Howto make your own plugins

Version : 1.0
Date : 19/08/2006
Author : ¨Patrick Germain Placidoux

SUMMARY

1 OBJECTIVE

This document will drive you on creating your own kikonf plugins,
and give you the necessary skills to adapt existing plugins to your own specific needs.

Kikonf supports two kind of plugins :

- Action plugins
- Extract plugins

As you know Kikonf deserves to support configuration for many softwares.

The purpose of a particular Action (or Exit) plugin is not to support the whole functionalities of the target software.
But to cover the most general used functionalities inspired from real life and production use cases.

We aware, that the functionalities covered by a given plugin, can appear restrictive, for you, and may need some extent, this is why:

- Kikonf makes it easy to create your own plugin
- Kikonf makes it easy to adapt its own plugin to yours.
- Will be pround to support your efforts in this process (questions at www.kikonf.com).
- Your plugins with whatever license they come, can be advertised at www.kikonf.com (*).

* On reserve of Kikonf organization approbation.

Note:
Before reading this document, you should have a global understanding about how Kikonf works.
For that, the reading of the Kikonf documentation and eventually the kikact and kikarc commands documentation is a good start.

2 INTRODUCTION

Kikonf supports two kind of plugins:

- Action plugins
- Exit plugins

From start a plugin comes into a zip package for instance myaction.zip (or myexit.zip).

The command to install a plugin is:
plug <PATH_TO_MY_ZIP>/myaction.zip

Note:
The plugin installation process is not the purpose of this document for more information see the command help (plug -h).

Once this command is issued, the plugin package is expanded under the general directory :
<KIKONF_INSTALL_DIR>/plugins

If the plugin is an Action plugin, it will be extracted under the directory :
<KIKONF_INSTALL_DIR>/plugins/actions
This last is called the plugin base directory (**PLUGIN_BASE_DIR**) for the Action plugins.

If the plugin is an Exit/Extract plugin, it will be extracted under the directory :
<KIKONF_INSTALL_DIR>/plugins/exit/extract
This last is called the plugin base directory **(PLUGIN_BASE_DIR)** for the Exit/Extract plugin.

Actually the only currently supported plugins for exit topic are Extract plugins.

2.1 STRUCTURE OF THE PLUGIN BASE DIRECTORY

- <SOFTWARE_CATEGORY>/
- <PLUGIN_NAME>/
- by/
- <WHO>

Syntax:
SOFTWARE_CATEGORY : an aka for a given software name.
For instance "was" would represent all the Actions provided to automate WebSphere Aplication Server ®.
Or "apa" would represent all the Actions provided to automate Apache HTTP Server ®.

PLUGIN_NAME : the plugin name that build "de facto" a new sub category.
For instance the name for Actions which configure JVMs may be "jvm",
or "datasrc for Actions which configures DataSources.

by : this directory with it affiliated WHO, is the key of the mechanism which allows more than one person or organization to provide their own plugins.
Kikonf is not stick to one single version for one action and can support any provider name as the WHO directory.

WHO : the provider name.

The Kikonf default plugins are provided under :
<PLUGIN_BASE_DIR>/<SOFTWARE_CATEGORY>/<PLUGIN_NAME>/by/**kikonf**

For instance:

<KIKONF_INSTALL_DIR>/plugins/**actions/**<SOFTWARE_CATEGORY>/<PLUGIN_NAME>/by/**kikonf**
for Action plugins.
<KIKONF_INSTALL_DIR>/plugins/**exit/extract**/<SOFTWARE_CATEGORY>/<PLUGIN_NAME>/by/**kikonf**
for Exit/Extract plugins.

Hence the Action plugin "jvm", for WebSphere ® is provided by Kikonf at :
<KIKONF_INSTALL_DIR>/plugins/actions/**was**/**jvm**/by/**kikonf**

Hence the Exit/Extract plugin "application", for WebSphere ® is provided by Kikonf at :
<KIKONF_INSTALL_DIR>/plugins/exit/extract/**was**/**application**/by/**kikonf**

If you want to provided your own version of the Action "jvm" for "was", you may store it here :
<KIKONF_INSTALL_DIR>/plugins/actions/**was**/**jvm**/by/**me**

If you want to provided your own version of the Exit/Extract plugin "application", for "was", you may store it here :
<KIKONF_INSTALL_DIR>/plugins/exit/extract/**was**/**application**/by/**me**

Note :
Even if you could do it yourself, **you do not have** to put your stuff directly under the plugin base directory once you content is zipped and provided to the plug command, it **will do it for you.**

2.2 HOW DOES KIKONF COMMANDS WORK WITH THE PLUGIN BASE DIRECTORY

As long as the plugin base directory structure is respected, Kikonf commands will always go along with it.

2.2.1 *Explicit calls*

To explicitly call you Action plugin (under <KIKONF_INSTALL_DIR>/plugins/actions/**was**/**jvm**/by/**me**) type:

> kikact was.jvm.by.me
or use the **"bal" tag : was.jvm.by.me** into your custom xml file (e.g.:c.xml) and call it like this:
> kikarc c.xml

Note:
For more information about the kikact and the kikarc commands refer to the kikact and kikarc doucments.

To explicitly call you Exit/Extract plugin
(under <KIKONF_INSTALL_DIR>/plugins/exit/extract/**was**/**application**/by/**me**) type:

> kikact -o extract -e was.jvm.by.me –name myapp
or use the **"bal" tag : was.jvm.by.me** into your custom xml file (e.g.:c.xml) and call it like this:
> kikarc c.xml -o extract

2.2.2 *Implicit calls*

As shown in the kikact and kikarc command documentation implicit calls like :
> kikact was.jvm
or using the "bal' tag : was.jvm into your custom xml file (e.g.:c.xml) like this:
> kikarc c.xml

will aslo work.

So Kikonf commands are able to locate provider plugin without mentioning the by.WHO clause.

To match the default provier, the commands look to a file at:

<PLUGIN_BASE_DIR>/<SOFTWARE_CATEGORY>/<PLUGIN_NAME>/by/default.txt

The default.txt contains just one the name, the default provider name.

By default this name is "kikonf".

If you switch this name to yours ("me"), you'll be able to make implicit calls to your own plugins, hence further calls to the Kikonf plugin would have to be explicit.

3 ACTION PLUGIN

3.1 INTRODUCTION

The following shows how an Action plugin is structured.

You can take a look at any of the actions plugin deployed under the
<ACTION_PLUGIN_BASE_DIR>/<SOFTWARE_CATEGORY>/<ACTION_NAME>/by/kikonf.
In the following, we'll refer this directory as the ACTION_BASE_DIR

For the purpose of this chapter will use the Action plugin "envars" provided for "was" at :
<ACTION_PLUGIN_BASE_DIR>/was/envars/by/kikonf (ACTION_BASE_DIR).

Note:
The kikact command with -p (or -P options) shows a (detailled) list -p of all the deployed plugins.

