# KIKONF
# Universal
# J2EE
# and related Servers
# Administration tool

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

# SUMMARY

## OBJECTIVES

Kikonf is a tools box that allows you to run simple and complexe Architecture configurations.

Kikonf objective is to provide a simple an easy way to configure, from one point, sofisticated softwares, that relay on heavy and complexe configurations.

The limit of  kikonf supported softwares is the number of plugins provided for it.

Kikonf also offers  facilities to help you to develop (and publish) you own plugins to join you own needs.

One more feature of kikonf is its ability to extract configurations from existing software configurations into its own scheme.

Note:
To see how many softwares currently are supported by kikonf take a look at www.kikonf.com.
Kikonf stick with real life  situations when you may have to configure more than one softwares in one operation.

# 1 PREREQUISITES

Oses (every where python works)

Linux, AIX ®, Windows ®

Langage

Python >= 2.4 < 3

Softwares

As Kikonf aims to administrate softwares, the target required software should be installed.

**e.g.:**
If you intend to use Kikonf to Administrate WebSphere Application Server using the Kikonf defaut set of plugins (provided by default).
WebSphere © Application Server version >= 6.1 must be installed.

# 2 INSTALLATION

Download kikonf at www.kikonf.com

Unzip the file kikonf_#.##.zip in the directory of your choice.

Put the path <KIKONF_INSTALL_PATH>/bin in your path environment variable.

Type
> kikact on Windows
or
> kikact.sh on Unixes
(dont forget to run chmod ugo+x *.sh in the bin directory)

# 3 INTRODUCTION

In kikonf scheme, running a singular operation on the software is called a configuration Action.
Actions are sorted per Category (aka software).

To see the list of the supported Software/Actions of your current kikonf instalation type :
**kikact -p** (or -P) (todo)

To see the list of all kikonf supported Software/Actions check out www.kikonf.com.

Note:
Kikonf also make it easy for you to realize your own custom plugins.
The Kikonf community is happy to help you to share plugins and then make them availables under watever licence you like, therefore the www.kikonf.com site supports facilities for that.

Each configuration Action is supported by an Action plugin and many Action plugins are availables.

Kikact and kikact commands allow you to run configuration actions, either :

- **one by one (see the kikact command)**
  In order to realise specific surgical operations.
  Also when providing a short list of actions likely for delegation to a special category of users or menu options.

or

- **as a whole (see the kikact command)**
  When applying a custom configuration globally which represent a typical configuration in your own business sheme..

Kikonf supports two principal command kikact and kikarc.

# 4 KIKONF PREFERENCE FILE

The Kikonf preference file is kikonf.attrs.

## 4.1 OBJECTIVES

The kikonf.attrs allows to :

- **Specify the path of the software beeing administrated**
  In order to configure softwares Kikonf needs to know where its binary is installed.

- **Specify global variables**
  Once defined into the kikonf.attrs file these global variables are available to the Actions and plugins.

## 4.2 CONTENT

The kikonf.attrs is a list of pair attribute/value elements.

The kikonf.attrs is divided in 3 parts :

<u>The software part:</u>

These values are path or access points (like ports) to the software to configure.
All these attributes are prefxed by :
**software_<SOFTWARE_CATEGORY>_**

Some of supported categories are: was, ihs, apa, mq.

<u>Note</u>:
For a more explicite list of softawre categories, run:
**kikonf -p** (or -P) or check the www.kikonf.com site.

For instance if you intend to configure WebSphere Application Server the following attributes must be include into the kikonf.attrs file :

```
software_was_version = 6.1 # This version is used in kikonf plugins to check compatibility.
software_was_dmgr_host = localhost # Host/Port to join the Dmgr.
software_was_dmgr_port = 8879
# software_was_dmgr_protocol = SOAP # (Optional, default SOAP) SOAP/RMI
software_was_profile_path = /usr/opt/WebSphere/AppServer/profiles/Dmgr01
software_was_user = wasadm # (Optional, default None) If WAS secturity is enabled.
software_was_password = wasadm # (Optional, default None)
```

**software_was_xms** = 512 # (Optional, default 256)
**software_was_xmx** = 1024 # (Optional, default 512)

For instance if you intends to use kiko action for IHS the following properties must be include into the kikonf.attrs file :

**software_ihs_alias** ='IHS' # This alias is used in kikonf plugin to check compatibility.
**software_ihs_version** ='6.0' # This version is used in kikonf plugins to check compatibility.
**software_ihs_path** = /opt/ibm/HttpServer

Note:
For a complete list of all the available software attributes please check the per software documentation at www.kikonf.com.

The commands part

The major purpose of this part is to restrict the use of the kikonf commands : **kikact** and **kikarc**.
These attributes starts either by **kikact** or **kikarc**.

**kikact**_cxml = /a/custom/path/for/action/files/directory.
**kikact**_crst = /a/path/for/action/restrictors/directory
**kikact**_restricted_actions_list=crtserver,jvm,jdbc,datasrc
**kikarc**_cxml = /a/custom/path/for/action/files/directory.
**kikarc**_crst = /a/path/for/action/restrictors/directory
**kikarc**_allowed=False

The custom part:
These attributes are prefixed with alias.
In this part any useful global variables can be specified.
These global variables are then available for the Actions and plugins.
For instance declaring here :
**alias**_myvariable=myvalue

makes $myvariable available for any Action that may need it, using either the kikact or the kikarc command.

Note(for kikarc only):
This global variable may be overwriten by a local variable guiven with the __alias__ special tag (for more information see the kikarc command below).

## 4.3 PATH RESOLUTION

By default the kikonf.attrs file is located into the <KIKONF_INSTALL_PATH>/conf directory.
But this file can be placed anywhere else to provide specific configurations.
For instance when you have to support more than one version for the same software or want to provide some user specific configurations.

In order to find the kikonf.attrs file the kikonf commands (either kikact or kikarc) operate a path resolution in that order :

- **As parameter**
  The path of the kikonf.attrs file can be guiven as parameters :

  **kikact -C /where/is/kikonf.attrs**
  or
  **kikact --cattrs /where/is/kikonf.attrs**

- **As environment variable**
  If the path of the kikonf.attrs file is not guiven as parameters,
  the kikonf.attrs file is retreived at this path.

  Example :
  **set KIKONF_CATTRS=/where/is/kikonf.attrs**

- **By default**
  If the path of the kikonf.attrs file is not guiven nether as parameter nor as an environment variable, the kikonf.attrs is retreveid at:

  **<KIKONF_INSTALL_PATH>/conf/kikonf.attrs**

# 5 KIKONF COMMANDS

Kikonf provide five commands : **kikact**, **kikarc, picxml, epicxml** (todo) **and plug**

kikact and kikarc allow you to run configuration actions, either :

- **one by one (see the kikact command)**
  In order to realise specific surgical operations.
  Also when providing a short list of actions likely for delegation to a special category of users or menu options.

or

- **as a whole (see the kikact command)**
  When applying a custom configuration globally which represent a typical configuration in your own business sheme..

Note:
Obviousliy both commands may impact many softwares dependindg on wich Actions they are calling for.

For those who don't have access to hight level xml scripting langages and want to perform there management operations upon xml files :
**picxml** and **epicxml** are provided (to query/update xml files).
plug manages plugins installation.

Note:
The purpose of this documentation is to describe the kikonf core implemention.
In the following chapters, the kikonf commands are explored througth this singular point of view.
For an exhaustive documention of the commands, please check out each command specific documentation at www.kikonf.com.

# 6 THE KIKACT COMMAND

Kikact stands for ***kick actions !***

The kikact syntax is:
***kikact <BAL>[,BAL]***

For more information about the BAL syntax see the kikact command documentation ANNEX 1 The BAL (Basic Action Locator).

Here is a simple use of kikact :

Running one Action:
**kikact was.crtserver**
Running more than one Action:
**kikact was.crtserver,was.jvm,was.jdbc,was.datasrc**

The single argument comes with a list of actions.
These actions are executed in the list order:
First was.crtserver then was.jvm, was.jdbc and last was.datasrc.

Note:
This order can be crucial because the server needs to pre-exist before to run jvm.
And the Jdbc driver must have been configured before to attach any datasource to it.

Behind each Action there is an Action plugin.

## 6.1 PLUGINS

Plugins are deployed into the **<KIKONF_INSTALL_PATH>\plugins\actions** directory.

Once deployed Action plugins are materialized by 4 aspects in kikonf :

- A simple sample for the Action is deployed at:
  **<KIKONF_INSTALL_PATH>/actions/<CATEGORY>.<ACTION_NAME>.xml**
- The Action's descriptor file is at:
  **<KIKONF_INSTALL_PATH>/actions/<CATEGORY>/<ACTION_NAME>/by/<WHO>/ACT_INF/action.xml**
- The plugin zip itself is at:
  **<KIKONF_INSTALL_PATH>/actions/<CATEGORY>/<ACTION_NAME>/by/<WHO>/<ACTION_NAME>.zip**
- The Action code and utilities are unziped into this directory
  **<KIKONF_INSTALL_PATH>/actions/<CATEGORY>/<ACTION_NAME>/by/<WHO>/<ACTION_NAME>**

Generally you don't bother with last the two aspects, except if you intend to develop Action plugins youself.

