Technical White Paper

VMware Cloud Foundation Automation

Extension of the JavaScript Runtime Environment to use Java Archive Files

Stefan Schnell

Table of Contents

Disclaimer

This document is provided without any warranty of any kind. No guarantee for the actuality, correctness, completeness or quality of the available information. Liability claims, which refer to damage by the use or not-use of information presented here respectively by the use of incorrect and incomplete information are in principle impossible. Any liability claims are declined. For damages no liability and no responsibility are assumed. Everyone is responsible for himself. This document is subject to change and may be changed at any time for any reason without notice.

Document History

Version	Name	Date	Description
1.00	Stefan Schnell	03.10.2023	Initial document creation
1.10	Stefan Schnell	05.10.2023	Add Write Base64 Encoded String
1.20	Stefan Schnell	29.10.2023	Add Store Binary File as an Action
1.30	Stefan Schnell	11.01.2024	Add Limitation of the Command Class and resulting changes in the entire document
1.40	Stefan Schnell	14.02.2024	Add Using JShell to use JAR, update of About the Author and minor corrections
2.00	Stefan Schnell	19.06.2025	Delete all chapters containing Java compilation, update of About the Author, add Using Scope to use JAR.

About the Author

Name: Stefan Schnell

Employer: BWI GmbH in Meckenheim, Germany,
IT system house of the German Federal Armed Forces

Occupation: Senior IT-Architect in Service Architecture Department

Vocation: The building of integration scenarios in VCF Automation, with different programming languages, interfaces and platforms.

Meet me at… my VCF Automation Blog
LinkedIn
my Private Site

Abstract

Many information is stored in different files with different formats. Many files are not in a text format, these are called binary files. This document describes in a first step different ways to store any kind of files, in the context of VMware Cloud Foundation (VCF) Automation JavaScript runtime environment (RTE), and to use them in a following processing sequence. This is to create an equivalent to the package capabilities of the other RTEs. In a second step it describes how to use Java archive (JAR) files, to be able to use the possibilities of the Java programming language to the full extent.

Target Audience

The target audience of this document are experienced VCF Automation JavaScript developers, with knowledge of the Java API specification, who wants to use binary files and / or Java archive (JAR) files in the context of their development.

Software Requirements

The necessary software requirement is VMware Aria Automation 8.* or VCF Automation 9.*. The system property `com.vmware.scripting.javascript.allow-native-object` must be set to true in the control center of Aria Automation or in the system settings of VCF Automation. Beside a local installed Java Development Kit (JDK) 17 or higher is required. Here it is recommended to use the same one with the corresponding version that is used in VCF Automation, the BellSoft Liberica JDK[1]. Additionally, the Mozilla Rhino JavaScript engine[2] is also required. Either release 1.7R4, which is included in Aria Automation 8.*, or 1.7.15, which is included in VCF Automation 9.*. To build a JAR file it is recommended to use exactly the same version of the JDK as used in VCF Automation, because there could be a version dependency when executing a JAR file.

Important Hint

Do not experiment in production systems. Develop and test only in the designated development systems.

Store Binary Files in the Context of JavaScript RTE

One of the basic approaches of this consideration is the fact, that there is no possibility for the programmer of accessing the VCF Automation file system. There is a strict separation of roles between the programmer and the administrator of the system. So, the only way to bring in files, for use in the JavaScript RTE context, is through VCF Automation itself. This is not a matter of finding a bypass. It is only intended to show the existing possibilities.

The first step is to look at how binary files can be persisted in VCF Automation. We will go three ways here. The first is to embed the binary file in the code as a byte array or as JSON. The second is to save the binary file in a Node.js package. And the third is to store the binary file as a base64 encoded string in an action.

Embed the Binary Files

Embedding binary files can easily be done by using an array. This means that the file to be embedded is read byte by byte and each of these bytes is stored in an array, number by number. With this procedure we create a copy of the file, which is represented in the array. If needed, this array can easily be written back to a file. That is a simple and compact approach. Everything is available in one source code. However, this approach has a disadvantage, the memory requirements of the class are very large. The file can also be stored as base64 encoded JSON in the source code, here no size limitation occurs.

