schema below, all packages have been developed, and libraries and web services used:



Note: In case of a problem using public REST services, it ’ s possible to start a docker instance of the converter and validator service.

What does each package do?

OpenAPI Suite has been designed in different packages to ease maintenance, improvement and future extension. Each package has a role. Let's have a look at it!

[openapi-common-lib]

This contains all the common code to the others packages. For example, openapi-client-gen and openapi-server-gen accept the following input for an OpenAPI specification:

- URL
- File path
- %Stream.Object
- %DynamicObject
- Format YAML
- Format JSON
- OpenAPI version 1.x, 2.x, 3.0.x.

However, only a specification 3.0.x in a %DynamicObject can be processed. The code for the transformation is located in this package. It also contains various utilities.

swagger-converter-cli

It ’ s a dependency of openapi-common-lib. This is an HTTP client using the public REST service converter.swagger.io in order to convert OpenAPI version 1.x or 2.x in version 3.0.

swagger-validator-cli

It ’ s also a dependency of openapi-common-lib, even if his name is “ validator ” , it ’ s not used to validate the specification. converter.swagger.io provide the service “ parse ” allowing to simplify the structure of an OpenAPI specification. For example: create a definition for a “ nested object definition ” and replace it with a “ $ref ” . This reduces the number of cases to be dealt with in the code generation algorithm.

openapi-client-gen

This package is dedicated to client-side code generation to help developers to consume REST services.

It includes a simple HTTP client or a Production client (business services, process, operation, Production classes). Originally created to support OpenAPI 2.x It was just completely refactored to support version 3.x.

openapi-server-gen

The opposite of openapi-client-gen, it is dedicated to server-side code generation. There is no interest in specification version 2.0 because ^%REST exists but the target of this package the version 3.0 support.

openapi-suite

It gathers all the above packages and provides a REST API to:

- Generate the code and compile code on the IRIS instance.
- Generate code without compiling for download only.

A web interface is also provided to consume this REST API and thus exploit the functionalities of the OpenAPI Suite.

And the libraries?

Here are some of the existing libraries on the DC that have been useful in this development:

objectscript-openapi-definition

A useful library to generate model classes from an OpenAPI specification. This is a very important piece of this project and I ’ m also a contributor.

ssl-client

Allows creating SSL Configuration. Mainly used to create a configuration name “ DefaultSSL ” for HTTPS requests.

yaml-utils

In the case of YAML format specification, this library is used to convert into JSON format. A must-have in this project. By the way, it was developed initially to test YAML specification with openapi-client-gen version 1.

io-redirect

This is one of my libraries, It allows redirecting “ write ” into a stream, file, global or string variable. It ’ s used by the REST service to keep a trace of the logs. It ’ s inspired of this community post.

Installation IPM

To install the suite, your best friend is IPM (zpm). There are many packages and dependencies, and using IPM is definitely convenient.

```
zpm "install openapi-suite"
; optional
; zpm "install swagger-ui"
```

Installation Docker

There is nothing special, this project uses the intersystems-iris-dev-template.

```
git clone git@github.com:lscalese/openapi-suite.git
cd openapi-suite
docker-compose up -d
```

If you have an error at Iris start, maybe this is a permission issue iris-main.log.

You can try:

```
touch iris-main.log && chmod 777 iris-main.log
```

Note: adding RW permission for the irisowner user should be enough.