3.2 THE ACTION PLUGIN CONTENT

ACTION_NAME.py
<ACT_INF>/
    action.attrs
    action.xml
    LICENSE_FILE

ACTION_NAME.py : the python code file of the Action.

<ACT_INF> : the Action Info directory.

action.attrs : this file reports the identification attributes of this Action plugin.

action.xml : the action descriptor file.
This file describes the xml content supported by this action.
An action descriptor file can be seen as a WYSIWYG dtd.
Note:
For more information about the action descriptor file see the kikonf documentation.

LICENSE_FILE : the license for this product.

3.2.1 The action.attrs file

The action.attrs file contains just one line an attribute named "action", which value is a coolTyped python dictionary.
For more information about the CoolTyping tools see the dedicated CoolTyping documentation.
To say simple a coolTyped value is a python type value with no "" and no "".

The line syntax:
action =
{**category**:SOFTWARE_CATEGORY \
,**name**:<ACTION_NAME>,**by**:WHO,**license**:<LICENSE_FILE>,**multiple**:<MULTIPLE>,**version**:VERSION}

SOFTWARE_CATEGORY : the software to which is dedicated this Action plugin.

ACTION_NAME : the Action name.

WHO : the Action plugin provider name.

LICENSE_FILE : the license file name for this Action plugin.
This file must exists under the ACT_INF directory.

MULTIPLE : true or false (see the note below).

VERSION : The version of this Action plugin, may be a number or a string.

For instance the content of the action.attrs file for the sample Action plugin "envars" is:

action = {**category**:was,**name**:envars,**by**:kikonf,**license**:COPYING,**multiple**:true,**version**:5.0}

Note about multiple :
Here is a sample Action file for the Action envars :

```
<envars type = 'action' prefix='myprefix'>
    <scope node = 'localhostNode01' server = 'server1'/>

    <var name='myvar1' value='value1' required='false' desc='mydesc1'/>
    <var name='myvar2' value='value2' required='true' desc='mydesc2'/>
    <var name='myvar3' value='value3' required='false' desc='mydesc3'/>
</envars>
```

As tells it's name, the envars Action is dedicated to configure JVM Environment Variables.
We see in this xml file file, that this Action allows to configures more than one Environment Variable at a time: this is what we call multiple : true.

3.3 THE ACTION PLUGIN INSTALLATION

Simply zip the content above and run the plug command on it.

> plug -i my.zip

Note:
For more information about the plug command, type: plug -h.

3.4 THE ACTION LIFE CYCLE

When the Kikonf commands are run, the Action classes are intanciated, and one operational methods is called on them, then they are deallocated. This is what we call the Action class life cycle.

e.g.:
> kikact was.crtserver,was.jdbc,was.datasrc,was.jmq
Will instanciate and call the method injects on each Action class in their appearance order in the command.

> kikarc c.xml
Will instanciate and call the method injects on each Action class in their appearance order in the c.xml file.

Note:
To know how to work with Action files and Action custom xml files please see the Kikonf documentation.

Operational methods
An Action instance can do 3 stuffs:

- configures the target software regarding the content of the Action file (or custom xml file).
- remove the configuration from the target software regarding the content of the Action file (or custom xml file).
- Extract the configuration from the target software and generate a corresponding Action file (or custom xml file).

To provide this facility, 5 standard methods are implemeted by the parent class and may be surdefined by the Action class.

- def **inject**(self) # Only Action with sub_type "configuration" is their desccriptor file.
- def **run**(self) # Only Action with sub_type "control" in their desccriptor file.
- def **extract**(self, scope_attrs=None, prefix=None, **keywords)
- def **remove**(self, no_name, no_name_no_prefix)

These methods managed by the Kikonf engine are called in specific situations.

When no option –operation (-o) is passed to the Kikonf commands (kikact and kikarc),
the default operation is **inrun :**
operation inrun will call:

- the **method inject** on the Action instance, if the Action sub_type is "configuration"
- **run** if the Action sub_type is "control".

e.g.: > kikact was.envars,was.starts or
kikact was.envars,was.starts -o inrun
kikact c.xml -o inrun

When the option –operation (-o) **inject** is passed to the Kikonf commands:
the **method inject** is called on the Action instance.
e.g.: > kikact was.envars -o inject
kikact c.xml -o inject

When the option –operation (-o) **run** is passed to the Kikonf commands:
the **method run** is called on the Action instance.
e.g.: > kikact was.envars -o run
kikact c.xml -o inject

When the option –operation (-o) **remove** is passed to the Kikonf commands:
the **method remove** is called on the Action instance.
e.g.: > kikact was.envars -o remove
kikact c.xml -o remove

When the option –operation (-o) **extract** is passed to the Kikonf commands:
the **method extract** is called on the Action instance.
e.g.: > kikact was.envars -o extract
kikact c.xml-o extract

Note:
For more information about the descriptor attribute sub_type check the kikonf documentation.

3.5 THE PYTHON CODE FILE

This is the python code file for the sample Action plugin "was.envars" :

```
1.  from actions.was.tools import *

2.  class Envars(wasAction):

3.      def extract(self, scope_attrs=None, prefix=None, **keywords):
4.          self_funct='extract'
5.          scope_id, scope_attrs, scope=self.getScope(scope_attrs=scope_attrs, indent=self.getIndent())
6.          envars_node=self.newTop()  #-- because in extract mode, each call returns a new blank top node.
7.          indent=self.getIndent() + 3*' '

8.          envars_node.setAttrs(prefix=prefix)

9.          mkNodeScope(envars_node, scope_attrs, isUnique=True) #-- creates a scope node.
10.
11.         pjd=AdminConfig.list('JavaProcessDef', scope_id)
12.         vs=getStartByNameAsDict(split(AdminConfig.showAttribute(pjd, 'environment')))

13.         if len(vs)==0:return

14.         for vname in vs.keys():
15.             if vname==":continue

16.             if prefix!=None and vname.startswith(prefix) \
17.             or prefix==None:

18.                 attrs={}
19.                 vars=getShowAsDict(AdminConfig.show(vs[vname]))
20.                 name=vname
21.                 if prefix!=None:name=vname.split(prefix + '_')[1]
22.                 envars_node.newNode('var', name=name, value=vars['value'], required=vars['required'],
23.                 desc=vars['description'])

24.     def remove(self, no_name, no_name_no_prefix):
25.         envars_node = self.getTop()
26.         envars_attrs = envars_node.getAttrs()
27.         scope_id, scope_attrs, scope=self.getScope(parent_node=envars_node, indent=self.getIndent())
28.         indent=self.getIndent() + 3*' '

29.         # no_name/no_name_no_prefix
30.         prefix=envars_attrs.prefix
31.         if no_name_no_prefix:prefix=None
32.         if no_name_no_prefix or prefix!=None:self.rmvVar(prefix=prefix, indent=indent)
33.         else:
34.             if envars_node.hasNode('var'):
35.                 var_nodes=envars_node.getNode('var')

36.                 for var_node in var_nodes:
37.                     var_attrs = var_node.getAttrs()
38.                     self.rmvVar(name=var_attrs.name, indent=indent)

39.     def inject(self):
```

```
40.     self_funct='inject'
41.     envars_node = self.getTop()
42.     envars_attrs = envars_node.getAttrs()
43.     scope_id, scope_attrs, scope=self.getScope(parent_node=envars_node, indent=self.getIndent())

44.     if envars_attrs.prefix!=None:self.rmvVar(prefix=envars_attrs.prefix, indent=indent)

45.     if envars_node.hasNode('var'):
46.        pjd=AdminConfig.list('JavaProcessDef', scope_id)
47.        var_nodes=envars_node.getNode('var')

48.        cmdvalues=[]
49.        for var_node in var_nodes:
50.           var_attrs = var_node.getAttrs()

51.           #-- name
52.           if envars_attrs.prefix==None:name=var_attrs.name
53.           else:name=envars_attrs.prefix + '_' + var_attrs.name
54.           indent=self.getIndent() + 3*' '
55.           if envars_attrs.prefix==None:self.rmvVar(name=var_attrs.name, indent=indent)

56.           vars=[]
57.           vars.append(['name', name])
58.           vars.append(['value', var_attrs.value])
59.           vars.append(['required', var_attrs.required])
60.           if var_attrs.desc!=None:vars.append(['description', var_attrs.desc])
61.           cmdvalues.append(vars)

62.        AdminConfig.modify(pjd, [['environment', cmdvalues]])

63.  def rmvVar(self, name=None, prefix=None, indent=None):
64.     scope_id, scope_attrs, scope=self.getScope()

65.     pjd=AdminConfig.list('JavaProcessDef', scope_id)
66.     vs=getStartByNameAsDict(split(AdminConfig.showAttribute(pjd, 'environment')))

67.     for vname in vs.keys():
68.        if prefix!=None and vname.startswith(prefix) \
69.        or prefix==None and name!=None and name==vname \
70.        or prefix==None and name==None:
71.           AdminConfig.remove(vs[vname])
```