Create Byte Array

Here is a function to convert a file into an array of bytes. It reads the selected file, creates an array of bytes and writes it to a file of the same name, with the extension txt, in the temporary file directory.

```
1  load("System.class.js");
2  load("File.class.js");
3
4  function fileToArrayOfBytes(fileName) {
5
6    try {
7
8      if (System.isWindows()) {
9        fileName = fileName.replace(/\\/g, "/");
10     }
11
12     var path = java.nio.file.FileSystems.getDefault().getPath("", fileName);
13     if (!java.nio.file.Files.exists(path)) {
14       throw new Error("Error: File does not exists");
15     }
16
17     var arrayOfBytes = java.nio.file.Files.readAllBytes(path);
18     if (arrayOfBytes.length > 8192) {
19       var keepGoing = javax.swing.JOptionPane.showConfirmDialog(
20         null,
21         "File size > 8192 Bytes, continue?",
22         "Important Hint",
23         javax.swing.JOptionPane.YES_NO_OPTION,
24         javax.swing.JOptionPane.WARNING_MESSAGE
25       );
26       if (keepGoing === 1) {
27         return;
28       }
29     }
30
31     var arrayAsString = fileName + " = [";
```

```
32      for (var i = 0; i < arrayOfBytes.length; i++) {
33        if (i < arrayOfBytes.length - 1) {
34          arrayAsString += arrayOfBytes[i].toString() + ",";
35        } else {
36          arrayAsString += arrayOfBytes[i].toString();
37        }
38      }
39      arrayAsString += "]";
40
41      var fileWriter = new FileWriter(
42        System.getTempDirectory() + "/" + java.io.File(fileName).getName() + ".txt"
43      );
44      fileWriter.open();
45      fileWriter.write(arrayAsString);
46      fileWriter.close();
47
48    } catch (exception) {
49      System.log(exception.message);
50    }
51
52  }
53
54  var fileChooser = new javax.swing.JFileChooser();
55  fileChooser.setCurrentDirectory(java.io.File("").getCanonicalFile());
56  var result = fileChooser.showOpenDialog(null);
57  if (result === javax.swing.JFileChooser.APPROVE_OPTION) {
58    var selectedFile = fileChooser.getSelectedFile();
59    fileToArrayOfBytes(String(selectedFile));
60  }
```

In lines 18 to 30 we see the check of the file size. If this exceeds a limit of 8k a message is displayed. Usually, with a larger array, the maximum Java class size of 64k is exceeded and an error occurs during compilation. The process flow is clearly visible: Select file, check if file exists, read file and perform size check, convert file to byte array and write byte array to file.

Hint: This program is executed in the local development environment. To use equivalent commands as in the VCF Automation environment, JavaScript source codes are loaded, in lines 2 and 3, as mockups[3]. If this program is executed in an VCF Automation environment, this is not necessary.

Write Byte Array

After the byte array is created, it can be used to write files in VCF Automation. Here is a function to write a file from a byte array. The created byte array can simply be copied here.

Hint: The name of the function is chosen for reasons of understanding. The function and file are combined here, a less general function name would certainly be more meaningful.

```
1   load("System.class.js");
2
3   function writeBinaryFile() {
4
5     // Name:   tinyImage.png
6     // Size:   93 Bytes
7     // SHA256: fceae91067a062c77a99b3b5c027cc5007da555709eca7f213a7e912e395ec05
8     var fileContent = [
9       -119,  80,  78,  71,  13,  10,  26,  10,   0,   0,   0,  13,  73,  72,  68,
10        82,   0,   0,   0,   3,   0,   0,   0,   3,   8,   0,   0,   0,   0, 115,
11        67, -22,  99,   0,   0,   0,   9, 112,  72,  89, 115,   0,   0,   5,-119,
12         0,   0,   5,-119,   1, 109, 104, -99,  -6,   0,   0,   0,  15,  73,  68,
13        65,  84, 120,-100,  99,  -7, -49, -64, -64, -62,   0, -63,   0,  11,  97,
14         1,  12, -52, -75,   0,  77,   0,   0,   0,   0,  73,  69,  78,  68, -82,
15        66,  96,-126
```

```
16    ];
17
18    var file = System.appendToPath(System.getTempDirectory(), "tinyImage.png");
19
20    try {
21
22      var contextFactory = org.mozilla.javascript.ContextFactory();
23      var context = contextFactory.getGlobal().enterContext();
24      var scope = context.initStandardObjects();
25      var byteArray = context.jsToJava(fileContent, java.lang.Class.forName("[B"));
26
27      var path = java.nio.file.Paths.get(file);
28      java.nio.file.Files.deleteIfExists(path);
29      java.nio.file.Files.createFile(path);
30      var outputStream = java.nio.file.Files.newOutputStream(path);
31      outputStream.write(byteArray);
32      outputStream.close();
33
34    } catch (exception) {
35      System.log(exception)
36    } finally {
37      context.exit();
38    }
39
40  }
41
42  writeBinaryFile();
```

Hint: If this program is executed in the local development environment, it is necessary to load a mockup[3] in line 2. If this program is executed in an VCF Automation environment, this is not necessary.

The process flow is the only one step: Writing the byte array to a file. After this action is ran in VCF Automation, the file is available in the temporary file directory.

```
aria-auto.corp.vmbeans.com - PuTTY
root@aria-auto [ /data/vco/usr/lib/vco/app-server/temp ]# ls -l
-rw-r----- 1 root root 93 Sep 27 18:32 tinyImage.png
root@aria-auto [ /data/vco/usr/lib/vco/app-server/temp ]#
```

