Article
Eduard Lebedyuk · Nov 20, 2019 9m read

Developing REST API with a spec-first approach

In this article, I would like to talk about the spec-first approach to REST API development.

While traditional code-first REST API development goes like this:

- Writing code
- REST-enabling it
- Documenting it (as a REST API)

Spec-first follows the same steps but reverse. We start with a spec, also doubling as documentation, generate a boilerplate REST app from that and finally write some business logic.

This is advantageous because:

- You always have relevant and useful documentation for external or frontend developers who want to use your REST API
- Specification created in OAS (Swagger) can be imported into a variety of tools allowing editing, client generation, API Management, Unit Testing and automation or simplification of many other tasks
- Improved API architecture. In code-first approach, API is developed method by method so a developer can easily lose track of the overall API architecture, however with the spec-first developer is forced to interact with an API from the position if API consumer which usually helps with designing cleaner API architecture
- Faster development - as all boilerplate code is automatically generated you won't have to write it, all that's left is developing business logic.
- Faster feedback loops - consumers can get a view of the API immediately and they can easier offer suggestions simply by modifying the spec

Let's develop our API in a spec-first approach!

Plan

1. Develop spec in swagger
    - Docker
    - Locally
    - Online
2. Load spec into IRIS
    - API Management REST API
    - ^REST
    - Classes
3. What happened with our spec?
4. Implementation
5. Further development
6. Considerations
    - Special parameters
    - CORS
7. Load spec into IAM

Develop specification

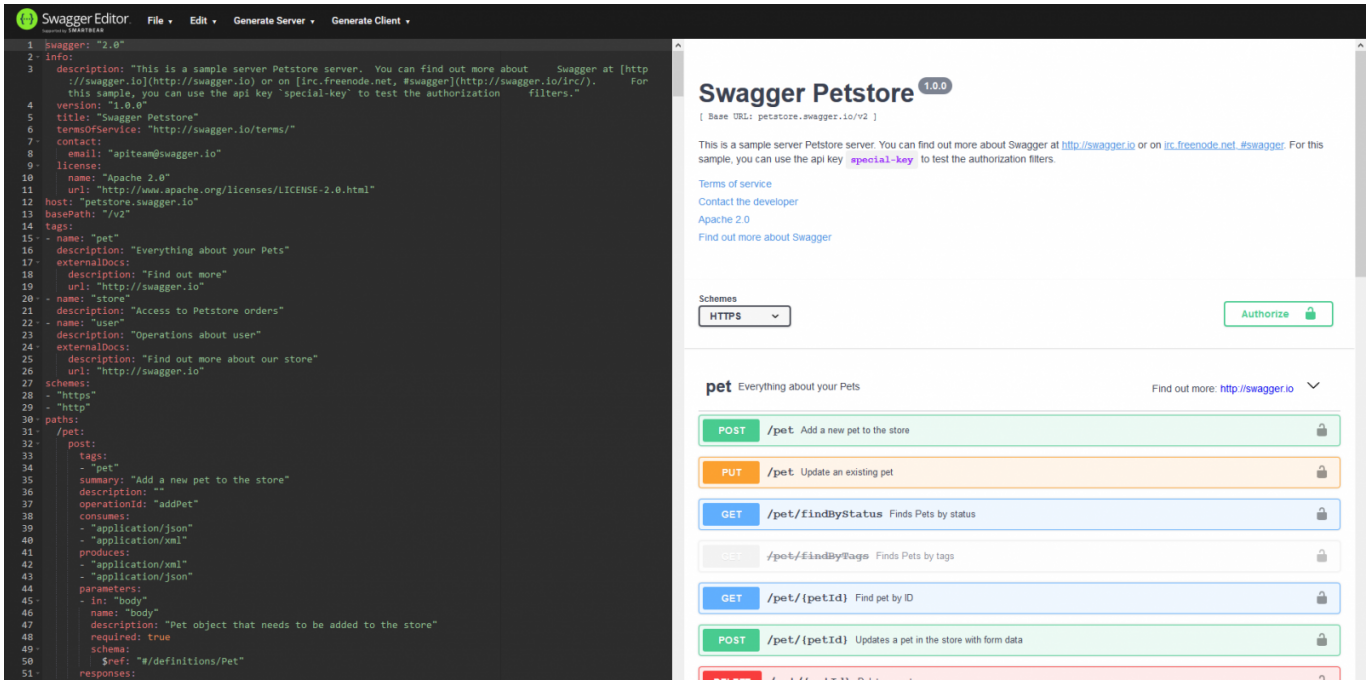
The first step is unsurprisingly writing the spec. InterSystems IRIS supports Open API Specification (OAS):