line 1 : from actions.was.tools import *
An Action import all stuff from its software dedicated module.
If the software was "**apa**", the import would have been :
from actions.**apa**.tools import *

Note about import :
The software dedicated module defines several helpfull functions that may simplify one Action coding. You may want to check the final chapter Annex : tools & utilities or run the python help on the module tools.

line 2 : class **Envars**(**wasAction**)

A "was" Action class inherites from the mother class: was Action
This class comes from the actions.**was**.tools module.
If the software was "**apa**", the mother class would have been: **apa**Action
from module actions.**apa**.tools import module.

An action class name is the name of the Action captitalized.
Here Envars is the class name for the Action named envars.

The action class envars surdefines 3 operational methods from the mother class :

lines: 3, 24, 39

- def **extract**(self, scope_attrs=None, prefix=None, **keywords)
- def **remove**(self, no_name, no_name_no_prefix)
- def **inject**(self)

All other methods are internal methods used by one of the 3 previous.
For instance the method **rmv**Var (line 72) is used by the method remove.

As best practice, we prefix the internal methods:

- used by remove with "rmv"
- used by extract with "ext"
- used by inject with "inj"

In fact, our sample class use more operational methods than required for the simplest Action class.
If an Action do not surdefine an operational method no operation is simply performed for this method.

3.6 THE INJECT METHOD

An inject method is called by default when no operation options is passed to the command or when the option –o inject (or -o inrun) is explicitly passed to the commands.
e.g.:
kikact was.envars
or
kikact was.envars -o inject

The inject method is where we defines the configuration for our Action.

Following our sample, the Action envars means to configure a JVM environment variable.

Remember that the Action class relay on an external xml file : the Action file for its attributes.

Note:
If the term of Action file do not mean any thing to you, please read the kikonf documentation and eventually the kikact and kikarc command documentation as well.

Here is a sample Action file for the Action envars :

```
<envars type = 'action' prefix='myprefix'>
    <scope node = 'localhostNode01' server = 'server1'/>

    <var name='myvar1' value='value1' required='false' desc='mydesc1'/>
    <var name='myvar2' value='value2' required='true' desc='mydesc2'/>
    <var name='myvar3' value='value3' required='false' desc='mydesc3'/>
</envars>
```

Action contract: The inject method for Action envars will created as much JVM environment variables as "var" nodes into the given Action file.

Lines (41 to 43)
line 41: *envars_node = self.getTop()*
Retreives the top node for the Action file.
This node is a picxml Node. Here the node with tag: "`envars`".
For more informattion about the picxml parser please read the picxml documentation.

line 42: *envars_attrs = envars_node.**getAttrs()***
This line retreives the attributes for the top node as an objects which attribnutes are the top node attributes. Here "type" and "prefix".

line 43: ***scope_id, scope_attrs, scope**=self.**getScope**(parent_node=envars_node, indent=self.getIndent())*
The getScope method from the mother class, digests the node with tag: scope :
<scope node = 'localhostNode01' server = 'server1'/>
scope_id is the WebSphere configuration id of the AdinConfig object matching this scope.
scope_attrs is a dict with this value : {node: 'localhostNode01', server: 'server1'}

scope is an instance of the class Scope from the imported module.

Line 44 : *if envars_attrs.prefix!=None:self.rmvVar(prefix=envars_attrs.prefix, indent=indent)*
An Action class as best practise manages to clear the place before performing one operation and destroys is configuration from the target software.

Line 46: *pjd=AdminConfig.list('JavaProcessDef', scope_id)*
This line retreives the Process Definition configuration id, for the server: server1 on node: localhostNode01.

Line 47: *var_nodes=envars_node.getNode('var')*
This retreives a list of picxml nodes for all nodes with Attribute "var" under the top node.

Line 49: *for var_node in var_nodes:*
For each node of the list.

Line 50: *var_attrs = var_node.getAttrs()*
We retreives the attributes as an object which attributes are the node with tag "var" attributes.

Lines 57 to 61:
vars.append(['name', name])
vars.append(['value', var_attrs.value])
vars.append(['required', var_attrs.required])
cmdvalues.append(vars)
We feed cmdvalus with pair of Attribute/Values from the xml file.

Finally cmdvalues would like :

```
[
[['name', 'myvar1'], ['value', 'value1'], ['required, 'false'], ['desc='mydesc1']],
[['name', 'myvar2'], ['value', 'value2'], ['required, 'false'], ['desc='mydesc2']],
[['name', 'myvar3'], ['value', 'value3'], ['required, 'false'], ['desc='mydesc3']]
]
```

Line 62: *AdminConfig.modify(pjd, [['environment', cmdvalues]])*
We feeded AdminConfig modify with cmdvalues.