Create Base64 Encoded File

Here is a function to convert a file into base64. It reads the selected file, encodes the content to base64 and writes it to a file of the same name, with the extension base64, in the temporary file directory.

```
1   load("System.class.js");
2   load("File.class.js");
3
4   function fileToBase64(fileName) {
5
6     try {
7
8       if (System.isWindows()) {
9         fileName = fileName.replace(/\\/g, "/");
10      }
11
12      var path = java.nio.file.FileSystems.getDefault().getPath("", fileName);
13
14      if (!java.nio.file.Files.exists(path)) {
15        throw new Error("Error: File does not exists");
16      }
17      var arrayOfBytes = java.nio.file.Files.readAllBytes(path);
18
```

```
    var base64Encoded = java.util.Base64.getEncoder().encodeToString(arrayOfBytes);

    var fileWriter = new FileWriter(
      System.getTempDirectory() + "/" + java.io.File(fileName).getName() + ".base64"
    );
    fileWriter.open();
    fileWriter.write(base64Encoded);
    fileWriter.close();

  } catch (exception) {
    System.log(exception.message);
  }

}

var fileChooser = new javax.swing.JFileChooser();
fileChooser.setCurrentDirectory(java.io.File("").getCanonicalFile());
var result = fileChooser.showOpenDialog(null);
if (result === javax.swing.JFileChooser.APPROVE_OPTION) {
  var selectedFile = fileChooser.getSelectedFile();
  fileToBase64(String(selectedFile));
}
```

The basically approach here is exactly the same as for create byte array, only with the difference that here base64 encoding is used.

Write Base64 Encoded JSON

After the base64 encoded string is created and copied to JSON, it can be written to a file as the byte array. Here is a function to write a file from a base64 encoded JSON.

```
load("System.class.js");

var binaryFile = '{' +
'"BinaryFile": {'+
'"E_FILENAME": "test256k.txt",' +
'"E_MIMETYPE": "application/octet-stream",' +
'"E_MD5HASH": "df9b9b561313bbe9608cd10a16bcf9e3",' +
'"E_SHA256HASH": "3898772616cb9b346d79f9a280195db71f27e2d6e95b35826ad13d6bba7b125a",' +
'"E_DATA": "MDEyMzQ1Njc4OUFCQ0RFRjAxMjM0NTY3ODlBQkNERUYwMTIzNDU2Nzg5QUJDREVG...MDEyMzQ1Njc4OUFCQ0RFRg=="}}';

function writeBinaryFile() {

  var objJSON = JSON.parse(binaryFile);

  try {

    var contextFactory = org.mozilla.javascript.ContextFactory();
    var context = contextFactory.getGlobal().enterContext();
    var byteArray = context.jsToJava(
      java.util.Base64.getDecoder().decode(java.lang.String(objJSON.BinaryFile.E_DATA).getBytes()),
      java.lang.Class.forName("[B")
    );

    var fileName = System.appendToPath(System.getTempDir(), "/test256k.txt");
    if (System.isWindows()) {
      fileName = fileName.replace(/\\/g, "/");
    }

    var path = java.nio.file.Paths.get(fileName);
    java.nio.file.Files.deleteIfExists(path);
    java.nio.file.Files.createFile(path);
    var outputStream = java.nio.file.Files.newOutputStream(path);
    outputStream.write(byteArray);
    outputStream.close();

```

```
36    } catch (exception) {
37      System.log(exception);
38    } finally {
39      context.exit();
40    }
41
42  }
43
44  writeBinaryFile();
```

After this action is ran in VCF Automation, the file is available in the temporary file directory.

```
aria-auto.corp.vmbeans.com - PuTTY
root@aria-auto [ /data/vco/usr/lib/vco/app-server/temp ]# ls -l
-rw-r----- 1 root root 262144 Oct  3 16:01 test256k.txt
root@aria-auto [ /data/vco/usr/lib/vco/app-server/temp ]#
```

As can be seen from this example, a file of 256 kByte length was used here and it worked without any problems.

Write Base64 Encoded String

Here is another function to write a file from a base64 encoded string. In this case only VCF Automation classes are used. With the class ByteBuffer a byte array can be created from a base64 encoded string. With the method write, of the class MimeAttachment, this byte array can be written to a file. The byte buffer must be assigned to the buffer attribute.

```
1   load("System.class.js");
2   load("File.class.js");
3   load("ByteBuffer.class.js");
4   load("MimeAttachment.class.js");
5
6   var thisIsATestBase64 = "VGhpcyBpcyBhIFRlc3Q";
7
8   try {
9
10    var byteBuffer = new ByteBuffer(thisIsATestBase64);
11
12    // Delete existing file
13    var file = new File(System.getTempDirectory() + "/Test.txt");
14    if (file.exists) {
15      file.deleteFile();
16    }
17
18    // Write file in temporary directory
19    var mime = new MimeAttachment();
20    mime.name = "Test.txt";
21    mime.mimeType = "text/plain";
22    // mime.mimeType = "application/java-archive";
23    mime.buffer = byteBuffer;
24    mime.write(System.getTempDirectory(), null);
25
26  } catch (exception) {
27    System.log(exception)
28  }
```

Hint: If this program is executed in the local development environment, it is necessary to load a mockups[3] , from line 2 to 5. If this program is executed in an VCF Automation environment, this is not necessary.

The corresponding mime type must be set, in this case text/plain, via the mimeType attribute.

Store the Binary Files in a Node.js ZIP Package

The embedding approach is suitable only for small files. If larger files are to be used, here is another approach. A Node.js ZIP package is simply used as a container. This means that all required files are stored in a Node.js ZIP package. This also contains a Node.js handler, that can be called by the JavaScript RTE. This handler function returns the desired file as a base64 encoded string, which is converted into a byte array and this is then saved as a file.