Note:
To see how to make your own Action plugin see the plugins_howto documentation at **www.kikonf.com.**

**A sample Action file is:**

```
<jvm>
   type = 'action'
   xms = '512'
   xmx = '1024'
   user='myuser' group='mygroup'
   >
   <scope node = 'localhostNode01' server = 'server1'/>
<jvm>
```

The **<KIKONF_INSTALL_PATH>/actions** directory lists a simple sample for all available actions. It is the default Action files directory.

When you run this command with no specific Action directory:
**kikact was.crtserver,was.jvm,was.jdbc,was.datasrc**

Kikonf picks up the was.crtserver.xml, was.jvm.xml, was.jdbc.xml, and was.datasrc.xml files from this directory.
There are many ways to specify another Action file directory (see the path resolution below).
If the Action file do not exist in the given directory an Error is thrown.

Obviously you need to setup each of these Action files
(was.crtserver.xml, was.jvm.xml, was.jdbc.xml, and was.datasrc.xml) before running the command.

And this is the point for delegation.

## 6.2 DELEGATION

### 6.2.1 *Restricting the use of Actions:*

Considering using this command (in a menu for instance) in order to delegate some actions to user paul : **kikact was.crtserver,was.jvm,was.jdbc,was.datasrc -c /paul/directory/actions**

The directory /paul/directory/actions is where are stored was.crtserver.xml, was.jvm.xml, was.jdbc.xml, and was.datasrc.xml.
Paul is allowed to update these files.

You can restrict the use of kikact only for these 4 Actions using a specific kikonf.attrs file.
For example by setting an environment variable for this user paul :

**set KIKONF_CATTRS=/where/is/the/secured/kikonf.attrs**

And kikonf.attrs may include these attributes:

**kikact_restricted_actions_list**=was.crtserver,was.jvm,was.jdbc,was.datasrc # Paul only have access to these 4 actions.
**kikarc_allowed**=False # Paul is not allowed to use the kikarc command.
**# kikact_cxml**=/paul/directory/actions # The action files directory can also be set into the kiko.attrs file, instead of being passed as option (e.g. -c /paul/directory/actions).

Now paul can run the command :

**kikact was.crtserver,was.jvm,was.jdbc,was.datasrc**

or later:

**kikact jvm**

But cannot run:

**kikact was.jmq**

Now we're happy cause paul can run a restricted list of actions.

But let's take a look to this Action file: was.jvm.xml

```
<jvm>
   type = 'action'
   xms = '512'
   xmx = '1024'
   user='myuser' group='mygroup'
   >
   <scope node = 'localhostNode01' server = 'server1'/>
<jvm>
```

The tag scope allows to specify a target node/server name for the jvm configuration

But Paul may not have authority to update any other servers of the cell.
You may also not want paul to change the xmx to fancy values, because your system memory is limited.

### 6.2.2 Restricting the Action file Tag/Attributes

Action file content restriction is based on Action restrictor file.
An Action restrictor file is a customized Action descriptor file

.
To restrict the use of some Action's Tag and/or Attributes,

- create a directory structured as this:
  **<MY_ACTION_RESTRICTOR_DIRECTORY>/<CATEGORY>/<ACTION_NAME>/by/<WHO>**
- copy the Action descriptor file from:
  **<KIKONF_INSTALL_PATH>/actions/<CATEGORY>/<ACTION_NAME>/by/<WHO>/ACT_INF/action.xml** to this directory.

For instance the result could be :
/path/to/my/restrictor/directory/was/jvm/by/kikonf/action.xml

Pass this directory to the kikact/kikarc commands either using the KIKACT_RST environment variable or with this entry into the kikonf.attrs file:
**kikact_crst =/my/restrictor/directory**

Note:
The restrictions only apply on Tag/Attributes included into the restrictor file all other Tag/Attributes still beeing checked in background by the descriptor file during the kiact/kikarc commands regular process.

In our situation the jvm.rst file could be :

```
<jvm>
   xms = {type:int,*lt:1024}
   xmx = {type:int,*lt:1024}
   >
   <scope node = '{*eq:localhostNode01}' server = '{*eq:server1}'/>
<jvm>
```

This restricts xms and xmx to a value leater than 1024,
node must be equal to:  localhostNode01 and server must be equal to server1.

# 7 ACTION FILES

Action files are per Action specific files defining the very setup we expect for each Action.

**The only command using Action files is the kikact command.**
Because the kikact command relays on an an arbitray list of Actions passed as argument,
it needs to know where is the setup for each of them.

An Action file is an xml file defining the setup for a specific Action, the default Action files directory is: **<KIKONF_INSTALL_PATH>/actions**.

When you run this command with no specific Action directory:
**kikact was.crtserver,was.jvm,was.jdbc,was.datasrc**

Kikonf picks up the was.crtserver.xml, was.jvm.xml, was.jdbc.xml, and was.datasrc.xml files from this directory.
There are many ways to specify another Action file directory (see the path resolution below).
If the Action file do not exist in the given directory an Error  is thrown.

The option –show (-s ) will show the Action file(s) content and there path(s):
**>kikact was.jvm,was.jdbc –s**

```
bal: was.jvm path: .../actions
<jvm>
   type = 'action'
   xms = '512'
   xmx = '1024'
   user='myuser' group='mygroup'
   >
   <scope node = 'localhostNode01' server = 'server1'/>
<jvm>

bal: was.jdbc path: .../actions
<jdbc type='action' bal='was.jdbc' name='myprovider'  path='/my/database/jdbc/path'>
   <scope node='localhostNode01' server='myserver'/>
   <db2 xa='true' jars='db2jcc.jar;db2jcc_license_cu.jar;Another_path'/>
</jdbc>
```

### 7.1.1 Content

As any xml file an Action file shows a set of Tag and Attributes.
These Tag and Attributes are strictely described into a specific file coming with the Action plugin : the Action descriptor file.

See the chapter below "Action descriptor file" to understand there structure.

**If you try to define a Tag an Attribute or a non supported value, the corresponding Action is rejected by the command with one explicite Error concerning this Tag/Attribue or value.**

### 7.1.2 Path resolution

In order to find the Action files kikact operates a path resolution in that order :

- **As an attribute of the kikonf.attrs file**
  The path of the directory of the Action files can be given as an attribute of
  the kikonf.attrs file:

  **kikact.cxml**=/where/are/actions/files
  Note:
  In this case all other ways to pass the action's directory are refused and a warning is sent.

- **As parameter**
  If kikact.cxml is not defined the path of the directory of the Action files can be given as parameters :

  **kikact -c /where/are/actions/files**
  or
  **kikact --cxml /where/are/actions/files**

- **As environment variable**
  If kikact.cxml is not defined and the directory of the Action files is not guiven
  as parameters, the action's directory can be set like this:

  Example :
  **set kikact_cxml=/where/are/actions/files**

- **By default**
  If kikact.cxml is not defined by any of the ways above by default the ACTION.xml files are searched into:

  **<KIKONF_INSTALL_PATH>/conf/actions**

# 8 THE XML CONTROLER CYCLE

Kikonf is based on two parsers :

- a Jython compatible ligth parser : picxml (previsouly called jxparser in kikonf early versions, picxml at sourceforge.net)
- a complete parser : epicxml (epicxml at sourceforge.net)

The frontal job is always made by epicxml while picxml will only run into jython.

When epicxml load a new xml file 3 scenarios are possible:

**First the xml as no descriptor file:**
The xml file is loaded as this, whith no more check than the standard xml structure.

**Second the file has a descriptor file:**

> The descriptor controller is run to check the xml file.
> All default value and other values resulting from the xml check, update the in memory loaded xml file.

**Third the file has a descriptor file and a restrictor file:**

> The restrictor controller is run to check the xml file.
> All default value and other values resulting from the xml check, update the in memory loaded xml file.
> The descriptor controller is run to check the xml file
> All default value and other values resulting from the xml check update the in memory loaded xml file.

Action files always come with one descriptor file, as illustrated into the next chapter.

Restrictor files can be provided to Action file to restrict user access to Action files.

# 9 ACTION DESCRIPTOR FILE

The Action descriptor file is defined by the plugin designer (or Action provider) who knows exactly what kind of setup is supported by its Action.

TheAction descriptor file path is always the fileACT_INF/action.xml into the plugin directory:
**<KIKONF_INSTALL_PATH>/actions/<CATEGORY>/<ACTION_NAME>/by/<WHO>/ACT_INF/action.xml**
and cannot be surdefined.