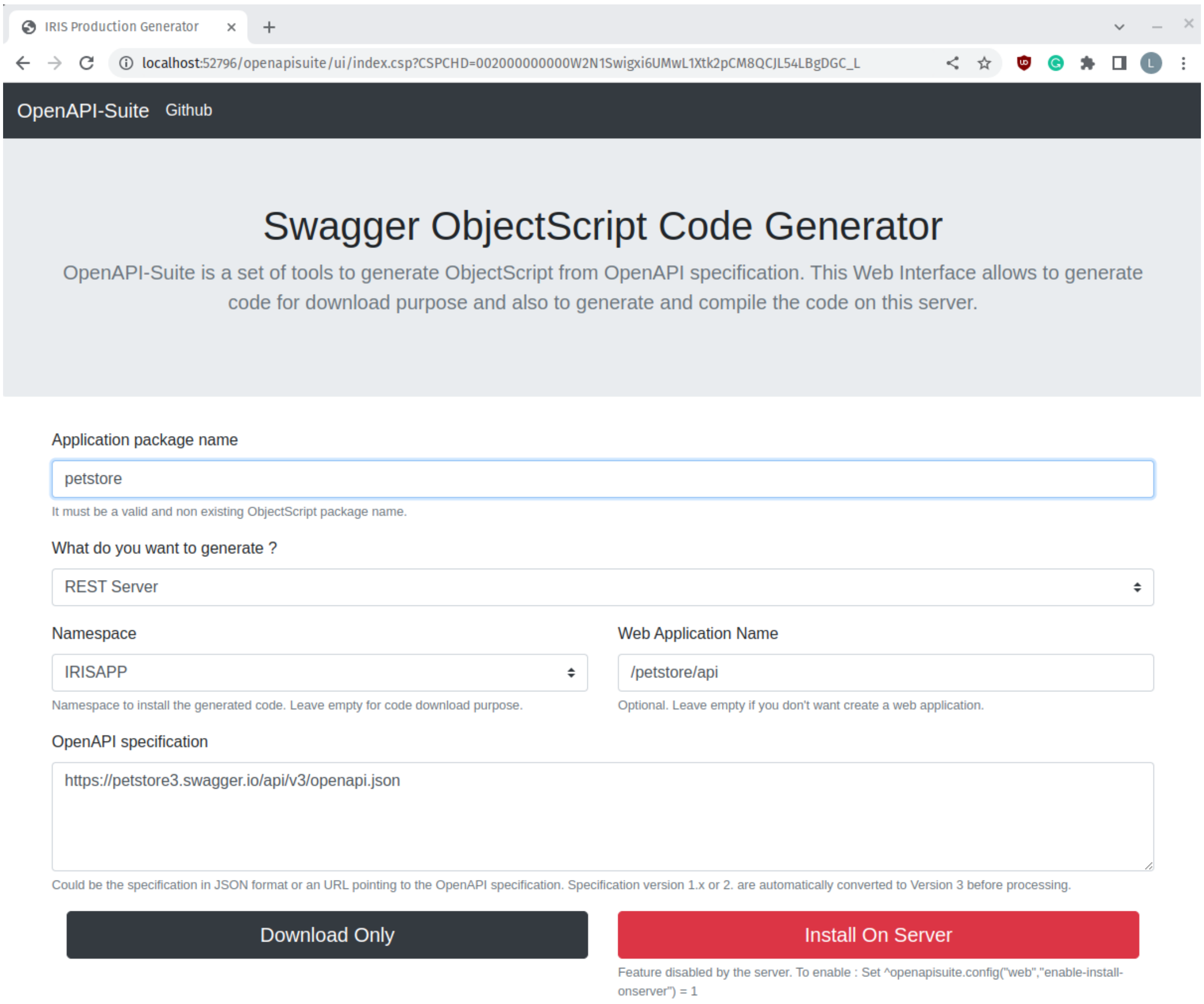
How to use It

OpenAPI-Suite provides a web interface to “ Generate and download ” or “ Generate and install ” code.

The interface is available at http://localhost:52796/openapisuite/ui/index.csp (*adapt with your port number if needed).

It ’ s very simple, fill out the form:

1. Application package name: this is the package used for the generated classes. It must be a non-existing package name.
2. Select what do you want to generate: HTTP Client, Client Production or REST server.
3. Select the namespace where the code will be generated. It makes sense only if you click “ Install On Server ” otherwise this field will be ignored.
4. Web Application Name is optional and available only if you select “ REST Server ” to generate. Leave empty if you don ’ t want to create a Web Application related to the REST dispatch class generated.
5. The OpenAPI specification field could be an URL pointing to the specification or a copy\paste of the specification itself (in this case the specification must be in JSON format).


IRIS Production Generator
localhost:52796/openapisuite/ui/index.csp?CSPCHD=002000000000W2N1Swigxi6UMwL1Xtk2pCM8QCJL54LBgDGC_L
OpenAPI-Suite
Github

Swagger ObjectScript Code Generator

OpenAPI-Suite is a set of tools to generate ObjectScript from OpenAPI specification. This Web Interface allows to generate code for download purpose and also to generate and compile the code on this server.

