Artigo
Andre Larsen Barbosa · Jul. 14, 2021 6min de leitura

Aprimoramentos JSON

O InterSystems IRIS 2019.1 já foi lançado há algum tempo e gostaria de abordar algumas melhorias para lidar com JSON que podem ter passado despercebidas. Lidar com JSON como um formato de serialização é uma parte importante da construção de aplicativos modernos, especialmente ao interagir com terminais REST.

Formatando JSON

Em primeiro lugar, ajuda se você pode formatar JSON para torná-lo mais legível. Isso é muito útil quando você precisa depurar seu código e observar o conteúdo JSON de um determinado tamanho. Estruturas simples são fáceis de procurar por um humano, mas assim que você tiver vários elementos aninhados, pode ficar complicado com bastante facilidade. Aqui está um exemplo simples:

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

Um formato mais legível torna mais fácil explorar a estrutura do conteúdo. Vamos dar uma olhada na mesma estrutura JSON, mas desta vez com quebras de linha e recuo adequados:

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

Mesmo este exemplo simples aumenta um pouco a saída, então você pode ver por que este não é o padrão em muitos sistemas. Mas com essa formatação detalhada, você pode identificar subestruturas facilmente e ter uma noção se algo está errado.

InterSystems IRIS 2019.1 introduziu um pacote com o nome %JSON. Você pode encontrar alguns utilitários úteis aqui, sendo um formatador, que permite que você obtenha exatamente o que viu acima: Formate seus objetos dinâmicos e matrizes e strings JSON em uma representação mais legível. %JSON.Formatter é uma classe com uma interface muito simples. Todos os métodos são métodos de instância, então você sempre começa recuperando uma instância.

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

A razão por trás dessa escolha é que você pode configurar seu formatador para usar certos caracteres para o recuo (por exemplo, espaços em branco x tabulações) e terminadores de linha uma vez e usá-los sempre que precisar.

O método **Format()** pega um objeto / array dinâmico ou uma string JSON. Vejamos um exemplo simples usando um objeto dinâmico:

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

E aqui está um exemplo com o mesmo conteúdo JSON, mas desta vez representado como uma string JSON:

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

O método **Format()** envia a string formatada para o dispositivo atual, mas você também verá os métodos **FormatToString()** e **FormatToStream()** caso queira direcionar a saída para uma variável.

Mudando a marcha

O texto acima é bom, mas pode não valer um artigo por si só. O InterSystems IRIS 2019.1 também apresenta uma maneira de serializar objetos persistentes e transitórios de e para JSON. A classe que você deseja examinar é%JSON.Adaptor. O conceito é muito semelhante a% XML.Adaptor, daí o nome. Qualquer classe que você gostaria de serializar de e para JSON precisa ter a subclasse%JSON.Adaptor. A classe herdará alguns métodos úteis, os mais notáveis são %JSONImport() e %JSONExport(). É melhor demonstrar isso com um exemplo. Suponha que temos as seguintes classes:

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

e

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

Como você pode ver, temos uma classe de evento persistente, que se vincula a um local. Ambas as classes herdam de %JSON.Adaptor. Isso nos permite preencher um gráfico de objeto e exportá-lo diretamente como uma string 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"}}
```

Claro, você também pode ir na outra direção com %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
```

Os métodos de importação e exportação funcionam para estruturas aninhadas arbitrariamente. Semelhante ao %XML.Adaptor, você pode especificar a lógica de mapeamento para cada propriedade individual definindo os parâmetros correspondentes. Vamos mudar a classe Model.Event para a seguinte definição:

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

Supondo que temos a mesma estrutura de objeto atribuída ao evento variável como no exemplo acima, uma chamada para %JSONExport () retornaria o seguinte resultado:

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

O **nome** da propriedade é mapeado para o nome do campo **eventName** e a propriedade **Location** é excluída da chamada %JSONExport(), mas será preenchida quando presente no conteúdo JSON durante uma chamada %JSONImport(). Existem vários parâmetros disponíveis para permitir que você ajuste o mapeamento:

- %JSONFIELDNAME corresponde ao nome do campo no conteúdo JSON.
- %JSONIGNORENULL permite que o desenvolvedor substitua o tratamento padrão de strings vazias para propriedades de string.
- %JSONINCLUDE controla se essa propriedade será incluída na saída / entrada JSON.
- If %JSONNULL for verdadeiro (= 1), as propriedades não especificadas serão exportadas como o valor nulo. Caso contrário, o campo correspondente à propriedade é apenas ignorado durante a exportação.
- %JSONREFERENCE especifica como as referências do objeto são tratadas. "OBJECT" é o padrão e

indica que as propriedades da classe referenciada são usadas para representar o objeto referenciado. Outras opções são "ID", "OID" e "GUID".

Isso fornece um alto nível de controle e é muito útil. Já se foi o tempo de mapear manualmente seus objetos para JSON.

Mais uma coisa

Em vez de definir os parâmetros de mapeamento no nível da propriedade, você também pode definir um mapeamento JSON em um bloco XData. O seguinte bloco XData com o nome **OnlyLowercaseTopLevel** tem as mesmas configurações de nossa classe de evento acima.

```
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>
 }
}
```

Há uma diferença importante: os mapeamentos JSON em blocos XData não alteram o comportamento padrão, mas você deve referenciá-los nas chamadas %JSONImport() e %JSONExport() correspondentes como o último argumento, por exemplo:

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

Se não houver bloco XData com o nome fornecido, o mapeamento padrão será usado. Com essa abordagem, você pode configurar vários mapeamentos e fazer referência ao mapeamento necessário para cada chamada individualmente, concedendo ainda mais controle e tornando seus mapeamentos mais flexíveis e reutilizáveis.

Espero que essas melhorias facilitem sua vida e aguardo seu feedback. Deixe-me saber o que você pensa nos comentários.

#JSON #Modelo de Dados Objeto #REST API #XML #InterSystems IRIS