facadedevice

Release

Jul 18, 2017

Contents

A descritive interface for reactive high-level Tango devices.

CHAPTER 1

Presentation

This python package provide a descriptive interface for reactive high-level Tango devices.

1.1 Requirements

The library requires:

- **python** >= 2.7 or >= 3.4
- **pytango** >= 9.2.1

1.2 Installation

Install the library by running:

```
$ python setup.py install # Or
$ pip install .
```

1.3 Unit-testing

Run the tests using:

```
$ python setup.py test
```

The following libraries will be downloaded if necessary:

- pytest
- pytest-runner
- pytest-mock

- pytest-xdist
- pytest-coverage

1.4 Documentation

Generating the documentation requires:

- sphinx
- sphinx.ext.autodoc
- sphinx.ext.napoleon

Build the documentation using:

```
$ python setup.py build_sphinx
$ sensible-browser build/sphinx/html/index.html
```

CHAPTER 2

Tutorial

This tutorial goes through most of the library features by presenting several facade devices with increasing complexity.

2.1 Creating an running a facade device

A facade device is an enhanced pytango HLAPI device. It provides the same methods and supports the same pytango object (device properties, attributes, commands, etc.). In order to create a new facade device class, simply inherit from the *Facade* base class:

```
from facadedevice import Facade


class Empty(Facade):
    pass

if __name__ == '__main__':
    Empty.run_server()
```

This example is already a working (empty) device. It is possible to run it without a database using the *tango.test_context* module:

```
$ cd examples/
$ python -m tango.test_context examples.empty.Empty --debug=3
Ready to accept request
Empty started on port 8888 with properties {}
Device access: tango://hostname:8888/test/nodb/empty#dbase=no
Server access: tango://hostname:8888/dserver/Empty/empty#dbase=no
```

It is now accessible through *itango*:

```
In [1]: d = Device('tango://hostname:8888/test/nodb/empty#dbase=no')
In [2]: d.state()
Out[2]: tango._tango.DevState.UNKNOWN
```

2.2 Adding extra logic at initialization

The default state is **UNKNOWN**. Since facade devices are regular devices, we can change it using the *set_state* method. However, the *init_device* method shouldn't be overridden because it performs specific exception handling. Instead, override *safe_init_device* if you have to add some extra logic. Don't forget to call the parent method since it performs other useful steps:

```
class On(Facade):

    def safe_init_device(self):
        super(On, self).safe_init_device()
        self.set_state(DevState.ON)
```

Now let's check the state:

```
In [2]: d.state()
Out[2]: tango._tango.DevState.ON
```

2.3 Local attributes

Good, but a static state is not really useful. Instead we'd like it to react to the values of other attributes. Let's create a device with a local counter using the *local_attribute* object.

```
class Counter1(Facade):

    @local_attribute(
        dtype=int,
        access=AttrWriteType.READ_WRITE,)
    def count(self):
        return 0
```

Note that *local_attribute* can be used as a decorator to set a default value for the attribute, although it is not mandatory. Also, *local_attribute* (and other facade-specific attributes) supports all the arguments of the standard pytango attribute object (e.g. *access* and *dtype* in the example above). Now let's try our counter:

```
In [2]: d.count
Out[2]: 0
In [3]: d.count += 1
In [4]: d.count
Out[4]: 1
```

See how the *count* attribute has been incremented successfully. Also note that the facade devices have a full support for events, meaning a change event has been pushed when the *count* value has been updated (no polling is required on the attribute).

2.4 Data model

Now, instead of a writable attribute, we'd like to use a command to increment the value of *count*. But first, we need to learn about the data model that allows reactivity and the propagation of changes. Every facade device instance has a graph of nodes that represents the different values that the device has to manage. For instance, every local attribute has a corresponding node that can be accessed through *self.graph[attr_name]*. A node can contain either:

- nothing
- a *triplet* result (value, stamp, quality)
- an exception

Accessing the node state is done through the following methods:

- *node.result() == None* if the node contains nothing
- *value, stamp, quality = node.result()* if the node contains a result
- *node.exception() == None* if the node doesn't contain an exception
- *exc = node.exception()* if the node contains an exception

Also note that calling *node.result()* on a node containing an exception will raise the corresponding exception. The node state is set using the following methods:

- *node.set_result(None)*
- *node.set_result(triplet(value, stamp, quality))*
- *node.set_exception(exc)*

Note that stamp and quality are optional. They respectively default to the current time and the **VALID** quality. The *increment* tango command can now be implemented:

```
class Counter2(Facade):

    @local_attribute(
        dtype=int)
    def count(self):
        return 0

    @command
    def increment(self):
        node = self.graph['count']
        value, stamp, quality = node.result()
        new_result = triplet(value+1)
        node.set_result(new_result)
```

Let's give it a try:

```
In [2]: d.count
Out[2]: 0
In [3]: d.increment()
In [4]: d.count
Out[4]: 1
```