Application package name: petstore

It must be a valid and non existing ObjectScript package name.

What do you want to generate ?: REST Server

Namespace: IRISAPP

Namespace to install the generated code. Leave empty for code download purpose.

Web Application Name: /petstore/api

Optional. Leave empty if you don't want create a web application.

OpenAPI specification: https://petstore3.swagger.io/api/v3/openapi.json

Could be the specification in JSON format or an URL pointing to the OpenAPI specification. Specification version 1.x or 2. are automatically converted to Version 3 before processing.

Feature disabled by the server. To enable : Set ^openapisuite.config("web","enable-install-onserver") = 1

If you click the " Download Only " button the code will be generated and returned in an XML file, and then the classes will be deleted from the server. The namespace used to store temporarily the generated classes is the namespace where OpenAPI-Suite is installed (By default IRISAPP if you use a docker installation).

However, if you click the " Install On Server " button the code will be generated and compiled, and the server returns a JSON message with the status of the code generation \compilation and also logs.

By default this feature is disabled, to enable just open an IRIS terminal and:

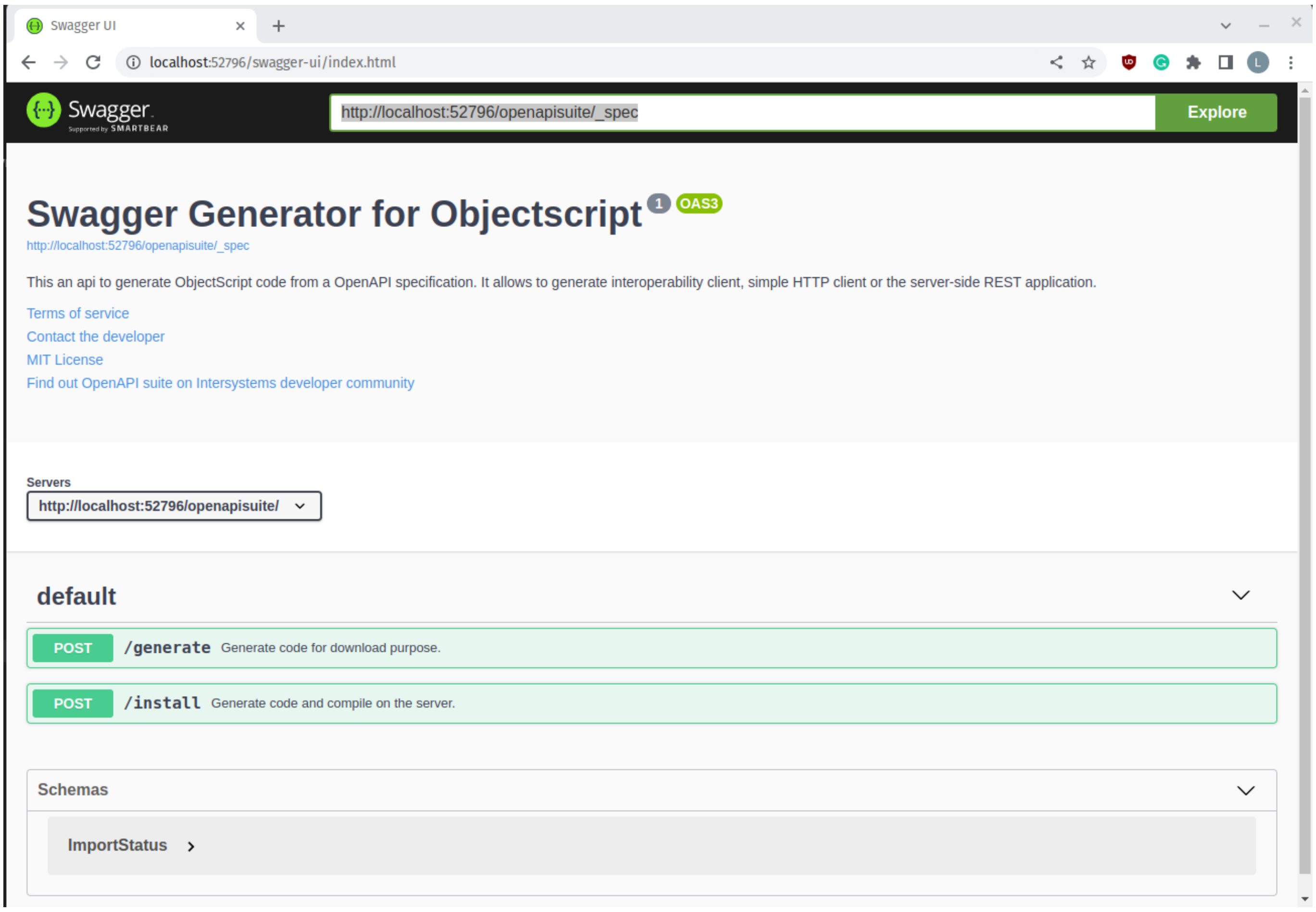
```
Set ^openapisuite.config("web","enable-install-onserver") = 1
```

