

API Documentation

September 19, 2006

Contents

Contents	1
1 Module conddbui	2
1.1 Functions	2
1.2 Class CondDB	3
1.2.1 Methods	3
1.2.2 Class Methods	10
1.3 Class Tag	10
1.3.1 Methods	10
Index	13

1 Module conddbui

The idea behind the CondDBUI module is to simplify the usage of PyCool for LHCb users. This means that you will find here many functions which are specific to the way LHCb is playing with its conditions database. This means as well that these functions are not supposed to be efficient (and they are not). For efficient manipulations, we advise you to use directly the PyCool classes and functions, or (even better) to use the COOL C++ API. Please note that most of the functions of the conddbui module are used in the conddb browser.

1.1 Functions

run_tests()

testDBAccess (<i>connectionString</i>)

Create a CondDB object, connect to a database and show its contents

testGetTagList (<i>connectionString</i>)

Connect to the db, create a node architecture and tag some tree elements, then retrieve some tag list.
--

testMD5 (<i>connectionString</i>)
--

connect to a db and compute an md5 checksum

testNodeCreation (<i>connectionString</i>)

Connect to a database and create the nodes given in the nodeList
--

testRecursiveTag (<i>connectionString</i>)

Connect to a db, create a node architecture and recursively tag the root folder set and its children.

testRemoveNode (<i>connectionString</i>)

connect to a db and remove a node from it

testTagLeafNode (<i>connectionString</i>)
--

Connect to a db, create a dummy folder with dummy condition and tag the HEAD
--

testTagWithAncestorTag (<i>connectionString</i>)

Connect to the db, create a node architecture and tag some tree elements using ancestor tags.

testXMLListStorage (<i>connectionString</i>)

connect to a db, create a folder and store a condition object list in it
--

testXMLStorage (<i>connectionString</i>)

connect to a db, create a folder and store a condition object in it

1.2 Class CondDB

Object allowing to manipulate a COOL database object in an LHCb way. This object contains a functions to open or create a database. It can then be manipulated either directly through the attribute 'db', or via a set of functions simplifying some operations, like creation and deletion of nodes, storage and retrieval of XML strings, etc.

1.2.1 Methods

<pre>__init__(self, connectionString='', create_new_db=0, defaultTag='HEAD', readOnly=1)</pre> <hr/> <p>Establishes the connection to the COOL database and store the object.</p> <p>inputs:</p> <p> connectionString: string; standard COOL connection string. An empty string does not initialise the database handle. -> Default = ''</p> <p> create_new_db: boolean; tells the constructor what to do if the connectionString doesn't open an existing database. -> Default = False</p> <p> defaultTag: string; tag which will be used by default if no other is precised. -> Default = 'HEAD'</p> <p> readOnly: boolean; open the conddb in read only mode if True, or read/write mode if False. -> Default = True</p> <p>outputs:</p> <p> none</p>	<pre>closeDatabase(self)</pre> <hr/> <p>Close the connection to the opened database</p> <p>inputs:</p> <p> none</p> <p>outputs:</p> <p> none</p>
<pre>createDatabase(self, connectionString)</pre> <hr/> <p>Create a new database and connect to it.</p> <p>inputs:</p> <p> connectionString: string; standard COOL connection string.</p> <p>outputs:</p> <p> none</p>	

```
createNode(self, path, description='', storageType='XML', versionMode='MULTI',
storageKeys=['data'])
```

Creates a new node (folder or folderset) in the database.

inputs:

```
path:          string; full path of the new node. Parents will be created if necessary.
description:   string; short description of the node.
               -> Default = ''
storageType:   string; data type to be stored in this node, implying the type of node
               to create. If the node is a folder, it will contain 'XML'. If it is a
               folderset, it will contain 'NODE'.
               -> Default = 'XML'
versionMode:   string; applies to folders only: is it multi version ('MULTI') or single
               version ('SINGLE') ?
               -> Default = 'MULTI'
storageKeys:   list of strings; the keys of the attribute list that will be stored in the folder.
               -> Default = ['data']
```

outputs:

```
none
```

```
createTagRelation(self, path, parentTag, tag)
```

Create a relation between the tag of the given node and a tag of its parent node.

inputs:

```
path:          string; path of the node
parentTag:     string; a tag associated to the parent node.
tag:           string; the tag which we want to relate to the parent tag.
```

outputs:

```
none
```

```
deleteNode(self, path, delete_subnodes=0)
```

Delete a node from the database permanently.