3.7 THE REMOVE METHOD

A remove method is called on remove operation.
e.g.:
kikact was.envars -o remove

The sample Action file for the Action envars is reproduced here for convenience:

```
<envars type = 'action' prefix='myprefix'>
    <scope node = 'localhostNode01' server = 'server1'/>

    <var name='myvar1' value='value1' required='false' desc='mydesc1'/>
    <var name='myvar2' value='value2' required='true' desc='mydesc2'/>
    <var name='myvar3' value='value3' required='false' desc='mydesc3'/>
</envars>
```

The remove method is where we defines the way to remove the configuration from the target software.

Lines 25 to 27 : same as method inject 41 to 43 :
Line 25: *envars_node = self.getTop()*
Retreives the top node for the Action file.
This node is a picxml Node. Here the node with tag: "`envars`".
For more informattion about the picxml parser please read the picxml documentation.

line 26: *envars_attrs = envars_node.**getAttrs()***
This line retreives the attributes for the top node as an objects which attribnutes are the top node attributes. Here "type" and "prefix".

line 27: ***scope_id**, **scope_attrs**, **scope**=self.**getScope**(parent_node=envars_node, indent=self.getIndent())*
The getScope method from the mother class, digests the node with tag: scope :
<scope node = 'localhostNode01' server = 'server1'/>
scope_id is the WebSphere configuration id of the AdinConfig object matching this scope.
scope_attrs is a dict with this value : {node: 'localhostNode01', server: 'server1'}
scope is an instance of the class Scope from the imported module.

Note about prefix:
You do not need to use the prefix concept to design you Action class.
The prefix notion wont be covered in this documentation.
But talking simple, when a prefix is provided :
On inject:
The created configuration element names are : envars_attrs.prefx + '_' + var_attrs.name
On remove:
All configuration element on this scope starting with "prefix" is destroyed.
On extract:
All configuration element on this scope starting with "prefix" is extracted.

Lines 34 to 38:
if envars_node.hasNode('var'):

```
var_nodes=envars_node.getNode('var')
for var_node in var_nodes:
    var_attrs = var_node.getAttrs()
    self.rmvVar(name=var_attrs.name, indent=indent)
```

If the top node has sub nodes with tag "var", they are retreived into var_nodes.
And for each var node the internal method rmvVar is run.

Line 63: *def rmvVar(self, name=None, prefix=None, indent=None):*
The method rmvVar is called with the name of the var to destroy.

Line 65: *pjd=AdminConfig.list('JavaProcessDef', scope_id)*
Retreives the process defiinition for the scope (the server: 'server1' on the node: 'localhostNode01').

Line 71: *AdminConfig.remove(vs[vname])*
Then the Environment variable is destroyed using AdminConfig remove.

Note:
For more information about ultility functions like getStartByNameAsDict, see the chapter Annex : tools & utilities.

3.8 THE EXTRACT METHOD

An extract method is called on extract operation.
e.g.:
kikact was.envars -o extract

This method extract the Action configuration from the software.

Based on the Action key read from the Action file e.g. :

```
<envars type = 'action' prefix='myprefix'>
    <scope node = 'localhostNode01' server = 'server1'/>

    <var name='myvar1' value='value1' required='false' desc='mydesc1'/>
    <var name='myvar2' value='value2' required='true' desc='mydesc2'/>
    <var name='myvar3' value='value3' required='false' desc='mydesc3'/>
</envars>
```

The kikonf engines call the extract method on the Action instance.

Action key :
To learn more about Action keys please check the kikact and kikarc documentation.
Talking simple the Action key is made of three elements :
_ **name** (optional) : if the Action has an Attribute name supported by the top Action tag it is taken in consideration to be part of the Action key.
_ **prefix** (optional) : if the Action has an Attribute prefix supported by the top Action tag it is taken in consideration to be part of the Action key.
_ **scope**

Line 1: *def extract(self, scope_attrs=None, prefix=None, **keywords):*
The Kikonf engine calls the extract method with parameters :
scope_attrs={node: 'localhostNode01', server: 'server1'}, prefix=None, using the content read from the Action file.

line 5: **scope_id, scope_attrs, scope**=self.getScope(scope_attrs=scope_attrs, indent=self.getIndent())
The getScope method from the mother class, digests the parameter scope_attrs into :
scope_id is the WebSphere configuration id of the AdinConfig object matching this scope.
scope_attrs is a dict with this value : {node: 'localhostNode01', server: 'server1'}
scope is an instance of the class Scope from the imported module.

At this point, remember that called on operation extract, the Kikonf engine do not generates any topNode, since it expects the extract method to do it.

Line 6: *envars_node=self.newTop()*
This create a blank top node.
The top node's tag is always the Action name in lower cases.

Note about the super method : newTop()
For each call this method creates a new blank picxml Node and stores it to the Action instance internal top node list. The getTops() method will return all the stored top Nodes.

Line 8: *envars_node.setAttrs(prefix=prefix)*
This set only one Attribute (prefix) for the newly creaty top node.

Note:
For more information about how to manage picxml nodes, please see the picxml project documention.

Line 9: *mkNodeScope(envars_node, scope_attrs, isUnique=True) #-- creates a scope node.*

This tools utility function add a new node with tag: "scope" and with attributes:
server= 'server1' and node='localhostNode01', to the parent node: envars_node (top node).

Line 11: *pjd=AdminConfig.list('JavaProcessDef', scope_id)*
This line retreive the Configuration id of the Process Definition for the scope (the server: 'server1' on the node: 'localhostNode01').

Line 12: *vs=getStartByNameAsDict(split(AdminConfig.showAttribute(pjd, 'environment')))*
This line retreives all the JVM Environment Variables from the Process Definition
into a python dict, where key is the Environment Variable name.

Note:
For more information about ultility functions like getStartByNameAsDict, see the chapter Annex : tools & utilities.

Lines 16 to 23:

```
for vname in vs.keys():
        attrs={}
        vars=getShowAsDict(AdminConfig.show(vs[vname]))
        name=vname
        envars_node.newNode('var', name=name, value=vars['value'], required=vars['required'],
        desc=vars['description'])
```

For each Environment Variable:
_ all Attributes are retreived using getShowAsDict into the python dict vars.
_ then a new node with tag: "var" and attributes: the found attributes is created and add to the father top node.

Note:
For more information about ultility functions like getShowAsDict, see the chapter Annex : tools & utilities.

4 EXIT/EXTRACT PLUGIN

4.1 INTRODUCTION

An Exit/extract plugin offers a custom way to extract bunches of configurations elements, from the target software.
The extracted informations may be organized either as

- standalone Action files.
- or a big custom file.

by the kikonf commands.

The following shows how an Exit/Extract plugin is structured.

You can take a look at any of the extract plugins deployed under the <EXIT_EXTRACT_PLUGIN_BASE_DIR>/<SOFTWARE_CATEGORY>/<PLUGIN_NAME>/by/kikonf.

For the purpose of this chapter will use the exit/extract plugin "server" provided for "was" at : <EXIT_EXTRACT_PLUGIN_BASE_DIR>/was/server/by/kikonf

Note:
The kikact command with -p (or -P options) shows a (detailled) list -p of all the deployed plugins.