Explore the OpenAPI-suite REST API

The CSP form uses REST services available at http://localhost:52796/openapisuite.

Open swagger-ui http://localhost:52796/swagger-ui/index.html and
explore http://localhost:52796/openapisuite/spec

This is the first step towards creating a more advanced front-end application with the Angular framework.


Swagger Generator for Objectscript 1 OAS3

http://localhost:52796/openapisuite/_spec

This an api to generate ObjectScript code from a OpenAPI specification. It allows to generate interoperability client, simple HTTP client or the server-side REST application.

Terms of service
Contact the developer
MIT License
Find out OpenAPI suite on Intersystems developer community

Servers: http://localhost:52796/openapisuite/

default

POST /generate Generate code for download purpose.

POST /install Generate code and compile on the server.

Schemas

ImportStatus

Generate code programmatically

Of course, it's not mandatory to use the UI, we see in this section how to generate the code programmatically and how to use the generated services.

All snippets are also available in the class dc.openapi.suite.samples.PetStore.

HTTP client

```
Set features("simpleHttpClientOnly") = 1
Set sc = ##class
(dc.openapi.client.Spec).generateApp("petstoreclient",
"https://petstore3.swagger.io/api/v3/openapi.json", .features)
```

The first argument is the package where the classes will be generated, so be sure to pass a valid package name. The second argument could be an URL pointing to the specification, a filename, a stream, or a %DynamicObject. “ features ” is an array, currently only subscripts are available :

simpleHttpClientOnly: if 1 only a simple HTTP client will be generated otherwise, a production will be also generated (default behaviour).

compile: if 0 the generated code won ’ t be compiled. It could be useful if you want to generate code for export purposes only. By default, compile = 1

Below is an example of how to use the service “ addPet ” with the just-generated HTTP client:

```
Set messageRequest = ##class(petstoreclient.requests.addPet).%New()
Set messageRequest.%ContentType = "application/json"
Do messageRequest.PetNewObject().%JSONImport({"id":456,"name":"Mittens",
"photoUrls":[
"https://static.wikia.nocookie.net/disney/images/c/cb/Profile_-_Mittens.jpg/revision/
latest?cb=20200709180903"],"status":"available"})

Set httpClient = ##class(petstoreclient.HttpClient).%New(
"https://petstore3.swagger.io/api/v3","DefaultSSL")
; MessageResponse will be an instance of petstoreclient.responses.addPet
Set sc = httpClient.addPet(messageRequest, .messageResponse)
If $$$ISERR(sc) Do $SYSTEM.Status.DisplayError(sc) Quit sc

Write !,"Http Status code : ", messageResponse.httpStatusCode,!
Do messageResponse.Pet.%JSONExport()
```

Click to show generated classes.

HTTP client Production

```
Set sc = ##class
(dc.openapi.client.Spec).generateApp("petstoreproduction",
"https://petstore3.swagger.io/api/v3/openapi.json")
```

The first argument is the package name if you test code generation of simple HTTP client and client production make sure to use a different package name. The second and third follow the same rules as the HTTP client.