Data Transfer to JavaScript RTE

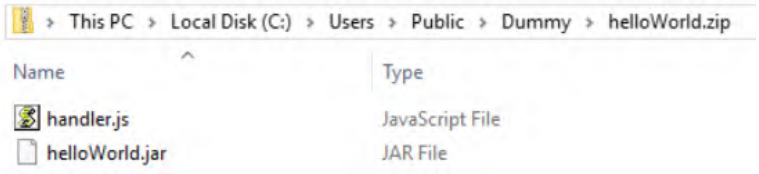
Here you can see a function to return a file from a Node.js ZIP package as base64 encoded string.

```
1  exports.handler = (context, inputs, callback) => {
2
3    const fs = require("fs");
4    const fileContentBase64 = fs.readFileSync(inputs.fileName, "base64");
5    callback(undefined, {status: "done", fileContentBase64: fileContentBase64});
6
7  }
```

This handler has as an input parameter fileName, this is the name of the file whose content should be returned, and as return value fileContentBase64, this is the base64 encoded file content.

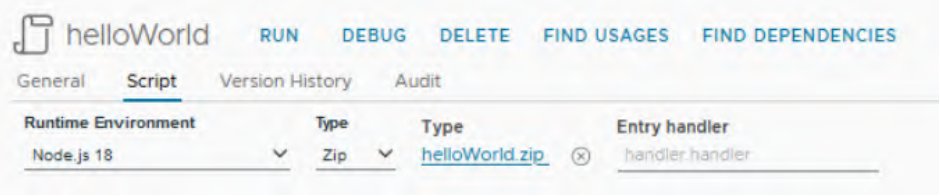
Build and Import a ZIP Package

First a ZIP file must be built, which contains handler.js and the required binaries.



Then an action is created ...

… where the ZIP file will be imported.



The input parameters and the return type, always Properties, must be defined.

Now this action can be called by another JavaScript action.

Write Byte Array

Here is a function to call the Node.js action and write the desired file to the temporary file directory.

```
1  function writeFileFromBase64ToTempDir(fileName) {
2
3    var res = System.getModule("de.stschnell.jarSilo").helloWorld(fileName);
4
5    if (res.status === "done") {
6
7      try {
8
9        var contextFactory = org.mozilla.javascript.ContextFactory();
10       var context = contextFactory.getGlobal().enterContext();
11
12       var byteArray = context.jsToJava(
13         java.util.Base64.getDecoder().decode(java.lang.String(res.fileContentBase64).getBytes()),
14         java.lang.Class.forName("[B")
15       );
16
17       var path = java.nio.file.Paths.get(System.appendToPath(System.getTempDirectory(), fileName));
18       java.nio.file.Files.deleteIfExists(path);
19       java.nio.file.Files.createFile(path);
20       var outputStream = java.nio.file.Files.newOutputStream(path);
21       outputStream.write(byteArray);
22       outputStream.close();
23
24     } catch (exception) {
25       System.log(exception);
26     } finally {
27       context.exit();
28     }
29
30   }
31
32 }
33
34 writeFileFromBase64ToTempDir("helloWorld.jar");
```

In line 4 the file content is returned as base64 encoded string. From line 13 to 16 the base64 encoded string is converted to an array of bytes. This is written to a file in the temporary file directory. And, after this action is run in VCF Automation, the file is available.

```
aria-auto.corp.vmbeans.com - PuTTY
root@aria-auto [ /data/vco/usr/lib/vco/app-server/temp ]# ls -l
-rw-r----- 1 root root 230 Sep 28 05:57 helloWorld.jar
root@aria-auto [ /data/vco/usr/lib/vco/app-server/temp ]#
```

Store the Binary File as an Action

It also possible to store a base64 encoded binary file as an action. The content of this action can be read from another action and written like Write Base64 Encoded String. There is no known length limit for the data that we can store in an action. So we can assume that we have a lot of potential with this approach.

Get Action as Text

Before the binary file can be stored as a base64 action, it is necessary to encode it with Create Base64 Encoded File. After this is done, the data can be read out with the following action.

```
var _getActionAsText = {

  main : function(moduleName, actionName) {

    var allActions =
      System.getModule("com.vmware.library.action").getAllActions();

    var allActionsInModule = allActions.filter( function(actionItems) {
      return actionItems.module.name === moduleName;
    });
    if (allActionsInModule.length === 0) {
      throw new Error("No actions were found in the module " +
        moduleName + ".");
    }

    var action = allActionsInModule.filter( function(actionItem) {
      return actionItem.name === actionName;
    });
    if (action.length === 0) {
      throw new Error("The action " + actionName +
        " was not found in the module " + moduleName + ".");
    } else if (action.length > 1) {
      System.warn("Too many actions, with the name " + actionName +
        ", were found in the module " + moduleName + ".");
      action.forEach( function(actionItem) {
        System.warn("ID: " + actionItem.id);
      });
      throw new Error("Too many actions, with the name " + actionName +
        ", were found in the module " + moduleName + ".");
    }

    return action[0].script;

  }

};

if (
  String(in_moduleName).trim() !== "" &&
  String(in_actionName).trim() !== ""
) {
  return _getActionAsText.main(
    in_moduleName,
    in_actionName
  );
} else {
  throw new Error(
    "in_moduleName or in_actionName argument can not be null"
```