4.2 THE EXIT/EXTRACT PLUGIN CONTENT

EXIT_NAME.py
<EX_INF>/
    action.attrs
    LICENSE_FILE

ACTION_NAME.py : the python code file of the exit.

<ACT_INF> : the Action Info directory.

exit.attrs : this file reports the identification attributes of this Action plugin.

LICENSE_FILE : the license for this product.

4.2.1 The exit.attrs file

The exit.attrs file contains just one line an attribute named "exit", which value is a coolTyped python dictionary (the same keys as for action).
For more information about the CoolTyping tools see the dedicated CoolTyping documentation.
To say simple a coolTyped value is a python type value with no "" and no "".

The line syntax:
exit =
{**category**:SOFTWARE_CATEGORY \
,**name**:<EXIT_NAME>,**by**:WHO,**license**:<LICENSE_FILE>,**multiple:**false,**version**:VERSION}

SOFTWARE_CATEGORY : the software to which is dedicated this Action plugin.

EXIT_NAME : the exit name.

WHO : the exit plugin provider name.

LICENSE_FILE : the license file name for this Exit plugin.
This file must exists under the ACT_INF directory.

MULTIPLE : always false.

VERSION : The version of this Exit plugin, may be a number or a string.

For instance the content of the exit.attrs file for the sample Exit plugin "envars" is:

exit = {**category**:was,**name**:envars,**by**:kikonf,**license**:COPYING,**multiple**:true,**version**:5.0}

4.3 THE EXIT/EXTRACT PLUGIN INSTALLATION

Simply zip the content above and run the plug command on it.

> plug -i -e my.zip

Note:
For more information about the plug command, type: plug -h.

4.4 THE EXIT LIFE CYCLE

When the Kikonf (kikact or kikarc) commands are run, the Exit module is loaded and the function "extract" is called on it, this is what we call the exit life cycle.

e.g.:
> kikact -o extract -e **was.server** --scope_**node** localhostNode01 --scope_**server** server1
or
> kikarc -o extract -e **was.server** --scope_**node** localhostNode01 --scope_**server** server1

Parameters
This command will load the python code file for the exit at :
<EXIT_EXTRACT_PLUGIN_BASE_DIR>/**was**/server/by/kikonf/**server.py**
and call the function extract on it with the parameters:
prefix=None, scope_attrs={**node:** 'localhostNode01', **server**:'server1'}

Other parameters all allowed, and if you use them they are also passed to the extract function. They are:

Exit extended options:
The following options are allowed combined with --exit (-e) option.

--name=NAME In conjontion with the --exit (-e) option. A name.
--prefix=PREFIX In conjontion with the --exit (-e) option. A prefix name.
--scope_server=SCOPE_SERVER In conjontion with the --exit (-e) option. A server name.
--scope_node=SCOPE_NODE In conjontion with the --exit (-e) option. A node name.
--scope_cluster=SCOPE_CLUSTER In conjontion with the --exit (-e) option. A cluster name.
--scope_application=SCOPE_APPLICATION In conjontion with the --exit (-e) option. An application name.
--scope_war=SCOPE_WAR In conjontion with the --exit (-e) option. An application war name.
--scope_type=SCOPE_TYPE In conjontion with the --exit (-e) option. A server type. Allowed values are
as: for Application Server, and
ws: for WebServer.
--scope_cell In conjontion with the --exit (-e) option. An application name.

Note:
For more information about the kikonf commands parameters check the kikact and the kikarc documentations.

4.4.1 The exit context:

While an Exit do not inherite from a particular managed class (like Actions with wasAction)
it still need a way to retreive the actual kikonf context.
This job is handled by a special class: Config.
A class Config is instanciated by default by the Kikonf engine and is made available into the exit module's name space.

Typical operations on the context are :

- Create a a new top node and fill it.
- Intanciate any Action from any Action plugin.

e.g.:

This sample creates a new top Node :
Line 7 to 8:
*top_node = **config**.newTop()*
servers_node=top_node.newNode('servers')

This sample run a trivial Action:
Line 22 to 26:
*ad=**config**.getActionDispatcher()*
action=ad.getAction('was', 'crtserver', byWho=None)
action.doExtract(scope_attrs=scope_attrs, prefix=prefix, doRaise=doRaise)
servers_node.adds(getTops()) (little variation for the course)

Note:
For more information about the methods and attributes of the Config instance see the chapter Annex : tools & utilities.

4.5 THE PYTHON CODE FILE

This is the python code file for the sample Exit plugin "was.server" :

```
1. from exits.extract.was import utils
2. from actions.was.tools import *
3.
4.
5. def extract(prefix=None, scope_attrs=None, doRaise=True, **keywords):
6.     self_funct='extract'
7.     top_node = config.newTop()
8.     servers_node=top_node.newNode('servers')
9.
10.          #-- Reduce the scope to what we need for this exit
11.          scope_attrs={'server': scope_attrs['server'], 'node':
   scope_attrs['node'],
12.          'cluster': scope_attrs['cluster']}
13.          mkServerNodes(servers_node, scope_attrs, prefix=prefix, doRaise=doRaise)
14.
15.          return top_node
16.
17.def mkServerNodes(parent_node, scope_attrs, prefix=None, doRaise=True):
18.          """
19.          Makes server nodes according scope_attrs,
20.          and add them to parent_node.
21.          """
22.          ad=config.getActionDispatcher()
23.
24.          action=ad.getAction('was', 'crtserver', byWho=None)
25.          action.doExtract(scope_attrs=scope_attrs, prefix=prefix,
   doRaise=doRaise)
26.          l=action.getTops()
27.
28.          for i in range(len(l)):
29.              server_node=l[i]
30.              scope_attrs=server_node.getNode('scope')[0].getdAttrs()
31.              eventually updated scope_attrs.
32.              server_node._setBalLabel(scope_attrs['server'] + '_' +
   scope_attrs['node']       + '_' + str(i))
33.              parent_node.add(server_node)
34.              utils.server_node(server_node, scope_attrs, prefix=prefix,
   config=config,
35.              doRaise=doRaise)      # Filling server with all resources.
```

line 2 : from actions.was.tools import *
An Action import all stuff from its software dedicated module.
If the software was "**apa**", the import would have been :
from actions.**apa**.tools import *

Note about import :
The software dedicated module defines several helpfull functions that may simplify one Action

coding. You may want to check the final chapter Annex : tools & utilities or run the python help on the module tools.

4.6 THE EXTRACT FUNCTION

Quickly speaking, the job of an extract function is to extract a bunch of configurations elements from the software and to dump them to big xml node.

At the function completion, this xml node is then handled by the Kikonf engine to cast the result eithers as standalone Action files or as a big custom file.

To extract configuration elements from the targret software the exit module relays on Action plugins.