> **OpenAPI Specification** (formerly Swagger Specification) is an API description format for REST APIs. An OpenAPI file allows you to describe your entire API, including:
>
> - Available endpoints (/users) and operations on each endpoint (GET /users, POST /users)
> - Operation parameters Input and output for each operation
> - Authentication methods
> - Contact information, license, terms of use and other information.
>
> API specifications can be written in YAML or JSON. The format is easy to learn and readable to both humans and machines. The complete OpenAPI Specification can be found on GitHub: OpenAPI 3.0 Specification
>
> - from Swagger docs.

We will use Swagger to write our API. There are several ways to use Swagger:

- Online
- Docker: docker run -d -p 8080:8080 swaggerapi/swagger-editor
- Local installation

After installing/running Swagger, you should see this window in a web browser:



On the left side, you edit the API specification and on the right, you immediately see rendered API documentation/testing tool.

Let's load our first API spec into it (in YAML). It is a simple API with one GET request - returning random number in a specified range.

Math API Specification

Here's what it consists of.

Basic information about our API and used OAS version.

```
swagger: "2.0"
info:
  description: "Math"
  version: "1.0.0"
  title: "Math REST API"
```

Server host, protocol (http, https) and Web application names:

```
host: "localhost:52773"
basePath: "/math"
schemes:
  - http
```

Next we specify a path (so complete URL would be http://localhost:52773/math/random/:min/:max) and HTTP request method (get, post, put, delete):

```
paths:
  /random/{min}/{max}:
    get:
```

After that, we specify information about our request:

```
      x-ISC_CORS: true
      summary: "Get random integer"
      description: "Get random integer between min and max"
      operationId: "getRandom"
      produces:
      - "application/json"
      parameters:
      - name: "min"
        in: "path"
        description: "Minimal Integer"
        required: true
        type: "integer"
        format: "int32"
      - name: "max"
        in: "path"
        description: "Maximal Integer"
        required: true
        type: "integer"
        format: "int32"
      responses:
        200:
          description: "OK"
```

In this part we define our request:

- Enable this path for CORS (more on that later)
- Provide *summary* and *description*
- *operationId* allows in-spec reference, also it's a generated method name in our implementation class
- *produces* - response format (such as text, xml, json)
- *parameters* specify input parameters (be they in URL or body), in our case we specify 2 parameters - range for our random number generator
- *responses* list possible responses form server

As you see this format is not particularly challenging, although there are many more features available, here's a specification.

Finally, let's export our definition as a JSON. Go To File → Convert and save as JSON. The specification should look like this:

Math API Specification

Load specification into IRIS

Now that we have our spec, we can generate boilerplate code for this REST API in InterSystems IRIS.

To move to this stage we'll need three things:

- REST Application name: package for our generated code (let's say math)
- OAS spec in a JSON format: we just created it in a previous step
- WEB Application name: a base path to access our REST API (/math in our case)

There are three ways to use our spec for code generation, they are essentially the same and just offer various ways to access the same functionality

1. Call ^%REST routine (Do ^%REST in an interactive terminal session), documentation.
2. Call %REST class (Set sc = ##class(%REST.API).CreateApplication(applicationName, spec), non-interactive), documentation.
3. Use API Management REST API, documentation.

I think documentation adequately describes required steps so just choose one. I'll add two notes:

- In case (1) and (2) you can pass a dynamic object a filename or a URL
- In cases (2) and (3) you **must** make an additional call to create a WEB application: set sc = ##class(%SYS.REST).DeployApplication(restApp, webApp, authenticationType), so in our case set sc = ##class(%SYS.REST).DeployApplication("math", "/math"), get values for authenticationType argument from %sySecurity include file, relevant entries are $$$Authe*, so for unauthenticated access pass $$$AutheUnauthenticated. If omitted, the parameter defaults to password authentication.

What happened with our spec?