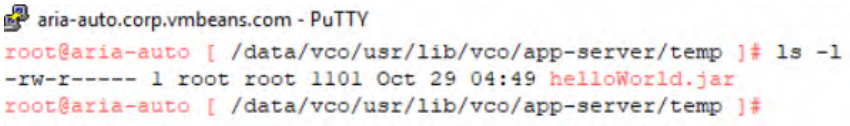
```
49    );
50  }
```

The action, to write the base64 encoded content in a file, is to modify a little. The content is read by the action call in lines 2 and 3. Otherwise, no modifications were made.

```
1   var jarFileBase64 = System.getModule("de.stschnell")
2     .getActionAsText(in_moduleName, in_actionName);
3
4   try {
5
6     var byteBuffer = new ByteBuffer(jarFileBase64);
7
8     var file = new File(System.getTempDirectory() + "/" + in_jarFileName);
9     if (file.exists) {
10      file.deleteFile();
11    }
12
13    var mime = new MimeAttachment();
14    mime.name = in_jarFileName;
15    mime.mimeType = "application/java-archive";
16    mime.buffer = byteBuffer;
17    mime.write(System.getTempDirectory(), null);
18
19  } catch (exception) {
20    System.log(exception)
21  }
```

Now it is possible to invoke the action, here an example.

After this action is executed in VCF Automation, the file is available.



Conclusion Store Binary Files

With the approaches presented here, we are now in a position to bring any file into an VCF Automation JavaScript RTE processing, without the need for administrative access to the file system. The resulting freedoms are naturally accompanied with a greater responsibility. A higher risk potential is not really visible, because the role of the programmer is a position of trust. Only approaches have been implemented here, that are already possible in the other runtime environments. And the independence, resulting from these approaches, helps us to expand our possibilities and to increases the speed of our development.

Execute Methods in JAR Files

The execution of methods of a JAR file are an important and normal part of Java development, because they contain the classes which provides the functionalities. Here different ways are presented in which JAR files can be used in the context of the JavaScript RTE.

Hint: The following approaches uses the technic of storing binary files.

What are JAR Files?

JAR is an acronym and means Java archive file. It is a container in ZIP file format, with lossless data compression, which can include one or more files or directories. It is used to store Java classes, as well as resources, metadata and a manifest. This bundling allows Java an efficient provisioning of their components. Each Java archive can be handled like a normal ZIP file.

JAR File for Testing

To demonstrate the use of a JAR file, here a Java source code that contains a method with and without parameters. The method simply returns a text, depending on the given parameters. The built JAR file must be placed in the VCF Automation file system, e.g. as described above. Below it is assumed that the JAR file is located in the temporary file directory.

```
1  Package de.stschnell;
2
3  public class helloWorld {
4
5    public static String say() {
6      return "Hello World from JAR";
7    }
8
9    public static String say(String name) {
10     if (name.trim().equals("")) {
11       return "Hello World from JAR";
12     } else {
13       return "Hello " + name + " from JAR";
14     }
15   }
16
17 }
```

Problem with Automatic Type Casting

Unfortunately, there seems to be a problem with the automatic type casting in VCF Automation. The Dunes framework converts specific native Java complex data types into in its opinion equivalent JavaScript data types[5]. This prevents a direct use of the following approach. For this reason, an indirect approach must be taken. The code to be processed is stored as a string in the program. This is directly read from an action or stored in the temporary file directory and invoked via an additional Rhino scope. On this way we can bring any code to execution, without reservation.

Limitations of the Command Class

The standard Command class has one crucial limitation. It delivers only the standard output stream (stdout) with the attribute output. But what happens if an error occurs and the output is take to the standard error stream (stderr)? The answer is simple, nothing. This information is not accessible. But often they contain important information and this knowledge is necessary to solve the errors. For this reason, the Command class is replaced by an own implementation.

Additionally offers this approach a time-out parameter, to abort the process when it exceeds. These advantages give us a better opportunity to integrate calls of operating system commands into the JavaScript runtime environment.

```
1  function executeCommand(command, timeOut) {
2
3    if (
4      typeof command === "undefined" ||
5      command === null ||
6      arguments.length === 0
7    ) {
8      throw new Error("command argument can not be undefined or null");
9    }
10
11   var _command;
12
13   if (Array.isArray(command)) {
14     _command = command;
15   } else if (typeof command === "string") {
16     _command = command.split(" ");
17   } else {
18     throw new Error(
19       "command argument must be string or array of string"
20     );
21   }
22
23   var _timeOut;
24
25   if (
26     typeof timeOut === "undefined" ||
27     timeOut === null ||
28     timeOut <= 1 ||
29     isNaN(timeOut)
30   ) {
31     _timeOut = -1;
32   } else {
33     _timeOut = timeOut;
34   }
35
36   var output = "";
37   var exitValue = -1;
38   var bufferedProcessInputStream;
39   var bufferedProcessErrorStream;
40
41   try {
42
43     var process = java.lang.Runtime.getRuntime().exec(_command);
44     if (_timeOut === -1) {
45       process.waitFor(); // Infinity
46     } else {
47       process.waitFor(
48         java.lang.Long(_timeOut),
49         java.util.concurrent.TimeUnit.MILLISECONDS
50       );
51     }
52     exitValue = process.exitValue();
```

```
    if (exitValue !== 0) {
      System.error("Exit value: " + exitValue);
    }

    var processInputStream =
      java.io.InputStreamReader(process.getInputStream());
    bufferedProcessInputStream =
      java.io.BufferedReader(processInputStream);

    var processErrorStream =
      java.io.InputStreamReader(process.getErrorStream());
    bufferedProcessErrorStream =
      java.io.BufferedReader(processErrorStream);

    var line = "";

    while ((line = bufferedProcessInputStream.readLine()) !== null) {
      output += line + "\n";
    }

    while ((line = bufferedProcessErrorStream.readLine()) !== null) {
      output += line + "\n";
    }

  } catch (exception) {
    output += exception.message;
    System.error(exception);
  } finally {
    if (bufferedProcessInputStream != null) {
      bufferedProcessInputStream.close();
    }
    if (bufferedProcessErrorStream != null) {
      bufferedProcessErrorStream.close();
    }
  }

  return { "output": String(output), "exitValue": exitValue };

}
```