Note about **by/<WHO**
WHO is the Action designer (or Action provider) name.
e.g.: by/andrew

Usually all the Action provided by the kikonf team are under by/kikonf.
This notation allows you to provide any of your Actions under by/<MY_NAME/>.
Hence your Actions may be published on www.kikonf.org under by/<MY_NAME/> (*)

At the same level as the directory "by" theres always is a file called default.txt saying the name of the default designer (e.g. kikonf).

When referecing an Action with a BAL with no bywho clause e.g.:was.jvm
this default is matched.

A byWho clause may be explicited in a BAL like this:
was.jvm**.by.kikonf** (e.g. was.jvm**.by.thomas**).
For more information about the BAL syntax see the kikact command documentation
ANNEX 1 The BAL (Basic Action Locator).

An Action descriptor file is an xml file.
The Action descriptor file describes allowed Tags, Attributes and Attribute's value types,
it and can be seen as a WYSISWYG dtd or a simplified .xsd.

Running kikonf commands, if you have defined into any Action file, a non described **Tag(s),** Attribute(s) or a non supported value(s), the corresponding Action is rejected with one explicite Error concerning this Tag/Attribue or value.

Except **you are the Action designe**r, you are not supposed to update those file. But they can be copied to make restrictor files.

To see the descriptor for a given (set of) Action(s) type:
**>kikonf was.jvm –show_descriptor** (todo)

*bal: was.jvm path: ..*

```
<crtserver
   type = '{\*eq:action,\*required:True\}'
   bal = ''
   sub_type = '{\
      *value:configuration,\
      *eq:configuration,\
      *required:True\
   }'
   softwares = "{\
      '*type':'dict',\
      '*dtype':\
         {\
            'was':\
            {\
               '*ge':'6.1',\
               '*le':'7.9',\
               '*required':True\
            }\
         }\
   }"

   template='{*value:default,*required:True}'
   weight='{*value:2,*type:int,*required:True}'
   >
```

## 9.1 CONTENT

With the previou sample.

- **Top tag** (e.g. crtserver)

The top Tag is always the Action name.

- **Reserved Top Tag Attributes**

The four starting Top tag's Attributes are **reserved Attributes** and come with every Action descriptor file.

**type = action. Always equal to action and must be defined into the Action file**.

**bal = ''.** Supported but only need to be defined when an Action file is **pasted into a custom file**.
For more information about the BAL syntax see the kikact command documentation
ANNEX 1 The BAL (Basic Action Locator).
For more information about sutom file, see the chapter custom files.

**sub_type** = configuration/control. **Not has to be defined into Action files**. Defines this Action as either a "configuration" or a "control" Action, for the kikonf engine (Only the run operation is run for "crontrol").

**softwares Not has to be defined into Action files**. Defines a dictionary of tsoftware and the supported versions by this Action, for the kikonf engine.

- **All other tags/Atrribut**e: may be defined into Action files.

## 9.2 ATTRIBUTES DESCRIPTION

The stuff coming after every "=" is a specific notation called wk (Wanted Keyword) notation.

wk (Wanted Keyword) is a tools and notation that help to define and control simple and complexe types.
For more information about the wk (wanted keyword) notation see the **wk (Wanted Keyword) notation documentation** into <kikonf_intall_root>/doc or at www.kikonf.com or the wk project at sourceforge.net.

A wk expression is a keys/values pair python dictionary.
Afterward This dictionary is python evaluated through a restricted eval.

**It can be given as a fully formatted python like expression :**

```
  weight='{
         "*value":2,
         "*type":int,
         "*required":True
}'
<=>
  weight='{ "*value":2, "*type":int, "*required":True }'
```

**or CoolTyped :**

```
  weight='{\
         *value:2,\
         *type:int,\
         *required:True\
}'
<=>
  weight='{*value:2,*type:int,*required:True}'
```

Note:
A CoolTyped expression is a python type expression with no "“" and no """.
Please note that it is also an expression with no space arround type separators characters like ",", ":", "{", "}", "[", "]" or "(", ")".
For instance:
**name={*value:this is my name}**, is valide and is translated to name={"*value": "this is my name"}
The following expression is not valide:
name={*value: this is my name }
Because there is a space after ":" and before"}".
For more information CoolTyping see the CoolTyping (or ct) documentation on www.sourceforge.net.

The general rule is to use Cooltyped expression wherever it is possible in order to simplify reading/writing, and python expression where it is not.

### 9.2.1 Wk definition samples

Here are some sample wk definitions, from the previous Action crtserver's descriptor file:

- **bal=''**

When nothing is given, this mean that:
This Attribute is supported.
Its type is the default type: str (string).
Its default value is None.

- **weight='{*value:2,*type:int,*required:True}'**

This Attribute is supported.
Because the key *value is given the default value for weigth is 2.
Its type is: int (integer).
Because the key: *required is given and is True, this attribute is required.
(None cannot be forced on this Attribute using weight="None" into an Action file).
All Attribute defined with a default value ("*value" key), do not need to ,be redefined into the Action file.

Note:
For more information about the wk (wanted keyword) notation see the **wk (Wanted Keyword) notation documentation** into <kikonf_intall_root>/doc or at www.kikonf.com or the wk project at sourceforge.net.

### 9.2.2 Extended wkkeys

Extended wkkeys are wkkeys that are not published into the wk notation documentation but that are implemented for kikonf.
Here is a list of wk extended wkkeys :

***help:short string**
Provide short help for the attribute.

***lhelp:long string**
Provide long help for the attribute.

help or lhelp multilangue support:
*help:%lang/<DICTIONARY_KEY>/ENTRY_KEY

Multilangue support is supplied switching the help effective value by a key.

DICTIONARY_KEY: guives a path where to search the dictionay,
actually under the <kikonf_install_root>/langs directory.
ENTRY_KEY: guives the entry into the dictionary.

e.g.:
*help:%lang/action.was.en/scope.help
The help string is retreived from the " entry scope.help" within the
dictionary: <kikonf_install_root>/langs/action.was.en.

***display:true|false**
Is this attribute displayable on gui inteerfaces.

Note:
help, lhelp and display wkkeys, are merely used for gui purpose, like in kikupd launched with option –console.

***deny: true|false**
If false this attribute is not allowed.

Hence if this attribute exists into the loaded xml file for the described tag, an exception is thrown.
So what is this dumb wkkey stands for ?

Assuming that the attribute "name" is described like this:
name:{*value:myqueue1,*deny:true}

The attribute: name wont be allow into the xml file, except if its value match the default value: "myqueue1".
And either the attribute name exists into the xml file, for the tag or not,
its value will always be forced to "myqueue1" by the descriptor controller.

This allows to force values for specific attributes, this functionality is merely used into restritors.

### 9.2.3 The special tag's Attribute __wk__

The special tag's Attribute: __wk__ provide the support of wk definition not for a specific attribute,
but for the tag itself.

e.g.:
__wk__ = {*help:Here is some help for my tag}

Will implement help for the tag.

**Node occurences number management**

The wkkeys *eq, *lt, *le, *gt, *ge and *between allow to restrain the number of node allowed for the described tag..

e.g.:
__wk__ = {*eq:1,*help:Here is some help for my tag}
Only one node is supported for this tag.

__wk__ = {*gt:3}
More than 3 nodes are required for this tag.

__wk__ = {*ge:3,*le:7}
The number of node for this tag must be greater or equal to 3
and leater or equal to 7.

This is equivalent to:
__wk__ = {*between:(3,7)}

Note:

For Action designers only.
Sometime you may find two ways to write the boolean type :
True/False or true/false.
The first syntax is the python syntax and is expected into all wkkey satements except for values wich are free and refers to what is expected by the target software configuration.

This exemple from the was.tp Action show this situation:
is_growable='{*value:**false**,*checkIn:(**false,true**),*required:**True**}'

Green is the python expectation do we want *required True or False.
Blue is the values expected by the target software for the Attribute is_growable either "true" or "false".

## 10 ACTION RESTRICTOR FILES

A restrictor file is a descriptor file but with a tighter range of allowed Tag/Attributes or Attributes values.

There is no default path for restrictor file.
To create a restrictor path do the following:

- create a directory structured as this:
  **<MY_ACTION_RESTRICTOR_DIRECTORY>/<CATEGORY>/<ACTION_NAME>/by/<WHO>**

- copy the Action descriptor file from:
  **<KIKONF_INSTALL_PATH>/actions/<CATEGORY>/<ACTION_NAME>/by/<WHO>/ACT_INF/action.xml** to this directory.

For instance the result could be :
/my/restrictor/directory/was/jvm/by/kikonf/action.xml

Note about **by/<WHO:**
See the by/WHO note of previous chapter.

