This article describes various "recipes" on how to accomplish certain tasks with the NiFi processor [ExecuteScript](https://nifi.apache.org/docs/nifi-docs/components/org.apache.nifi.processors.script.ExecuteScript/index.html), with examples given in [Groovy](http://www.groovy-lang.org/), [Jython](http://www.jython.org/), Javascript ([Nashorn](http://openjdk.java.net/projects/nashorn/)), and [JRuby](http://jruby.org/). This is Part 2 in the series, I will be discussing reading from and writing to flow file contents, as well as error handling.

[Part 1](https://community.hortonworks.com/articles/75032/executescript-cookbook-part-1.html)*- Introduction to the NiFi API and FlowFiles*

* Getting a flow file from an incoming queue
* Creating new flow files
* Working with flow file attributes
* Transferring a flow file
* Logging

***Part 2 - FlowFile I/O and Error Handling***

* **Reading from a flow file**
* **Writing to a flow file**
* **Reading and writing to/from a flow file**
* **Error Handling**

[Part 3](https://community.hortonworks.com/articles/77739/executescript-cookbook-part-3.html)*- Advanced Features*

* Using Dynamic Properties
* Adding Modules
* State Management
* Accessing Controller Services

Introduction to FlowFile I/O

Flow files in NiFi are made of two major components, attributes and content. Attributes are metadata about the content / flow file, and we saw how to manipulate them using ExecuteScript in [Part 1](https://community.hortonworks.com/articles/75032/executescript-cookbook-part-1.html) of this series. The content of a flow file is, at its heart, simply a collection of bytes and has no inherent structure, schema, format, etc. Various NiFi processors assume the incoming flow files have a particular schema/format (or determine it from attributes such as "mime.type" or infer it in other ways). These processors can then act upon the content based on the assumption that the files really do have that format (and will often transfer to a "failure" relationship if they do not). Also processors may output flow files in a specified format, this is described in the processors' descriptions in the [NiFi documentation](https://nifi.apache.org/docs.html" \t "_blank).

Input and Output (I/O) for the contents of flow files is provided via the ProcessSession API and thus the "session" variable for ExecuteScript (see Part 1 for more information). One mechanism for this is to pass a callback object into a call to session.read() or session.write(). An InputStream and/or OutputStream will be created for the FlowFile object, and the callback object will be invoked using the corresponding callback interface, with the InputStream and/or OutputStream references passed in for use by the callback. There are three main callback interfaces, each with its own use case:

*InputStreamCallback*

This interface is used by the session.read( *flowFile*, *inputStreamCallback*) method to provide an InputStream from which to read the contents of the flow file. The interface has a single method:

void process(InputStream in) throws IOException

This interface provides a managed input stream for use. The input stream is automatically opened and closed though it is ok to close the stream manually. This is the form you would use if you are only reading from a particular flow file, and not writing back out to it.

An example is when you want to process an incoming flow file, but create many output flow files, such as the [SplitText](https://nifi.apache.org/docs/nifi-docs/components/org.apache.nifi.processors.standard.SplitText/index.html" \t "_blank) processor does.

*OutputStreamCallback*

This interface is used by the session.write( *flowFile*, *outputStreamCallback*) method to provide an OutputStream to which to write the contents of the flow file. The interface has a single method:

void process(OutputStream out) throws IOException

This interface provides a managed output stream for use. The output stream is automatically opened and closed though it is ok to close the stream manually - and quite important if any streams wrapping these streams open resources which should be cleared.

An example is when ExecuteScript will be generating data, either from within or from an external file, but not a flow file. Then you would use session.create() to create a new FlowFile, then session.write(*flowFile*, *outputStreamCallback*) to insert content.

*StreamCallback*

This interface is used by the session.write(*flowFile*, *streamCallback*) method to provide an InputStream and OutputStream, from which to read from and/or write to the contents of the flow file. The interface has a single method:

void process(InputStream in, OutputStream out) throws IOException

This interface provides managed input and output streams for use. The input stream is automatically opened and closed though it is ok to close the streams manually - and quite important if any streams wrapping these streams open resources which should be cleared.

