Scripting from scratch: Creating a Jython administrative script for IBM WebSphere Application Server

Skill Level: Intermediate

Bob Gibson (bgibson@us.ibm.com)
Advisory Software Engineer
IBM

07 Apr 2010

Developing IBM® WebSphere® Application Server administrative scripts from scratch is not difficult, but practical information to step you through the process is difficult to come by. This article helps you do just that, beginning with an idea of what you want your script to do, and then proceeding through iterative development steps that result in a complete sample script that includes comments, usage information, and command line parameter processing.

Introduction

Even before writing a book about WebSphere Application Server Administration Using Jython, I was aware of how few examples exist that describe how you might go about creating a complete script. Therefore, I thought it might be helpful for aspiring wsadmin script writers if I described the process that I have used many times to create such scripts.

In general, these are the high level steps you need to perform to create a wsadmin script from scratch:

1. Decide what administrative task you want to accomplish.
2. Review the documentation to learn about the task and the various methods you can use.
3. Identify the method parameters and understand how to use them.

4. Use an interactive wsadmin session to verify and experiment with the method, its use, and its parameter values.

5. Using a script template, determine the best technique for providing method parameter values to the script (for example, in a property file or via command line parameters).

6. Add the parameter processing code and usage details to the script.

7. Add code to execute the actual script method(s) that will perform the desired task.

For demonstration purposes, this article describes the steps you would perform to build a script that enables you to create cluster members. That’s a pretty straightforward task, but like mosts tasks, there can be factors that could make the task more complicated. For example, should the script be able to create just a single cluster member, or would you also want the ability to use it to create more than one member at a time?

Following the steps described in this article, you’ll start out simply to ultimately create a sample script that lets you create a single cluster member.

This article applies to both IBM WebSphere Application Server V7.0 and V6.1.

Decide on an approach

As you begin, the idea is to first find information about the task you are interested in so you can understand the process and the parameters that are involved. This will enable you to create a script that performs the task in a proper and efficient manner.

For example, using a Web browser, navigate to the IBM WebSphere Application Server V7 Network Deployment Information Center and search for the phrase create cluster member. The result is encouraging because items titled Creating cluster members using scripting display at the top of the search results. Select the first such link to display the related documentation:

- The Before you begin section explains that there are multiple approaches you can take to do this job. For example, to create new cluster members, you can:
  - Use the AdminConfig scripting object.
• Use the ClusterConfigCommands command group in the AdminTask scripting object.

• Use the createClusterMember method in the AdminClusterManagement scripting library.

• Further, the About this task section raises some interesting questions, such as:
  • How general do you want (or need) the script to be?
  • Do you want the script to be able to create the first member of a cluster, just subsequent members of the cluster, or both?
  • Does the cluster need to already exist in order for the script to succeed, or do you want (or need) the script to be able to create a new cluster as part of the process?

Let’s take a look at each of these approaches before deciding how to proceed.

Using the AdminConfig scripting object

Because it’s clear that you’ll be using some sort of AdminConfig scripting object for this task, it’s a good idea to start by searching the Information Center for information about it.

Searching for AdminConfig produces several results, including one at the top of the results list called Commands for the AdminConfig object using wsadmin scripting. Reviewing this item reveals that a createClusterMember method exists on the AdminConfig scripting object. Looking further, you’ll find that this method has some required parameters. Take a quick look at the examples farther down that page to better understand how they are used. Interestingly enough, the list of required parameters (shown in Table 1) doesn’t match the sample code.

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clusterID</td>
<td>Configuration ID for the cluster to which the member is to be added.</td>
</tr>
<tr>
<td>nodeID</td>
<td>Configuration ID for the node on which the member is to be created.</td>
</tr>
<tr>
<td>memberAttributes</td>
<td>Specifies the attributes to be used to create the new member.</td>
</tr>
<tr>
<td>templateID</td>
<td>Configuration ID for the template to be used by the member creation process.</td>
</tr>
</tbody>
</table>

Table 1. AdminConfig.createClusterMember() required parameters

How is it that the templateID parameter can be left off the sample code if it is really required? The templateID parameter is actually only permitted for the first member of the cluster.
the cluster. The creation of subsequent cluster members uses the first member as a template.

About the sample scripts provided
Using the scripts provided enables you to maintain focus on the process of creating the desired script rather than the mechanics of creating the required cluster containing a single member. This article acknowledges that a cluster (Cluster1) and its first member (Member1) were created previously. Also, since these scripts were created using the processes described in this article, you can also use them as additional examples for study purposes.

In order to create a cluster member, though, you’re first going to have to create a cluster, as well as the first cluster member. To help you do that, you can use these scripts, which are included in the download materials accompanying this article (and which were created using the techniques described here):

- **createCluster.py**: Creates an empty cluster.
- **createFirstClusterMember.py**: Creates the first cluster member using a user specified template name.

Now, let’s use an interactive wsadmin session to see what steps would be needed to use the `AdminConfig.createClusterMember()` method to create a new cluster member. Listing 1 shows this interactive session, and Table 2 explains the session in detail.

