Artículo
Alberto Fuentes · 14 oct, 2019 Lectura de 6 min

Mejoras en procesamiento JSON

¡Hola a tod@s!

Me gustaría comentar con vosotros algunas de las mejoras en procesamiento JSON que incorpora IRIS desde la versión 2019.1. Utilizar JSON como formato de serialización es muy común a la hora de construir aplicaciones hoy en día, especialmente si desarrollamos o interactuamos con servicios REST.

Dar formato a cadenas a JSON

Ayuda mucho poder dar un formato fácilmente interpretable por una persona a una cadena JSON. Especialmente cuando queremos depurar código y acabamos teniendo que examinar por ejemplo una respuesta JSON de un tamaño considerable. Al principio, las estructuras simples son fáciles de leer, pero a medida que comenzamos a tener múltiples elementos anidados unos en otros puede complicarse.

Aquí tenemos un ejemplo sencillo:

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

Si la tuviésemos formateada de manera que nos facilitase la lectura, nos permitiría explorar de forma sencilla su contenido. Echemos un vistazo a la misma cadena JSON pero formateada de forma más apropiada:

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

Incluso con este ejemplo tan simple ya vemos como el resultado es un bastante más largo, así que nos hacemos a la idea de por qué en muchas ocasiones este formato no es el que se utiliza por defecto en muchos sistemas. Pero sin embargo, con este formato tan "explicativo" podemos identificar muy fácilmente subestructuras y hacernos una idea rápidamente de si algo no está del todo bien.

En InterSystems IRIS 2019.1 se incluye un paquete con el nombre %JSON donde podemos encontrar un conjunto

de herramientas, entre ellas un formateador, que nos ayudará a conseguir lo que acabamos de ver en el ejemplo anterior. Dar formato a nuestros objetos dinámicos, arrays y cadenas JSON para obtener una representación más fácilmente interpretable por una persona. %JSON.Formatter es una clase con una interfaz muy simple. Todos los métodos son métodos de instancia, así que el primer paso es obtener una instancia de la clase:

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

La razón para desarrollarlo de esta forma es que así podremos configurar nuestra instancia del formateador de manera que incluyamos por ejemplo ciertos caracteres para la indentación (espacios en blanco o tabuladores), caracteres de fin de línea, etc. y después lo podremos reutilizar donde nos haga falta.

El método Format()recibe bien un objeto dinámico / array o una cadena JSON. Echemos un vistazo a un ejemplo sencillo utilizando un objeto dinámico

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

Ahora el mismo ejemplo pero utilizando una cadena JSON:

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

El método Format()devuelve la cadena formateada al dispositivo actual, pero también tenemos disponibles los métodos FormatToString() y FormatToStream()en caso de necesitar el resultado en una variable.

Algo más complicado

Lo que hemos visto hasta ahora está bien, pero no merece un artículo solamente para ello :). InterSystems IRIS 2019.1 incluye también una manera muy sencilla de serializar objetos persistentes y no-persistentes hacia o desde JSON.

La clase que nos interesa para este fin es %JSON.Adaptor. El concepto es muy similar al de %XML.Adaptor, de ahí el nombre. Cualquier clase que quisiéramos serializar hacia o desde JSON necesita heredar de %JSON.Adaptor. La clase heredará entonces unos cuantos métodos, de los cuales cabe reseñar %JSONImport() y %JSONExport(). Para explicarlos, lo mejor es verlo con un ejemplo.

Asumamos como punto de partida que tenemos las siguientes clases:

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

y

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

```
 Property Country As %String;
}
```

Tenemos una clase Event persistente, que se relaciona con una Location. Ambas clases heredan de %JSON.Adaptor. Esto les permite instanciar, rellenar un objeto y exportarlo directamente como una cadena JSON:

```
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"}}
```

Por supuesto, también podemos ir en sentido inverso utilizando %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
```

Los métodos para importar y exportar funcionan para estructuras de datos anidadas arbitrariamente. De forma análoga a %XML.Adaptor puedes especificar la lógica de mapeo para cada propiedad individual a través de su correspondiente parámetro. Vamos a cambiar la definición de la clase Model.Event para ver cómo funciona:

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

Asumiendo que tenemos la misma estructura de objeto asignado a la variable event que en el ejemplo anterior, una llamada a %JSONExport() retornaría lo siguiente:

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

La propiedad Name se mapea al campo eventName y la propiedad Location se excluye de la exportación que hace %JSONExport(), pero sin embargo estará presente cuando se carguen datos a partir de una cadena JSON

cuando se llame a %JSONImport() . Hay además otros parámetros que nos permitirán ajustar el mapeo:

- %JSONFIELDNAME corresponde al nombre del campo en el contenido JSON.
- %JSONIGNORENULL permite sobreescribir el comportamiento por defecto a la hora de manejar cadenas vacías para las propiedades tipo *string*.
- %JSONINCLUDE controla si la propiedad se incluirá o no en la salida / entrada JSON.
- Si %JSONNULL es *true* (=1), entonces las propiedades sin rellenar son exportadas con valor *null*. En otro caso, el campo correspondiente a la propiedad simplemente se ignorará durante la exportación.
- %JSONREFERENCE especifica cómo se tratan las referencias a objetos. "OBJECT" es el valor por defecto e indica que las propiedades de las clases referenciadas se utilizan para representar al objeto referenciado. Otras opciones disponibles son "ID", "OID" y "GUID".

De esta forma tenemos un gran nivel de control de forma muy cómoda. Quedan atrás los días en que manualmente tenías que mapear tus objetos a JSON.

Una cosa más

En lugar de configurar los parámetros de mapeo a nivel de las propiedades, puedes configurar tu mapeo JSON también en un bloque XData. El siguiente bloque XData con el nombre OnlyLowercaseTopLevel tiene la misma configuración que nuestra clase Event anterior.

```
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>
 }
}
```

Sin embargo, aquí hay una diferencia importante: los mapeos JSON en bloques XData no cambian el comportamiento por defecto, y además, para utilizarlos, tienes que referenciarlos en el correspondiente %JSONImport() y %JSONExport() como último argumento, por ejemplo:

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

Si no existe un bloque XData con el nombre que le hemos pasado, utilizará el mapeo por defecto. Con esta aproximación, podemos tener configurados diferentes mapeos y referenciar aquel que nos haga falta en cada caso, permitiendo aún mayor control a la vez que nos permite tener mapeos más flexibles y reutilizables.

#API REST #JSON #Modelo de datos de objetos #XML #InterSystems IRIS

URL de fuente:https://es.community.intersystems.com/post/mejoras-en-procesamiento-json