Introducing new JSON capabilities in Caché 2016.1

Caché, Interoperability, JSON, Web Development

This post is intended to guide you through the new JSON capabilities that we introduced in Caché 2016.1. JSON has emerged to a serialization format used in many places. The web started it, but nowadays it is utilized everywhere. We've got plenty to cover, so let's get started.

Our JSON support so far was tightly coupled with the ZEN framework, our own framework for building web applications. Over time, we recognized a rising demand for accessing the JSON functionality outside of ZEN. Though it was possible to be used outside of ZEN, it was confusing and the old JSON support included a couple of shortcomings that required the introduction of a new API. I want to describe the new API in this post and provide you with some background information why we made certain design choices.

The first change you will notice is the introduction of two new classes, `%Object` and `%Array`, both extending`%AbstractObject`. These classes are your primary interface for handling dynamic data, that is, data with a dynamic structure like JSON. Both classes are located in the `%Library` package as they implement general-purpose functionality.

`%Object` allows you to create what we call a dynamic object. You will not implement a subclass, but rather you will instantiate a dynamic object and add properties you require at runtime. You can also remove properties and we provide an iterator interface for introspection and discovery of existing properties. Properties are always unordered and we will not guarantee any order.

```
USER>set object = ##class(%Object).$new()  USER>set object.name = "Ste
fan Wittmann"  USER>set object.lastSeriesSeen = "Daredevil"  USER>set
object.likes = "Galaxy"
```

The above code sample creates a new dynamic object with three properties `name`, `lastSeriesSeen` and `likes`.

`%Array` is an ordered collection of values. You can add new values to an array, you can remove them and iterate over it. I think you get the picture. We call these dynamic arrays. Arrays do support

sparseness, that means you can assign a value to slot 100 while slot 0 to 99 are left unassigned and we only allocate space for one value, instead of 101.

```
USER>set array = ##class(%Array).$new()  USER>do array.$push(1)  USER>
do array.$push("This is a string")  USER>do array.$push(object)
```

The above example creates a dynamic array with three values. The first is a number, the second is a string and the last is the dynamic object from our previous example. We are building a dense dynamic array by pushing new values to the array. A sparse array can be built by setting values at a specific key. But let's save sparseness for later.

Producing JSON

If you want to serialize a dynamic entity to JSON you just call the method `$toJSON()` on it. The method is pretty smart and returns a string if it fits into a single string and a stream object otherwise. If it is used in a DO statement it will send the output to the current device. The output will be assigned to the left-hand variable if it is used in a SET statement.

```
USER>do object.$toJSON()  {"name":"Stefan Wittmann","lastSeriesSeen":
"Daredevil","likes":"Galaxy"}  USER>do array.$toJSON()  [1,"This is a
string.",{"name":"Stefan Wittmann","lastSeriesSeen":"Daredevil","like
s":"Galaxy"}]
```

Consuming JSON

The other direction is pretty simple as well. There are plenty of ways how you can receive a JSON object, but almost always it will end up in a variable, either as a string or a stream, depending on the size. Just call `$fromJSON()` and pass in your JSON string or stream. We will take care of the rest.

```
USER>set someJSONstring = "{""firstname"":""Stefan"",""lastname"":""W
ittmann""}"  USER>set consumedJSON = ##class(%AbstractObject).$fromJS
ON(someJSONstring)  USER>write consumedJSON.$size() 2 USER>write consu
medJSON.$toJSON() {"firstname":"Stefan","lastname":"Wittmann"}
```

The above code example makes use of the method `$size()` to get the number of properties for the `consumedJSON` object.`$size()` is also available for arrays and returns the number of assigned values in the array.

Moving on

Now that you have seen the basics for the new API for dynamic objects and arrays, let's explore some of the more advanced features and topics. As the data structures are defined at runtime it is very important to provide proper tooling for discovering content. The most important utility is an iterator that allows you to loop through the properties of a dynamic object.

```
USER>set iter = object.$getIterator()  USER>while iter.$getNext(.key,.
value) { write "key "_key_":"_value,! } key name:Stefan Wittmann key l
astSeriesSeen:Daredevil key likes:Galaxy
```

A very important aspect while designing the API was consistency. A generic functionality like an iterator should behave the same for objects and arrays.

```
USER>set iter = array.$getIterator()  USER>while iter.$getNext(.key,.v
alue) { write "key "_key_":"_value,! } key 0:1 key 1:This is a string.
key 2:2@%Library.Object
```

