Using Groovy

Groovy scripting is available for quickly creating new pipeline modules without having to write Java Code and create a Jar for the component. Groovy scripts can do many things that a regular Java pipeline stage can do.

- Groovy scripts are pooled and reused to minimize re-compiling of scripts.
- Therefore, all scripts are thread safe.

On this page:

Related Groovy scripting pages:

- Groovy home page
- Getting started guide
- User guide, topics on many subjects
- API guide

Custom Groovy Workflow Rules

1. When not to use a Groovy script:
   Anything Big or Complex - Size and complexity can result in greater opportunity for errors, it is recommended to code in Java where it is possible to do more careful unit testing with a Java based JUnit test bench.

2. More information:
   Groovy is a Java-based scripting language that compiles directly into Java byte code. It's advantages are that it runs very fast in the Java JVM and it has full access to all Java objects, classes, and methods.

Variables

Your Groovy script will automatically have a number of variables predefined before the script is called. These variables can be used directly from your script (no special setup or imports or anything required).

Variable	When Available	Description
job	always	Java Type = Job References the job which is being procesed. You can use this variable to create new sub-jobs, check on job status, wait for sub-jobs, etc.
doc	always	Java Type = AspireObject The AspireObject which holds all of the metadata for the current document being processed. This is the same as job.get() - the job's data object.
component	always	Java Type = StageImpl which is derived from ComponentImpl This variable provides access to the component itself. This can be used for a variety of useful tasks, such as logging, accessing other components, getting Aspire Home, and turning relative paths to Aspire Home into absolute paths. Note that StageImpl extends ComponentImpl, where all of the most useful methods are located. This variable refers to the Workflow section which is running the rule, this sections can be: **AfterScanWorkflow**, **AddUpdateWorkflow**, **Publish Workflow**, **DeleteWorkflow** or **ErrorWorkflow**
globals	always	Java Type = HashMap This variable contains global variables you can pass between rules if required.
rule	always	Java Type = Rule This variable reference the rule that is being executed.

Job Variables

All of the variables in the AspireObject instance map are automatically available as variables in your Groovy Script. This makes it possible for you to create variables which are attached to your document which can then be used by your own script, or other Groovy scripts in other pipeline modules. See AspireObject for more details about the AspireObject instance.

Note: These variables are attached to the AspireObject instance within the *job* and *not* the groovy stage itself. Therefore, the values for these variables are passed down the pipeline with the *job* and are therefore available to any later Groovy stage which processes the same job.

Within Groovy code, you can access variables just like regular variables.

Example 1: Setting a document variable (*will* be stored in the Job)

```
myVar = 12;
```

Example 2: Using a document variable

```
print myVar + 33;
```

Example 3: Setting a local variable (will *not* be stored in the Job)

```
def myVar = 32;
```

Example 4: Setting the variable with putVariable() (works the same as example 1)

```
job.putVariable("myVar",12);
println myVar + 32;
```

Note that, once the "myVar" variable is set with job.putVariable() (all of the examples above *except* example #3), it is attached to the document. All Groovy scripts which process the same document will have access to the "myVar" variable.

Example 5: Setting the variable with *job.* prefix (works the same as example 1)

```
job.myVar = 12;
```

Hierarchical Job Variables

In Aspire, jobs can be split into a series of smaller jobs called "sub jobs". Every sub-job maintains a link to the parent-job from whence it was derived.

This forms a "job hierarchy". For example, you might have a large XML file which has multiple data records within it. Each of the records within the XML file may be processed by a sub job. Sub jobs are typically extracted with "Sub Job Extractors" such as the Tabular Files Extractor or the XML Sub Job Extractor.

When accessing a variable, the following procedure is followed:

1. Check to see if the variable is on the current job's AspireObject
2. If not, then check to see if the job has a parent job.
    a. If so, then look for the variable in the parent job's AspireObject
3. Continue checking up the job hierarchy until the variable is found or you have reached a job with no parent.

What this means is that referencing a variable as follows will first check the current job, and then will automatically check the parent job and grandparent jobs (if any):

```
println myVar
```

If no job has the variable, then the variable will return *null*.

All new variables set, for example, will be place in the current job, and *not* the parent job:

```
myNewVar = 12
```

Groovy Programming

How do I access a job?

The Job being processed by the Scripting application is made available to Groovy in the *job* variable. You may then call any Java method on the Job object, for example

```
myVar = job.getJobId()
```

How do I access the Aspire document?

The document (an AspireObject) associated with the job being processed by the Scripting application is made available to Groovy in the *doc* variable. While you can then access the Java methods of the *doc*, the implementation allows a shortcut to children.

Get a value

If you want to get the child name **content**, use:

```
myVar = doc.content
```

NOTE: Children are returned as AspireObjects, so if you want to get the text of **content**, use:

```
myVar = doc.content.text()
```

Groovy also provides a convenient method of protecting against null values. The above code will give a null pointer exception if there is no **content** child for the document - as there's no child named **content**, it returns null and then you try to access the *text()* method on null.