An example is when you want to process an incoming flow file and overwrite its contents with something new, such as the [EncryptContent](https://nifi.apache.org/docs/nifi-docs/components/org.apache.nifi.processors.standard.EncryptContent/index.html" \t "_blank) processor does.

Since these callbacks are Java objects, the script will have to create one and pass it into the session method(s), the recipes will illustrate this for the various scripting languages. Also there are other methods of reading from and writing to flow files, which include:

* + Using session.read(*flowFile*) to return an InputStream. This alleviates the need for an InputStreamCallback, instead it returns an InputStream that you can read from. In exchange you must manage (close, e.g.) the InputStream manually.
* + Using session.importFrom(*inputStream*, *flowFile*) to write from an InputStream to a FlowFile. This replaces the need for a session.write() with an OutputStreamCallback passed in.
* Now, on to the recipes :)

*Recipes*

Recipe: Read the contents of an incoming flow file using a callback

Use Case: You have incoming connection(s) to ExecuteScript and want to retrieve the contents of a flow file from the queue(s) for processing.

Approach: Use the read(*flowFile*, *inputStreamCallback*) method from the session object. An InputStreamCallback object is needed to pass into the read() method. Note that because InputStreamCallback is an object, the contents are only visible to that object by default. If you need to use the data outside the read() method, use a more globally-scoped variable. The examples will store the full contents of the incoming flow file into a String (using Apache Commons' IOUtils class). NOTE: For large flow files, this is not the best technique; rather you should read in only as much data as you need, and process that as appropriate. For something like SplitText, you could read in a line at a time and process it within the InputStreamCallback, or use the session.read(flowFile) approach mentioned earlier to get an InputStream reference to use outside of a callback.

Examples:

*Groovy*

import org.apache.commons.io.IOUtils

import java.nio.charset.StandardCharsets

flowFile = session.get()

if(!flowFile)return

def text = ''

// Cast a closure with an inputStream parameter to InputStreamCallback

session.read(flowFile, {inputStream ->

text = IOUtils.toString(inputStream, StandardCharsets.UTF\_8)

// Do something with text here

} as InputStreamCallback)

*Jython*

from org.apache.commons.io import IOUtils

from java.nio.charset import StandardCharsets

from org.apache.nifi.processor.io import InputStreamCallback

# Define a subclass of InputStreamCallback for use in session.read()

class PyInputStreamCallback(InputStreamCallback):

def \_\_init\_\_(self):

pass

def process(self, inputStream):

text = IOUtils.toString(inputStream, StandardCharsets.UTF\_8)

# Do something with text here

# end class

flowFile = session.get()

if(flowFile != None):

session.read(flowFile, PyInputStreamCallback())

# implicit return at the end

*Javascript*

var InputStreamCallback = Java.type("org.apache.nifi.processor.io.InputStreamCallback")

var IOUtils = Java.type("org.apache.commons.io.IOUtils")

var StandardCharsets = Java.type("java.nio.charset.StandardCharsets")

var flowFile = session.get();

if(flowFile != null) {

// Create a new InputStreamCallback, passing in a function to define the interface method

session.read(flowFile,

new InputStreamCallback(function(inputStream) {

var text = IOUtils.toString(inputStream, StandardCharsets.UTF\_8);

// Do something with text here

}));

}

*JRuby*

java\_import org.apache.commons.io.IOUtils

java\_import org.apache.nifi.processor.io.InputStreamCallback

# Define a subclass of InputStreamCallback for use in session.read()

class JRubyInputStreamCallback

include InputStreamCallback

def process(inputStream)

text = IOUtils.toString(inputStream)

# Do something with text here

end

end

jrubyInputStreamCallback = JRubyInputStreamCallback.new

flowFile = session.get()

if flowFile != nil

session.read(flowFile, jrubyInputStreamCallback)

end

Recipe: Write content to an outgoing flow file using a callback

Use Case: You want to generate content for an outgoing flow file.