To check the function executeCommand in a Windows environment you can use the following source:

```
load("System.class.js");

var command = [
  "cmd.exe",
  "/c",
  "echo",
  "Hello World"
];

var result = executeCommand(command);

System.log(result.exitValue); // 0
System.log(result.output); // Hello World
```

The test script opens the command prompt and calls the echo command which displays the message Hello World.

Using JShell to use JAR

The Java Shell tool (JShell), which was introduced with JDK 9, is an interactive tool for learning and prototyping Java. JShell scripts can also be called up directly. This enables Java code to be executed without a separate compilation process. The use of JAR files is also possible on the

simplest way. The class path to the JAR file can be set with the /env command. After the import the methods of the class can be used as usual.

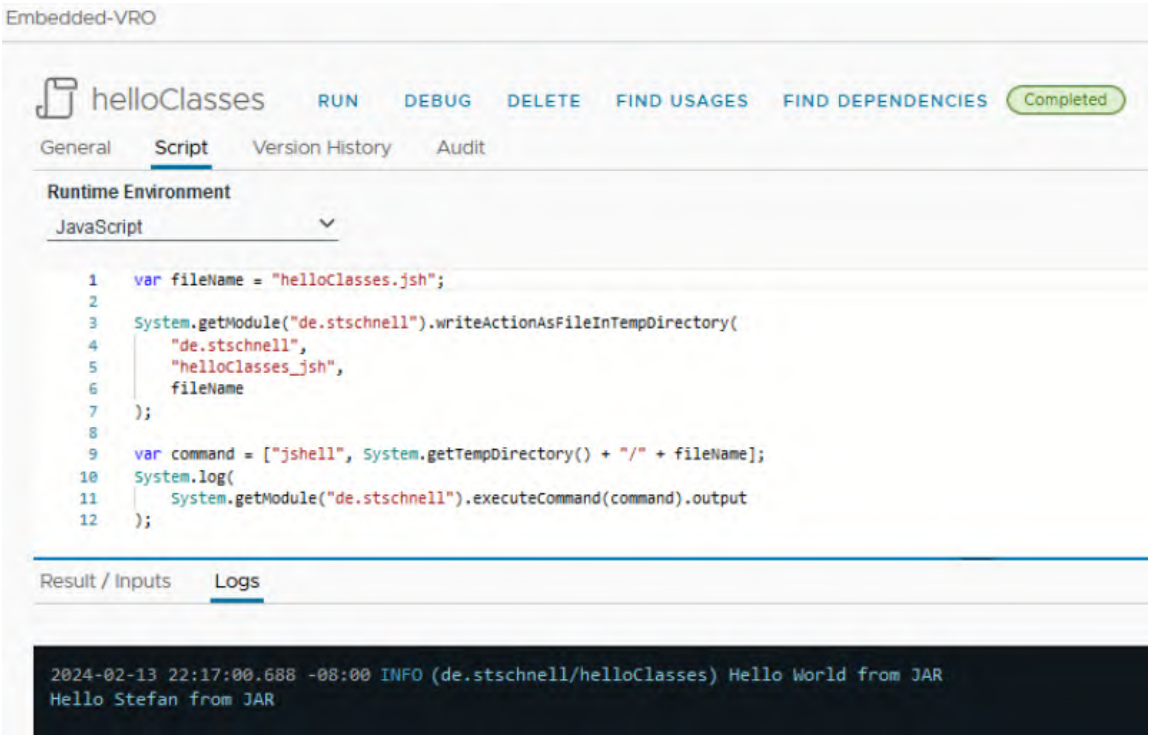
```
/env --class-path /lib/vco/app-server/temp/helloClasses.jar

import de.stschnell.*;

System.out.println( helloWorld.say() );

System.out.println( helloWorld.say("Stefan") );

/exit
```

This JShell script is saved as an action, can therefore be read out as text and saved in the temporary directory. Then JShell is called with the script and it is executed.

```
var fileName = "helloClasses.jsh";

System.getModule("de.stschnell").writeActionAsFileInTempDirectory(
  "de.stschnell",
  "helloClasses_jsh",
  fileName
);

var command = [ "jshell", System.getTempDirectory() + "/" + fileName ];
System.log(
  System.getModule("de.stschnell").executeCommand(command).output
);
```

Hint: The action `writeActionAsFileInTempDirectory` is an extension of Get Action as Text. The text is read and written to the temporary directory as a file.

