Article
Bernd Mueller · Jan 30, 2018 13m read

A story from support - how the quest for a raw DEFLATE compression/decompression function leads to a Node callout server by REST

Some time ago I got a WRC case transferred where a customer asks for the availability of a **raw** DEFLATE compression/decompression function built-in Caché.

When we talk about DEFLATE we need to talk about Zlib as well, since Zlib is the de-facto standard free compression/decompression library developed in the mid-90s.

Zlib works on particular DEFLATE compression/decompression algorithm and the idea of encapsulation within a wrapper (gzip, zlib, etc.).
https://en.wikipedia.org/wiki/Zlib

In Caché Object Script (COS) we already have GZIP support available in using /GZIP=1 on file- or tcp-devices or our Streamclasses for use with gzip files.
http://docs.intersystems.com/latest/csp/docbook/DocBook.UI.Page.cls?KEY=GOBJ_propstream_gzip

Our "CSP-Gateway/Web-Gateway" webserver module also make use of Zlib library to GZIP compress/decompress http-data transmissions coming through and from Caché-Server. (CSP, Zen, SOAP, REST, etc.)

But the GZIP format included additional header and trailer wrapped upon the raw DEFLAT-compressed body.

This is not what the customer wants. He have a use case where he **only** needs to be able to create and decompress raw DEFLATE-compressed content.

This is supported by the Zlib library but not exposed through from within Caché API/functions currently.

So what can we do to add it?

"We need to access the Zlib library somehow."

Can we make Zlib available from within Caché by a callout?

"Yes, we can do it."

A Caché callout enables you to invoke executables, operating system commands or functions from other libraries (a DLL on windows, a SO on Unix) written in other languages that supports C/C++ calling conventions.

Caché callout is provided by $ZF functions, see here:
http://docs.intersystems.com/latest/csp/docbook/DocBook.UI.Page.cls?KEY=BGCL

For example, to issue operating systems commands, we can use $ZF(-1) and $ZF(-2) functions.

While $ZF(-1, command) executes a program or operating system command as a spawned child process and suspends execution of the current process while waiting for the child process to return it's exit status, $ZF(-2, command) works asynchronously, that means it does not await completion of the spawned child process and therefore cannot receive status information from that process directly.

Another alternative is to use **command pipes** to communicate with processes, just as on the operating system level. Here you can send output through the pipe to control the process and read the pipe to receive input, to fetch the process output.
http://docs.intersystems.com/latest/csp/docbook/DocBook.UI.Page.cls?KEY=GIOD_ipc_pipes

PLEASE NOTE: In the future we plan to deprecate and replace current Caché callout mechanism's by $ZF and pipe to a more secure way to callout. So stay tuned.

Since i am a "Web guy" i prefer to use JavaScript. But we need something to execute it on the **Server** instead of client-execution of JavaScript inside our favorite webbrowser in context of a web-page, we all probably know well.

A very popular and commonly used javascript server runtime environment/engine is **Node.js**.
It is a community driven javascript runtime environment built on chrome's V8 javascript engine. Node.js uses an event-driven, non-blocking asynchronous I/O model that makes it lightweight and very efficient.
https://nodejs.org/en/

The good news is, Node.js comes with a zlib module included which plays great with our plan.
https://nodejs.org/api/zlib.html

Caché also supports Node.js in a slightly different way. It comes with a powerful **cache.node** connector/interface to make Data and Methods inside Caché easily available from within Node.js.
http://docs.intersystems.com/latest/csp/docbook/DocBook.UI.Page.cls?KEY=BXJS

In our particular use case/requirement this is **not** what we are looking for.

We need to run pure javascript through Node.js and return the results back to Caché.
So this is the other way around.

The command pipe callout mechanism we've mentioned earlier seems the right way to go.

After downloading and installing Node.js let's try if that plan could work.

```
USER>set cmd="node -v",oldIO=$IO open cmd:"QR" use cmd read result close prog use old
IO
USER>w result
v8.9.1
```

As you can see in that test, it is working as expected. The "node -v" command gives us back the version information on the currently installed Node.js runtime environment.

"Great!"

Let's now start to code a node script to compress/decompress a files content using zlib module and raw DEFLATE/INFLATE algorithms from given command-line-arguments.