inputs:

```
path:          string; node's path in the database
delete_subnodes: boolean; only useful for foldersets. If True, all the subnodes
                    will be destroyed as well. If False, a node can be deleted only
                    if it has no children.
                    -> Default = False
```

outputs:

```
none
```

deleteTag(*self*, *path*, *tagName*, *delete_relations=1*)

Delete a tag from the database, and its relations if asked for.

inputs:

path: string; path to the node
 tagName: string; name of the tag to delete
 delete_relations: boolean; this has a meaning only for folders. If True, delete also the relations with the parent tag.
 -> Default = True

outputs:

none

deleteTagRelation(*self*, *path*, *parentTag*)

Delete a relation between the tag of the given node and a tag of its parent node.

inputs:

path: string; path of the node
 parentTag: string; the tag we no longer want to be related to.

outputs:

none

generateUniqueTagName(*self*, *baseName*, *reservedNames=[]*)

Generate a random tag name based on a given one.

inputs:

baseName: string; ideally, this is the "parent tag" name. If this name is an automatically generated one (i.e. starting with '_auto_' and finishing with '-' and 6 alphanumeric characters), the function will automatically strip the name from its random parts.
 reservedNames: list of strings; list of name that can't be chosen.
 -> Default = []

outputs:

string; the generated tag name. Its format is:
 '_auto_' + baseName + '-' + 6 random alphanumeric characters.

getAllChildNodes(*self*, *path*)

Return all the nodes of the tree lying under "path"

inputs:

path: string; path of the parent node. Must be a folderset.

outputs:

list of strings; the paths of all the elements of the tree under the given node.

getAllNodes(*self*)

Convenience function: returns all the nodes of the database.

inputs:

none

outputs:

list of strings; the paths of all the nodes of the database

getAttributeList(*self*, *path*, *when*, *channelID*=0, *tag*='')

Retrieve the Coral Attribute List of the condition object valid at a given time.

inputs:

path: string; path to the condition data in the database.
when: integer; time stamp (most likely an event time) at which the value of the condition is requested.
channelID: integer; ID of the channel in which the condition data are stored.
 -> Default = 0
tag: string; name of the version. If empty, defaultTag is used.
 -> Default = ''

outputs:

dictionary; the contents of the attribute list

getAttributeListList(*self*, *path*, *fromTime*, *toTime*, *channelID*=0, *tag*='')

Retrieve the coral attribute list of the condition objects valid during a given time interval.

inputs:

path: string; path to the condition data in the database.
fromTime: integer; lower bound of the studied time interval.
toTime: integer; upper bound of the studied time interval. Note that an object with start of validity equal to this upper bound value will be returned as well.
channelIDs: integer; IDs of the channel in which the condition data are stored. If None is given instead, all channels will be browsed.
 -> Default = 0
tag: string; name of the version. If empty, defaultTag is used.
 -> Default = ''

outputs:

list of [dictionary, integer, integer, integer, integer]; the dictionary is the coral attribute list. The first two integers are the since and until values of the interval of validity. The third integer is the channel ID, and the last integer is the insertion time.

getChildNodes(*self*, *path*)

Return a list of the children of the given node.

inputs:

path: string; path of the parent node. Must be a folderset.

outputs:

list of strings; the paths of the child nodes.

getFolderStorageKeys(*self*, *path*)

Retrieve the keys of the Coral Attribute List Specification for the given folder.

inputs:

path: string; path to the folder.

outputs:

list of strings; the list of storage keys.

getTagList(*self*, *path*)

Return all the tag objects defined for the given node.

inputs:

path: string; path to the leaf node

outputs:

tagList: list of Tag; the list of Tag objects defined for this node.
They contains links to their parent Tag objects.

getXMLString(*self*, *path*, *when*, *channelID*=0, *tag*='', *payloadKey*='data')

Retrieve the XML string of the condition object valid at a given time.

inputs:

path: string; path to the condition data in the database.

when: integer; time stamp (most likely an event time) at which the value of the condition is requested.

channelID: integer; ID of the channel in which the condition data are stored.

-> Default = 0

tag: string; name of the version. If empty, defaultTag is used.

-> Default = ''

payloadKey: string; key of the coral attribute list element we want to retrieve.