If you've created the app successfully, new math package should be created with three classes:

- *Spec* - stores the specification as-is.
- *Disp* - directly called when the REST service is invoked. It wraps REST handling and calls implementation methods.
- *Impl* - holds the actual internal implementation of the REST service. You should edit only this class.

Documentation with more information about the classes.

Implementation

Initially our implementation class math.impl contains only one method, corresponding to our /random/{min}/{max} operation:

```
/// Get random integer between min and max<br/>
/// The method arguments hold values for:<br/>
///     min, Minimal Integer<br/>
///     max, Maximal Integer<br/>
ClassMethod getRandom(min As %Integer, max As %Integer) As %DynamicObject
{
    //(Place business logic here)
    //Do ..%SetStatusCode(<HTTP_status_code>)
    //Do ..%SetHeader(<name>,<value>)
    //Quit (Place response here) ; response may be a string, stream or dynamic object
}
```

Let's start with the trivial implementation:

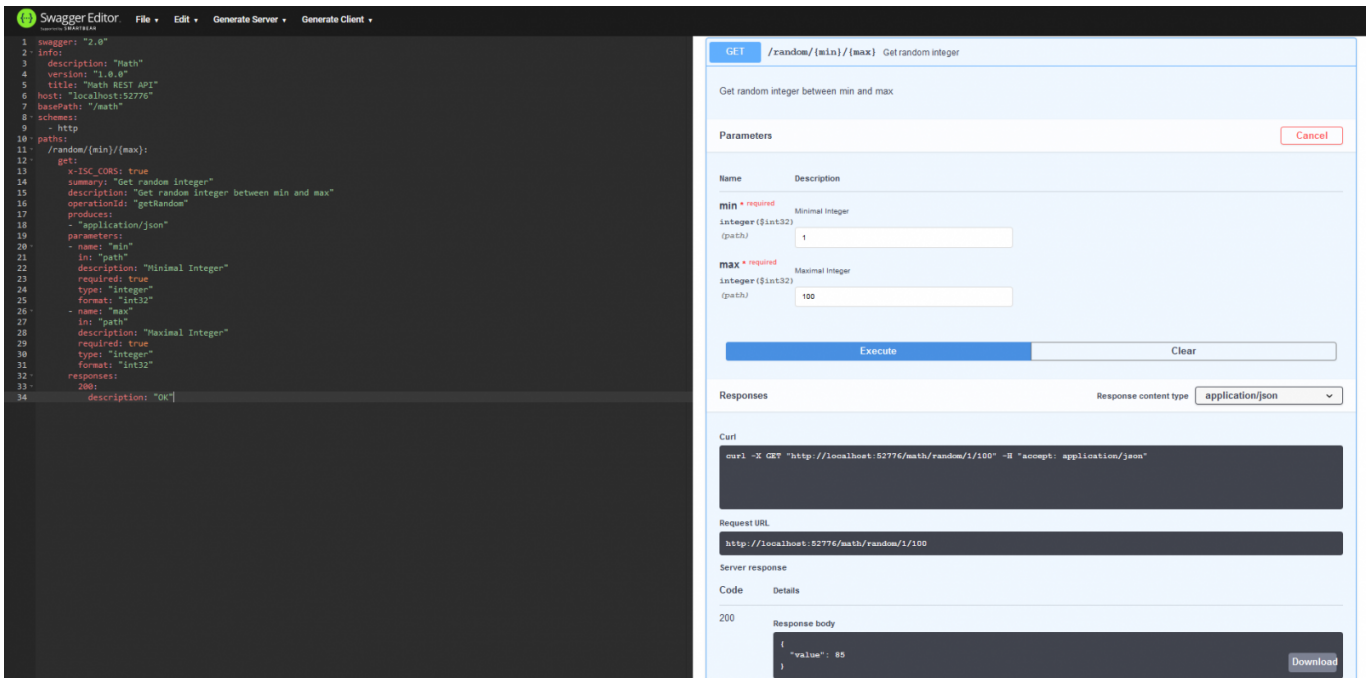
```
ClassMethod getRandom(min As %Integer, max As %Integer) As %DynamicObject
{
    quit {"value":($random(max-min)+min)}
}
```

And finally we can call our REST API by opening this page in browser : http://localhost:52773/math/random/1/100

The output should be:

```
{
    "value": 45
}
```

Also in the Swagger editor pressing Try it out button and filling the request parameters would also send the same request:



Congratulations! Our first REST API created with a spec-first approach is now live!