### Listing 1. Using `AdminConfig.createClusterMember()`

```
1 [root@ragdoll bin]#/wsadmin.sh -lang jython
2 WASX7209I: Connected to process "dmgr" on node ragdollCellManager02 using SOAP
3 connector; The type of process is: DeploymentManager
4 WASX7031I: For help, enter: "print Help.help()"
5 |
6 wsadmin> print AdminConfig.list( 'ClusterMember' )
7 Member1(cells/ragdollCell02/clusters/Cluster1|cluster.xml#ClusterMember_1261105354124)
8 |
9 wsadmin> clusterID = AdminConfig.getid('/ServerCluster:Cluster1/')
10 |
11 wsadmin> nodeID = AdminConfig.getColor( '/Node:ragdollNode03/' );
12 |
13 wsadmin> memberID = AdminConfig.createClusterMember(clusterID, nodeID, '[memberName Member2]')
14 |
15 wsadmin> print AdminConfig.list( 'ClusterMember' )
16 Member1(cells/ragdollCell02/clusters/Cluster1|cluster.xml#ClusterMember_1261105354124)
17 Member2(cells/ragdollCell02/clusters/Cluster1|cluster.xml#ClusterMember_1261145939139)
```

### Table 2. Using `AdminConfig.createClusterMember()` explained

<table>
<thead>
<tr>
<th>Line(s)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start wsadmin and specify the language as Jython.</td>
</tr>
<tr>
<td>2-4</td>
<td>Information generated by wsadmin showing a successful connection to a deployment manager has occurred.</td>
</tr>
<tr>
<td>5</td>
<td>A blank line added for readability.</td>
</tr>
</tbody>
</table>
Here is what a script would need to do to create a single cluster member using this technique:

1. Verify the command line parameters.
2. Lookup the cluster configuration ID, given the clusterName command line parameter. (The script should ask the user to specify the cluster and node names, but not the configuration IDs.)
3. Lookup the node configuration ID, given the nodeName command line parameter.
4. Possibly verify that the specified memberName does not already exist in the cluster.
5. Execute the createClusterMember method to perform the desired action.
6. Verify the success or failure of the request.

Of course, this is just a rough approximation of the process, but it gives you an idea about the kind of coding that is required to use this particular technique. But before you invest any time and effort implementing a script that uses this approach, let’s take a look at the alternatives to see how different they are, and if one might be better or easier to implement than the others.

**Using the AdminTask scripting object**

Search the Information Center again, this time for the term **AdminTask**. With more than 200 results, you have to scroll down a bit to find an item called **ClusterConfigCommands command group for the AdminTask object**. Selecting this item provides a link to the createClusterMember method, which is described further down the page. This section helps you better understand how to use this method:

- To create the first cluster member, you need to either:
• Specify an application server template name (if you specify the template name, you don’t have to lookup the configuration ID because the AdminTask scripting object will do this for you).

• Specify an existing application server that should be used as a template.

• You can specify either:
  - The name of the cluster.
  - The configuration ID of the cluster to which the member is to be added.

• You can have cluster members at different WebSphere Application Server versions (V6.1 and V7.0), but this scenario is beyond the scope of this article.

To make sure that your comparison of the different approaches is valid, be sure that the actual task you’re comparing is the same one in all approaches, which in this case is how to create a script that enables you to create additional cluster members. This means you can ignore the template-related parameters and focus on just those parameters related to the creation of subsequent cluster members.

If you are going to have your script use scripting object methods to create cluster members, then you need understand the parameters that are available. Because the documentation on this can be confusing, try using interactive wsadmin sessions to get a better understanding.

One of the really powerful and useful things about the AdminTask scripting object is that almost every method includes a way to "step through" the parameter definitions. You do this by calling the method and specifying -interactive as the only parameter. Use this technique now to learn the difference between using the cluster configuration ID and the cluster name, as they relate to the createClusterMember() method.

Start by calling the method, similar to what you see on Line 1 of Listing 2a. (Because of the number of lines generated during this wsadmin session, it has been divided here into Listings 2a through 2e.)

**Listing 2a. Interactive cluster member creation**

```python
wsadmin> M1 = AdminTask.createClusterMember( '-interactive' )
Create Cluster Member
Creates a new member of an application server cluster.
Cluster Object ID: Cluster1(cells/ragdollCell102/clusters/Cluster1|cluster.xml#ServerCluster_1261065198076)
```
The result of the interactive call is assigned to a variable (Listing 2a, Line 1) because when you are done with your interactive parameter specification, the resulting call to AdminTask.createClusterMember() will be the actual call you want to make, including all the parameters you eventually specify. The result of the call will be the configuration ID of the newly created cluster member.

Line 6 is where the AdminTask scripting object prompts you for information, telling you that it is looking for the value of Cluster Object ID (that is, the configuration ID of the cluster to which a new member is being added). Because you want to find out what this command looks like when you specify a configuration ID, copy and paste the complete configuration ID of the cluster in response to this prompt. (An exception will occur if an invalid or unknown configuration ID is specified, and control will return to the wsadmin> prompt. The AdminTask interactive technique doesn’t support or permit the use of variable names, so values like the configuration ID shown here must be explicitly specified.)

Continuing with this interactive session in Listing 2b, you are prompted for the name of the cluster in which the member is to be created. Because you already identified the cluster by its configuration ID, you don’t need to specify the cluster name. Therefore, you can simply press Enter (Listing 2b, Line 1).

**Listing 2b. Interactive cluster member creation**

```
1| Cluster Name (clusterName):
2| Create Cluster Member
3| -> *1. Member Configuration (memberConfig)
4|  2. First Member Configuration (firstMember)
5|  3. Promote Proxy Server Settings To Cluster (promoteProxyServer)
6|  4. Configure the event service during cluster member creation. (eventServiceConfig)
11| S (Select)
12| N (Next)
13| C (Cancel)
14| H (Help)
17| Select [S, N, C, H]: [S]
```

This results in the display of the available steps as a submenu (Lines 6 through 9), the available input options (Lines 11 through 14), and the command prompt requesting your input (Line 17, with the default selection shown in square brackets).

The asterisk * next to step 1 indicates that input is required. Notice how the default command is S (for Select). If you press Enter, or the letter S followed by Enter, then the specified step will be preformed, and you will see the information shown in Listing 2c.

**Listing 2c. Interactive cluster member creation**
In Listing 2c, the selected step description and step name are displayed in Line 1. The current values for the current step attributes are shown in Lines 3 through 9. Again, required values are prefixed with an asterisk. Because the values of some required attributes have yet to be defined, the default command is **E for Edit** (Line 12). When you press **Enter** at this point, you will be prompted for the missing values that are required. Listing 2d shows these prompts and the responses that were provided (Lines 1 through 8). In this case, the only values that were entered were for the required attributes (Lines 1 and 2).