Approach: Use the write(*flowFile*, *outputStreamCallback*) method from the session object. An OutputStreamCallback object is needed to pass into the write() method. Note that because OutputStreamCallback is an object, the contents are only visible to that object by default. If you need to use the data outside the write() method, use a more globally-scoped variable. The examples will write a sample String to a flowFile.

Examples:

*Groovy*

import org.apache.commons.io.IOUtils

import java.nio.charset.StandardCharsets

flowFile = session.get()

if(!flowFile) return

def text = 'Hello world!'

// Cast a closure with an outputStream parameter to OutputStreamCallback

flowFile = session.write(flowFile, {outputStream ->

outputStream.write(text.getBytes(StandardCharsets.UTF\_8))

} as OutputStreamCallback)

*Jython*

from org.apache.commons.io import IOUtils

from java.nio.charset import StandardCharsets

from org.apache.nifi.processor.io import OutputStreamCallback

# Define a subclass of OutputStreamCallback for use in session.write()

class PyOutputStreamCallback(OutputStreamCallback):

def \_\_init\_\_(self):

pass

def process(self, outputStream):

outputStream.write(bytearray('Hello World!'.encode('utf-8')))

# end class

flowFile = session.get()

if(flowFile != None):

flowFile = session.write(flowFile, PyOutputStreamCallback())

# implicit return at the end

*Javascript*

var OutputStreamCallback = Java.type("org.apache.nifi.processor.io.OutputStreamCallback");

var IOUtils = Java.type("org.apache.commons.io.IOUtils");

var StandardCharsets = Java.type("java.nio.charset.StandardCharsets");

var flowFile = session.get();

if(flowFile != null) {

// Create a new OutputStreamCallback, passing in a function to define the interface method

flowFile = session.write(flowFile,

new OutputStreamCallback(function(outputStream) {

outputStream.write("Hello World!".getBytes(StandardCharsets.UTF\_8))

}));

}

*JRuby*

java\_import org.apache.commons.io.IOUtils

java\_import java.nio.charset.StandardCharsets

java\_import org.apache.nifi.processor.io.OutputStreamCallback

# Define a subclass of OutputStreamCallback for use in session.write()

class JRubyOutputStreamCallback

include OutputStreamCallback

def process(outputStream)

outputStream.write("Hello World!".to\_java.getBytes(StandardCharsets::UTF\_8))

end

end

jrubyOutputStreamCallback = JRubyOutputStreamCallback.new

flowFile = session.get()

if flowFile != nil

flowFile = session.write(flowFile, jrubyOutputStreamCallback)

end

Recipe: Overwrite an incoming flow file with updated content using a callback

Use Case: You want to reuse the incoming flow file but want to modify its content for the outgoing flow file.

Approach: Use the write(*flowFile*, *streamCallback*) method from the session object. An StreamCallback object is needed to pass into the write() method. StreamCallback provides both an InputStream (from the incoming flow file) and an outputStream (for the next version of that flow file), so you can use the InputStream to get the current contents of the flow file, then modify them and write them back out to the flow file. This overwrites the contents of the flow file, so for append you'd have to handle that by appending to the read-in contents, or use a different approach (with session.append() rather than session.write() ). Note that because StreamCallback is an object, the contents are only visible to that object by default. If you need to use the data outside the write() method, use a more globally-scoped variable. The examples will reverse the contents of the incoming flowFile (assumed to be a String) and write out the reversed string to a new version of the flowFile.

Examples:

*Groovy*

import org.apache.commons.io.IOUtils

import java.nio.charset.StandardCharsets

flowFile = session.get()

if(!flowFile) return

def text = 'Hello world!'

