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" \t "_blank), with examples given in [Groovy](http://www.groovy-lang.org/), [Jython](http://www.jython.org/" \t "_blank), Javascript ([Nashorn](http://openjdk.java.net/projects/nashorn/" \t "_blank)), and [JRuby](http://jruby.org/" \t "_blank). This is Part 3 in the series, I will be discussing advanced features such as dynamic properties, modules, state management, and accessing/using Controller Services.

[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](https://community.hortonworks.com/articles/75545/executescript-cookbook-part-2.html)*- 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 - Advanced Features***

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

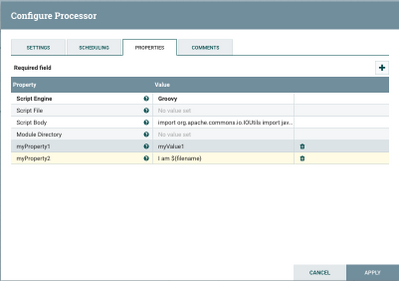
Advanced Features

The first two articles in the series covered the basics of flow file operations, such as reading/writing attributes and content, as well as retrieving and transferring flow files using the "session" variable (a ProcessSession object). There are many more capabilities available to ExecuteScript; I will talk about a few of them here.

*Dynamic Properties*

One such capability is the concept of dynamic properties, also called User-Defined properties. These are properties for a processor for which a user can set both the property name and value. Not all processors support/use dynamic properties, but ExecuteScript will pass dynamic properties as variables which reference a PropertyValue object corresponding to the property's value. There are two important things to note here:

1. Because the property name is bound as-is to a variable name, the naming convention for dynamic properties must be supported for the specified programming language. For example, Groovy does not support period (.) as a valid variable character, so a dynamic property such as "my.value" will cause the processor to fail. A valid alternative in this case is "myValue".
2. The PropertyValue object is used (rather than a String representation of the value) to allow the script to perform various operations on the property's value before evaluating it to a String. If the property is known to contain a literal value, you can call the getValue() method on the variable to get its String representation. If instead the value could contain Expression Language or you want to cast the value to something other than String (such as the value 'true' to a Boolean object), there are methods for these operations too. These examples are illustrated in the recipes below, assuming we have two properties 'myProperty1' and 'myProperty2' defined as such:



Recipe: Get the value of a dynamic property

Use Case: A user has entered a dynamic property for use in the script (configuration parameter, e.g.).

Approach: Use the getValue() method from the variable's PropertyValue object. This method returns a String representation of the value of the dynamic property. Note that if Expression Language is present in the value, getValue() will not evaluate it (see the following recipe for that functionality).

Examples:

*Groovy*

def myValue1 = myProperty1.value

*Jython*

myValue1 = myProperty1.getValue()

*Javascript*

var myValue1 = myProperty1.getValue()

*JRuby*

myValue1 = myProperty1.getValue()

Recipe: Get the value of a dynamic property after evaluating Expression Language constructs

Use Case: A user has entered a dynamic property for use in the script (configuration parameter, e.g.), and it refers to attribute(s) in an incoming flow file.

Approach: Use the evaluateAttributeExpressions(*flowFile*) method from the variable's PropertyValue object. This method, followed by getValue(), returns a String representation of the value of the dynamic property after any Expression Language constructs have been evaluated. If a flow file is not available, but variables have been defined in the environment or Variable Registry, you can use evaluateAttributeExpressions() without a parameter

Examples:

*Groovy*

def myValue1 = myProperty1.value

def myValue2 = myProperty2.evaluateAttributeExpressions(flowFile).value

*Jython*

myValue1 = myProperty1.getValue()myValue2 = myProperty2.evaluateAttributeExpressions(flowFile).getValue()

*Javascript*

var myValue1 = myProperty1.getValue()var myValue2 = myProperty2.evaluateAttributeExpressions(flowFile).getValue()

*JRuby*

myValue1 = myProperty1.getValue()myValue2 = myProperty2.evaluateAttributeExpressions(flowFile).getValue()

*Adding Modules*

Another feature of ExecuteScript is the ability to add external "modules" to the classpath, which allows you to leverage various third-party libraries, scripts, etc. However each of the script engines handles the concept of modules differently, so I will discuss them separately. In general though, there are two types of modules, Java libraries (JARs) and scripts (written in the same language as the one in ExecuteScript. Here is how the various script engines handle these:

*Groovy*

The Groovy script engine (at least for use in ExecuteScript) does not support the importing of other Groovy scripts, instead it allows for JARs to be added to its classpath. So for external Groovy projects, consider compiling into bytecode and pointing at the classes folder or packaging into a JAR.