Further development

Of course, our API is not static and we need to add new paths and so on. With spec-first development, you start with modifying the specification, then updating the REST application (same calls as for creating the application) and finally writing the code. Note that spec updates are safe: your code is not affected, even if the path is removed from a spec, in implementation class the method would not be deleted.

Considerations

More notes!

Special parameters

InterSystems added special parameters to swagger specification, here they are:

Name	Datatype	Default	Place	Description
x-ISCDispatchParent	classname	%CSP.REST	info	Superclass for dispatch class.
x-ISCCORS	boolean	false	operation	Flag to indicate that CORS requests for this endpoint /method combination should be supported.
x-ISCRequiredResource	array		operation	Comma-separated

				list of defined resources and their access modes (resource:mode) that are required for access to this endpoint of the REST service. Example: ["%Development:USE"]
x-ISCServiceMethod	string		operation	Name of the class method called on the back end to service this operation; default is operationId, which is normally suitable.

CORS

There are three ways to enable CORS support.

1. On a route by route basis by specifying x-ISCCORS as true. That's what we have done in our Math REST API.

2. On per API basis by adding

```
Parameter HandleCorsRequest = 1;
```

and recompiling the class. It would also survive spec update.

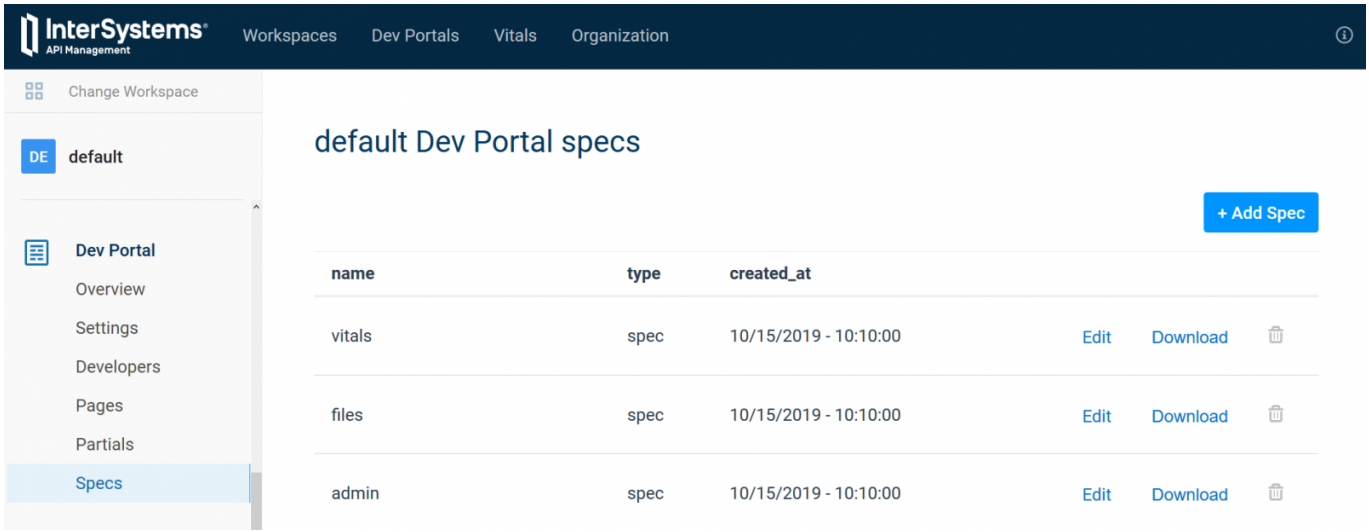
3. (Recommended) On per API basis by implementing custom dispatcher superclass (it should extend %CSP.REST) and writing CORS processing logic there. To use this superclass add x-ISCDispatchParent to your specification.