// Cast a closure with an inputStream and outputStream parameter to StreamCallback

flowFile = session.write(flowFile, {inputStream, outputStream ->

text = IOUtils.toString(inputStream, StandardCharsets.UTF\_8)

outputStream.write(text.reverse().getBytes(StandardCharsets.UTF\_8))

} as StreamCallback)

session.transfer(flowFile, REL\_SUCCESS)

*Jython*

from org.apache.commons.io import IOUtils

from java.nio.charset import StandardCharsets

from org.apache.nifi.processor.io import StreamCallback

# Define a subclass of StreamCallback for use in session.write()

class PyStreamCallback(StreamCallback):

def \_\_init\_\_(self):

pass

def process(self, inputStream, outputStream):

text = IOUtils.toString(inputStream, StandardCharsets.UTF\_8)

outputStream.write(bytearray('Hello World!'[::-1].encode('utf-8')))

# end class

flowFile = session.get()

if(flowFile != None):

flowFile = session.write(flowFile, PyStreamCallback())

# implicit return at the end

*Javascript*

var StreamCallback = Java.type("org.apache.nifi.processor.io.StreamCallback");

var IOUtils = Java.type("org.apache.commons.io.IOUtils");

var StandardCharsets = Java.type("java.nio.charset.StandardCharsets");

var flowFile = session.get();

if(flowFile != null) {

// Create a new StreamCallback, passing in a function to define the interface method

flowFile = session.write(flowFile,

new StreamCallback(function(inputStream, outputStream) {

var text = IOUtils.toString(inputStream, StandardCharsets.UTF\_8)

outputStream.write(text.split("").reverse().join("").getBytes(StandardCharsets.UTF\_8))

}));

}

*JRuby*

java\_import org.apache.commons.io.IOUtils

java\_import java.nio.charset.StandardCharsets

java\_import org.apache.nifi.processor.io.StreamCallback

# Define a subclass of StreamCallback for use in session.write()

class JRubyStreamCallback

include StreamCallback

def process(inputStream, outputStream)

text = IOUtils.toString(inputStream)

outputStream.write((text.reverse!).to\_java.getBytes(StandardCharsets::UTF\_8))

end

end

jrubyStreamCallback = JRubyStreamCallback.new

flowFile = session.get()

if flowFile != nil

flowFile = session.write(flowFile, jrubyStreamCallback)

end

Recipe: Handle errors during script processing

Use Case: An error occurs in the script (either by data validation or a thrown exception), and you want the script to handle it gracefully.

Approach: For exceptions, use the exception-handling mechanism for the scripting language (often they are try/catch block(s)). For data validation, you can use a similar approach, but define a boolean variable like "valid" and an if/else clause rather than a try/catch clause. ExecuteScript defines "success" and "failure" relationships; often your processing will transfer "good" flow files to success and "bad" flow files to failure (logging an error in the latter case). It is possible

Examples:

*Groovy*

flowFile = session.get()

if(!flowFile) return

try {

// Something that might throw an exception here

// Last operation is transfer to success (failures handled in the catch block)

session.transfer(flowFile, REL\_SUCCESS)

} catch(e) {

log.error('Something went wrong', e)

session.transfer(flowFile, REL\_FAILURE)

}

*Jython*

flowFile = session.get()

if(flowFile != None):

try:

# Something that might throw an exception here

# Last operation is transfer to success (failures handled in the catch block)

session.transfer(flowFile, REL\_SUCCESS)

except:

log.error('Something went wrong', e)

session.transfer(flowFile, REL\_FAILURE)

# implicit return at the end

*Javascript*

var flowFile = session.get();

if(flowFile != null) {

try {

// Something that might throw an exception here

// Last operation is transfer to success (failures handled in the catch block)

session.transfer(flowFile, REL\_SUCCESS)

} catch(e) {

log.error('Something went wrong', e)

session.transfer(flowFile, REL\_FAILURE)

}

}

*JRuby*

flowFile = session.get()

if flowFile != nil

begin

# Something that might raise an exception here

# Last operation is transfer to success (failures handled in the rescue block)

session.transfer(flowFile, REL\_SUCCESS)

rescue Exception => e

log.error('Something went wrong', e)

session.transfer(flowFile, REL\_FAILURE)

end

end