When using Groovy, the Module Directory property can be set to a comma-separated list of files (JARs) and folders. If a folder is specified, ExecuteScript will find any JARs in that folder and add those as well. This allows you to include third-party software that consists of a large number of JARs. For an example in Groovy, see [this blog post](http://funnifi.blogspot.com/2016/02/executescript-using-modules.html).

*Jython*

The Jython script engine (at least for use in ExecuteScript) currently only supports the importing of pure Python modules, not natively-compiled modules (CPython, e.g.) such as numpy or scipy. It also does not currently support JARs, although this may change in an upcoming release. See [this HCC post](https://community.hortonworks.com/questions/75921/how-to-add-additional-libraries-in-executescript-p.html) for more details. Under the hood, the entries in the Module Directory property are prepended to your script before execution, using "import sys" followed by "sys.path.append" for each of the specified module locations.

If you have Python installed, you can make use of all its installed pure Python modules by adding its site-packages folder to the Module Directory property, such as

/usr/local/lib/python2.7/site-packages

Then in your script you can import from the various packages, such as

from user\_agents import parse

*Javascript*

The Javascript script engine (at least for use in ExecuteScript), allows for the same kinds of JARs/folders as the Groovy engine. It looks for JARs and if a folder is specified, it also looks in that folder for JARs.

*JRuby*

The JRuby script engine (at least for use in ExecuteScript) currently only allows for the specification of individual JARs, if a folder is specified it must have classes in it (same as the java compiler would want to see it), if the folder contains JARs they will not be automatically picked up. Also currently, no pure Ruby modules can be imported.

I intend to improve on all the engines in the future, to give a more powerful yet consistent user experience.

*State Management*

NiFi (as of 0.5.0 I believe) offers the capability for Processors and other NiFi components to persist some information to enable some stateful functionality around the component. For example, the [QueryDatabaseTable](https://nifi.apache.org/docs/nifi-docs/components/org.apache.nifi.processors.standard.QueryDatabaseTable/index.html" \t "_blank) processor keeps track of the largest values it has seen for the specified columns, such that the next time it runs, it will only fetch rows whose values are larger than those that have been seen so far (i.e. stored in the State Manager).

An important concept in terms of state management is Scope. NiFi components can choose to store their state at the cluster level or the local level. Note that in a standalone NiFi instance, "cluster scope" is identical to "local scope". The choice of scope is usually about whether in a flow, the same processor on each node can share the state data. If the instances across the cluster do not need to share state, then use local scope. In Java these options are provided as an enum called Scope, so when I refer to Scope.CLUSTER and Scope.LOCAL, I mean cluster and local scope respectively.

To employ state management features in ExecuteScript (language-specific examples are below), you get a reference to the StateManager by calling ProcessContext's getStateManager() method (recall that each engine gets a variable named "context" with an instance of ProcessContext). You can then call the following methods on the StateManager object:

void setState(Map<String, String> *state*, Scope *scope*) - Updates the value of the component's state at the given scope, setting it to given value. Notice that the value is a Map; the concept of "component state" is the Map of all key/value pairs that each constitute a lower-level of state. The Map is updated all at once to provide atomicity.

StateMap getState(Scope *scope*) - Returns the current state for the component at the given scope. This method never returns null; rather it is a StateMap object and if the state has not yet been set, the StateMap's version will be -1, and the map of values will be empty. Often a new Map<String,String> will be created to store the updated values, then setState() or replace() will be called.

boolean replace(StateMap *oldValue*, Map<String, String> *newValue*, Scope *scope*) - Updates the value of the component's state (at the given scope) to the new value if and only if the value currently is the same as the given oldValue. If the state was updated to the new value, true is returned; otherwise false is returned if the state's value was not equal to oldValue.

void clear(Scope *scope*) - Clears all keys and values from the component's state at the given scope.