**Listing 2d. Interactive cluster member creation**

```
1  *Node Name (memberNode): ragdollNode03
2  *Member Name (memberName): Member2
3  Member Weight (memberWeight):
4  Member UUID (memberUUID):
5  Generate Unique HTTP Ports (genUniquePorts): [true]
6  enable data replication (replicatorEntry): [false]
7  Specific short name of cluster member (specificShortName):
8  Create Cluster Member
9  
10 Creates a new member of an application server cluster.
11 
12  1. Member Configuration (memberConfig)
13  2. First Member Configuration (firstMember)
14  3. Promote Proxy Server Settings To Cluster (promoteProxyServer)
15  4. Configure the event service during cluster member creation. (eventServiceConfig)
16  
17 S (Select)
18 N (Next)
19 P (Previous)
20 F (Finish)
21 C (Cancel)
22 H (Help)
23 
24 Select [S, N, P, F, C, H]: [F]
```

After the final response, the Create Cluster Member submenu is displayed again, this time indicating with an arrow (->) which step is next; in this case, step 2 (Line 13). The important thing to note is that since all required attribute values have been provided, a new command, **F (Finish)** is now available (Line 20) and is also the default command (Line 25).
Selecting the **Finish** command initiates the last phase of the interactive method and ultimately displays the generated command. The result is shown in Listing 2e.

**Listing 2e. Generated createClusterMember command**

```
WASX7278I: Generated command line: AdminTask.createClusterMember('Cluster1(cells/ragdollCell02/clusters/Cluster1|cluster.xml#ServerCluster_1261065198076)', '[-memberConfig [-memberNode ragdollNode03 -memberName Member2 -genUniquePorts true -replicatorEntry false] ]')
```

This a fairly lengthy command, so let's take it apart to make it a little easier to interpret. The basic form of the command is:

```
AdminTask.createClusterMember( TargetObject, Parameters )
```

The **TargetObject**, as you can readily see, is the configuration ID of the target cluster to which the new member will be added. The **Parameters** value is a little more complicated, but is a string in the form:

```
‘[-memberConfig [values]]’
```

You might remember seeing **memberConfig** before; it appeared in Listing 2b (Line 6), Listing 2c (Line 1), and in the Steps subsection of the Information Center documentation. You can now correlate the documentation to the interactive createClusterMember interactive session, and see how the pieces of the puzzle fall into place. You can also see that the generated command includes some of the default values (for example, genUniquePorts and replicatorEntry). Since they are default, you can decide later whether or not your script needs to provide these values.

At this point, you have used the interactive form of the createClusterMember() method to determine exactly how the command would look should you decide to use the cluster configuration ID. You can do the same thing to see how the command would look if you used the clusterName parameter instead. All you have to do is use the AdminConfig.reset() method to discard this newly created cluster member, and run the AdminTask.createClusterMember( ‘-interactive’ ) method again, only this time specifying the clusterName parameter instead of the cluster configuration ID. The result of doing this is shown in Listing 3.

**Listing 3. Alternate createClusterMember command**

```
WASX7278I: Generated command line: AdminTask.createClusterMember('[-clusterName Cluster1 -memberConfig [-memberNode ragdollNode03 -memberName Member1 -genUniquePorts true -replicatorEntry false] ]')
```
This form of the command looks like:

AdminTask.createClusterMember( Parameters )

The Parameters string of the form looks like:

`[-clusterName value -memberConfig [values]]`

In fact, the -memberConfig values are identical to what you saw earlier. This might help you better understand the on-line documentation where the -clusterName parameter was identified as required. (The description for the –clusterName parameter indicates that either the TargetObject or the –clusterName value should be specified. If both are actually provided, an exception is thrown, and the command is not executed.)

Hopefully, this exercise has helped you to understand the createClusterMember() method, its -interactive execution, and the documentation that supports it. You can use this same technique to create a second cluster member; you’ll find that the only difference from the command you just ran will be the name of the member to be created.

Having done this, you now know that the parameters your script would need to handle to use the AdminTask.createClusterMember() method to create an additional cluster member are:

- The name of the cluster to be changed.
- The name of the node on which the member is to be created.
- The name of the member to be created.

For the purpose of this article, and for simplicity, the default values will be accepted for all other values.

You should now have a fairly good understanding of what it would take to use this approach to create a script.

**Using the scriptLibraries**

The third approach you might consider for accomplishing the task of creating cluster members is using the createClusterMember method in the AdminClusterManagement scripting library. Because the scriptLibraries already exist in WebSphere Application Server V7, this approach might enable you to meet your requirement with a minimum of effort. Let’s take a look and see if indeed it will work for you.

Searching the Information Center for **AdminClusterManagement** will lead you to an
item called **Jython script library: Cluster configuration scripts using wsadmin scripting**, which describes some of the methods in that library module, including the createClusterMember method -- which appears to be exactly the kind of information for which you might be looking.

To test this, you need to have a deployment manager configuration that has at least one cluster with at least one member. Since this is the configuration you already have, it appears that you’re all set to try using this approach. Listing 4 shows a sample interactive wsadmin session used to test this situation, and Table 3 describes these steps in detail.