Line 5: *def extract(prefix=None, scope_attrs=None, doRaise=True, **keywords):*
As seen in the introduction of this chapter, the extract function is called with these parameters:
prefix=None, scope_attrs={**node:** 'localhostNode01', **server**:'server1'}

Line 7 to 8:
*top_node = config.**newTop**()*
This create a blank top node.
The top node's tag is always the Exit name in lower cases (here envars).
*servers_node=top_node.**newNode**('servers')*
A new Node "server" is created and added to the top node.
Afterwards this servers_node will be filled with whatever is extracted from the software.

Line 22 *ad=config.**getActionDispatcher**()*
This line retreives an ActionDispatcher from the context.
This instance will allow to call any Action plugin of the Kikonf reprository.

Line 24: *action=ad.**getAction**('was', 'crtserver', byWho=None)*
The method getAction instanciates a new Action instance from the Kikonf repository.
The parameters are :

- category : "was"
- action name = "crtserver"
- byWho = None (e.g.: kikonf)

Please note that these parameters are the same in the bal (Basic Action Locator) used into Kikonf regular commands.
e.g. :
> kikonf was.envars -v 3

Line 25: *action.doExtract(scope_attrs=scope_attrs, prefix=prefix, doRaise=doRaise)*
Then any method can be called on the Action instance.

Line26: *servers_node.**adds**(action.getTops())* (little variation for the course)
This particular pcixml method "adds": adds a list of nodes to the parent node, here server_node.

5 ANNEX I : THE BASE ACTION CLASS

All per cateory Action classes inherite from this one.

cd <KIKONF_INSTALL_DIR>/core
python
>>> from lib import action
>>> help(action)
Help on module lib.action in lib:

package: lib.action

class Action

| This class must be inherited by all Action class from the "was" category.
|
| **__init__**(self, action_dir=None, restrictor_dir=None, kikonf_attrs=None, verbose=None, indent='
| ', logFile=None)
| Must not be surdefine or at your own risks.
|
| **init(self)**
| May be surdefined.
| This method is called by the Kikonf engine, after instantiation.
|
| **getName(self)**
| This returns the Action name.
|
| **getCAttrs(self)**
| This retreives the kikonf.attrs file as a dict of key/values pair attributes.
|
| **getCAttr(self, attr)**
| This retreives a specific attribute from the kikonf.attrs file.
|
| **hasCAttr(self, attr)**
| This checks a specific attribute from the kikonf.attrs file.
|
| **getCurrentDir(self)**
| Returns the Action plugin directory.
|
| **getCxmlDir(self)**
| This shows the Actions directory.
|
| **getRstDir(self)**
| This shows the Restrictors directory.
|

getTop(self)
Returns the Action file's top node.

newTop(self)
Generates a new top from scratch for this Action and add it to the internal list.

getTops(self)
Returns the list of all nodes which has been generated by the instance using the method newTop.

doEnter(self)
Internal use only/Managed by the Kikonf engine.
When called by kikonf, this will run the **enter method** on the Action instance.

doInject(self)
Internal use only/Managed by the Kikonf engine.
When called by kikonf, this will run the **inject method** on the Action instance.

doRun(self)
Internal use only/Managed by the Kikonf engine.
When called by kikonf, this will run the **run method** on the Action instance.

doRemove(self, no_name, no_name_no_prefix, doRaise=True)
Internal use only/Managed by the Kikonf engine.
When called by kikonf, this will run the **remove method** on the Action instance.

doExtract(self, **keywords)
Internal use only/Managed by the Kikonf engine.
When called by kikonf, this will run the **extract method** on the Action instance.

doLeave(self)
Internal use only/Managed by the Kikonf engine.
When called by kikonf, this will run the **leave method** on the Action instance.

isTest(self)
True if the kikonf.attrs Test attribute is true.

isMultiple(self)
Idem as the action.attrs attribute multiple.

extractIsFilled(self)
True if the extract method has filled the generated blank top node, with at least one node.

getVerbose(self)
Returns the verbose level, if the –verbose (-v) parameter has been used with the kikonf command.

getIndent(self)

| Returns the indent value for verbose.
|
| **getLogFile(self)**
| Returns the logfile value, if the –log_file (-f) parameter has been used by the kikonf command.

6 ANNEX II : PER CATEGORY ACTION CLASSES

6.1 THE WAS ACTION CLASS

package: actions.was.tools

class wasAction(lib.action.Action)

| This is the main mother class Action for the "was" category.
| Every Action class for the "was" category, must implement this method.
|
| **getCell(self)**
| This returns the cell name.
|
| **getScope(self, parent_node=None, scope_attrs=None, indent=None)**
|
| Use cases:
| When a typical Action file supports a scope node (child of its top node).
| Actually 90% of the Acton plugins supports scope node. e.g.:
| <scope node = 'localhostNode01' server = 'server1'/>
|
| This the way to use the method getScope into an Action class:
| Called from inject method:
| scope_id, scope_attrs, scope=self.getScope(parent_node=envars_node, indent=self.getIndent())
|
| Called from extract method:
| scope_id, scope_attrs, scope=self.getScope(scope_attrs=scope_attrs, indent=self.getIndent())
|
| Parameters:
| parent_node: the top node
| scope_attrs: a dict of the scope attrs.
| parent_node and scope_attrs are exclusives together.
|
| Returned values:
| The getScope method digests the scope representation into a WebSphere (R) representation.
| scope_id: is the WebSphere configuration id of the object matching this scope.
| e.g.: myserver(cells/localhostCell01/nodes/localhostNode01/servers/myserver|
| server.xml#Server_1292955380265)
|
| scope_attrs: is a complete dict representation of a scope:
| e.g.: {'cluster': None, 'cell': 'false', 'node': 'localhostNode01', 'server': 'myserver'}
|
| scope is an instance of the class Scope (defined below in this module).

| e.g.: scope: <actions.was.tools.Scope instance at 905983488>
|
| **getScopeCP(self)**
| When a typical Action file supports a scope node (child of its top node),
| this method is a proxy above the scope instance getCP method.
| See the class Scope below for more information.
|
| **getScopeId(self)**
| When a typical Action file supports a scope node (child of its top node),
| this method is a proxy above the scope instance getId method.
| See the class Scope below for more information.
|
| **getScopeON(self)**
| When a typical Action file supports a scope node (child of its top node),
| this method is a proxy above the scope instance getON method.
| See the class Scope below for more information.
|
| **getScopeType(self)**
| When a typical Action file supports a scope node (child of its top node),
| this method is a proxy above the scope instance getType method.
| See the class Scope below for more information.
|
| **getScopes(self, parent_node=None, list_scope_attrs=None, indent=None)**
| Use cases:
| When a typical Action file supports a scopes node (child of its top node).
| If fact they are not so much, at the writing on this documentation there is currently
| only one Action plugin supporting multiple scopes : insapp. e.g.:
| <scopes>
| <scope node = 'localhostNode01' server = 'server1'/>
| <scope node = 'localhostNode01' server = 'server2'/>
| <scope cluster = 'mycluster'/>
| </scopes>
|
| This the way to use the method getScope into an Action class:
| Called from inject method:
| scopes=self.getScope(parent_node=insapp_node, indent=self.getIndent())
|
| Called from extract method:
| scopes=self.getScope(list_scope_attrs=list_scope_attrs, indent=self.getIndent())
|
| Parameters:
| parent_node: the top node
| list_scope_attrs: a list of scope attrs.
| parent_node and list_scope_attrs are exclusives together.
|
| Returned value:
| A list of scope instances of the class Scope (defined below in this module).
| e.g.: scope: [<actions.was.tools.Scope instance at 905983488>, ...]