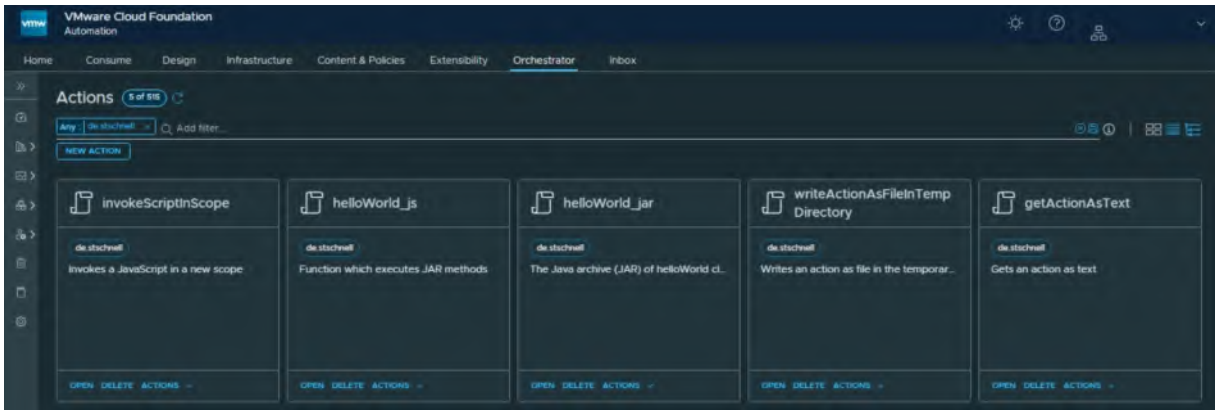
JShell is part of the JDK, so it is available in VCF Automation and can be used seamlessly with it. This simplifies the use immensely.



Using Scope to use JAR

The Rhino JavaScript Context class represents the runtime context of an executing script. When accessed to this it is necessary to enter to the current thread context. It stores the execution state of the JavaScript engine. After entering the context it is necessary to initialize the JavaScript scope. When the standard objects are initialized in this context, new instances and constructors are created. An initialization must be done before a script can be evaluated in that scope. This approach allows us to execute scripts in a Rhino scope that is independent of the Dunes framework. But we also lose access to the objects of the Dunes framework, e.g. like System.

Here an overview of the necessary actions to invoke JavaScript code in a new scope of Rhino in VCF Automation.



The following function enters the current thread context in line 22 and initializes a scope in line 23. The function to be invoked is compiled in line 25 and called in line 26. In this example it is loaded via `getActionAsText`.

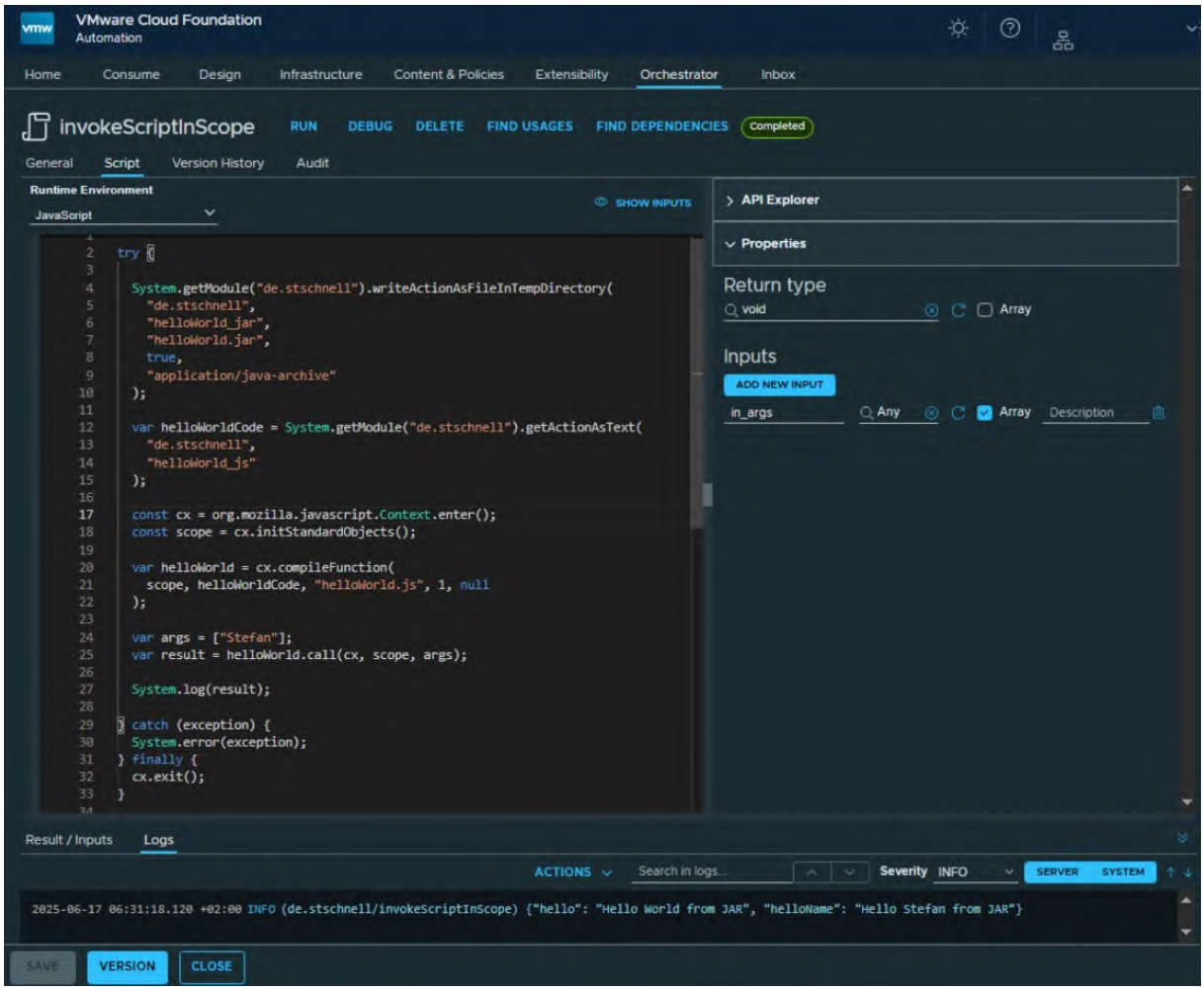
```
1  function invokeScriptInScope(moduleName, actionName, arguments) {
2
3  /**
4   * @param {string} moduleName Module name which contains the action
5   * @param {string} actionName Action name which contains the text
6   * @param {Array/Any} arguments Arguments as array
7   * @returns {Any}
8   *
9   * @example
10  * invokeScriptInScope("de.stschnell", "helloWorld_js", [java.lang.String("Stefan")])
11  */
12
13    var result = null;
14
15    try {
16
17      const script = System.getModule("de.stschnell").getActionAsText(
18        moduleName,
19        actionName
20      );
21
22      const cx = org.mozilla.javascript.Context.enter();
23      // cx.setLanguageVersion(org.mozilla.javascript.Context.VERSION_ES6);
24      const scope = cx.initStandardObjects();
25
26      var func = cx.compileFunction(scope, script, "script", 1, null);
27      result = func.call(cx, scope, arguments);
28
29    } catch (exception) {
30      System.warn(exception);
31    } finally {
```

```
    cx.exit();
  }

  return result;

}
```