**Listing 4. Interactive wsadmin test of a scriptLibrary method**

```
C:\IBM\WebSphere\AppServer70\bin>wsadmin -lang jython
WASX7209I: Connected to process "dmgr" on node ragweedCellManager02 using SOAP connector; The type of process is: DeploymentManager
WASX7031I: For help, enter: "print Help.help()"
wsadmin>cluster, node, member = 'Cluster1', 'ragweedNode03', 'Member2'
wsadmin>AdminClusterManagement.createClusterMember( cluster, node, member )
AdminClusterManagement: Create a new cluster server member
Cluster name: Cluster1
Member node name: ragweedNode03
New member name: Member2
Usage: AdminClusterManagement.createClusterMember("Cluster1", "ragweedNode03", ...
Return: The configuration ID of the new cluster member.
----------------------------------------------------------------------
'Member2(cells/ragweedCell02/clusters/Cluster1|cluster.xml#ClusterMember_...)'
wsadmin>
```

**Table 3. Interactive wsadmin session explained**

<table>
<thead>
<tr>
<th>Line(s)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start wsadmin and specify the language as Jython.</td>
</tr>
<tr>
<td>2-4</td>
<td>Information generated by wsadmin showing a successful connection to a deployment manager.</td>
</tr>
<tr>
<td>5</td>
<td>Local variable assignment so the next line won’t be too long.</td>
</tr>
<tr>
<td>6</td>
<td>Call to the createClusterMember() method to create the new member.</td>
</tr>
<tr>
<td>7-16</td>
<td>Information generated by the method to detail what is being done. (Line 12 has been truncated here.)</td>
</tr>
<tr>
<td>17</td>
<td>Truncate result returned by the createClusterMember() method call.</td>
</tr>
</tbody>
</table>

It appears this method has both advantages and disadvantages:

- The most significant advantage is probably the simple interface. Using only three parameters, you can quickly and easily create a new member for an existing cluster. In fact, these parameters are identical to those
required by the AdminTask.createClusterMember() method you saw earlier.

- As far as disadvantages go, you might find the generation of the banner section (Listing 4, Lines 7 through 16) to be distracting or annoying. While this might be acceptable during script development, there doesn’t appear to be an easy way to disable these lines. If this is enough to keep you from using this method on a regular basis, there is an alternative you can consider that involves one of the other advantages of scripting libraries, which is the availability of the source code for you to review.

Working with the source

Finding the source code for a specific method is relatively easy. When a library module is used, you can add a __file__ attribute to the command so that the file from which this library was loaded will be identified. Listing 5 shows an example of how you can use this attribute.

Listing 5. Finding a library source file

```
1|...>wsadmin -lang jython -conntype none -c "print AdminClusterManagement.__file__"
2|WASX7357I: By request, this scripting client is not connected to any server process.
3|Certain configuration and application operations will be available in local mode.
4|C:\IBM\WebSphere\AppServer70\scriptLibraries\servers\V70\AdminClusterManagement.py
```

Of course, you shouldn’t modify these script library files directly. Make a copy of any library file that you want to view, and then edit only your copy.

Examining the script source code, you can learn a number of things, such as:

- How to verify parameter values.
- Which WebSphere Application Server scripting object method calls can be used.
- How to provide parameters for these scripting object methods.
- How to set up error handlers.

Checking parameters

Looking closely at the AdminClusterManagement.createClusterMember() method can give you some insight on how to verify parameter values. For example, you might notice that no default value is provided for any of the first three parameters (clusterName, nodeName, or newMember). Therefore, you must specify a value for each when the function is called. This is apparent when you review the code because, except for the method banner that displays information about the method and how it was called, the values of these parameters are checked to see if they
contain an empty string. Listing 6 shows some of this code. (The specific details of
the parameters provided to the AdminUtilities._formatNLS() method simply provide
values to identify the parameter value with which an error was found, and aren’t
required in order to understand how the parameter checking actually occurs.)

Listing 6. Parameter checking

```python
# check the required arguments
if (clusterName == ""):  
  raise AttributeError(AdminUtilities._formatNLS(...))

if (nodeName == ""):  
  raise AttributeError(AdminUtilities._formatNLS(...))

if (newMember == ""):  
  raise AttributeError(AdminUtilities._formatNLS(...))

# check if cluster exists
cluster = AdminConfig.getid("/ServerCluster: " +clusterName+"/")
if (len(cluster) == 0):
  raise AttributeError(AdminUtilities._formatNLS(...))

# check if node exists
node = AdminConfig.getid("/Node:"+nodeName+"/")
if (len(node) == 0):
  raise AttributeError(AdminUtilities._formatNLS(...))

# check if cluster exists
cluster = AdminConfig.getid("/ServerCluster: /")
if (len(cluster) == 0):
  raise AttributeError(AdminUtilities._formatNLS(...))

# check if node exists
node = AdminConfig.getid("/Node:"+node+"/")
if (len(node) == 0):
  raise AttributeError(AdminUtilities._formatNLS(...))
```

Are these checks necessary -- or sufficient -- for your environment? That will depend
on your requirements and how thorough you need to be; for example, do you need
to check for:

- A value of None.
- A data type other than string.
- Whitespace-only values.

The scripting library modules raise a ScriptLibraryException when a parameter error
occurs. If you don’t want this to occur, you must decide what you want to have
happen when an error is detected.

The code then determines whether the specified values are valid. You could,
however, make some improvements to the code shown in Listing 6. For example,
Line 2 shows a check that determines if an empty string has been provided. If not,
Lines 12 and 13 check to see if the specified cluster exists. A problem with this code
occurs when a clusterName containing a single blank is specified; the empty string
check in Line 2 returns false, and the result of the call to the AdminConfig.getid(  
"/ServerCluster: /") method will be a list of all cluster configuration IDs. Also, the test
in Line 13 will fail to detect an error because the result of the call will not be an
empty string.
Making the call

When the parameter checking is complete, all that remains is the actual call to the AdminTask.createClusterMember() method. Having recently investigated this call, it should be easy to understand the statement itself:

```python
clusterMember = AdminTask.createClusterMember(['-clusterName', clusterName, '-memberConfig', ['-memberNode', nodeName, '-memberName', newMember]])
```