Recipe: Get the current map of key/value pairs

Use Case: The script needs to get the current key/value pairs from the state manager for use in the script (update, e.g.).

Approach: Use the getStateManager() method from ProcessContext, then getStateMap() from the StateManager, then toMap() to convert to a Map<String,String> of key/value pairs. Note that StateMap also has a get(*key*) method for simply retrieving a value, but this is not often used as the Map is usually updated and must be set for the StateManager once this is done.

Examples:

*Groovy*

import org.apache.nifi.components.state.Scope

def oldMap = context.stateManager.getState(Scope.LOCAL).toMap()

*Jython*

from org.apache.nifi.components.state import Scope

oldMap = context.stateManager.getState(Scope.LOCAL).toMap()

*Javascript*

var Scope = Java.type('org.apache.nifi.components.state.Scope');

var oldMap = context.stateManager.getState(Scope.LOCAL).toMap();

*JRuby*

java\_import org.apache.nifi.components.state.Scope

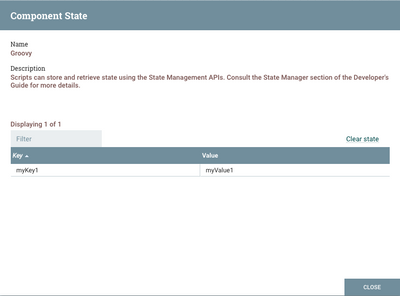
oldMap = context.stateManager.getState(Scope::LOCAL).toMap()

NOTE: Only the Scope class was referenced explicitly in the script, so it is the only one imported. If you refer to StateManager, StateMap, etc. you will need to import those classes as well.

Recipe: Update the map of key/value pairs

Use Case: The script wants to update the state map with a new Map of key/value pairs.

Approach: To get the current StateMap object, again use the getStateManager() method from ProcessContext, then getStateMap() from the StateManager. The examples assume a new Map, but using the above recipe (with the toMap() method), you could create a new Map using the existing values, then just update the desired entries. Note that replace() will not work if there is no current map (i.e. the StateMap.getVersion() returns -1), so the examples check and call setState() or replace() accordingly. When running from a new instance of ExecuteScript, the StateMap version will be -1, so after a single execution, if you right-click on the ExecuteScript processor and choose View State, you should see something like this:



Examples:

*Groovy*

import org.apache.nifi.components.state.Scope

def stateManager = context.stateManager

def stateMap = stateManager.getState(Scope.CLUSTER)

def newMap = ['myKey1': 'myValue1']

if (stateMap.version == -1) {

stateManager.setState(newMap, Scope.CLUSTER);

} else {

stateManager.replace(stateMap, newMap, Scope.CLUSTER);

}

*Jython*

from org.apache.nifi.components.state import Scope

stateManager = context.stateManager

stateMap = stateManager.getState(Scope.CLUSTER)

newMap = {'myKey1': 'myValue1'}

if stateMap.version == -1:

stateManager.setState(newMap, Scope.CLUSTER)

else:

stateManager.replace(stateMap, newMap, Scope.CLUSTER)

*Javascript*

var Scope = Java.type('org.apache.nifi.components.state.Scope');

var stateManager = context.stateManager;

var stateMap = stateManager.getState(Scope.CLUSTER);

var newMap = {'myKey1': 'myValue1'};

if (stateMap.version == -1) {

stateManager.setState(newMap, Scope.CLUSTER);

} else {

stateManager.replace(stateMap, newMap, Scope.CLUSTER);

}

*JRuby*

java\_import org.apache.nifi.components.state.Scope

stateManager = context.stateManager

stateMap = stateManager.getState(Scope::CLUSTER)

newMap = {'myKey1'=> 'myValue1'}

if stateMap.version == -1

stateManager.setState(newMap, Scope::CLUSTER)

else

stateManager.replace(stateMap, newMap, Scope::CLUSTER)

end

Recipe: Clear the state map

Use Case: The script wants to clear all the key/value pairs for the state map.

Approach: Use the getStateManager() method from ProcessContext, then call StateManager's clear(scope) method.