The iterator allows us to easily introspect the content of an array and, therefore, we can now discuss sparse arrays. Let's assume we set another value at the positional index 10. Keep in mind that arrays are zero-based.

```
USER>do array.$set(10,"This is a string in a sparse array")  USER>writ
e array.$toJSON() [1,"This is a string.",{"name":"Stefan Wittmann","la
stSeriesSeen":"Daredevil","likes":"Galaxy"},null,null,null,null,null,
null,null,"This is a string in a sparse array"]
```

You can observe that the 11th value is set to the string "`This is a string in a sparse array`" and all slots between index 3 and 9 are serialized with null values. These null values do not exist in memory and they are only serialized as null values because JSON does not support undefined values. You can easily prove this by iterating over the array as the following code snippet.

```
USER>set iter = array.$getIterator()  USER>while iter.$getNext(.key,.v
alue) { write "key "_key_":"_value,! } key 0:1 key 1:This is a string.
key 2:2@%Library.Object key 10:This is a string in a sparse array
```

As you can see only 4 keys are defined and keys 3 to 9 are not set to any value. The code sample demonstrates this concept with an array, but the same is true for dynamic objects. It is very important for various environments to handle sparse data efficiently and you will see some references to this in blog posts I will share later.

Error handling and some sugar

Another important fact is that we are throwing exceptions in the case of an error instead of returning a `%Status` value. Let's see what happens if we try to parse an invalid JSON string.

```
USER>set invalidObject = ##class(%AbstractObject).$fromJSON("{,}")  <
THROW>zfromJSON+24^%Library.AbstractObject.1 *%Exception.General Pars
ing error 3 Line 1 Offset 2
```

You can see that the thrown exception includes enough information to conclude that the second character on the first line is invalid. Therefore, any code that makes use of the new JSON API should be surrounded with a try/catch block at some level. If you think about it, this makes sense as we are dealing with dynamic data and the data may not fit your assumptions.

There are multiple benefits for using exceptions for the report mechanism, but the most important reason is that it allows each method to return a reference to the output, therefore allowing chaining of methods:

```
USER>do array.$push(11).$push(12).$push(13)  USER>write array.$toJSON
() [1,"This is a string.",{"name":"Stefan Wittmann","lastSeriesSeen":
"Daredevil","likes":"Galaxy"},null,null,null,null,null,null,null,"Thi
s is a string in a sparse array",11,12,13]
```

Tight COS integration

All the code samples I provided so far created the dynamic objects and arrays explicitly. We called the constructor of the corresponding class - `%Object` or `%Array` – and started to manipulate the in-memory object.

With the new API there is an even simpler way to create dynamic objects and arrays by implicitly creating them with the JSON syntax:

```
USER>set object = {"name":"Stefan Wittmann","lastMovieSeen":"The Mart
ian","likes":"Writing Blogs"}  USER>write object.$toJSON() {"name":"S
tefan Wittmann","lastMovieSeen":"The Martian","likes":"Writing Blogs
"} USER>set array = [1,2,3,[4,5,6],true,false,null]  USER>write array.
$toJSON() [1,2,3,[4,5,6],true,false,null]
```

Isn't that exciting? It is a very clear, compact and human-readable way of describing what you want to create. You may have realized that the array is initialized with the JSON values `true`, `false` and `null`. These values are not directly accessible in COS, but they can be used within the JSON syntax.

We did not stop there. To really make this useful and dynamic we allow values to be COS expressions. Consider this example:

```
USER>set name = "Stefan"  USER>set subObject = {"nationality":"German
","favoriteColors":["yellow","blue"]}  USER>set object = {"name":nam
e,"details":subObject,"lastUpdate":$ZD($H,3)}  USER>write object.$toJ
SON() {"name":"Stefan","details":{"nationality":"German","favoriteCol
ors":["yellow","blue"]}," lastUpdate ":"2016-01-31"}
```

This allows you to easily produce and alter JSON structures on the server. There is one thing you should consider, though: Accessing values will always produce a COS friendly value. Let's explore an example to understand what this actually means:

```
USER>set array = [1,2,3,[4,5,6],true,false,null]  USER>set iter = arra
y.$getIterator()  USER>while iter.$getNext(.key,.value) { write "key "
_key_":"_value,! }          key 0:1 key 1:2 key 2:3 key 3:5@%Library.Ar
ray key 4:1 key 5:0 key 6:
```

The output until key 4 should be expected. Key 4 returns the COS value `1` for the JSON value `true`. Similar for key 5, which returns the COS value `0` for the JSON value `false`. Probably less obvious is that key 6 is returning an empty string for the JSON value `null`.