As you might know, the Jython wsadmin scripting object methods can pass either a list of strings (as you see here) or a string of lists (which you saw earlier). The difference is in the syntax. If you look closely at the statement above, you will see that all of the individual values are either literal strings (surrounded by single quotes) or variable names, all separated by commas. This enables you to use variable names (such as clusterName, nodeName, and newMember) at those places where you want to use the value of a variable.

```python
Parameters = '[-clusterName ' + clusterName + ' –memberConfig [-memberNode ' + nodeName + ' –memberName ' + newMember + ']]'
```

If you prefer not to use a long string concatenation statement like this, an alternative is to use the string formatting operator which enables you to have an expression like:

```python
FormatString % ( values )
```

The string format operation involves the processing of the FormatString, looking for format specification sequences, and using these sequences to format the associated data from the right operand. For any other text in the format string, the data is copied as is to the result string. For this simple example, you can use the format specification sequence %s in your FormatString to indicate that a string substitution should occur. These characters will be replaced by the string representation of the next value in the tuple (an ordered sequence of values enclosed in parentheses and separated by commas). To build this same parameter string using string formatting instead of string concatenation, then, would look something like this:

```python
Parameters = '[-clusterName %s –memberConfig [-memberNode %s –memberName %s]]' % ( clusterName, nodeName, newMember )
```

All of this explanation, however, is just background. What it enables you to do is simplify the actual method call to something like this:

```python
memberID = AdminTask.createClusterMember( Parameters )
```

This is the real heart of the script you should be trying to create.
Choosing a technique

Now that you have investigated the various approaches available for performing the task you want to accomplish in your script, it’s time to make a decision. Which method do you want to use:

- AdminConfig.createClusterMember() method.
- AdminTask.createClusterMember() method.
- AdminClusterManagement.createClusterMember() scriptingLibrary method.

If you choose the first technique, then you need to provide the method with configuration IDs for both the cluster and the node. If you choose the third technique, then the scriptingLibrary banner will be displayed when the method is called. It appears, then, that the better option is the second technique.

In general, when choosing between an AdminTask method and some other method to perform a specific action, it is usually better to go with the AdminTask method, because AdminTask methods tend to do more things for the script writer, resulting in more robust scripts that are easier to write.

Now that you know the actual command (or method) to call to perform the desired action, you can now create a script to use this command. One more thing that you have to decide is how robust you want the script to be. In other words, is it okay if your script performs only a minimal amount of checking, or must it check and verify absolutely everything? The answer depends on how much risk you can (or should) accept.

One of the first things you can check is whether the script was executed or imported. You can make this determination in Jython by checking the value of a special global variable called __name__. If the script was executed, the value of this variable will be __main__ and if it was imported, the value will be name of the imported file. Therefore, to verify that the script was executed, your script file should contain a simple test that looks something like that shown in Listing 7.

Listing 7. Checking whether script was executed or imported

```python
if (__name__ == '__main__') :
    # The script was executed
else :
    # The script was imported
```

The approach we’re taking here is to build up the script as you go along, so you need to decide what you want the script to do next. Add these items to the code
shown in Listing 7, continuing your script:

- A comment block (prologue) describing the script.
- An empty function as a placeholder for the code that will actually perform the desired work.
- A Usage() function that you should start filling out.

The result is shown in Listing 8.

**Listing 8. Iteration 1**

```python
#---------------------------------------------------------------------
# Name: createClusterMember.01.py
# Role: Example script, created from scratch.
# Author: Robert A. (Bob) Gibson
# Iteration: 1
# What's new? Verify that the script was executed, not imported, and start populating the Usage information
#---------------------------------------------------------------------
import sys

#---------------------------------------------------------------------
# Name: createClusterMember()
# Role: Placeholder for routine to perform the desired action
#---------------------------------------------------------------------
def createClusterMember() :
    print 'createClusterMember() - iteration 1'

#---------------------------------------------------------------------
# Name: Usage()
# Role: Routine used to provide user with information necessary to use the script.
#---------------------------------------------------------------------
def Usage( cmdName = None ) :
    if not cmdName :
        cmdName = 'createClusterMember'

    print '''Command: %(cmdName)s
Purpose: wsadmin script used to create an additional member to an existing cluster.
Usage: %(cmdName)s [options]
Example: ./wsadmin.sh -lang jython -f %(cmdName)s.py ...'''
locals();

#---------------------------------------------------------------------
# This is the point at which execution begins
#---------------------------------------------------------------------
if ( __name__ == '__main__' ) :
    createClusterMember();
else :
    print 'Error: this script must be executed, not imported.';
    Usage(__name__);
```

Save your script so that you can test it; name it `createClusterMember.py`. (UNIX® systems have a symbolic link facility where the filename can actually point to the current iteration; for example, `createClusterMember.01.py`.)
The test is shown in Listing 9. It shows that the script successfully determined whether it was executed or imported, and displays either a message indicating that things are OK (Line 3) or the usage information (Lines 11 through 18).

**Listing 9. Test iteration 1**

```
1 [root@ragdoll bin]# ./wsadmin.sh -lang jython -f createClusterMember.py
2 WASX7209I: Connected to process "dmgr" ...
3 createClusterMember() - iteration 1
4 [root@ragdoll bin]# ./wsadmin.sh -lang jython
5 WASX7209I: Connected to process "dmgr" ...
6
7 wsadmin>import createClusterMember
8 Error: this script must be executed, not imported.
9 Command: createClusterMember
10 Purpose: wsadmin script used to create an additional member to an existing cluster.
11 Usage: createClusterMember [optios]
12 Example: ./wsadmin.sh -lang jython -f createClusterMember.py ...
13 wsadmin>
```

**Processing the command line parameters and options**

One of the more challenging aspects of writing scripts is handling command line parameters. One frequently used technique is having the command line parameters be position-dependent. This means that you need to know that the first parameter is supposed to represent clusterName, the second parameter is supposed to represent nodeName, the third parameter is supposed to be the name of the member to be created, and so on.