To pass this directory to the kikact/kikarc commands either use the KIKACT_RST environment variable or this entry into the kikonf.attrs file:
**kikact_crst =/my/restrictor/directory.**

To see if any Action restrictor file apply to a set of Action type:
**>kikonf was.jvm –show_restrictor** (todo)

```
bal: was.jvm path: ..
<jvm>
   ...
   xms = {type:int,*lt:1024}
   xmx = {type:int,*lt:1024}
  ...
   >
   <scope node = '{*eq:localhostNode01}' server = '{*eq:server1}'/>
<jvm>
```

This restricts xms and xmx to a value leater than 1024,
node must be equal to: localhostNode01 and server must be equal to server1.

## 10.1 CONTENT

A restrictor file is a descriptor file but with a tighter range of allowed Tag/Attributes or Attributes values.

See the the chapter "The xml controler cycle" to understand how the check sequence occures when both descriptor and restrictor files are provided.

### *10.1.1 Restriction rules*

The two reserved attributes **type** and **bal** should always be surdefined like the following :

```
   type = '{\
      *eq:action,\
      *required:True,\
      *display:False\
   }'
   bal = '{*display:False}'
```

The two others: configuration and sub_type are not to be surdefined.

For tags and attributes defined into the restrictor file these rules applies :

If a tag or an attribute **is not defined** into the restrictor file **it is denied.**
To allow an Attribute juste describe it with the more permissive wk: ''.
e.g.:
Guiven the attribute size defined at the descriptor level like this:
**size**='{*value:50,*type:int,*required:True}'
Just to allow size, with no more restriction into the restritor file just type it like this:
**size**=''

Assuming the xml controler (see chapter above) checks the xml file twice, once on the restrictor and second on the descriptor file, any wk values are supported to redefined tags and attributes.

### *10.1.2 Sample*

Samples of restrictor files are provided into the <kikonf_install_root>/tests/tests/restrictors_dir directory.

For the  purpose of this explaination we take the cache Action
and its sample provided at <kikonf_install_root>/tests\tests\restrictors_dir\was\cache.rst.xml.

**Here is the cache Action's descriptor file:**
(at <kikonf_install_root>/plugins/actions/was/cache/by/kikonf/ACT_INF/action.xml)
The reserved Attributes are skipped.

```
<cache
    type = ... skipped
    bal = ... skipped
    sub_type = ... skipped
    softwares = ... skipped

    name='{*required:True}'
    jndi_name='{*required:True,*label:jndi name}'
    size='{*value:50,*type:int,*required:True}'
    prefix='{*help:%lang/action.was.en/prefix.help}'
    desc=''
    __wk__='{\
        *help:%lang/action.was.en/cache.help,\
        *lhelp:%lang/action.was.en/cache.lhelp\
    }'
    >

    <scope
        cell = '{*value:false,*checkIn:(false,true),*required:True}'
        node = '' server = '' cluster = ''
        __wk__='{\
            *eq:1,\
            *help:%lang/action.was.en/scope.help,\
            *lhelp:%lang/action.was.en/scope.lhelp,\
        }'
    />

</cache>
```

**Here is the restrictor file provided for the cache Action:**
(at <kikonf_install_root>/tests/tests/restrictors_dir/was/cache.rst.xml)

```
 1. <cache
 2.     type = '{\
 3.        *eq:action,\
 4.        *required:True,\
 5.        *display:False\
 6.     }'
 7.     bal = '{*display:False}'
 8.
 9.     name='{*value:mycache,*eq:mycache,*deny:True}'
10.     jndi_name='{*value:cache/mycache2,*eq:cache/mycache2,*deny:True}'
11.     size='{*between:(50,100),*type:int,*required:True}'
12.     prefix=''
13.     desc=''
14.     __wk__='{*help:new help,*lhelp:new lhelp}'
15.     >
16.
17.     <scope
18.        cell = '{*value:false,*checkIn:(false,true),*required:True}'
19.        node = ''
20.        server = ''
21.        cluster = ''
22.     />
23.
24. </cache>
```

Line 9:
The attribute name is defined into the previous cache descriptor file like this:
**name**='{*required:True}'
Surdefining name with this wk into the restrictor file:
**name**='{*value:mycache,*eq:mycache,*deny:True}'
will force its value to mycache.

This wkkey: ***value:mycache**, defaults the name attribute's value to "mycache", if the attribute name do not exist into the xml file.
This wkkey: ***eq:mycache**, force the value to be "mycache", if the attribute name exists into the xml file.

Line 10:
The same as for line 9.

Line 11:
The attribute size is defined into the previous cache descriptor file like this:
**size**='{*value:50,*type:int,*required:True}'
Surdefining size with this wk into the restrictor file:
**size**='{*between:(50,100),*type:int,*required:True}'
will force restriction of its values between 50 and 100.

Lines 12 and 13:
Simply allow the attributes prefix and desc.

Line 14:
The tag's descriptor attribute: __wk__ is defined into the previous cache descriptor file like this:

```
__wk__='{\
         *help:%lang/action.was.en/cache.help,\
         *lhelp:%lang/action.was.en/cache.lhelp\
     }'
```

Surdefining __wk__ with this wk into the restrictor file:
**__wk__**='{*help:new help,*lhelp:new lhelp}'
will change the help and long help values for this tag.

Line 17 to 22:
To force the tag scope to a **specific node and server** it could have been defined like this:

```
<scope
    node = '{*value:mynode,*eq:mynode}'
    server = '{*value:myserver,*eq:myserver}'
/>
```

**Here is the sample provided for Action:**
(at <kikonf_install_root>/actions/cache.xml)

```
<cache
   type = 'action'
   name='mycache'
   jndi_name='cache/mycache2'
   size='100'
   >

   <scope node='localhostNode01' server='server1'/>

</cache>
```

This command
**kikact was.cache -r /my/restrictors/dir -v3**

(by default) would run this Action file on the two previsous descriptor and restrictor files.

## 10.2 PATH RESOLUTION

In order to find the Action restrictor files kikact operates a path resolution in that order :

- **As an attribute of the kikonf.attrs file**
  The path of the directory of the action restrictor files can be given as an attribute of the kikonf.attrs file:

  **kikact_crst =/my/restrictors/dir**
  Note:
  In this case all other ways to pass the action restrictors directory are refused and a warning is sent.

- **As parameter**
  If kikact.cxml is not defined the path of the directory for the Action restrictor files can be given as parameters :

  **kikact -r /my/restrictors/dir**
  or
  **kikact --crst /my/restrictors/dir**

  e.g.:
  ```
  kikact was.cache -r <kikonf_install_root>/tests/tests/restrictors_dir -v10
  kikupd was.cache -r <kikonf_install_root>/tests/tests/restrictors_dir --console
  kikarc c.xml -r <kikonf_install_root>/tests/tests/restrictors_dir -v10
  ```

- **As environment variable**
  If kikact.cxml is not defined and the directory of the Action restrictor files is not given as parameters, the Action restrictors directory can be set like this:

  Example :
  **set kikact_cxml=/where/are/actions/files**

# 11 THE KIKARC COMMAND

Kikarc stands for ***kick architecture !***

Here is a simple use of kikarc :

```
kikarc myapplication.xml
```

The argument is a custom xml file.

## 11.1 Introduction

In industrialized context, some Organization/Company may need to declare all the resources requiered for one business operation in just one file.
Lately, this configuration file, stored in a safe place for backup and recovery,
will garantee and represent the configuration of this operation.
This operation may cover several distincts softwares.

Note:
Farther in this doc we will refer to this complexe operation as the configuration needs to configure an Application, because this J2EE concept is taken as exemple (as many others could be).
And this exemple is relevant because middle ware Applications by design interact with many softwares.
Talking about Application here is not simply talking about ear files, but about
all the resources need in the Enterprise to setup the Application.
An Application may need :
WebSphere configuration, MQ configurations and IHS configurations, (and any other Kikonf supported software), to run.

Kikarc allows to declare all these resources, as a whole, in one file : the custom xml file.

## 11.2 The custom xml file

The kikarc command only supports one file: the custom file (and not many Action files like kikact).
The custom file is an xml file.
This xml file can incorporate as much Actions tag as need.

A custom file can be seen as a big xml file where several Action files are pasted in.
This xml file can include as much custom (non Action) tags as need.

### 11.2.1 *Overview and Content*

#### 11.2.1.1 *simple custom file*

This is a simple custom file for Application invoice:

invoices.xml :

```
<myapplication>

   <jvm type='action' bal='was.jvm' xms = '512' xmx = '1024'  user='myuser' group='mygroup'>
        <scope server='srv_invoices_uat_01' node='axaneUatNode01'/>
   </jvm>

   <jmq type='action' bal='was.jmq'>
        <scope server='srv_invoices_uat_01' node='axaneUatNode01'/>
       <qcfs>
           <qcf name='myqcf' jndi_name='jms/myqcf' queue_manager='myqm' host='myhost'  port='1136'/>
        </qcfs>
       <queues>
            <queue name='myqueue' queue='myqueue' jndi_name='jms/myqueue' queue_manager='myqm1' host='myhost' port='1136'/>
       </queues>
   </jmq>

   <conf type='action'  bal='ihs.conf'  host='dmz_uat'>
        <scope server='ihs_invoices_uat'/>
   </conf>

</myapplication>
```

Three Actions are setup here:

- was.jvm : an WebSphere Application Server ®  jvm configuration Action.
- was.jmq : an WebSphere Mq ®  configuration Action.
- ihs.conf : an IBM HttpServer ®  configuration Action.