This is easy. Create a *zlib.js* in your project folder with the following lines of code:

```
//zlib.js
Const
  func = process.argv[2],
  infile = process.argv[3],
  outfile = process.argv[4];
```

```
const zlib = require('zlib');
const fs = require('fs');

if (func=='DEFLATERAW') {
  var wobj = zlib.createDeflateRaw();
}
else  {
  var wobj = zlib.createInflateRaw();
}

const instream = fs.createReadStream(infile);
const outstream = fs.createWriteStream(outfile);

instream.pipe(wobj).pipe(outstream);

console.log(func + ' ' + infile + ' -> ' + outfile);
```

You can run it from OS console with a command like this to compress an already existing file input.txt into output.zz using raw DEFLATE:

```
C:\projects\zlib>node zlib.js DEFLATERAW input.txt output.zz
DEFLATERAW input.txt -> output.zz
```

Please note: for my convenience my code only supports compress/decompress of files located in the folder where the node script is running, e.g. c: \ projects \ zlib. So please make sure to create or copy at least a input.txt file into this location.

At first, the script-code brings the "zlib" (node Zlib library) and "fs" (node File-System for file access/operations) modules in place in order to use their functionality.

The code then uses *process.argv* to access the incoming command-line arguments.
argv stands for "argument vector", it's an array containing "node" and the full path to the script-file as it is first two elements. The third element (that is, at index 2) is the "function/algorithm name", the fourth and fifth elements (that is, at index 3 and 4) will be the input-file "infile" and output-file "outfile" arguments.

Finally, we are using appropriate zlib methods on both the input and output file streams using pipe processing.

To return back the result of the function, we simply print out a result message trough the console.

"That's it".

Let's try if it works from within Caché.

```
USER>set cmd="node c:\projects\zlib\zlib.js DEFLATERAW input.txt output.zz",oldIO=$IO
 open cmd:"QR" use cmd read result close cmd use oldIO
USER>w result
DEFLATERAW input.txt -> output.zz
```

"Yes, it works as expected".

With the following command you can try to decompress (inflate) the previous compressed file output.zz into output.txt.

```
USER>Set cmd="node c:\projects\zlib\zlib.js INFLATERAW output.zz output.txt",...
```

After that, the output.txt file-content and file-size should result in exactly the same as the input.txt file.

"Issue solved."

We've made raw DEFLATE compression/decompression of files available in Caché by callout to a node script by a command pipe.

But let's consider, in terms of performance, the callout mechanism comes with the overhead of starting a new child process for every callout.

If performance does not matter or if the processing work to be done is time intensive, compressing/decompressing is, if file-size increases, it might be ok and the time overhead to start the process could be ignored. But for compressing/decompressing many and relatively small files one after the other and in masses, this overhead is better to be avoided.

So how could we make this happen?

We need to avoid the creation of a new child process every time the callout is made.

"How can we achieve this?"

We need our script to run as a **server**, listening for incoming orders to proceed with the desired operation as requested.

Sounds plausible and familiar, yes, this is what nowadays a **RESTful HTTP API/service** can offer and is designated for.

With Node.js it is very easy to write a simple server, based on HTTP protocol.

Node.js comes with support for low-overhead HTTP servers out of the box using the built-in "http" module.

To include the "http" module, use node's require() method as usual and as shown here in *simple_https.js* script file:

```
//simple_https.js
const
  http = require('http'),
  server = http.createServer(function (request, response) {
    response.writeHead(200, {'Content-Type' : 'text/plain'});
    response.end('Hello World!\n');
  });
server.listen(3000, function(){
    console.log('ready captain!');
});
```

Use the following command to start our simple http-server from the OS console:

```
C:\projects\zlib>node simple_http.js
ready captain!
```

I am using **"curl"** now to test it. curl is a common and useful command-line tool for issuing HTTP requests to a given server.
https://curl.haxx.se/

Adding "-i" flag tells curl that it should output HTTP headers in addition to the response body.

```
C:\curl>curl -i http://localhost:3000
HTTP/1.1 200 OK
Content-Type: text/plain
Date: Mon, 22 Jan 2018 13:07:06 GMT
Connection: keep-alive
Transfer-Encoding: chunked

Hello World!
```

So, this works like a charm but writing http services against the low-level "http" module directly can be a pain and a lot of work.

Since Node.js has a thriving open source community which has produced many excellent modules to add additional capabilities to Node.js applications, we'll use **"Express"** for developing our servers RESTful API.

"Express.js" or simply **"Express"** is a web application framework for Node.js designed for building web-apps and API's.

It provides a lot of plumbing code that you'd otherwise end up writing yourself. It can route incoming requests based on URL paths, parsing incoming data and rejecting malformed requests, etc.