One of the really good things about Jython is that it comes with library routines that can make your life easier, and your scripts better and more user friendly. One such library is the `getopt` library, which is based on work that was done long ago for the C programming language to enable programs written in C to easily process command line parameters.

There are a few things you need to do so that you can use the getopt library. First, you must select **short form parameter letters** and **long form parameter names** for each parameter value. With a short form (single letter) option, you can execute your script like this:

```
... -f createClusterMember.py -X clusterName -Y nodeName -Z memberName
```

In the above command, X, Y, and Z are used as placeholders. Identify convenient
letters that would help identify the parameters in the command. For example, -c for
clusterName, -n for nodeName, and -m for memberName. In this case, however, -c
is not available, as this is a command line option that is reserved for the wsadmin
utility and therefore is not available for scripts. Instead, use -L for the short form
command line option to specify the clusterName parameter value. (Using uppercase
avoids confusion with the number 1.)

Next, you need to tell the getopt utility routine to look for these options. To do this:

1. Import the getopt library module.
2. Invoke the getopt.getopt() function and pass it the required parameters.

The getopt() function requires these parameters:

- List of user-specified command line parameters.
- String identifying the short form options.
- List containing the long form options.

The short form string should include each valid option letter, followed by a colon (:) to
indicate that the command line option should be followed by a value. Remember
that the getopt library routine is general in nature, and not every command line
option is expected to have an associated value. Consider the situation where you
might want to have a command line option (for example, to enable debugging) that
doesn’t require a value. This could very well be indicated with an option parameter
something like ‘-X’.

Hopefully, you can see how this makes sense. In this case, then, you have decided
to have three short form option letters, each of which should be followed by a value.
Here is what that the shortForm option string would look like:

\[
\text{shortForm} = \text{'L:m:n:'};
\]

In this example, you want the getopt function to check for the existence of any of the
option letters in that string, and each should have a value. You don’t need to do
doing anything special to define the order in which these parameters occur, and you don’t
need to do anything special in the string provided to the getopt() function to indicate
how many times each option might occur. You will see shortly how to decide
whether or not this makes sense for your script.

Now, what about the long form command line options? Remembering that short form
command line options are each represented by a single character, preceded by a
single hyphen (-), and followed by an optional colon (:) which indicates that the
command line option will be followed by a value. Long form command line options
have a similar yet slightly different format. All long form options begin with two hyphens (---) and are represented by a variable number of alphanumeric characters. In this example, it makes sense to use the following long form option identifiers:

- clusterName
- nodeName
- memberName

You also need to tell the getopt() function that each of these options should be followed by a value. You can do this with a list of string values, each of which identifies the option identifier, followed by an equal sign. The longForm option list for this example will look something like this:

```python
longForm = [ 'clusterName=', 'nodeName=', 'memberName=' ];
```

Another, more compact way to represent the same expression is to use an assignment statement similar to this:

```python
longForm = 'clusterName=,nodeName=,memberName='.split( ',' );
```

Here, split() is a method that processes an input string (of the type string) that contains a specified delimiter character (in this case a comma) and returns a list of strings. The two assignment statements above, therefore, both create the same list of string values. You can use whichever one you find easier to read and understand.

Finally, the actual call to the getopt routine will look like this:

```python
getopt.getopt( sys.argv, shortForm, longForm );
```

That’s pretty slick, but how does the function tell you what options were actually encountered? The answer is that the function returns two things:

- A list of name/value pairs of valid options.
- A list of the remaining (unprocessed) options.

Listing 10 shows what should the code to handle this situation should look like.

### Listing 10. Using the getopt() function

```python
try :
    opts, args = getopt.getopt( sys.argv, shortForm, longForm );
except getopt.GetoptError :
    # code to handle the exception situation
for name, value in opts :
    # code to handle each value option letter/identifier
```
That appears pretty easy, except there are places where code has been left out (Lines 4, 7, 10):

- An exception will be thrown by the getopt() function when an unrecognized option is encountered, or if an option doesn’t contain a required value. The code in the exception handler (Line 4) should display an appropriate message and terminate the script.

- The code in the for loop (Line 7) is where you process each of the supported options. This is where your script would check for multiple occurrences of a command line option, if this is important to you.

- In the last section (Line 10), your script can check for and handle any extra information that was provided but isn’t associated with any of the valid options.

This information should help you to understand the parseOpts() routine shown in Listing 11. When you call this routine, a dictionary is returned that reflects the user-specified command line options.

Listing 11. The complete parseOpts() routine

```python
#---------------------------------------------------------------------
# Name: parseOpts()
# Role: Process the user specified (command line) options
# Returns: A dictionary containing the user specified values
#---------------------------------------------------------------------
def parseOpts( cmdName ) :
    shortForm = 'L:m:n:';
    longForm = 'clusterName=,memberName=,nodeName='.split( ',' );
    badOpt = '%(cmdName)s: Unknown/unrecognized parameter%(plural)s: %(argStr)s';
    optErr = '%(cmdName)s: Error encountered processing: %(argStr)s';
    try :
        opts, args = getopt.getopt( sys.argv, shortForm, longForm );
        print opts, args
        for name in longForm :
            if name[ -1 ] == '=' :
                name = name[ :-1 ];
            Opts[ name ] = None;
        Usage( cmdName );
        # Initialize the Opts dictionary using the longForm key identifiers
        #-----------------
        Opts = {};
        for name in longForm :
            if name[ -1 ] == '=' :
                name = name[ :-1 ];
            Opts[ name ] = None;
        # Process the list of options returned by getopt()
        #-----------------
```

```
Creating a Jython administrative script for IBM WebSphere Application Server
Page 20 of 28 © Copyright IBM Corporation 2010. All rights reserved.
```
for opt, val in opts:
    if opt in ( '-L', '--clusterName' ) : Opts[ 'clusterName' ] = val
    elif opt in ( '-m', '--memberName' ) : Opts[ 'memberName' ] = val
    elif opt in ( '-n', '--nodeName' ) : Opts[ 'nodeName' ] = val

# Check for unhandled/unrecognized options

if ( args != [] ) : # If any unhandled parms exist => error
    argStr = ' '.join( args );
    plural = '';
    if ( len( args ) > 1 ) : plural = 's';
    print badOpt % locals();
    Usage( cmdName );
    return Opts;

return Opts;

The complete script, including the parseOpts() routine above, is included in the createClusterMember.02.py file included in the download materials.