Load spec into IAM

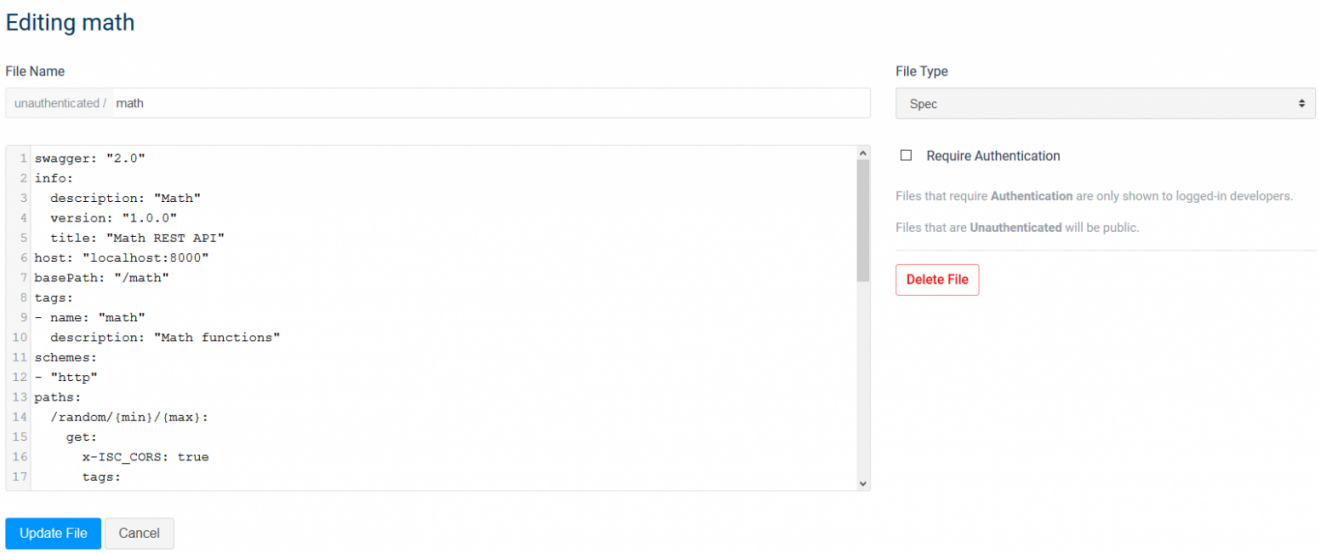
Finally, let's add our spec into IAM so it would be published for other Developers.

If you have not started with IAM, check out this article. It also covers offering REST API via IAM so I'm not describing it here. You might want to modify spec host and basepath parameters so that they point to IAM, rather than the InterSystems IRIS instance.

Open the IAM Administrator portal and go to the Specs tab in the relevant workspace.

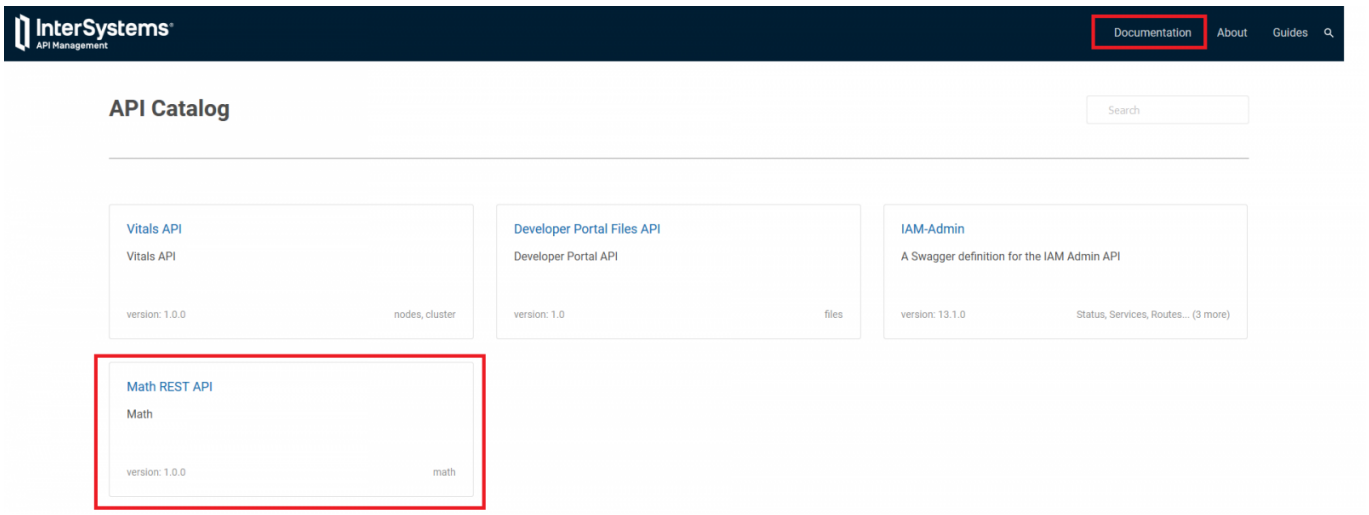


Click the Add Spec button and input the name of the new API (math in our case). After creating new Spec in IAM click Edit and paste the spec code (JSON or YAML - it doesn't matter for IAM):