To protect against returned nulls, use the following syntax:

```
 myVar = doc.content?.text()
```

You can also chain calls together to get children of children. If you want the text of the **myProperty** child of the **properties** child, use:

```
  myVar = doc.properties?.myProperty?.text()
```

Set a value

If you want to set the value of the **content** child on the document, use:

```
 doc.content = "my text value"
```

Use a Third-Party Jar from a Groovy script (using Reflection)

Below is a base/example script that will allow you to incorporate functionality from external jar files into a groovy stage in the pipeline.

Normally, 3rd Party jars must be "wrapped" to be used in Aspire. This requires a wrapper stage to be coded, but if you only want to use a small number of methods in the jar file, this Groovy stage method can be used.

The script assumes that the jar(s) file (and dependencies) has/have been copied into the "lib" folder as part of the Aspire distribution. Please note an example folder structure below:

Aspire libconfigdatabinfelix-cache

```
//=============================================================================================================
// Details of 3rd Party Jar
//=============================================================================================================

// The third party jar
def basePathLangDetect = "lib/langdetect.jar";
def basePathJSONIC = "lib/jsonic-1.3.0.jar";
def basePathCatCon = "lib/tgcatcon.jar";
def basePaths = [basePathLangDetect, basePathJSONIC, basePathCatCon] as String[];

// Classes to use
def classDetectorFactory = "com.cybozu.labs.langdetect.DetectorFactory";
def classJSON = "net.arnx.jsonic.JSON";
```

```
def classJSONException = "net.arnx.jsonic.JSONException";

//================================================================================================
// Base methods to load 3rd Party Classes
//================================================================================================
//There should not be major reason to change this code.

//Obtains the relative path
def getRelativePath = { basePath ->
        def path = "file://" + new File(basePath).toURI().getPath();
}

// Obtains the classloader consisting of the original class loader with the jar file added.
// Use this if you only have a single 3rd Party jar file.
// param: path - the path to the jar file
def getClassLoader = { path ->
        def classLoader = ClassLoader.systemClassLoader
        while (classLoader.parent) {
                classLoader = classLoader.parent
        }
        def newClassLoader = new URLClassLoader([new File(path).toString().toURL()] as URL[], classLoader);
        return newClassLoader;
}

// Obtains the classloader consisting of the original class loader with the jar files added.
// Use this if you have multiple 3rd Party jar files.
// param: paths - an array of the paths to the jar file
def getMultiClassLoader = { paths ->
        def classLoader = ClassLoader.systemClassLoader
        while (classLoader.parent) {
                classLoader = classLoader.parent
        }
        ArrayList<URL> urls = new ArrayList<URL>(paths.length);
        for (String path:paths) {
                urls.add(new File(getRelativePath(path)).toString().toURL());
        }
        def newClassLoader = new URLClassLoader(urls.toArray([] as URL[]), classLoader);
        return newClassLoader;
}

// Obtains a new instance of a class using the classloader and a constructor with an empty argument list
def getClass = { cName, cLoader ->
        // load the class
        Class clazz = cLoader.loadClass(cName)
        return clazz.newInstance()
}

// Obtains a new instance of a class using the classloader and a constructor with the specified array of
arguments.
// paramTypes is an array of classes representing the Constructor parameters.
// paramValues is an array of Objects representing the Constructor parameter values.
def getClassWithArgs = { cName, cLoader, paramTypes, paramValues ->
        // load the class
        Class clazz = cLoader.loadClass(cName)
        return clazz.getConstructor(paramTypes).newInstance(paramValues)
}

// Load the class using the classloader.
// Use this if you don't need an instance of the class but the script needs to be aware of it.
def loadClass = { cName, cLoader ->
        return cLoader.loadClass(cName)
}
//================================================================================================
// End Base methods to load 3rd Party Classes
//================================================================================================


//================================================================================================
// Example usage
//================================================================================================
```

```
// Get a Class Loader for all 3rd Party Jars. Need to use the same one for all classes.
classLoader = getMultiClassLoader(basePaths);

// Load the extra classes required by LangDetect
jsonException = loadClass(classJSONException, classLoader);
json = loadClass(classJSON, classLoader);

// Get a DetectorFactory and load the files from the LangDetect profiles directory
detectorFactory = loadClass(classDetectorFactory, classLoader);
detectorFactory.loadProfile("C:/Profiles");

// Instantiate a class with no constructor parameters
catHandle = getClass(classname, classLoader);

// Call a method
catHandle.addServer("127.0.0.1", 6500);

// Example of instantiating a class with constructor parameters (equivalent to cal = new SimpleTimeZone(0,
"en"))
cal = getClassWithArgs("java.util.SimpleTimeZone", classLoader, [int.class, String.class] as Class[], [0,
"en"] as Object[]);
```