Verifying and using the command line options

At this point, you have a nice little function that checks the user-specified command line parameters and returns the result in a dictionary. But what do you do with it now?

First, understand that the parseOpts() function returns a dictionary result only when no error condition is detected – meaning that the parseOpts() routine didn’t encounter any problems for those conditions that it checked, which doesn’t necessarily mean that no problems exist.

Listing 12 shows how the second iteration of this script processes these parameters.

**Listing 12. Iteration 2: Calling the parseOpts() routine**

```python
missingParms = '%(cmdName)s: Insufficient parameters provided.';

#-------------------------------------------------------------------
# How many user command line parameters were specified?
#-------------------------------------------------------------------
argc = len( sys.argv );
if ( argc < 3 ) :
    print missingParms % locals();
    Usage( cmdName );
else :
    Opts = parseOpts( cmdName );
```

In Listing 12, the code shows how the parseOpts() routine is only called if three or more parameters were specified; three is the minimum number of parameters that should be processed by the getopt() routine:
The print statement in Line 8 is another of those string formatting operations that was discussed earlier. The difference here is that the format string must identify, with each format specification, which dictionary entry will be used to provide the substitution. Therefore, the format string containing \%(cmdName)s will be replaced by the local variable named \texttt{cmdName}, and the value will be displayed as a string.

This is a very powerful idiom, which is also used in the Usage() routine. For example, if you take a look at Listing 8, Lines 27-32, you will find another of these mapped messages. The interesting thing to note about the format string in this example is that the same local variable (cmdName) occurs multiple times in the same string.

What can you do with the dictionary returned by the parseOpts() routines? You can access the individual dictionary values using the corresponding index, but it would be much easier if you assigned the dictionary values to local variables. Compare the two statements in Listing 13 and see which is simpler and easier to understand.

**Listing 13. Which is easier to read and understand?**

```
if ( not Opts[ 'clusterName' ] ) :
    print 'Required option missing: clusterName'

if ( not clusterName ) :
    print 'Required option missing: clusterName'
```

An easy way to convert the dictionary returned by the parseOpts() routine into the corresponding local variables is to use a set of assignment statements (Listing 14).

**Listing 14. Simple conversion of dictionary entries to local variables**

```
clusterName = Opts[ 'clusterName' ];
nodeName = Opts[ 'nodeName' ];
memberName = Opts[ 'memberName' ];
```

This approach is fine provided the number of valid command line (long form) identifiers is small. But there is another way you can do this using the identifiers that exist in the dictionary, shown in Listing 15.

**Listing 15. Generic conversion of dictionary entries to local variables**

```
#-----------------------------
# Assign values from the user Options dictionary, to make value
# access simpler, and easier. For example, instead of using:
# Opts[ 'clusterName' ]
# we will be able to simply use:
# clusterName
```

[ `'-LclusterName', `'-nNodeName', `'-mMemberName' ]
# to access the value. Additionally, this allows us to make use of
mapped error messages that reference local variable values.

for key in Opts.keys() :
    cmd = '%s=Opts["%s"]' % ( key, key );
    exec( cmd );

The key to this code segment is a string that gets created, which looks like the
assignment statements in Listing 14, and the exec() routine to execute the
assignment statement and perform the actual variable assignment.

At this point, all that this iteration of the script does is display a message that shows
the user-specified values. You can use this version of the script to test and verify
that the command line processing does what is expected of it.

Performing the desired action

All that remains to do is add code to check the user-specified values, and then use
these values to perform the desired action.

If one of the required values is None, this means that one of the command line
option was not specified. How could this occur? The code to process the command
line parameters in the parseOpts() function doesn’t check for duplicates, so if the
user specifies command line parameters such as:

    --clusterName=C1 -L C2 -L C3

then the value of the nodeName and memberName options will both be None. You
could modify the parseOpts() function to check for multiple occurrences of the same
command line option, but you’ll have to decide how valuable adding that kind of
check would be.

You might also consider checking for empty string values. A user could specify a
blank value surrounded by double quotes:

    ./wsadmin.sh -f createClusterMember.py -L " " -n " " -m " 

If you test iteration 2 of the script, the output could be something like:

    createClusterMember: --clusterName --nodeName --memberName

Since all of the values are required, you should perform the same checks on each
value. (You could even add checks to verify that the specified clusterName and
nodeName already exist, and that the specified memberName does not. These
checks are beyond the scope of this article, but you can always add these later, if
you decide they would be worthwhile.)
Another thing to realize is that you can add the checks to the same loop that you use
to create local variables from the Opts dictionary. Listing 16 shows what this
modified code would look like.

Listing 16. Adding checks for invalid required values

```
badReqdParam = '%(cmdName)s: Invalid required parameter: %(key)s.'

#-------------------------------------------------------------------
# Assign values from the user Options dictionary, to make value
# access simpler, and easier. For example, instead of using:
# Opts[ 'clusterName' ]
# we will be able to simply use:
# clusterName
# to access the value. Additionally, this allows us to make use of
# mapped error messages that reference local variable values.
#-------------------------------------------------------------------
for key in Opts.keys() :
    val = Opts[ key ];
    if ( not val ) or ( not val.strip() ) :
        print badReqdParam % locals();
        Usage();
    cmd = '%s=Opts["%s"]' % ( key, key );
    exec( cmd );
```

In Listing 16, Line 1 shows the creation of a new mapped error message that is
displayed when a value of None (or an empty string) is encountered. The modified
lines occur in Lines 13 through 16, where the value of the specified identifier is
retrieved.