Examples:

*Groovy*

import org.apache.nifi.components.state.Scope

context.stateManager.clear(Scope.LOCAL)

*Jython*

from org.apache.nifi.components.state import Scope

context.stateManager.clear(Scope.LOCAL)

*Javascript*

var Scope = Java.type('org.apache.nifi.components.state.Scope');

context.stateManager.clear(Scope.LOCAL);

*JRuby*

java\_import org.apache.nifi.components.state.Scope

context.stateManager.clear(Scope::LOCAL)

*Accessing Controller Services*

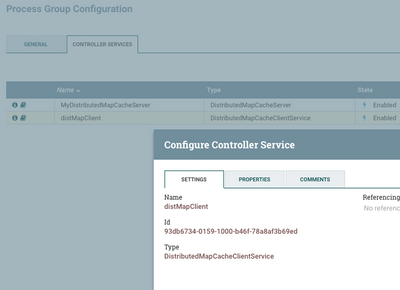
In the NiFi ARchive (NAR) structure, Controller Services are exposed as interfaces, usually in an API JAR. For example, the DistributedCacheClient is an interface extending from ControllerService, and it resides in the nifi-distributed-cache-client-service-api JAR, which is located in the nifi-standard-services-api-nar NAR. Other NARs that wish to reference interfaces (to create a new type of client implementation, e.g.) must specify the nifi-standard-services-api-nar as its parent NAR, and it then refers to provided instances of the API JARs in the processor submodule.

That's a little more low-level detail than you need to be able to leverage Controller Services, but I mention it for two reasons:

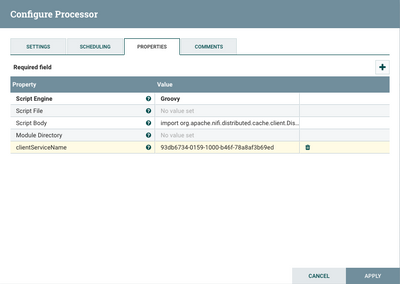
1. Prior to NiFi 1.0.0, the scripting NAR (which includes ExecuteScript and InvokeScriptedProcessor) did not specify nifi-standard-services-api-nar as its parent. That meant that only implicit references could be used for ControllerServices interfaces (and their implementations), and for the same reason, only interface methods that do not require other unavailable classes can be used. This limits the usefulness of ExecuteScript in terms of using Controller Services, but for a working example you can read [my blog post](http://funnifi.blogspot.com/2016/04/sql-in-nifi-with-executescript.html).
2. As of NiFi 1.0.0, the scripting processors do have access to *some* of the Controller Service interfaces (and associated classes) in the nifi-standard-services-api-nar. These include DBCPService, DistributedMapCacheClient, DistributedSetCacheClient, HttpContextMap and SSLContextService. However I don't believe the other APIs included in the nifi-standard-services-api-nar will be available, and no custom ControllerService interfaces will be recognized. Having said that, you may be able to use the approach from the blog post in #1 to do so.

Processors that always intend on using a Controller Service instance create a property (i.e. PropertyDescriptor object) and call identifiesControllerService(*class*) on it. When the UI component is rendered, it will find all the Controller Services that implement the desired interface, and under-the-hood the component's ID is used, but its friendly name is displayed to the user.

For ExecuteScript, we can let the user choose a Controller Service instance by letting them specify the name or ID. If we allow the user to specify the name, then the script will have to perform a lookup trying to match the name to a (single!) element in the list of Controller Service instances of that type. That technique is described in the blog post above, so I won't cover it here. If the user enters the ID of the instance, then (as of NiFi 1.0.0) it is much easier to get access to the object as we will see below. The examples will use a DistributedMapCacheClientService instance called "distMapClient", connecting to a DistributedMapCacheServer instance (with standard defaults, localhost:4557), where the client instance has an ID of 93db6734-0159-1000-b46f-78a8af3b69ed:



In the ExecuteScript config, a dynamic property is created, called "clientServiceId" and set to 93db6734-0159-1000-b46f-78a8af3b69ed:



We can then use clientServiceId.asControllerService(*DistributedMapCacheClient*), where the parameter is a reference to the Class object for DistributedMapCacheClient. For the examples, I have pre-populated the cache with a String key 'a' set to the String value 'hello'. For a handy Groovy script to work with a DistributedMapCacheServer, check out [my article here](https://community.hortonworks.com/articles/71837/working-with-a-nifi-distributedmapcache.html).