Before testing, please start the production via the management portal using this command

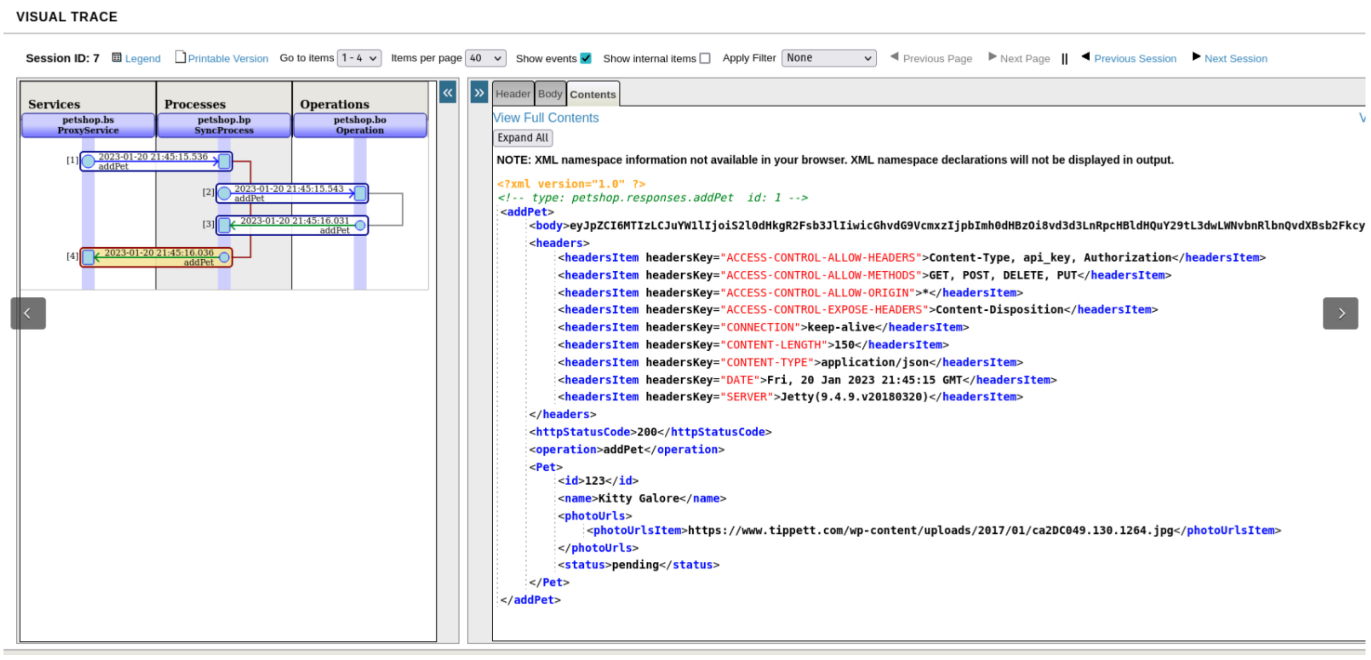
```
Do ##class(Ens.Director).StartProduction("petstoreproduction.Production")
```

Below is an example of how to use the service “ addPet ” , but this time with the generated production:

```
Set messageRequest = ##class(petstoreproduction.requests.addPet).%New()
Set messageRequest.%ContentType = "application/json"
Do messageRequest.PetNewObject().%JSONImport({"id":123,"name":
"Kitty Galore","photoUrls":["https://www.tippett.com/wp-
content/uploads/2017/01/ca2DC049.130.1264.jpg"],"status":"pending"})
; MessageResponse will be an instance of petstoreclient.responses.addPet
Set sc = ##class
```

```
(petstoreproduction.Utils).invokeHostSync(
"petstoreproduction.bp.SyncProcess", messageRequest,
"petstoreproduction.bs.ProxyService", , .messageResponse)
Write !, "Take a look in visual trace (management portal)"
If $$$ISERR(sc) Do $SYSTEM.Status.DisplayError(sc)
Write !,"Http Status code : ", messageResponse.httpStatusCode,!
Do messageResponse.Pet.%JSONExport()
```

Now, you can open the visual trace to see the details:



NOTE: XML namespace information not available in your browser. XML namespace declarations will not be displayed in output.

```
<?xml version="1.0" ?>
<!-- type: petshop.responses.addPet  id: 1 -->
<addPet>
    <body>eyJpZCI6MTIzLCJuYW1lIjoiS2l0dHkgR2Fsb3JlIiwicGhvdG9VcmxzIjpbImh0dHBzOi8vd3d3LnRpcHBldHQuY29tL3dwLWNvbnRlbnQvdXBsb2Fkcy
    <headers>
        <headersItem headersKey="ACCESS-CONTROL-ALLOW-HEADERS">Content-Type, api_key, Authorization</headersItem>
        <headersItem headersKey="ACCESS-CONTROL-ALLOW-METHODS">GET, POST, DELETE, PUT</headersItem>
        <headersItem headersKey="ACCESS-CONTROL-ALLOW-ORIGIN">*</headersItem>
        <headersItem headersKey="ACCESS-CONTROL-EXPOSE-HEADERS">Content-Disposition</headersItem>
        <headersItem headersKey="CONNECTION">keep-alive</headersItem>
        <headersItem headersKey="CONTENT-LENGTH">150</headersItem>
        <headersItem headersKey="CONTENT-TYPE">application/json</headersItem>
        <headersItem headersKey="DATE">Fri, 20 Jan 2023 21:45:15 GMT</headersItem>
        <headersItem headersKey="SERVER">Jetty(9.4.9.v20180320)</headersItem>
    </headers>
    <httpStatusCode>200</httpStatusCode>
    <operation>addPet</operation>
    <Pet>
        <id>123</id>
        <name>Kitty Galore</name>
        <photoUrls>
            <photoUrlsItem>https://www.tippett.com/wp-content/uploads/2017/01/ca2DC049.130.1264.jpg</photoUrlsItem>
        </photoUrls>
        <status>pending</status>
    </Pet>
</addPet>
```

The generated classes in the packages model, requests and responses are pretty similar to the generated code for a simple HTTP client. Classes from package requests inherit from Ens.Request and classes in package responses inherit Ens.Response. The default implementation of the business operation is very simple, see this snippet:

```
Class petstoreproduction.bo.Operation Extends
 Ens.BusinessOperation [ ProcedureBlock ]
{

Parameter ADAPTER = "EnsLib.HTTP.OutboundAdapter";
Property Adapter As EnsLib.HTTP.OutboundAdapter;
/// Implement operationId : addPet
/// post /pet
Method addPet(requestMessage As
 petstoreproduction.reque
sts.addPet, Output responseMessage As petstoreproduction.responses.addPet) As %Status
{
 Set sc = $$$OK, pHttpRequestIn = ##class(%Net.HttpRequest).%New
(), responseMessage = ##class(petstoreproduction.responses.addPet).%New()
 $$$QuitOnError(requestMessage.LoadHttpRequestObject(pHttpRequestIn))
 $$$QuitOnError(..Adapter
.SendFormDataArray(.pHttpResponse, "post", pHttpRequestIn, , , ..Adapter
.URL_requestMessage.%URL))
 $$$QuitOnError(responseMessage.LoadFromResponse(pHttpResponse, "addPet"))
 Quit sc
}
...
}
```

```
}
```

HTTP server-side REST application

```
Set sc = ##class
(dc.openapi.server.ServerAppGenerator
).Generate("petstoreserver",
"https://petstore3.swagger.io/api/v3/openapi.json", "/petstore/api")
```

The first argument is the package name to generate classes. The second follows the same rules as the HTTP client. The third argument is not mandatory, but if present a web application will be created with the given name (be careful to give a valid web application name).

