Updated JScript Capabilities

Technical Article

In just about every implementation of Workflow, scripting tasks are required in order to accomplish things that are not available with native plugins. Workflow provides an embedded version of Windows' Active Scripting engine, which supports multiple scripting languages. Over the years, the primary scripting language of choice has moved from VBScript to JScript, reflecting the massive gain of popularity of JavaScript that has made it the de facto standard in both front-end and back-end implementations.

In just about every implementation of Workflow, scripting tasks are required in order to accomplish things that are not available with native plugins. Workflow provides an embedded version of Windows' Active Scripting engine, which supports multiple scripting languages. Over the years, the primary scripting language of choice has moved from VBScript to JScript, reflecting the massive gain of popularity of JavaScript that has made it the de facto standard in both front-end and back-end implementations.

Moving away from JScript

For 2019.2, we wanted to provide a better scripting experience for users, by allowing them to use the latest and greatest features of JavaScript. This means moving away from JScript and implementing a more modern engine at run time. Over the past few months, our R&D teams have experimented with a number of different engines and tried to figure out which one would provide the best experience when embedded inside Workflow. Among the candidates were Google V8, ChakraCore and NodeJS, any of which would represent a massive improvement over the current JScript engine. We have since eliminated ChakraCore from the list since Microsoft, who lead that project initially, is now moving towards the V8 engine as well.

But unfortunately, we were not able to complete our work in time for the 2019.2 release (we are now targeting 2020.1), so we turned to plan B: how could we improve JScript's current abilities to modernize its functionality, while we wait for the new engine to be embedded? The answer: ***polyfills.***

What's a polyfill anyway?

The word ***polyfill*** is a bit of a misnomer. You can google all about the numerous geeky debates over the proper/improper naming of that feature. But what it does can be explained rather simply: it allows an outdated JavaScript engine to become forward-compatible (within limits) with modern engines. That's due to the very nature of JavaScript, which allows anyone to create new object methods or replace existing ones. Perhaps the best way to explain is through the simplest of examples: the `trim()` method.

If you use JScript regularly, it's highly probable you've cursed it repeatedly when trying to remove white space at the beginning or end of any string. After all, the `String.trim()` method has been available in JavaScript since biblical times, right? But no, not in JScript. So what most of us do is write a `quick trim()` method of our own and then we move on to something else:

```
function trim( myString ){
   return this.replace(/^[\s\uFEFF\xA0]+|[\s\uFEFF\xA0]+$/g, '');
}
Watch.log(trim("   this is a trimmed line   "),2);
// Outputs "this is a trimmed line"
```

That approach works just fine but it doesn't follow the JavaScript spec, in which trim() is a method of the String object. That means if you copy/paste code from anywhere else, it won't work because that code is likely to expect something like `"  this is a trimmed line  ".trim()` to work.

The proper way to implement `trim()` is therefore to use a polyfill: a method that follows the JavaScript spec and provides the same functionality across all engines:

```
if (!String.prototype.trim) {
   String.prototype.trim = function () {
      return this.replace(/^[\s\uFEFF\xA0]+|[\s\uFEFF\xA0]+$/g, '');
   };
}
Watch.log("   this is a trimmed line   ".trim(),2);
// Outputs "this is a trimmed line"
```

This piece of code first checks if the `trim()` method exists for the String object and if it doesn't, it defines it as a native method for all String object and from this point on, the method becomes available for all Strings. If you are not familiar with JavaScript prototypes, now is a good me to start reading about them because they are at the heart of why/how polyfills work. So with that out of the way, let's see how polyfills can make your life with JScript easier.

From ECMASCRIPT-3 to (almost) ECMASCRIPT-8 in a single leap

With a couple of minor exceptions, JScript is ECMASCRIPT-3 compatible. Describing ECMASCRIPT is beyond the scope of this article, but suffice it to say version 3 came out in 1999, which in computer terms means dinosaurs still roamed the earth. Over the last several years, a new edition has been published every year in June, with the latest ECMASCRIPT 2019 revision being the 10th edition. The last really major change, however, occurred with ECMASCRIPT 2015 (6th edition). At that point, the language had evolved significantly from JScript, adding arrow functions, class declarations, typed arrays, collections and a slew of other features.

Luckily, Objectif Lune is not the only organization that has to grapple with an outdated JS engine. Many members of the open-source community have created and maintained libraries of polyfilled methods for each new version of JavaScript. It is one such library that we decided to implement in Workflow, making it leap forward into the 21th scripting century.

With this polyfill library, basic methods like `String.trim()`, or `btoa()/atob()`, or more advanced methods like `Array.forEach()` can now be used natively inside any script. In fact, all methods and objects that are part of the ECMASCRIPT 2017 specification can now be used inside a Workflow scripting task.

Caution, though: this doesn't mean to say the Scripting task is now fully ECMASCRIPT 2017-compliant. Several syntactical elements (i.e. other than methods and objects) are still unavailable because they cannot be polyfilled (for instance: `let, const, class, await, async` or arrow functions). However, given that the vast majority of JavaScript samples found on the web do not necessarily make use of these newer syntactical elements yet, it's much more likely that copy/pasting these samples into a scripting task will work with little to no changes required, making development easier, faster and much less frustrating.

For instance, this 6-line piece of code which makes use of previously unavailable methods like `padStart(), padEnd(), trim(), atob(), btoa()` and `forEach()` can now be used inside the Scripting task:

```
var notEncoded = "SOME KIND OF STRING TO ENCODE".padStart(50,"*").padEnd(100);
var encoded = btoa(notEncoded.trim());

Watch.Log("*"+notEncoded+"*",2);
Watch.Log(encoded,2);
Watch.Log("*"+atob(encoded)+"*",2);

[1,2,3,4,5].forEach(function(item){Watch.Log(item,2)});
```

The equivalent script in previous Workflow versions, using standard JScript, would have required a whopping 150 lines of code!

The Workflow Scripting Task

To ensure backward-compatibility, the Workflow Scripting task still allows you to select the standard version of ***JScript*** as the scripting language. And although it has been renamed from ***JavaScript*** to ***JScript*** to better reflect its ... uhm ... older nature, it is exactly the same as before. However, a new language option labeled ***Enhanced JScript*** is now available, with the polyfill library automatically loaded. The Workflow Preferences also allow you to set ***Enhanced JScript*** as your default scripting language when dropping a new Scripting task onto a process.

Both the ***JScript*** and the ***Enhanced JScript*** options will still remain available in Workflow, even after we implement, in a future version, a new scripting engine which will be named, appropriately, ***JavaScript.***