Article
Stefan Wittmann · Oct 1, 2019 6m read

JSON Enhancements

InterSystems IRIS 2019.1 has been out for a while and I would like to cover some enhancements for handling JSON which might have gone unnoticed. Dealing with JSON as a serialization format is an important part of building modern applications, especially when you interact with REST endpoints.

Formatting JSON

First of all, it helps if you can format JSON to make it more human-readable. This is very handy when you have to debug your code and look at JSON content of a certain size. Simple structures are easy to scan for a human, but as soon as you have multiple nested elements it can get tricky quite easily. Here is a simple example:

```
{"name":"Gobi","type":"desert","location":{"continent":"Asia","countries":["China","Mongolia"]},"dimensions":{"length":1500,"length_unit":"km","width":800,"width_unit":"km"}}
```

A more human-readable format makes it easy to explore the structure of the content. Let ’ s take a look at the same JSON structure, but this time with proper line breaks and indentation:

```
{
  "name":"Gobi",
  "type":"desert",
  "location":{
    "continent":"Asia",
    "countries":[
      "China",
      "Mongolia"
    ]
  },
  "dimensions":{
    "length":1500,
    "length_unit":"km",
    "width":800,
    "width_unit":"km"
  }
}
```

Even this simple example blows up the output by quite a bit, so you can see why this is not the default in many systems. But with this verbose formatting, you can easily spot substructures and get a sense if something is wrong.

InterSystems IRIS 2019.1 introduced a package by the name %JSON. You can find a couple of useful utilities here, one being a formatter, which allows you to achieve exactly what you have seen above: Format your dynamic objects and arrays and JSON strings into a more human-readable representation. %JSON.Formatter is a class with a very simple interface. All methods are instance methods, so you always start by retrieving an instance.

```
USER>set formatter = ##class(%JSON.Formatter).%New()
```

The reason behind this choice is that you can configure your formatter to use certain characters for the indentation (e.g. whitespaces vs. tabs) and line terminators once and then use it wherever you need it.

The method Format() takes either a dynamic object/array or a JSON string. Let ’ s look at a simple example using a dynamic object:

```
USER>do formatter.Format({"type":"string"})
{
  "type":"string"
}
```

And here is an example with the same JSON content, but this time represented as a JSON string:

```
USER>do formatter.Format("{""type"":""string""}")
{
  "type":"string"
}
```

The Format() method outputs the formatted string to the current device, but you will also see the methods FormatToString() and FormatToStream() in case you want to direct the output to a variable.

Shifting Gears

The above is nice, but may not be worth an article by itself. InterSystems IRIS 2019.1 also introduces a way to conveniently serialize persistent and transient objects from and to JSON. The class you want to take a look at is %JSON.Adaptor. The concept is very similar to %XML.Adaptor, hence the name. Any class you would like to serialize from and to JSON needs to subclass %JSON.Adaptor. The class will inherit a couple of handy methods, most noteworthy are %JSONImport() and %JSONExport(). It ’ s best to demonstrate this with an example. Assume we have the following classes:

```
Class Model.Event Extends (%Persistent, %JSON.Adaptor)
{
 Property Name As %String;
 Property Location As Model.Location;
}
```

and

```
Class Model.Location Extends (%Persistent, %JSON.Adaptor)
{
 Property City As %String;
 Property Country As %String;
}
```

As you can see, we have a persistent event class, which links to a location. Both classes inherit from %JSON.Adaptor. This enables us to populate an object graph and directly export it as a JSON string:

```
USER>set event = ##class(Model.Event).%New()

USER>set event.Name = "Global Summit"

USER>set location = ##class(Model.Location).%New()
USER>set location.City = "Boston"

USER>set location.Country = "United States of America"

USER>set event.Location = location

USER>do event.%JSONExport()
{"Name":"Global Summit","Location":{"City":"Boston","Country":"United States of Ameri
ca"}}
```

Of course, you can also go the other direction with %JSONImport():

```
USER>set jsonEvent = {"Name":"Global Summit","Location":{"City":"Boston","Country":"U
nited States of America"}}

USER>set event = ##class(Model.Event).%New()

USER>do event.%JSONImport(jsonEvent)

USER>write event.Name
Global Summit
USER>write event.Location.City
Boston
```

The import and export methods work for arbitrarily nested structures. Similar to the %XML.Adaptor you can specify the mapping logic for each individual property by setting corresponding parameters. Let ’ s change the Model.Event class to the following definition:

```
Class Model.Event Extends (%Persistent, %JSON.Adaptor)
{
 Property Name As %String(%JSONFIELDNAME = "eventName");
 Property Location As Model.Location(%JSONINCLUDE = "INPUTONLY");
}
```

Assuming we have the same object structure assigned to the variable event as in the example above, a call to %JSONExport() would return the following result:

```
USER>do event.%JSONExport()
{"eventName":"Global Summit"}
```

The property Name is mapped to the field name eventName and the Location property is excluded from the %JSONExport() call, but will be populated when present in the JSON content during a %JSONImport() call. There are various parameters available to allow you to tweak the mapping:

- %JSONFIELDNAME corresponds to the field name in the JSON content.
- %JSONIGNORENULL allows the developer to override the default handling of empty strings for string properties.
- %JSONINCLUDE controls if this property will be included in the JSON output/input.
- If %JSONNULL is true (=1), then unspecified properties are exported as the null value. Otherwise, the field corresponding to the property is just skipped during export.
- %JSONREFERENCE specifies how object references are treated. "OBJECT" is the default and indicates that the properties of the referenced class are used to represent the referenced object. Other options are "ID", "OID" and "GUID".

This provides you with a high level of control and is very handy. Gone are the days of manually mapping your objects to JSON.

One more thing

Instead of setting the mapping parameters on the property level you can also define a JSON mapping in an XData block. The following XData block with the name OnlyLowercaseTopLevel has the same settings as our event class above.

```
Class Model.Event Extends (%Persistent, %JSON.Adaptor)
{
 Property Name As %String;
 Property Location As Model.Location;
 XData OnlyLowercaseTopLevel
 {
 <Mapping xmlns="http://www.intersystems.com/jsonmapping">
     <Property Name="Name" FieldName="eventName"/>
     <Property Name="Location" Include="INPUTONLY"/>
 </Mapping>
 }
}
```

There is one important difference: JSON mappings in XData blocks do not change the default behavior, but you have to reference them in the corresponding %JSONImport() and %JSONExport() calls as the last argument, e.g.:

```
USER>do event.%JSONExport("OnlyLowercaseTopLevel")
{"eventName":"Global Summit"}
```

If there is no XData block with the provided name, the default mapping will be used. With this approach, you can configure multiple mappings and reference the mapping you need for each call individually, granting you even more control while making your mappings more flexible and reusable.

I hope these enhancements make your life easier and I am looking forward to your feedback. Let me know what you think in the comments.

#Best Practices #JSON #Object Data Model #REST API #XML #InterSystems IRIS