The reason for this is that we want to return COS friendly values that can directly be used in COS conditionals. By mapping`true` to `1` and `false` to `0` you can directly test for truthiness and falseness in an if-statement. An empty string is as close as you can get to a JSON `null`.

But how do you distinguish the JSON value pairs `true` and `1`, `false` and `0` and `null` and "" from each other? You do so by checking the type. The method you want to use for this is `$getTypeOf()`.

```
USER>w array.$getTypeOf(5) boolean USER>w array.$getTypeOf(6) null
```

To close the gap you can pass in a type in the setter for dynamic objects and arrays to specify which type the value represents:

```
USER>do array.$set(7,1)  USER>write array.$toJSON() [1,2,3,[4,5,6],tr
ue,false,null,1] USER>do array.$set(7,1,"boolean")  USER>write array.
$toJSON() [1,2,3,[4,5,6],true,false,null,true]
```

First we set the key 7 to the value `1`, which obviously translates to the JSON number `1`. If we want to set the value `true`instead, we can specify the type `"boolean"` to the setter. I leave the exercise with the JSON value `null` for the reader.

Behind the scenes

Congratulations for getting this far. This is pretty much information and you are still reading. Obviously there are more API calls to learn, but I would like to refer you to the documentation as well as the class documentation for this topic. There are obviously more advanced topics we can cover and I would like to cover two if them.

Performance

Performance is a difficult word. It means so many things and you have to be very careful to state what you mean when you make use of it. One of the shortcomings of the old API, which includes the `zenProxyObject` is its non-linear runtime behavior for serializing and deserializing JSON. While consuming smaller JSON content is just fine, consuming a raw JSON content as large as 100MB on disk could take a couple of minutes. One reason for this is that parsing is implemented in COS, which is not the most efficient language for parsing character streams. In addition, the complete object graph had to be constructed in memory.

We took great care to ensure that we addressed this issue with the new API. Parsing is directly implemented in the kernel and we invented a highly optimized in-memory structure to manage dynamic entities. The following table shows the results from a performance test we conducted recently.

(Each company has an average of 20 employees and 20 customers each with several addresses)	Caché 2015.1 zenProxyObject	Caché 2016.1 JSON support	Node.JS 0.12.2 v8: 3.28.73
Load 1000 companies JSON file (10.8MB)	28,000ms	94ms	97ms
Find how many employees called "Robert" (126 employees)	55ms	55ms	2ms
Load 10,000 companies JSON file (108MB)	386,700ms	904ms	892ms
Find how many employees called "Robert" (1346 employees)	554.5ms	567ms	13ms

We tested the `zenProxyObject` on Caché 2015.1, the new JSON support on Caché 2016.1 and as a third-party comparison, we ran the same test against Node.JS. We loaded a 10.8 MB file with 1,000 JSON companies and several embedded objects and you can see that the numbers went down from 28,000ms to 94ms with the new JSON support. Even better the new JSON support is as fast as running the same operation in Node. Scaling up the test makes the improvement even more clear. Consuming a file with 10,000 JSON companies improved from 386,700ms to 904ms, which is on par with Node again. In-memory operations are pretty fast and linear, but as expected Node is a clear winner here, as there is a native support for JSON objects in-memory.

Overall we are pretty happy with the numbers. Let us know about your experience!

System Methods

This is the last topic I want to shed some light on before I conclude. You may have wondered why all method names started with a dollar character, `$new()`,`$set()`, `$pop()`, `$size()` and so on. This is a new category of methods we introduced, called system methods. There are two different types of system methods, similar to standard methods: instance and class methods.

System methods cannot be overridden by standard methods as they live in a separate namespace. If you've worked with the zenProxyObject, you are probably aware that we reserved some property names. The reason is that the `zenProxyObject`defined some internal properties and methods that allowed it manage its state and provide the API. These stepped on the users namespace and had to be reserved.

With the introduction of system methods we got rid of this problem and there are no reserved property names. Your JSON data can be truly dynamic from now on. System methods are always prefixed with a dollar character.

Conclusion

We think the new JSON support is a huge step forward and will help you to build better interfaces faster. Expect to see some benefits for handling huge sets of data as the API gets adopted by other parts of our stack, e.g. DeepSee. If you haven't had a chance to experience the new API yet make sure to give it a try. I hope you are as excited about it as we are, as this is the very foundation for another feature that I will be discussing in a separate blog post soon. Stay tuned.

I am looking forward for your feedback.

Stefan