Including Actions into the custom file is the same as pasting Action files into the custom xml file, except that the Attribute **bal** is required.
Because the bal (Basic Action Locator) cannot be passed as argument like it's done for the kikact

command, Action locations are signaled to the kikonf engine using this Attribute.

Every tags coming with the Tag/Attribute type="action" are Action Tags.
The structure of the Action Tags are exactly the same as defined by their Action descriptor file.

### 11.2.1.2 custom tags

The xml file can include any custom tags, here is a sample :

```
<myapplication>

   <server host='axane'  ip='10.100.1.1' >
      <jvm type='action' bal='was.jvm' xms = '512' xmx = '1024'  user='myuser' group='mygroup'>
          <scope server='srv_invoices_uat_01' node='axaneUatNode01'/>
      </jvm>

      <jmq type='action' bal='was.jmq'>
          <scope server='srv_invoices_uat_01' node='axaneUatNode01'/>
         <qcfs>
            <qcf name='myqcf' jndi_name='jms/myqcf' queue_manager='myqm' host='myhost'  port='1136'/>
          </qcfs>
         <queues>
             <queue name='myqueue' queue='myqueue' jndi_name='jms/myqueue' queue_manager='myqm1' host='myhost' port='1136'/>
         </queues>
      </jmq>
   </server>

   <conf type='action'  bal='ihs.conf' host='dmz_uat'>
       <scope server='ihs_invoices_uat'/>
   </conf>

</myapplication>
```

The Tags marked in green are custom Tags.
You may use as many custom Tags you want because this xml reflects your business scheme structure for this operation.
Any custom Tags at any level can be added into the xml file.

Note:
This is possible, because the kikarc parser will sequentially read the whole xml file,
and only execute the actions Tags.

At this point we configure a jvm named:  srv_invoices_uat_01 on node:  axaneUatNode01 for physical server: axane.

### 11.2.1.3 Action tags imbrication

Imbrication of Action nodes is allowed only at the first level (or at the Action level).

The following imbricates the Action node "jmq" into the Action node "jvm" at the first level as a direct child of the Action node: jvm, is allowed.

```
<jvm type='action' bal='was.jvm' xms = '512' xmx = '1024'  user='myuser' group='mygroup'>
            <scope server='srv_invoices_uat_01' node='axaneUatNode01'/>
            <jmq type='action' bal='was.jmq'>
                     <scope server='srv_invoices_uat_01' node='axaneUatNode01'/>
                     <qcfs>
                                 <qcf name='myqcf' jndi_name='jms/myqcf' queue_manager='myqm' host='myhost'  port='1136'/>
                      </qcfs>
                      <queues>
                                 <queue name='myqueue' queue='myqueue' jndi_name='jms/myqueue' queue_manager='myqm1'
                                    host='myhost'  port='1136'/>
                     </queues>
         </jmq>
</jvm>
```

The following imbricates the Action node "jmq" into the Action node: "jvm" not at the first level, but as a children of the node "scope", wich is a children of theAction node: "jvm".
This forbidden !

```
<jvm type='action' bal='was.jvm' xms = '512' xmx = '1024'  user='myuser' group='mygroup'>
            <scope server='srv_invoices_uat_01' node='axaneUatNode01'/>
                  <jmq type='action' bal='was.jmq'>
                      <scope server='srv_invoices_uat_01' node='axaneUatNode01'>
                      <qcfs>
                                  <qcf name='myqcf' jndi_name='jms/myqcf' queue_manager='myqm' host='myhost'  port='1136'/>
                       </qcfs>
                       <queues>
                                  <queue name='myqueue' queue='myqueue' jndi_name='jms/myqueue' queue_manager='myqm1'
                                     host='myhost'  port='1136'/>
                      </queues>
                 </jmq>
          </scope>
</jvm>
```

Note:
The informations coming now in the rest of this chapter are advanced technics.
You do not need to know these tricks to efficiently use kikonf.
Although using them may save your time and help custom file readibility.

The patterns "invoices" from  srv_**invoices**_uat_01 cand be deduced from the Application'name invoices, we'll now use Alias for that.

The Application Server may not exist, so let's create it with action: crtserver.

#### 11.2.1.4 special tag alias

Alias are variables that can be declared at any level of the xml file and that are reusable like this : $myvar or $[myvar].

```
<myapplication>
   <__alias__  name='app'   value='invoices'/>

   <server host='axane'  ip='10.100.1.1' >
       <__alias__  name='node'   value='axaneUatNode01'/>

       <crtserver type='action' bal='was.crtserver'>
            <scope server='srv_$[app]_uat_01' node='$[node]'/>
       </crtserver>

       <jvm type='action' bal='was.jvm' xms = '512' xmx = '1024'  user='myuser' group='mygroup'>
            <scope server='srv_$[app]_uat_01' node='$[node]'/>
       </jvm>

       <jmq type='action' bal='was.jmq'>
            <scope server='srv_$[app]_uat_01' node='$[node]'/>
           <qcfs>
               <qcf name='myqcf' jndi_name='jms/myqcf' queue_manager='myqm' host='myhost'  port='1136'/>
            </qcfs>
           <queues>
                <queue name='myqueue' queue='myqueue' jndi_name='jms/myqueue' queue_manager='myqm1' host='myhost' port='1136'/>
           </queues>
       </jmq>
   </server>

   <conf type='action'  bal='ihs.conf'  host='dmz_uat''>
        <scope server='ihs_$[app]_uat'/>
   </conf>

</myapplication>
```

The Aliases (marked in green) are declared and reused to defined the server and node names later in the xml file.

Suppose we had 2 Application Servers to create for the host axane, the file would be :

```
<myapplication>
   <__alias__  name='app'   value='invoices'/>

   <server host='axane'  ip='10.100.1.1' >
       <__alias__  name='node'   value='axaneUatNode01'/>

      <!-- JVM n°1 -->
       <crtserver type='action' bal='was.crtserver'>
            <scope server='srv_$[app]_uat_01' node='$[node]'/>
       </crtserver>
       <jvm type='action' bal='was.jvm' xms = '512' xmx = '1024'  user='myuser' group='mygroup'>
            <scope server='srv_$[app]_uat_01' node='$[node]'/>
       </jvm>
       <jmq type='action' bal='was.jmq'>
            <scope server='srv_$[app]_uat_01' node='$[node]'/>
           <qcfs>
               <qcf name='myqcf' jndi_name='jms/myqcf' queue_manager='myqm' host='myhost'  port='1136'/>
```

```
          </qcfs>
         <queues>
              <queue name='myqueue' queue='myqueue' jndi_name='jms/myqueue' queue_manager='myqm1' host='myhost' port='1136'/>
         </queues>
    </jmq>

    <!-- JVM n°2 -->
     <crtserver type='action' bal='was.crtserver'>
          <scope server='srv_$[app]_uat_02' node='$[node]'/>
    </crtserver>
     <jvm type='action' bal='was.jvm' xms = '512' xmx = '1024'  user='myuser' group='mygroup'>
          <scope server='srv_$[app]_uat_02' node='$[node]'/>
     </jvm>
     <jmq type='action' bal='was.jmq'>
          <scope server='srv_$[app]_uat_02' node='$[node]'/>
         <qcfs>
             <qcf name='myqcf' jndi_name='jms/myqcf' queue_manager='myqm' host='myhost'  port='1136'/>
          </qcfs>
         <queues>
              <queue name='myqueue' queue='myqueue' jndi_name='jms/myqueue' queue_manager='myqm1' host='myhost' port='1136'/>
         </queues>
     </jmq>

   </server>

   <conf type='action'  bal='ihs.conf'  host='dmz_uat'>
        <scope server='ihs_$[app]_uat'/>
   </conf>

</myapplication>
```

The file becomes quite bloated ?
Now if  we had 4 jvms to create for server axane, it would be better to use the special Tag **foreach** combinated with **alias** :