The first part of the expression in Line 14 is used to check for a value of None. If the
value of the variable `val` is None, then the result of `( not val )` will be true, and the
error message will be displayed. The rest of the expression is used to see if the
value (without leading or trailing blanks) is an empty string. If it is, then the error
message will be displayed, the `Usage()` routine will be called, and the script will
terminate.

Calling AdminTask.createClusterMember()

Now that you have verified that the user has provided values for all of the required
parameters, all you have left to do is use them in your call to the
AdminTask.createClusterMember() method. You saw earlier how to build the
parameter string for the method, so it looks like you’re all set.

Well, almost. Be sure your script is protected from an error; if the method call fails, it
throws an exception. Listing 17 shows the part of the script where the actual
AdminTask.createClusterMember() method is called.

Listing 17. Calling AdmingTask.createClusterMember()
Call the AdminTask.createClusterMember() method using the command line parameter values.

```python
def createClusterMember(clusterName='myCluster', nodeName='node1', memberName='member1'):
    cmdName = 'cmdName'
    parms = '%(cmdName)s --clusterName %(clusterName)s --nodeName %(nodeName)s --memberName %(memberName)s';
    print parms % locals();
    try:
       Parms = '\[-clusterName %s -memberConfig [-memberNode %s -memberName %s]\]' %
                     ( clusterName, nodeName, memberName )
        AdminTask.createClusterMember(Parms);
        AdminConfig.save();
        print '%(cmdName)s success. Member %(memberName)s created successfully.' % locals();
    except:
        # An exception occurred. Convert the exception value to a string using the backtick operator, then look for the presence of a WebSphere message, which start with 'ADMG'. If one is found, only display the last part of the value string.
        val = `sys.exc_info()[ 1 ]`;
        pos = val.rfind( 'ADMG' )
        if pos > -1:
            val = val[ pos: ]
        print '%(cmdName)s Error. %(val)s' % locals();
```

Table 4 explains the code segment shown in Listing 17 to help you better understand what the code is trying to do, and how it goes about doing its job.

<table>
<thead>
<tr>
<th>Line(s)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-4</td>
<td>Comment block identifying the action to be performed.</td>
</tr>
<tr>
<td>5</td>
<td>Mapped message used to display the action being performed and the user specified values.</td>
</tr>
<tr>
<td>6</td>
<td>Display the mapped message using local variable values.</td>
</tr>
<tr>
<td>7-23</td>
<td>Call to the createClusterMember() method protected by an exception handler (for example, try/except block).</td>
</tr>
<tr>
<td>8</td>
<td>Mapped format string used to build the method parameter string.</td>
</tr>
<tr>
<td>9</td>
<td>Actual call to the createClusterMember() method.</td>
</tr>
<tr>
<td>10</td>
<td>If the createClusterMember() succeeds, you need to call AdminConfig.save() to save the modified configuration.</td>
</tr>
<tr>
<td>11</td>
<td>Print a message indicating the successful completion of the requested action.</td>
</tr>
<tr>
<td>12-18</td>
<td>Should an exception occur, the most likely statement to fail is the call to the createClusterMember() routine.</td>
</tr>
<tr>
<td>19</td>
<td>Call sys.exc_info() to find out what happened. Only the second value (the error value) is really important in this instance, so you only keep it. Use the backtic operators to convert the returned value to a string.</td>
</tr>
<tr>
<td>20</td>
<td>Call to rfind() to locate the last occurrence of the ADMG message number prefix in the string.</td>
</tr>
<tr>
<td>21-22</td>
<td>If the message prefix is present, discard all characters before it, and keep only the important error message generated by the exception.</td>
</tr>
</tbody>
</table>
Conclusion

This article showed an approach for creating a `wsadmin` Jython script to perform a specific IBM WebSphere Application Server administrative task; in this case, to create an additional member on an existing cluster. It described how to use Information Center documentation to learn about various options for performing the task, how to compare the options to determine the most practical approach, and stepped through building a sample script in manageable stages. This article also provided information on common cautions, better practices, how to test a script, and how to handle error conditions.

Hopefully, you find this information useful and valuable.

Read the author's book: WebSphere Application Server Administration Using Jython
## Downloads

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
<th>Download method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample scripts</td>
<td>createClusterMember.scripts.zip</td>
<td>16 KB</td>
<td>HTTP</td>
</tr>
</tbody>
</table>

Information about download methods
Resources

Learn

- WebSphere Application Server Administration Using Jython
- WebSphere Application Server support
- WebSphere Application Server Information Center (all versions)
- WebSphere Application Server Network Deployment (distributed) documentation
- Creating cluster members using scripting
- Commands for the AdminConfig object using wsadmin scripting
- ClusterConfigCommands command group for the AdminTask object.
- Jython script library: Cluster configuration scripts using wsadmin scripting
- IBM developerWorks WebSphere Application Server zone
- IBM developerWorks WebSphere

Get products and technologies

- Trial download: Get IBM WebSphere Application Server V7 (trial version)
- No-charge offering: Download WebSphere Application Server for Developers

About the author

Bob Gibson
Bob Gibson is an Advisory Software Engineer who has over a quarter century of experience in numerous software related roles at IBM, including: Application Programmer, Architect, Developer, Instructor, Technical Support Analyst, and Tester. He is currently a team leader for the technical support group responsible for IBM WebSphere Distributed Application Server. He holds both a Bachelor of Science degree in Engineering Science and a Master of Science degree in Computer Science from the University of Virginia.