The class petstoreserver.disp (dispatch %CSP.REST class) looks like a code generated by ^%REST, performs many checks to accept or reject the request and calls the related service ClassMethod implementation in petstoreserver.impl. The main difference is the argument passed to the implementation method, this is petstoreserver.requests object. Example :

```
Class petstoreserver.disp Extends %CSP.REST [ ProcedureBlock ]
{

Parameter CHARSET = "utf-8";
Parameter CONVERTINPUTSTREAM = 1;
Parameter IgnoreWrites = 1;
Parameter SpecificationClass = "petstoreserver.Spec";
/// Process request post /pet
ClassMethod addPet() As %Status
{
 Set sc = $$$OK
 Try{
  Set acceptedMedia = $ListFromString("application/json,application/xml,application/x-
www-form-urlencoded")
  If '$ListFind(acceptedMedia,$$$LOWER(%request.ContentType)) {
    Do ##class(%REST.Impl).%ReportRESTError
(..#HTTP415UNSUPPORTEDMEDIATYPE,$$$ERROR($$$RESTContentType,
%request.ContentType)) Quit
  }
  Do ##class(%REST.Impl).%SetContentType($Get(%request.CgiEnvs("HTTP_ACCEPT")))
  If '##class(%REST.Impl).%CheckAccepts(
"application/xml,application/json") Do ##class(%REST.Impl).
%ReportRESTError(..#HTTP406NOTACCEPTABLE,$$$ERROR($$$RESTBadAccepts)) Quit
  If '$isobject(%request.Content) Do ##class(%REST.Impl).
%ReportRESTError(..#HTTP400BADREQUEST,$$$ERROR($$$RESTRequired,"body")) Quit
  Set requestMessage = ##class(petstoreserver.requests.addPet).%New()
  Do requestMessage.LoadFromRequest(%request)
  Set scValidateRequest = requestMessage.RequestValidate()
  If $$$ISERR(scValidateRequest) Do ##class(%REST.Impl).
%ReportRESTError(..#HTTP400BADREQUEST,$$$ERROR(5001,
"Invalid requestMessage object.")) Quit
  Set response = ##class(petstoreserver.impl).addPet(requestMessage)
  Do ##class(petstoreserver.impl).%WriteResponse(response)
 } Catch(ex) {
  Do ##class(%REST.Impl).%ReportRESTError
(..#HTTP500INTERNALSERVERERROR,ex.AsStat
us(),$parameter("petstoreserver.impl","ExposeServerExceptions"))
 }
 Quit sc
}
...
}
```

As you can see, the dispatch class call " LoadFromRequest " and " RequestValidate " before calling the implementation method. These methods have a default implementation, but the code generator cannot cover all cases. Currently, the most common cases are automatically handled as parameters in "query", "headers", "path" and body with the content type " application/json " , " application/octet-stream " or " multipart/form-data " . The developer has to check the implementation to check \complete if needed (by default the code generator set $$$ThrowStatus($$$ERROR($$$NotImplemented)) for unsupported case).

Example of request class :

As with the ^%REST usage, the "petstoreserver.impl" class contains all the methods related to the services that the developer has to implement.

```
Class petstoreserver.impl Extends %REST.Impl [ ProcedureBlock ]
{

Parameter ExposeServerExceptions = 1;
/// Service implemntation for post /pet
ClassMethod addPet(messageRequest As
 petstoreserver.requests.addPet) As %DynamicObject
{
 ; Implement your service here.
 ; Return {}
 $$$ThrowStatus($$$ERROR($$$NotImplemented))
}

...
}
```

Short description of the generated packages

Package name \Class Name	Type
petstoreclient.model petstoreproduction.model	Client-side and server-side
petstoreclient.requests	Client-side and server-side

petstoreproduction.requests		
petstoreclient.responses petstoreproduction.responses	Client-side and server-side	
petstoreclient.HttpClient	Client-side	
petstoreproduction. bo.Operation	Client-side	
petstoreproduction.bp	Client-side	
petstoreproduction.bs	Client-side	
petstoreproduction.Production	Client-side	
petstoreserver.disp	Server-side	
petstoreserver.Spec	Server-side	
petstoreserver.impl	Server-side	

Development status

OpenAPI-Suite is still a very young product and needs to be more tested and then improved. The support of OpenAPI 3 is partial, more possibilities could be supported.

The tests were done with the public specification https://petstore3.swagger.io/api/v3/openapi.json and two other relatively simple ones. Of course, this is not enough to cover all cases. If you have any specifications to share, I would be happy to use them for my tests.

I think the foundation of the project is good and it can easily evolve, for example, be extended to support AsyncAPI.

Do not hesitate to leave feedback.

I hope you will enjoy this application and deserves your support for the developer tools contest.

Thank you for reading.

#Interoperability #REST API #InterSystems IRIS #InterSystems Ideas Portal #Open Exchange
Check the related application on InterSystems Open Exchange

Source URL:https://community.intersystems.com/post/openapi-suite-part-1