### 11.2.1.5 special tag foreach

Foreach allows to loop over the several occurences of a given contextual tag.

```
<myapplication>

  <__alias__ name='app'  value='invoices'/>

  <server host='axane'  ip='10.100.1.1' >
           <__alias__ name='node'  value='axaneUatNode01'/>
          <index  value='01'/>
          <index  value='02'/>
          <index  value='03'/>
          <index  value='04'/>

          <__foreach__  tag='index'>
                     <__alias__  name='index'  value='pxq:bu.index@value'/>

                    <crtserver type='action' bal='was.crtserver'>
                           <scope server='srv_$[app]_uat_$[index]' node='$[node]'/>
                    </crtserver>
                    <jvm type='action' bal='was.jvm' xms = '512' xmx = '1024'  user='myuser' group='mygroup'>
                           <scope server='srv_$[app]_uat_$[index]' node='$[node]'/>
                    </jvm>

                    <jmq type='action' bal='was.jmq'>
                        <scope server='srv_$[app]_uat_$[index]' node='$[node]'/>
                        <qcfs>
                             <qcf name='myqcf' jndi_name='jms/myqcf' queue_manager='myqm' host='myhost'  port='1136'/>
                        </qcfs>
                        <queues>
                             <queue name='myqueue' queue='myqueue' jndi_name='jms/myqueue' queue_manager='myqm1'
                                 host='myhost' port='1136'/>
                        </queues>
                    </jmq>
          </__foreach__ >
  </server>

  <conf type='action'  bal='ihs.conf'  host='dmz_uat'>
      <scope server='ihs_$[app]_uat'/>
  </conf>

</myapplication>
```

We just replaced the arbitrary indice:01 value by the alias: index value.
And we incorporated the server configuration (Actions jvm and jmq) into a foreach tag.

The Attribute "tag" of this foreach special Tag has the value: "index".
This means that at parsing time:
foreach encountred index tag at the same context (at the same level) as the foreach tag,
the kikonf engine will incorporate all the content of the foreach tag, than destroy the empty foreach Tag.
Hence the special Alias Tag: __alias__ name='index', will retreive each index value (from 01 to 04).

Note:
To show the resulting xml file after kikonf parsed special tags, without running any operation type the command: **kikarc -s (--show)** (todo)

Here, the 3 Actions crtserver, jvm and jmq are applied to the 4 new jvms :
srv_invoices_uat_01, srv_invoices_uat_02, srv_invoices_uat_03 and srv_invoices_uat_04.
:
Note
alias index retreives the index value using a **PxQuery** instruction: pxq:bu.index@value.
For more information about PxQuery see chapter PxQuery.

This sample could even more complicated.
Now if we had 3 physical servers to configure :

```
<myapplication>

   <__alias__  name='app'   value='invoices'/>

   <server host='axane1'  ip='10.100.1.1'  node='axaneUatNode01'/>
   <server host='axane2'  ip='10.100.1.2'   node='axaneUatNode02'/>
   <server host='axane3'  ip='10.100.1.3'   node='axaneUatNode03'/>

   <__foreach__   tag='server'>
             <__alias__  name='node'   value='pxq:bu.server@node'/>


            <index  value='01'/>
            <index  value='02'/>
            <index  value='03'/>
            <index  value='04'/>

            <__foreach__   tag='index'>
                        <__alias__  name='index'   value='pxq:bu.index@value'/>

                        <crtserver type='action' bal='was.crtserver'>
                                <scope server='srv_$[app]_uat_$[index]' node='$[node]'/>
                        </crtserver>
                        <jvm type='action' bal='was.jvm' xms = '512' xmx = '1024'  user='myuser' group='mygroup'>
                                <scope server='srv_$[app]_uat_$[index]' node='$[node]'/>
                        </jvm>

                        <jmq type='action' bal='was.jmq'>
                             <scope server='srv_$[app]_uat_$[index]' node='$[node]'/>
                             <qcfs>
                                    <qcf name='myqcf' jndi_name='jms/myqcf' queue_manager='myqm' host='myhost'  port='1136'/>
                             </qcfs>
                             <queues>
                                    <queue name='myqueue' queue='myqueue' jndi_name='jms/myqueue' queue_manager='myqm1'
                                         host='myhost' port='1136'/>
                             </queues>
                        </jmq>
            </__foreach__>
    </__foreach__>

   <conf type='action'  bal='ihs.conf'  host='dmz_uat'>
        <scope server='ihs_$[app]_uat'/>
   </conf>

</myapplication>
```

Know we have seen the main characteristics of the writing of custom xml file for kikarc,
let dive into their mechanism.

## 11.3 SPECIAL TAGS

Note:
_ The informations coming now in the rest of this chapter are advanced technics.
You do not need to know these tricks to efficiently use kikonf.
Although using them may save your time and help custom file readibility.

Special tags always appear betwen two : "__".

### *11.3.1 Showing the resulting custom file after special expressions evaluation*

To Shows the resulting custom file after the evaluation of all special tags,
type the command: **kikarc c.xml -s (--show)** (todo)

### *11.3.2 Alias*

Alias are variables wich can be declared at any level of the xml and wich are reusable like this : $myvar or $[myvar].

#### *11.3.2.1 Simple alias*

Here is a sample use of alias.

```
1.  <myapplication>
2.     <__alias__  name='host'   value='axane'/>
3.     <server host='$host'  ip='10.100.1.1' />
4.  </myapplication>
```

Line 2. The alias named **host** is given an explicit value **axane**.
Line 3. The alias **host** is used to define the host attribute value of the node server : **host ↔ axane .**

The nodes __alias__ and server are child of the same node (kikarc), they are called contextual node.
Because they are child of the same father node (they share the same context).

#### *11.3.2.2 Alias context and inheritance*

Alias are contextuals and inheriteds.

Here are two aliases defined at two distinct levels :

- alias **app** at node level : kikarc

app is seeable by all nodes at the same level or context (direct childs of kikarc), ex: server.
app is seeable by all sub nodes, ex: server/__alias__ or server/crtserver.

- alias **node** at tag level : server.
  node is seeable by all nodes at the same level or context (direct childs of server), ex: crtserver.
  node is seeable by all sub nodes.

```
1.  <myapplication>
2.     <__alias__ name='app'   value='invoices'/>
3.
4.     <server host='axane'  ip='10.100.1.1' >
5.               <__alias__  name='node'   value='axaneUatNode01'/>
6.              <crtserver type='action' bal='was.crtserver'>
7.                          <scope server='srv_$[app]_uat_01'   node='$[node]"/>
8.              </crtserver>
9.      </server>
10. </myapplication>
```

Line 2. The alias named **app** is guiven an explicit value **axane**.
Line 7. The alias **app** is used to define the attribute server of the sub node scope : server ↔ **srv_invoices_uat_01 .**
Line 5 The alias named **node** is guiven an explicit value **axaneUatNode01** .
Line 7. The alias **node** is used to define the attribute node of the node scope : **node ↔ axaneUatNode01.**

#### *11.3.2.3 Alias Surdefition*

An alias defined at a level can be surdefined at a sub level.

```
1.  <myapplication>
2.     <__alias__ name='app'   value='invoices'/>
3.
4.     <server host='axane'  ip='10.100.1.1' >
5.               <__alias__  name='app'   value='contracts'/>
6.               <__alias__  name='node'   value='axaneUatNode01'/>
7.              <crtserver type='action' bal='was.crtserver'>
8.                          <scope server='srv_$[app]_uat_01'   node='$[node]'>
9.              </crtserver>
10.     </server>
11. </myapplication>
```

Line 8. Because the alias app has been surdefined (line 9) , the attribute server of the node scope becomes : server ↔ **srv_contracts_uat_01 .**

#### 11.3.2.4 *Alias substring and concatenation (experimental)*

```
1.  <myapplication>
2.     <__alias__ name='app'   value='invoices'/>
3.     <__alias__ name='node'   value='axaneUatNode01'/>
4.     <server host='$[node:0:4]'  ip='10.100.1.1' >
5.              <__alias__  name='env'   value='$[node:5:7]'/>
6.             <crtserver type='action' bal='was.crtserver'>
7.                  <scope server='srv_$[app]_uat_$[node:12:13]'   node='$[node]'/>
8.             </crtserver>
9.     </server>
10. </myapplication>
```

Line 4. The value of the attribute host of the tag server is obtained from a susbstituion : **host ↔ axane.**
Line 5. Note, that an alias can be defined from another alias.
Line 7. The value of the attribute server of the tag scope is obtained from a concatenation and a substitution : **server ↔ srv_invoices_uat_01.**

**The substitution syntax is :**

```
$[ALIAS:POS_START:POS_END]
```

POS_START : 0 is the first character.
POS_END : is inclusive.

#### 11.3.2.5 *Alias and PXQUERY*

Alias values can be retreived from the value of other attributes, of other tags, of the xml file.
This the purpose of PXQUERY. For more information about aliases and PXQUERY please see the chapter PXQUERY.

### 11.3.3 *Foreach*

Foreach allows to loop over the several occurences of a given contextual tag.

The kikarc parser sequentially read the whole xml file and when a foreach tag is met its content is duplicated for all occurences of the refenced tag.