-> Default = 'data'

outputs:

string; the contents of the condition data.

getXMLStringList(*self*, *path*, *fromTime*, *toTime*, *channelID*=0, *tag*='', *payloadKey*='data')

Retrieve the payload of the condition objects valid during a given time interval.

inputs:

path: string; path to the condition data in the database.

fromTime: integer; lower bound of the studied time interval.

toTime: integer; upper bound of the studied time interval. Note that an object with start of validity equal to this upper bound value will be returned as well.

channelIDs: integer; IDs of the channel in which the condition data are stored. If None is given instead, all channels will be browsed.

-> Default = 0

tag: string; name of the version. If empty, defaultTag is used.

-> Default = ''

payloadKey: string; key of the coral attribute list element we want to retrieve.

-> Default = 'data'

outputs:

list of [string, integer, integer, integer, integer]; the string is the payload. The first two integers are the since and until values of the interval of validity. The third integer is the channel ID, and the last integer is the insertion time.

isSingleVersionFolder(*self*, *path*)

Check if path corresponds to a single version folder

inputs:

path: string; path to the node to check

outputs:

boolean; True if the node is a single version folder, False in all other cases (i.e. if the node is a multi version folder OR if it is a folderset or doesn't exist).

openDatabase(*self*, *connectionString*, *create_new_db*=0, *readOnly*=1)

Closes the current database and connects to a new one. Creates it if asked to.

inputs:

connectionString: string; standard COOL connection string.

create_new_db: boolean; if True, creates a new database on failure to connect.
-> Default = False

readOnly: boolean; open the conddb in read only mode if True, or read/write mode if False.
-> Default = True

outputs:

none

payloadToMd5(*self*, *path*='/', *tag*='', *initialMd5Sum*=None)

Computes the md5 sum for the payload stored under the given node.

inputs:

path: string; path to the top of the database subtree to check.
-> Default = '/'

tag: string; version of the data to check. If set to '', defaultTag is used. If set to 'ALL', will check all the tags associated to this node (NOT YET IMPLEMENTED !!)
-> Default = ''

md5Sum: md5 object; starting point for the check. If none is given, a new one is created.
-> Default = None

outputs:

md5 object; result from the md5 check sum.

recursiveTag(*self*, *path*, *tagName*, *description*='', *reserved*=None)

Tag the given node and recursively tag the child nodes and their HEAD revisions with randomly generated tags.

inputs:

path: string; full path to the node

tagName: string; name of the tag to apply. It must be unique in the database.

description: string; details about the tagging operation.
-> Default = ''

reserved: list of strings; list of reserved tags.
-> Default = None

outputs:

none

setDefaultTag(self, tagName)

Set the value of the default tag.

inputs:

 tagName: string; the name of the default tag.

outputs:

 none

storeXMLString(self, path, data, since, until, channelID=0, key='data')

Adds a new condition object to the database.

inputs:

 path: string; path of the folder where the condition will be stored.

 data: string; XML string to store in the database.

 since: integer; lower bound of the interval of validity.

 until: integer; upper bound of the interval of validity. It is excluded from the interval.

 channelID: integer; ID of the channel where to store the condition.

 -> Default: 0

 key: string; key of the attribute list where xml data will be stored.

 -> Default: 'data'.

outputs:

 none

storeXMLStringList(self, path, XMLList)

Allows to store a list of XML string into a given folder.

inputs:

 path: string; the path to the folder

 XMLList: list of dictionary of 'payload': dictionary: a dictionary version of the attribute list to store in the database.

 'since': integer: the lower bound of the IOV.

 'until': integer: the upper bound of the IOV.

 'channel': integer: the channel ID.

outputs:

 none

tagLeafNode(self, path, tagName, description='')

Apply a new tag to the head of the given folder.

inputs:

 path: string; full path to the folder

 tagName: string; name of the tag to apply. It must be unique in the database.

 description: string; details about the tagging operation.

 -> Default = ''

outputs:

 none

tagWithAncestorTag(*self*, *path*, *ancestorTag*, *description*='')

Recursively tag (with automatically generated tag names) the given node and associate the tags with the ancestor tag given.

inputs:

path: string; path of the node to tag
 ancestorTag: string; tag to associate the node with. It must be an existing tag and a relation must exist with the parent of the given node. Otherwise, an exception is raised.
 description: string; description to associate with the tagged elements. If empty, the description of the closest ancestor tag will be used.
 -> Default = ''

outputs:

none

1.2.2 Class Methods

dropDatabase(*cls*, *connectionString*)

drop the database identified by the connection string.

inputs:

connectionString: string; standard COOL connection string.

outputs:

none

1.3 Class Tag

Basic class allowing to manipulate more easily the tags in the tag hierarchy. The rule is that a tag has only one child tag and can have many parent tag.