getScopesON(self, doIhs=True)
When a typical Action file supports a scopes node (child of its top node), this method is a joined reprsentation of the scopes Object names as expected by the MapModulesToServers attribute using for instance AdminApp.install.

getPwdFromJAAS(self, jaas_alias)
This retreives the xor representation of the password of a given JAAS name from the repository. Use at your own risk

---

Methods inherited from lib.action.Action:

doEnter(self)

doExtract(self, **keywords)

doInject(self)

doLeave(self)

doRemove(self, no_name, no_name_no_prefix, doRaise=True)

doRun(self)

extractIsFilled(self)

getCAttr(self, attr)

getCAttrs(self)

getConfDir(self)

getCurrentDir(self)

getCxmlDir(self)

getIndent(self)

getLogFile(self)

getName(self)

getRstDir(self)

| getTop(self)
|
| getTops(self)
|
| getVerbose(self)
|
| hasCAttr(self, attr)
|
| init(self)
|
| isMultiple(self)
|
| isTest(self)
|
| newTop(self)
|
| setTop(self, node)

class Scope

| **getCell(self)**
| Returns the cell name.
|
| **getType(self)**
| Returns a type in cell, cluster, ClusterMember, server, Application
| todo: lower caps for all !
|
| **getSType(self)**
| If acurates, Returns a sever_tyhpe in as (for Application Sever), ws (for Web Server).
|
| **getId(self)**
| Returns the Scope object Config Id.
| The syntax for a wsadmin config Id is: name(directory|filename#XML-fragment-id)
| e.g.: myserver(cells/localhostCell01/nodes/localhostNode01/servers/myserver
| server.xml#Server_1292955380265)
|
| **getCP(self)**
| Returns the Scope object Containment Path.
| The syntax for a wsadmin Containment Path is: /type:name/type:name/.../
| e.g.: /Node:localhostNode01/Server:myserver
|
| **getON(self)**
| If acurates, returns the Scope Object name.
| The syntax for a wsadmin Object Name (JMX Object Name syntax) is:
| WebSphere:key=value,key=value,key=value,...
| e.g.: WebSphere:cell=localhostCell01,node=localhostNode01,server=myserver

getVerbose(self)
Returns the verbose level, if the –verbose (-v) parameter has been used with the kikonf command.

getIndent(self)
Returns the indent value for verbose.

getLogFile(self)
Returns the logfile value, if the –log_file (-f) parameter has been used by the kikonf command.

7 ANNEX III: THE CONFIG CLASS

package: lib.tools

class Config

| Dedicated to Exit plugins.
| Because Exit plugins do not inheritate from any mother class like it is
| required for Action plugins,
| the config instance manages the life cycle management for the Exit plugins.
| When running an Exit plugin, a Config instance is generated for its into its global namespace.
| Hence the Exit module can use it to perform operations.
| This a sample use :
|
| Use case (excerpt from the Exit/extract from was.pplication.by.kikonf):
| Creating and feeding the xml tree:
| top_node = config.newTop()
| This generates a blank top node (a picxml node),
| application_node=top_node.newNode('application', name=name)
| and feeds it.
|
| Arbitrary calls of Actions:
| ad=config.getActionDispatcher()
| This retreives the Action dispatcher (described below).
|
| action=ad.getAction('was', 'clsloader', byWho=None)
| This arbitrary instanciates the Action: clsloader.
|
| action.doExtract(scope_attrs={'application':name}, prefix=prefix, doRaise=doRaise)
| This calls the extract method on the Action.
| This method generates a picxml tree node, from what it has extracted from the target
| configuration.
|
| if action.extractIsFilled():application_node.adds(action.getTops())
| This Action instance node is then retreived and added to the main exit tree.
|
| **getExitName(self)**
| This shows the exit name <=> getBelName.
|
| **getBelCategory(self)**
| This shows the Basic Exit Locator attribute: category.
|
| **getBelName(self)**
| This shows the Basic Exit Locator attribute: name.
|

| **getBelByWho(self)**
| This shows the Basic Exit Locator attribute: byWho.
|
| **getCAttrs(self)**
| This retreives the kikonf.attrs file as a dict of key/values attributes.
|
| **getCAttr(self, attr)**
| This retreives a specific attribute from the kikonf.attrs file.
|
| **hasCAttr(self, attr)**
| This checks a specific attribute from the kikonf.attrs file.
|
| **getCxmlDir(self)**
| This shows the Actions directory.
|
| **getRstDir(self)**
| This shows the Restrictors directory.
|
| **getInfos(self)**
| This shows the exit.attrs infos.
|
| **newTop(self)**
| Creates a new top picxml node.
|
| **getActionDispatcher(self)**
| Retreives the Action Dispatcher.
| To allow Action plugin trivial calls.
|
| **isTest(self)**
| True if the kikonf.attrs Test attribute is true.
|
| **getVerbose(self)**
|
| **getLogFile(self)**
|
| **getIndent(self)**

class ActionDispatcher

| Dedicated to Exit plugins.
| Works with the Config class, see below.
| An Action dispatcher is generatd for each Exit module instance,
| and is available througth the Config instance.
|
| An Action dispatcher allows to arbitrary instanciate any Action plugin.
|

| This a sample use :
|
| Use case (excerpt from the Exit/extract from was.pplication.by.kikonf):
|
| Arbitrary calls of Actions:
| ad=config.getActionDispatcher()
| This retreives the Action dispatcher.
|
| action=ad.getAction('was', 'clsloader', byWho=None)
| This arbitrary instanciates the Action: clsloader.
|
| action.doExtract(scope_attrs={'application':name}, prefix=prefix, doRaise=doRaise)
| This calls the extract method on the Action.
| This method generates a picxml tree node, from what it has extracted from the target
| configuration.
|
| if action.extractIsFilled():application_node.adds(action.getTops())
| This Action instance node is then retreived and added to the main exit tree.
|
| **getAction(self, category, action, byWho=None)**
| Instanciates an arbitrary Action from the Kikonf repository.
| The parameters are :
| category : the bal (Basic Action Locator) category.
| action : the bal Action name.
| byWho : the bal byWho.

8 ANNEX VI : PER CATEGORY TOOLS & UTILITIES

8.1 WAS TOOLS

package: actions.was.tools

DESCRIPTION

This module defines a set of utility methods (except these marked "internal use only") which can be used straigth away from plugin classes.
Because the main lib.tools is imported in the global namespace, utility methods from this library can be used as well. See the help on this module for more information about those extra methods.