The following function, which is loaded via `getActionAsText` from `invokeScriptInScope`, loads the class from the JAR file, gets the class and the method, executes its method, in our example say with and without a parameter, and delivers the result as JSON string.

```
function helloWorld(self, args) {

  var name = args[0]

  var jarFileName = "./helloWorld.jar";
  var jarFile = java.io.File(jarFileName);

  var urls = java.lang.reflect.Array.newInstance(java.net.URL, 1);
  urls[0] = jarFile.toURI().toURL();

  var classLoader = java.net.URLClassLoader(
    urls, java.lang.ClassLoader.getSystemClassLoader()
  );
  java.lang.Thread.currentThread().setContextClassLoader(classLoader);

  var cHelloWorld = java.lang.Class.forName(
    "de.stschnell.helloWorld", true, classLoader
  );

  var say = cHelloWorld.getDeclaredMethod("say");
  var sayName = cHelloWorld.getDeclaredMethod("say", java.lang.String);

  var hello = say.invoke(null);
  var helloName = sayName.invoke(null, [name]);

  return "{\"hello\": \"" + hello +
    "\", \"helloName\": \"" + helloName + "\"}";

}
```

This function must be executed in a new scope in VCF Automation so that the problem with automatic type casting cannot occur. Otherwise the classes java.io.File and java.net.URL throw an error. This integration approach allows a seamlessly solutions, to use the full range of the Java language, that does not leave the JavaScript environment. But, as already mentioned, with the disadvantage that we lose access to the objects of the Dunes Framework, e.g. like System, Server, Action, Workflow, etc.



Conclusion Execute Methods in JAR Files

With the approaches presented here, we are now in a position to include any JAR files and to call its methods seamlessly from the VCF Automation JavaScript RTE. The variety of possibilities arising from this, including the use of the full Java repertoire, offer many approaches for many use cases. The way via JShell or an additional scope seems cumbersome, but is a result of the circumstance of the automatic type casting problem.

Conclusion

These approaches shows us on the one hand on which ways we can store binary files in the context of the JavaScript RTE. For smaller file sizes they can be included directly in the source code as a byte array, and for large files they can be stored as base64 encoded JSON, in a Node.js ZIP package or as base64 encoded string in an action. On the other hand, based on this, it is shown how Java archive files can be integrated and used by the JavaScript RTE. This allows us to design our develop and test scenarios differently and to expand the scope of our development with all the possibilities offered by the Java programming language.

List of References

No.	Title	Reference
1	BellSoft Liberica JDK	Link
2	Mozilla Rhino JavaScript Engine	Link
3	Mock-up classes	Link
4	Using Java Reflection	Link
5	Automatic Conversion of Java Data Types in JavaScript Data Types	Link
6	Use Java Output Stream	Link
7	JShell	Link

Rights