The Express framework helps with these and a myriad other tasks. It is in fact the standard server framework for Node.js.
http://expressjs.com/

As for all Node modules, to use "Express" you have to first install it with npm (node package manager) before you can use it.

```
C:\projects\zlib>node install express
...
```

To include the "express" and other needed modules, use node's require() method as usual and as shown in the *zlibserver.js* script file:

```
//zslibserver.js
const express = require('express');
const zlib = require('zlib');
const fs = require('fs');

var app = express();

 app.get('/zlibapi/:func/:infile/:outfile', function(req, res) {
    res.type('application/json');

    var infile=req.params.infile;
    var outfile=req.params.outfile;

    try {

        var stats = fs.statSync(infile);
        var infileSize = stats.size;

        switch(req.params.func) {
          case "DEFLATERAW":
```

```
            var wobj = zlib.createDeflateRaw();
            break;
          case "INFLATERAW":
            var wobj = zlib.createInflateRaw();
            break;
          case "DEFLATE":
            var wobj = zlib.createDeflate();
            break;
          case "INFLATE":
            var wobj = zlib.createInflate();
            break;
          case "GZIP":
            var wobj=zlib.createGzip();
            break;
          case "GUNZIP":
            var wobj=zlib.createGunzip();
            break;
          default:
            res.status(500).json({ "error" : "bad function" });
            return;
        }

        const instream = fs.createReadStream(infile);
        const outstream = fs.createWriteStream(outfile);

        var d = new Date();
        console.log(d.toLocaleDateString() + ' ' + d.toLocaleTimeString() + ' : ' + r
eq.params.func + ' ' + infile + ' -> ' + outfile + '...');

        instream.pipe(wobj).pipe(outstream).on('finish', function(){

            var d = new Date();
            console.log(d.toLocaleDateString() + ' ' + d.toLocaleTimeString() + ' : '
 + 'finished!');

            var stats = fs.statSync(outfile);
            var outfileSize = stats.size

            res.status(200).json( { "result" : "OK" , "infileSize" : infileSize, "out
fileSize" : outfileSize, "ratio" : (outfileSize / infileSize * 100).toFixed(2) + "%"
} );
            return;
      });

    }
    catch(err) {
      res.status(500).json({ "error" : err.message});
      return;
    }

});
app.listen(3000, function(){
    console.log("zlibserver is ready captain.");
});
```

First, it brings in the "zlib", "fs" and "express" modules and creates an express "app"lication context.

Express functionality is provided through a "middleware", which are asynchronous functions that can manipulate request and response objects and do processing.

With app.get() we tell Express how we want to **handle** HTTP GET requests to the *route /zlibapi/:func/:infile/:outfile* path. With app.get() you can register multiple handlers for your routes/paths. The *:variable* chunk in the path is called a "named route parameter".
When the API is hit, express grabs that part of the URL and make it available in req.params.

In addition to RAWDEFLATE/RAWINFLATE the code added support for other zlib supported compressing/decompression wrapper formats GZIP/GUNZIP, DEFLATE/INFLATE as well.

I also added a basic Try/Catch error handling as a starting point.

To send back a JSON object with the result we use the response object res and res.status() which is equivalent to res.sendStatus().
See the Express documentation for more details.

Finally, we start listen on TCP port 3000 for incoming HTTP requests.

Let's run our "zlibserver" app to see if it works:

```
C:\projects\zlib>node zlibserver.js
zlibserver is ready captain.
```

Now that it runs, we can try to use it as a service.

I will try it from within Caché, but you could use "curl" or any other 3rd-party tool like "Postman", etc. to test our "zlibserver" RESTful API as well.

We need to use *%Net.HttpRequest* to implement a simple REST client in Caché COS for doing the GET request, which is not much effort but needs some lines of coding. See here my class *utils.Http:getJSON()* method:

```
Include %occErrors
Class utils.Http [ Abstract ]
{
  ClassMethod getJSON(server As %String = "localhost", port As %String = "3000", url
As %String = "",
    user As %String = "", pwd As %String = "", test As %Boolean = 0) As %DynamicAbstr
actObject
  {
     set prevSLang=##class(%Library.MessageDictionary).SetSessionLanguage("en")

     set httprequest=##class(%Net.HttpRequest).%New()
     set httprequest.Server=server
     set httprequest.Port=port

     if user'="" do httprequest.SetParam("CacheUserName",user)
     if pwd'="" do httprequest.SetParam("CachePassword",pwd)

     set sc=httprequest.SetHeader("Accept","application/json")
     if $$$ISERR(sc) $$$ThrowStatus(sc)
     set sc=httprequest.SetHeader("ContentType","application/json")
     if $$$ISERR(sc) $$$ThrowStatus(sc)

     try {
         set sc=httprequest.Get(url,test)
         if $$$ISERR(sc) $$$ThrowStatus(sc)
         if (httprequest.HttpResponse.StatusCode \ 100) = 2 {
             set response = ##class(%DynamicAbstractObject).%FromJSON(httprequest.Htt
```

```
pResponse.Data)
         }
         else {
             Throw ##class(%Exception.General).%New(httprequest.HttpResponse.ReasonPh
rase, $$$GeneralError,,httprequest.HttpResponse.StatusLine)
         }
     }
     catch exception  {
         set response = $$$NULLOREF
         throw exception
     }
     Quit response
  }
}
```