Trademarks:
WebSphere and WebSphere Aplication Server are a registred trademarks of International Business Machines Corporation.
Apache HTTP Server is a registred trademark of the Apache Software Foundation.
Other names may be trademarks of their respective owners.

FUNCTIONS

getShowAsDict(showOutput)

Returns AdminConfig.show as a python dict.

split(string)

Use cases:
Anywhere.

When wsadmin command returns a string representation for a list (using Adminconfig.getid or Adminconfig.list for instance),
the split function makes a well formatted python list of it.
e.g.:

```
> AdminConfig.getid('/VirtualHost:/')
'admin_host(cells/localhostCell01|virtualhosts.xml#VirtualHost_2)
default_host(cells/localhostCell01|virtualhosts.xml#VirtualHost_1)
myvhost(cells/localhostCell01|virtualhosts.xml#VirtualHost_1289071443671)'

> split(AdminConfig.getid('/Cell:localhostCell01/VirtualHost:/'))
['admin_host(cells/localhostCell01|virtualhosts.xml#VirtualHost_2)',
'default_host(cells/localhostCell01|virtualhosts.xml#VirtualHost_1)',
'myvhost(cells/localhostCell01|virtualhosts.xml#VirtualHost_1289071443671)']
```

getStartByNameAsDict(split_outputs)

Use cases:
Anywhere.

When wsadmin command returns a string representation for a list (using Adminconfig.getid or Adminconfig.list for instance),
and each entry starts by the effecive name of the attribute, getStartByNameAsDict function makes a well formatted python dict of it.
e.g.:
getStartByNameAsDict(split(AdminConfig.getid('/Cell:localhostCell01/VirtualHost:/')))

{'admin_host':'admin_host(cells/localhostCell01|virtualhosts.xml#VirtualHost_2)',
'default_host':'default_host(cells/localhostCell01|virtualhosts.xml#VirtualHost_1)',
'myvhost':'myvhost(cells/localhostCell01|virtualhosts.xml#VirtualHost_1289071443671)'}

Note:
For more information about the split function see the split function above.

trsToAdminTask(cmdvalues)

Translate an AdminConfig like imbricated list of [Attribute, value] ...
to an AdminTask like string of [Attribute value] ...

getClusterMember(cluster=None, node=None, member=None, asServer=True, asMember=False, doRaise=False)

For a given cluster member returns either
SERVER_ID, NODE, CLUSTER_ID if asServer is True (==> asMember is False)
or
MEMBER_ID, NODE, CLUSTER_ID if asMember is True (==> asServer is False)

Parameters:
cluster: (required), cluster name.
node: (optional), node name.
member: (required), member name.
asServer: (optional) True/False
asMember: (optional) True/False
doRaise: (optional) True/False

mkNodeScope(parent_node, list_scope_attrs, isUnique=True)

Use cases:
Called from extract method:
mkNodeScope(envars_node, scope_attrs, isUnique=True)
This function will feed the picxml top node parameter with either a
scope or a scopes node.
This new node becomes the child of the parent_node.

Parameters:
parent_node : (required) the top parent node to feed.
list_scope_attrs : (required) this parameter is hybride, it can receive easer a

_ scop_attrs : a dict of scope attrs, or a
_ list_scop_attrs : a list of dict of scope attrs.

getConnectionPoolAttrs(con_id, node, level=None, indent=None, logFile=None)

Typically used from an Action instance's extract method:
From an a AdminConfig Connection pool id, feeds a picxml connection pool node.
e.g. :

```
getConnectionPoolAttrs(confacts['connectionPool'], connection_pool_node,
    level=self.getVerbose(), logFile=self.getLogFile())
```

getConnectionPoolCfgs(conpool_node, level=None, indent=None, logFile=None)

Typically used from an Action instance's inject method:
From a picxml connection pool node, Returns a well formated python list of pair [Attribute, Value] as expected by AdminConfig.
e.g.:

```
getConnectionPoolCfgs(connection_pool_node, level=self.getVerbose(),
    logFile=self.getLogFile())
```

getInstTaskInfoAsDict(taskInfoOutput, type='MapModulesToServers')

Returns a python dict representation of a TaskInfo content.

TaskInfo content sample:

```
Module: Increment Enterprise Java Bean
URI: Increment.jar,META-INF/ejb-jar.xml
Server: WebSphere:cell=WOOSH,node=WOOSH,server=server1

Module: Default Web Application
URI: DefaultWebApplication.war,WEB-INF/web.xml
Server: WebSphere:cell=WOOSH,node=WOOSH,server=server1
```

getViewTaskInfoAsDict(taskInfoOutput, type='MapWebModToVH')

Returns a python dict representation of a TaskInfo content.

TaskInfo content sample:

```
Web module:  Default Web Application
URI:  DefaultWebApplication.war,WEB-INF/web.xml
Virtual host:  default_host
```

rtvAppTargets(application=None, doClsMbrs=False)

Retreiving all defined targets for this Application.

Returns a list of dicts of scopes like the one expect(by the action instapp for instance).
[{'cluster': CLUSTER, 'node': NODE, 'server': SERVER, 'server_type':SERVER_TYPE}]

SERVER_TYPE is one of:
ws : for the WEB_SERVER WAS type.
as : for the APPLICATION_SERVER WAS type.

getWasVersion()
Returns the WebSphere (TM) product version.

getWasFullVersion()
Returns the WebSphere (TM) full product version.

8.2 ANNEX V : GENERIC TOOLS & UTILITIES

cd <KIKONF_INSTALL_DIR>/core
python
>>> from lib import tools
>>> help(tools)
Help on module lib.tools in lib:

package: lib.tools

FUNCTIONS

getInstallDir()
Retreives the kikonf installation directory.

getEnvironmentvariable(env)
Returns the value of a given system environment varibale name.

getOsType()
Returns the os type.
unix for unixes or
windows for win32.

zipGetFile(src, path, check=False)
Extract a file from a zip.
src : zip file.
paths : path of the file in the zip.

unzip(src=None, destDir=None, toSign=None, toLightSign=None, ignoreErrors=True)
Unzip a file.
src : the zip file.
destDir : the destination directory where to uncompress.
ignoreErrors : ignore errors.
toSign : a path inside the destination directory where to dump the signature.

genUid()
Generats a Unique Id.

verbose(*args, **keywords)
arg Parameters:
If verbose has one hargument : this argument is printed at the indent position.
If verbose has two harguments : these argument are considered to be a title and its value and these argument are printed like this: title: value

keywords Parameters:
level : (required), numeric value, the current verbose level.
ifLevel : (required), numeric value, print arg only if ifLevel>=level.
indent : (optional) BEWARE must be a (blank) string not numeric.

9 TRADEMARKS:

AIX, WebSphere and WebSphere Aplication Server are registred trademarks of International Business Machines Corporation.

Windows is a registred trademark of Microsoft Corporation

Other names may be trademarks of their respective owners.