Once we have a DistributedMapCacheClient instance, then to retrieve a value we can call its get(*key*, *serializer*, *deserializer*) method. In our case since the keys and values are Strings, we just need an instance of Serializer<String> and Deserializer<String> to pass into the get() method. The approach for all the languages is similar to the creation of the StreamCallback instances described in Part 2 of this article series. The examples will get the value for key 'a' from the pre-populated server and log the result ("Result = hello")

Recipe: Get the value of a property stored in a DistributedMapCacheServer

Use Case: A user has populated values into a DistributedMapCacheServer (configuration data, e.g.) and the script wants access to them.

Approach: Use the approach described above, creating a StringSerializer and StringDeserializer object, then getting the DistributedMapCacheClientService instance by ID, then calling get() on the service. Logging of the result is included here for convenience.

Examples:

*Groovy*

import org.apache.nifi.distributed.cache.client.DistributedMapCacheClient

import org.apache.nifi.distributed.cache.client.Serializer

import org.apache.nifi.distributed.cache.client.Deserializer

import java.nio.charset.StandardCharsets

def StringSerializer = {value, out -> out.write(value.getBytes(StandardCharsets.UTF\_8))} as Serializer<String>

def StringDeserializer = { bytes -> new String(bytes) } as Deserializer<String>

def myDistClient = clientServiceId.asControllerService(DistributedMapCacheClient)

def result = myDistClient.get('a', StringSerializer, StringDeserializer)

log.info("Result = $result")

*Jython*

from org.python.core.util import StringUtil

from org.apache.nifi.distributed.cache.client import DistributedMapCacheClient, Serializer, Deserializer

# Define a subclass of Serializer for use in the client's get() method

class StringSerializer(Serializer):

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

pass

def serialize(self, value, out):

out.write(value)

# Define a subclass of Deserializer for use in the client's get() method

class StringDeserializer(Deserializer):

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

pass

def deserialize(self, bytes):

return StringUtil.fromBytes(bytes)

myDistClient = clientServiceId.asControllerService(DistributedMapCacheClient)

result = myDistClient.get('a', StringSerializer(), StringDeserializer())

log.info('Result = ' + str(result))

*Javascript*

var DistributedMapCacheClient = Java.type('org.apache.nifi.distributed.cache.client.DistributedMapCacheClient');

var Serializer = Java.type('org.apache.nifi.distributed.cache.client.Serializer');

var Deserializer = Java.type('org.apache.nifi.distributed.cache.client.Deserializer');

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

var StringSerializer = new Serializer(function(value, out) {

out.write(value.getBytes(StandardCharsets.UTF\_8));

})

var StringDeserializer = new Deserializer(function(arr) {

// For some reason I had to build a string from the character codes in the "arr" array

var s = "";

for(var i = 0; i < arr.length; i++) {

s = s + String.fromCharCode(arr[i]);

}

return s;

})

var myDistClient = clientServiceId.asControllerService(DistributedMapCacheClient.class);

var result = myDistClient.get('a', StringSerializer, StringDeserializer);

log.info("Result = "+ result);

*JRuby*

java\_import org.apache.nifi.distributed.cache.client.DistributedMapCacheClient

java\_import org.apache.nifi.distributed.cache.client.Serializer

java\_import org.apache.nifi.distributed.cache.client.Deserializer

java\_import java.nio.charset.StandardCharsets

# Define a subclass of Serializer for use in the client's get() method

class StringSerializer

include Serializer

def serialize(value, out)

out.write(value.to\_java.getBytes(StandardCharsets::UTF\_8))

end

end

# Define a subclass of Deserializer for use in the client's get() method

class StringDeserializer

include Deserializer

def deserialize(bytes)

bytes.to\_s

end

end

myDistClient = clientServiceId.asControllerService(DistributedMapCacheClient.java\_class)

result = myDistClient.get('a', StringSerializer.new, StringDeserializer.new)

log.info('Result = ' + result)