Let take a look to the previous example:

```
<myapplication>

   <__alias__  name='app'   value='invoices'/>

   <server host='axane'  ip='10.100.1.1' >
```

```
        <__alias__  name='node'   value='axaneUatNode01/'>
        <index  value='01'/>
        <index  value='02'/>

        <__foreach__   tag=index>
               <__alias__  name='index'   value='pxq:bu.index@value/'>

               <crtserver type='action' bal='was.crtserver'>
                    <scope server='srv_$[app]_uat_$[index]' node='$[node]'/>
               </crtserver>
               <jvm type='action' bal='was.jvm' xms = '512' xmx = '1024'  user='myuser' group='mygroup'>
                    <scope server='srv_$[app]_uat_$[index]' node='$[node]'/>
               </jvm>

               <jmq type='action' bal='was.jmq'>
                  <scope server='srv_$[app]_uat_$[index]' node='$[node]'/>
                  <qcfs>
                      <qcf name='myqcf' jndi_name='jms/myqcf' queue_manager='myqm' host='myhost'  port='1136'/>
                  </qcfs>
                  <queues>
                      <queue name='myqueue' queue='myqueue' jndi_name='jms/myqueue' queue_manager='myqm1'
                         host='myhost' port='1136'/>
                  </queues>
               </jmq>
        </__foreach__>
   </server>

   <conf type='action'  bal='ihs.conf'  host='dmz_uat'>
       <scope server='ihs_$[app]_uat'/>
   </conf>

</myapplication>
```

When the parser meet the tag <__foreach__ tag=index>, it will duplicate its content, foreach contextual tag index, and the xml actually becomes this :

```
<myapplication>

  <__alias__  name='app'   value='invoices'/>

  <server host='axane'  ip='10.100.1.1' >
         <__alias__  name='node'   value='axaneUatNode01'/>

        <index  value='01'>
               <__alias__  name='index'   value='pxq:bu.index@value'/>

               <crtserver type='action' bal='was.crtserver'>
                    <scope server='srv_$[app]_uat_$[index]' node='$[node]'/>
               </crtserver>
               <jvm type='action' bal='was.jvm' xms = '512' xmx = '1024'  user='myuser' group='mygroup'>
                    <scope server='srv_$[app]_uat_$[index]' node='$[node]'/>
               </jvm>
               <jmq type='action' bal='was.jmq'>
                    <scope server='srv_$[app]_uat_$[index]' node='$[node]'/>
                    <qcfs>
                        <qcf name='myqcf' jndi_name='jms/myqcf' queue_manager='myqm' host='myhost'
                            port='1136'/>
                    </qcfs>
                    <queues>
```

```
24.                                <queue name='myqueue' queue='myqueue' jndi_name='jms/myqueue'
25.                                     queue_manager='myqm1'  host='myhost' port='1136'/>
26.                          </queues>
27.                  </jmq>
28.          </index>
29.
30.          <index  value='02'>
31.                    <__alias__  name='index'   value='pxq:bu.index@value'>
32.
33.                   <crtserver type='action' bal='was.crtserver'>
34.                          <scope server='srv_$[app]_uat_$[index]' node='$[node]'/>
35.                  </crtserver>
36.                  <jvm type='action' bal='was.jvm' xms = '512' xmx = '1024'  user='myuser' group='mygroup'>
37.                          <scope server='srv_$[app]_uat_$[index]' node='$[node]'/>
38.                   </jvm>
39.                   <jmq type='action' bal='was.jmq'>
40.                           <scope server='srv_$[app]_uat_$[index]' node='$[node]'/>
41.                           <qcfs>
42.                                 <qcf name='myqcf' jndi_name='jms/myqcf' queue_manager='myqm' host='myhost'
43.                                      port='1136'/>
44.                           </qcfs>
45.                          <queues>
46.                                <queue name='myqueue' queue='myqueue' jndi_name='jms/myqueue'
47.                                     queue_manager='myqm1'  host='myhost' port='1136'/>
48.                          </queues>
49.                  </jmq>
50.          </index>
51.   </server>
52.
53.   <confihs type='action'  host='dmz_uat'>
54.  <scope server='ihs_$[app]_uat'/>
55.   </conf>
56.
57. </myapplication>
58.
```

Note:

In this line 9. : <**__alias__** name='index'   value='pxq:bu.index@value'>,
the fact that index finally becomes a father node, in the genealogy of node __alias__, makes it possible to retreive the attribute value from tag index with PXQUERY Bottom Up.
For more information about PXQUERY see the chapter PXQUERY.

## 11.4 PxQuery

Note:
The informations coming now in the rest of this chapter are advanced technics.
You do not need to know these tricks to efficiently use kikonf.
Although using them may save your time and help custom file readibility.

PxQuery stands for Picxml Query tools.
For more information about picxml see the picxml documentation on www.sourceforge.net.

The kikonf engine use the picxml tools.

Spec:An Attribute value can be defined from the value of another.
To comply with this the kikonf parser needs a mechanism to reach any Tag/Attribute into the xml file.
This is the purpose of PxQuery.

### *11.4.1 PxQuery expressions*

**Syntax**

**pxq:TYPE.QUERY_STRING**

**pxq:** is required

**TYPE** must be in :

- **td** for top down.
- **tdc** for complete top down.
- **bu** for bottom up.

**QUERY_STRING :** the path of the tag/attribute to retreive.

Here is sample with __alias__ :

Note:
Sample are given with Aliases, but any Attribute's value from any Tag (action or not) can be expressed with PxQuery.

### 11.4.2 top down query

```
1.  <applications>
2.      <myapplication>
3.          <__alias__ variable='environment'  value='pxq:td.myapplication/appli@env'/>
4.          <__alias__ variable='node' value='pxq:td.myapplication/node@type'/>
5.
6.          <appli name='costs' env='int'/>
7.          <servers type='backend'>
8.                  <server name='axaneIntNode01' host='axane' ip='10.100.1.1'/>
9.          </servers>
10.
11.         node type='frontend'/>
12.     </myapplication>
13. </applications>
```

Line 3. Alias environment is defined from the top down jxquery expression :
**pxq:td.myapplication/appli@env**
This part is required : pxq:td.
This part **myapplication/appli@env**, retreives the child node appli, from contextual top node **myapplication**
and from appli retreives the Attribute env.
The value returned is : **int.**

Line 4. Alias node is defined from the top down jxquery expression :
**pxq:td.myapplication/node@type**
This part is required : pxq:td.
This part **myapplication/node@type**, retreives the child node "node", from contextual top node **myapplication** and from "node" retreives the Attribute type.

**Syntax**
**pxq:TYPE.QUERY_STRING**

TYPE=**td:**, top down
QUERY_STRING=**TAG[/TAG][@ATTR]**

Each elements (except the last one) of the patern is a tag name, the last element is the returned attribute.

PxQuery assume that each tag **appears only once at each level** otherwise an exception is raised. Because the parser wouldn't know from wich to retreive the value.

### 11.4.3 Top down complete query

```
1.  <myapplication>
2.          <__alias__ variable='appli1'   value='pxq:tdc.myapplication/appli,@env=int,@name'/>
3.          <__alias__ variable='appli2'   value='pxq:tdc.myapplication/appli@env=uat,@name'/>
4.          <__alias__ variable='node1'
            value='pxq:tdc.myapplication /servers@type=backend/server@host=axane,@node'/>
5.          <__alias__ variable='node2'
            value='pxq:tdc.myapplication /servers@type=backend/server@host=buxane,@ip=10.100.1.2,@node'/>
6.
7.          <appli name='costs' env='int'/>
8.          <appli name='invoices' env='uat'/>
9.          <servers type='backend'>
10.                 <server node='axaneIntNode01' host='axane' ip='10.100.1.1'/>
11.                 <server node='buxaneIntNode01' host='buxane' ip='10.100.1.2'/>
12.         </servers>
13.
14.         <nodes type='frontend'/>
15. </myapplication>
```

Line 2. Alias appli1 is defined from the complete top down jxquery expression :
**pxq:tdc.myapplication/appli,@env=int,@name**

This part is required : pxq:tdc.
This part **myapplication/appli,@env=int,@name**, retreives from contextual top node myapplication the node
with tag appli, coming with attribute env equal “**int**“, the value of the attribute name.
**The value returned is : costs.**

Line 3. Alias appli2 is defined from the complete top down jxquery expression :
**pxq:tdc.myapplication /appli@env=uat,@name**

This part is required : pxq:tdc.
This part **myapplication /appli@env=uat,@name**, retreives from contextual top node myapplication the
node with tag appli, coming with attribute env equal “**uat**“, the value of the attribute name.
**The value returned is : invoices.**

Line 4. Alias node1 is defined from the complete top down jxquery expression :
**pxq:tdc.myapplication /servers@type=backend/server@host=axane,@node**