1.3.1 Methods

__init__(*self*, *tagName*, *nodePath*)

Create a new tag object.

inputs:

tagName: string; name of the tag
 nodePath: string; path to the node which own this tag

outputs:

none

__repr__(*self*)

Standard object representation. Returns a string representation of all the object's attributes, as well as its relations with its ancestors.

__str__(*self*)

Standard string conversion. Returns the name of the tag

connectChild(*self*, *child*)

Connect a child tag to the current tag, and update the parent list of the child.

inputs:

child: Tag object; the child tag object

outputs:

none

getAncestors(*self*)

Return the names of the ancestor tags.

inputs:

none

outputs:

ancestors: list of strings; the names of all the ancestor tags of the current tag. This is equivalent to a list of aliases for this tag.

getAncestorsBranches(*self*, *currentBranche*=[], *brancheList*=None)

Recursive function returning the list of ancestor branches of the tag.

inputs:

currentBranche: list of strings; stores the names of the ancestors of the current branch.
-> Default = []

brancheList: list of lists of strings; variable storing the list of completed ancestor branches.
-> Default = None

outputs:

brancheList: list of list of strings; each sublist contains a branch of the tag "family".

getAncestorTags(*self*)

Return the ancestor tags as a list of tag objects.

inputs:

none

outputs:

ancestors: list of tags; all the ancestor tags of the current tag.

getAncestorTagsDict(*self*, *currentBranche*={}, *brancheList*=None)

Recursive function returning the list of ancestor tags dictionaries.

inputs:

currentBranche: dictionary of tags; stores the the ancestors tags, referenced by names, for the current branch.
-> Default = {}

brancheList: list of dictionaries; variable storing the list of completed ancestor branches.
-> Default = None

outputs:

brancheList: list of dictionaries; each sublist contains a branch of the tag "family".

printAncestors(*self*, *branche*='')

Recursive function printing the relation between the tag and its ancestors.

inputs:

branche: string; current status of the ancestor branch. If other ancestors exist, this value is updated. Otherwise, it is printed.
-> Default = ''

outputs:

none; results are sent to the standard output.

Index

conddbui (*module*), 2–12

- CondDB (*class*), 2–10
 - `__init__` (*method*), 3
 - `closeDatabase` (*method*), 3
 - `createDatabase` (*method*), 3
 - `createNode` (*method*), 3
 - `createTagRelation` (*method*), 4
 - `deleteNode` (*method*), 4
 - `deleteTag` (*method*), 4
 - `deleteTagRelation` (*method*), 5
 - `dropDatabase` (*method*), 10
 - `generateUniqueTagName` (*method*), 5
 - `getAllChildNodes` (*method*), 5
 - `getAllNodes` (*method*), 5
 - `getAttributeList` (*method*), 5
 - `getAttributeListList` (*method*), 6
 - `getChildNodes` (*method*), 6
 - `getFolderStorageKeys` (*method*), 6
 - `getTagList` (*method*), 6
 - `getXMLString` (*method*), 7
 - `getXMLStringList` (*method*), 7
 - `isSingleVersionFolder` (*method*), 7
 - `openDatabase` (*method*), 8
 - `payloadToMd5` (*method*), 8
 - `recursiveTag` (*method*), 8
 - `setDefaultTag` (*method*), 8
 - `storeXMLString` (*method*), 9
 - `storeXMLStringList` (*method*), 9
 - `tagLeafNode` (*method*), 9
 - `tagWithAncestorTag` (*method*), 9

`run_tests` (*function*), 2

Tag (*class*), 10–12

- `__init__` (*method*), 10
- `__repr__` (*method*), 10
- `__str__` (*method*), 10
- `connectChild` (*method*), 10
- `getAncestors` (*method*), 11
- `getAncestorsBranches` (*method*), 11
- `getAncestorTags` (*method*), 11
- `getAncestorTagsDict` (*method*), 11
- `printAncestors` (*method*), 12

`testDBAccess` (*function*), 2

`testGetTagList` (*function*), 2

`testMD5` (*function*), 2

`testNodeCreation` (*function*), 2

`testRecursiveTag` (*function*), 2

`testRemoveNode` (*function*), 2

`testTagLeafNode` (*function*), 2

`testTagWithAncestorTag` (*function*), 2

`testXMLListStorage` (*function*), 2

`testXMLStorage` (*function*), 2