You can use it from within Caché the following way:

```
USER>try { set res="",res = ##class(utils.Http).getJSON(,,"/zlibapi/DEFLATERAW/input.
txt/output.zz"),result=res.result } catch (exc) { Set result=$system.Status.GetOneErr
orText(exc.AsStatus()) }
USER>w result
OK
USER>w res.%ToJSON()
{"result":"OK","infileSize":241243,"outfileSize":14651,"ratio":"6.07%"}
```

"Great!, it works"

Here is the curl way of testing the api: (using existent test.log file)

```
C:\curl>curl -i http://localhost:3000/zlibapi/GZIP/test.log/test.gz

HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 76
ETag: W/"4c-iaOk5W3g6IlIEkzJaRbf3EmxrKs"
Date: Fri, 26 Jan 2018 07:43:17 GMT
Connection: keep-alive

{"result":"OK","infileSize":36771660,"outfileSize":8951176,"ratio":"24.34%"}

C:\curl>curl -i http://localhost:3000/zlibapi/GUNZIP/test.gz/test.txt

HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 77
ETag: W/"4d-tGgowYnW3G9ctHKcpvWmnMgnUHM"
Date: Fri, 26 Jan 2018 07:43:36 GMT
Connection: keep-alive

{"result":"OK","infileSize":8951176,"outfileSize":36771660,"ratio":"410.80%"}

C:\curl>curl -i http://localhost:3000/zlibapi/DEFLATERAW/test.log/test.zz
```

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 76
ETag: W/"4c-4svUs7nFvjwm/JjYrPrSSwhDklU"
Date: Fri, 26 Jan 2018 07:44:26 GMT
Connection: keep-alive

{"result":"OK","infileSize":36771660,"outfileSize":8951158,"ratio":"24.34%"}

C:\curl>curl -i http://localhost:3000/zlibapi/INFLATERAW/test.zz/test.txt

HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 77
ETag: W/"4d-7s7jwh1nxCU+6Qi7nX2TB3Q1IzA"
Date: Fri, 26 Jan 2018 07:44:42 GMT
Connection: keep-alive

{"result":"OK","infileSize":8951158,"outfileSize":36771660,"ratio":"410.80%"}
```

Here you can see the zlibserver's console output while running and receiving/processing incoming jobs:

```
C:\projects\zlib>node zlibserver
zlibserver is ready captain.
2018-1-26 08:43:14 : GZIP test.log -> test.gz...
2018-1-26 08:43:17 : finished!
2018-1-26 08:43:36 : GUNZIP test.gz -> test.txt...
2018-1-26 08:43:36 : finished!
2018-1-26 08:44:23 : DEFLATERAW test.log -> test.zz...
2018-1-26 08:44:26 : finished!
2018-1-26 08:44:42 : INFLATERAW test.zz -> test.txt...
2018-1-26 08:44:42 : finished!
```

To recap and summarize the story and what we have achieved:

We've learned how easy you can enhance Caché by a Node.js callout using REST.

If you "blind out" our initial and specific use-case in general we started with in the first place, and think forward in terms of the wide and great Node.js ecosystem where there are hundreds of awesome node modules available out there, offering and providing you a wide range of functionality and possibilities by API's, you can now easily access/control them from within Caché with a charming solution.

See this link for a list of popular Node.js modules/API's to give you a taste:
http://www.creativebloq.com/features/20-nodejs-modules-you-need-to-know

"Story from support CLOSED!" :)

I hope you've find it interesting and worthwhile,

Bernd

#Best Practices #Callout #JavaScript #Node.js #ObjectScript #REST API #Caché

Source URL: https://community.intersystems.com/post/story-support-how-quest-raw-deflate-compressiondecompression-function-leads-node-callout-server