This part is required : pxq:tdc.
This part **myapplication /servers@type=backend/server@host=axane,@node**,
retreives from contextual top node myapplication the node with tag servers, coming with attribute type equal “backend“, the node with tag server.
And retreives from sub node with tag server, coming with attribute host equal “axane“, the value of attribute node
**The value returned is : axaneIntNode01.**

Line 5. Alias node2 is defined from the complete top down  jxquery expression :
**pxq:tdc.myapplication /servers@type=backend/server@host=buxane,@ip=10.100.1.2,@node**

This part is required : pxq:tdc.
This part
**myapplication /servers@type=backend/server@host=buxane,@ip=10.100.1.2,@node,**
retreives from contextual top node myapplication the  node with tag servers, coming with attribute type equal  “backend“,  the node with tag server.
And retreives from sub node with tag server, coming with attribute host equal  “buxane“, and with attribute ip equal  “10.100.1.2“ the value of attribute node.
**The value returned is : buxaneIntNode01.**

**Syntax**
**pxq:TYPE.QUERY_STRING**

TYPE=**tdc:**, complete top down
QUERY_STRING=**TAG[@ATTR=VALUE[,@ATTR=VALUE]],@ATTR Tag sep:/**

Each "/" separates a tag level.
Each "/"  is follown by a tag name.
A tag name is follown by a "/" than another tag,
or is follown by "@".
Each "@" is followed by an  attribute name.
Intermediates attribute names are folown by "=" then
their value.
The expression always finishes by an attribute name.

A simplest sample can be : tdc.appli@name
this is the equivalent tdc query pattern to the previous td query pattern : td.appli@name.

Note :
This works only if there is one node coming with tag name “appli“ and their is no distinction to be made on attributes values.

This syntax allow to match nodes not only by their tag names but also by there attribute name/values.
This syntax is complete and can reach any node of the xml.

The last element a:ATTR is the returned attribute name.

If the request retreives more than one node (at one level),
because at least two nodes for the same tag may have the same attributes Kikonf assume that is not what you want and raise an error.

### 11.4.4 Bottom up query

This query is run accross the node tree, towards the top.

```
1.  <myapplication>
2.
3.     <server host='axane'  ip='10.100.1.1'  jvm='srv_invoices_uat_01'>
4.              <__alias__  name='host' value='pxq:bu.server@host'/>
5.              <jmq type='action' bal='was.jmq'>
6.                   <scope server='pxq:bu.jmq/server@jvm' node='axaneUatNode01'/>
7.                   <qcfs>
8.                        <qcf name='myqcf' jndi_name='jms/myqcf' queue_manager='myqm' host='myhost'  port='1136'/>
9.                  </qcfs>
10.                 <queues>
11.                 <queue name='$[host]_queue1' queue='$[host]_queue1' jndi_name='jms/myqueue1'
12.                        queue_manager='myqm1' host='myhost' port='1136'/>
13.                 <queue name=''$[host]_queue2' queue='$[host]_queue2' jndi_name='jms/myqueue2'
14.                        queue_manager='myqm1' host='myhost' port='1136'/>
15.                 </queues>
16.             </jmq>
17.    </server>
18.
19. </myapplication>
```

Line 4. Alias host is defined from the botton up jxquery expression :

**pxq:bu.server@host**

This part is required : pxq:bu.

This part **server@host**, retreives from the father node: server, the attribute host.

**The value returned is : axane.**

Line 6. The value of the Attribute server of node scope is defined from the botton up jxquery expression :

**pxq:bu.jmq/server@jvm**

This part is required : pxq:bu.

This part **jmq/server@jvm**, retreives from the gran father node: server, the attribute jvm.

**The value returned is : srv_invoices_uat_01.**

**Syntax**

**pxq:TYPE.QUERY_STRING**

TYPE=**bu**:, bottom up query.
QUERY_STRING=**TAG[/TAG]@ATTR**.

Each elements (except the last one) of the pattern is a tag name, the last element is the returned attribute.
The bu pattern has the same structure as the td pattern except that its cinematic is different.

Note:
The bu cinematic works at the opposite of the td pattern :
The kiko request engine will start from the querying child node towards the top (and not from the top like in the td query).

This means that a bu request is always structured like, that :
FATHER_TAG[.GRANFATHER_TAG[.GRANGRANFATHER_TAG]...].attr.

### 11.4.5 PxQuery restrictions

As seen PxQuery expressions allows to jump from node to node.
**A main rule is that PxQuery never allows multiple intermediate nodes.**
If this happens an Exception is always thrown.

We seen td, tdc and bu PxQuery expressions are allowed to define xml Attribute values.
But PxQuery expressions **are also available as parameter of the kikarc (see kikarc selective calls) and the picxml commands**.

When using PxQuery expressions to define xml Attribute values, PxQuery always assumes that the target is an Attribute and is unique.

When using PxQuery expressions as parameter, PxQuery allows the target to be either a Node or an Attribute. The target may be multiple.

## 11.5 THE OR OPERATOR (|)

As we saw, in kikarc custom xml file, a value can by an explicite value e.g.:

**name = srv_invoivces_uat_01**,

from aliases e.g :

**name = srv_$[app]_$[env]_01**

from PxQuery e.g :

**name = 'pxq:bu.server@name'**.

Kikarc allows the use of alternated values, using the or operator : |.

**Syntax:**

$[VALUE1]|$[VALUE2]|$[VALUE3]

If trying to evaluate VALUE1 an exception is thrown, the parser tries to evaluate VALUE2, if trying to evaluate VALUE2 an exeption is thrown, the parser tries to evaluate VALUE3, and so on.

Example:

```
<jvm type = 'action' bal='was.jvm' xms = '512' xmx = '1024' >
   <scope server='pxo:pxq:bu.member@name|pxq:bu.jvm/server@jvm|$jvm'  node='axaneUatNode01'/>
<jvm>
```

This line means that the attribute “name“ of the node jvm can retreive its value from :

- the attribute name if its parent node is a tag named "member".
- the attribute jvm if its parent node is a tag named "server".
- the alias named “jvm“.

This line works in any of those following situations :

jvm Action is embedded into no wrapping tag :

```
<myapplication>
   <__alias__  name='app'   value='invoices'/>
   <__alias__  name='jvm'   value='srv_$[app]_uat_01'/>
   <__alias__  name='node'   value='axaneUatNode01'/>

    <jvm type = 'action' bal='was.jvm' xms = '512' xmx = '1024' >
        <scope server='pxo:pxq:bu.jvm/member@name|pxq:bu.jvm/server@jvm|$[jvm]'  node='axaneUatNode01'/>
    </jvm>
</myapplication>
```

jvm Action can be embedded into a server tag :

```
<myapplication>
   <__alias__ name='app'   value='invoices'/>

   <server host='axane'  ip='10.100.1.1' jvm='srv_$[app]_uat_01' node='axaneUatNode01'>
      <jvm type = 'action' bal='was.jvm' xms = '512' xmx = '1024' >
         <scope server='pxo:pxq:bu.jvm/member@name|pxq:bu.jvm/server@jvm|$[jvm]'  node='axaneUatNode01'/>
      </jvm>
   </server>

</myapplication>
```

jvm Action can be embedded into a crtluster tag :

```
<myapplication>
   <__alias__ name='app'   value='invoices'/>
   <__alias__ name='env'   value='uat'/>
   <__alias__ name='node01'   value='axaneUatNode01'/>
   <__alias__ name='node02'   value='buxaneUatNode01'/>

   <crtcluster type='action' name='cls_$appli_01'>
      <member name='jvm_$[app]_$[env]_01' node='$[node01]'/>
      <member name='jvm_$[app]_$[env]_02' node='$[node02]'/>
      <member name='jvm_$[app]_$[env]_03' node='$[node01]'/>

      <__foreach__ tag='member'>
         <jvm type = 'action' bal='was.jvm' xms = '512' xmx = '1024' >
            <scope server='pxo:pxq:bu.jvm/member@name|pxq:bu.jvm/server@jvm|$[jvm]'  node='axaneUatNode01'/>
         </jvm>
      </__foreach__>
   </crtcluster>

</myapplication>
```

## 11.6 Kikarc Selective calls

Kikarc allows you to run a fraction of a guiven xml file.
To run just a part of an xml file you use the parameter -t (--td) or -T (--tdc).

**The final target of this top down PxQuery is always assume to be a node or a list of node.**

Example:

```
kikarc -x custom.xml -t td.servers/server/jdbc/dtsrc
kikarc -x custom.xml -T tdc.servers@side=backend/server/jdbc/dtsrc
```

e.g.:

```
kikarc /<kikonf_install_root>/tests/c.xml -t application/servers/server -v3
kikarc /<kikonf_install_root>/tests/c.xml -T application/servers/server@name=server2
```

See the kikarc and the picxml  documentation for more information.

# 12 TRADEMARKS:

AIX, WebSphere, WebSphere Aplication Server, IBM HttpServer and WebSphere Mq 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.