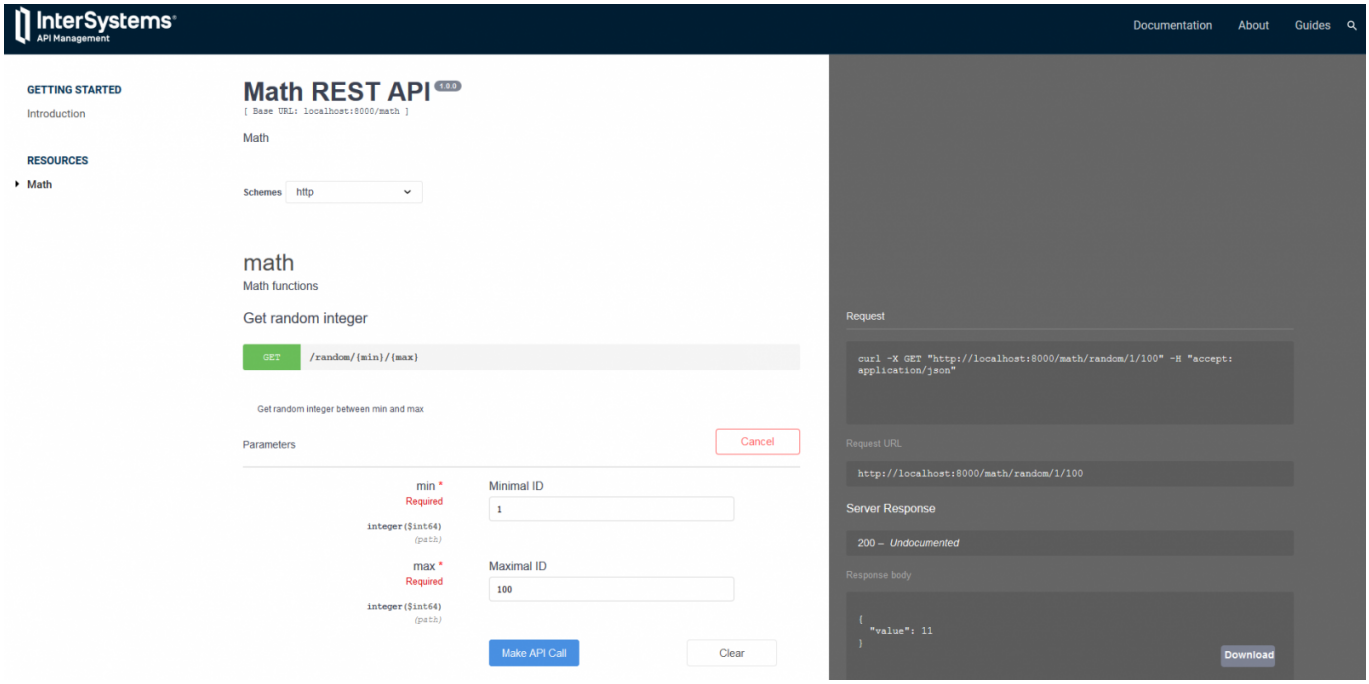


Don't forget to click Update File.

Now our API is published for Developers. Open Developer Portal and click Documentation in the upper right corner. In addition to the three default APIs our new Math REST API should be available:



Open it:



Now Developers can see the documentation for our new API and try it at the same place!

Conclusion

InterSystems IRIS simplifies the development process for a REST API and the spec-first approach allows faster and easier REST API life cycle management. With this approach, you can use a variety of tools for a variety of related tasks, such as client generation, unit testing, API Management, and many others.

Links

- OpenAPI 3.0 Specification

- Creating REST Services
- Starting with IAM
- IAM Documentation

#API #Best Practices #InterSystems API Manager (IAM) #REST API #InterSystems IRIS