2.5 State attribute

Now, we'd like to have the state react to the value of *count*. This can be achieved using the *state_attribute* facade object. It is used as a decorator and takes the list of the nodes to bind to as an argument:

```
class Counter3(Facade):

    @local_attribute(
        dtype=int)
    def count(self):
```

```
        return 0

    @command
    def increment(self):
        node = self.graph['count']
        value, stamp, quality = node.result()
        new_result = (value+1,)
        node.set_result(new_result)

    @state_attribute(
        bind=['count'])
    def state_and_status(self, count):
        if count == 0:
            return DevState.OFF, 'The count is 0'
        return DevState.ON, 'The count is {}'.format(count)
```

Note that it's possible to return the status along with the state, although it is not mandatory. Let's run the counter:

```
In [2]: d.state()
Out[2]: tango._tango.DevState.OFF
In [3]: d.status()
Out[3]: 'The count is 0'
In [4]: d.increment()
In [5]: d.state()
Out[5]: tango._tango.DevState.ON
In [6]: d.status()
Out[6]: 'The count is 1'
```

See how the state is updated automatically. Remember that there is no polling or periodic update involved: the changes are simply propagated through the device graph.

2.6 Logical attributes

State and *Status* are not the only attributes that can react to changes. It is possible to declare logical attributes using the same binding approach. Let's write a device that performs a division:

```
class Division1(Facade):

    A = local_attribute(
        dtype=float,
        access=AttrWriteType.READ_WRITE)

    B = local_attribute(
        dtype=float,
        access=AttrWriteType.READ_WRITE)

    @logical_attribute(
        dtype=float,
        bind=['A', 'B'])
    def C(self, a, b):
        return a / b
```

Here we defined the relationship $C = A / B$. Note how the arguments of the method C are simply the value A and B. Let's give it a try:

```
In [2]: d.A = 1
In [3]: d.B = 4
In [4]: d.C
Out[4]: 0.25
In [5]: d.B = 0
In [6]: d.C
PyDs_PythonError: Exception while updating node <C>:
  float division by zero
```

Remember that the computation of *C* does not happen when the attribute *C* is being read but when the values of *A* and *B* are changing. For instance, the zero division exception has been set to the node *C* right after we set *B* to zero.

They are special rules about aggregation depending on the state of the different input nodes:

- if node *A* or node *B* is empty, node *C* is empty too
- if node *A* or node *B* contains an exception, it's propagated to *C*
- if the quality of *A* or the quality of *B* is invalid, the quality of *C* is invalid
- otherwise, the *C* method is executed and the return value is used as a result

Note that the return value of the *C* method can be:

- a single value (timestamp and quality are computed from the input nodes)
- a *triplet* result, in order to set the timestamp and/or the quality

2.7 The triplet structure

The triplet is a named tuple provided by the facade device. All the node results are guaranteed to be a triplet when they exist. This is how it is used:

```
from time import time
from tango import AttrQuality
from facadedevice import triplet

# A triplet from a single value
result = triplet(1.)

# A triplet from a value and a stamp
result = triplet(1., stamp=time())

# A triplet from a value and a quality
result = triplet(1, quality=AttrQuality.ATTR_ALARM)

# A triplet from value, a stamp and a quality
result = triplet(1, time(), AttrQuality.ATTR_CHANGING)

# Triplets can be unpacked
value, stamp, quality = result

# The values can be accessed through attributes
result.value, result.stamp, result.quality
```

The default quality is **VALID** and the default stamp is the time at the triplet creation. It has another interesting property: a *None* value will cause the quality to be **INVALID** and an **INVALID** quality will cause the value to be *None*. This is enforced at triplet creation.

Warning: An empty node and a none (invalid) triplet can easily be confused! They are however very different:

- *node.set_result(None)* empty the node
- *node.set_result(triplet(None))* set an **INVALID** result with a timestamp

The both behave differently when reading the corresponding attribute or when used as an input node to propagate changes.

2.8 Proxy attribute

The division device is working nicely but it doesn't really communicate with the outside world. More precisely, the *A* and *B* might come from another device. In this case, we can simply replace the local attributes with proxy attributes:

```
class Division2(Facade):

    A = proxy_attribute(
        dtype=float,
        property_name='AAttribute')

    B = proxy_attribute(
        dtype=float,
        property_name='BAttribute')

    @logical_attribute(
        dtype=float,
        bind=['A', 'B'])
    def C(self, a, b):
        return a / b
```

The only special argument we need to provide a proxy attribute with is *property_name*: its the name of the device property that will contain the access to the remote attribute. In this case, the device properties could be:

- *AAttribute*: some/device/somewhere/x
- *BAttribute*: some/other/device/y

Those remote attributes are expected to push either change or periodic events. Facade devices have an expert command called *GetInfo* that provides extra information about the event subscription, e.g:

```
In [2]: print(d.getinfo())
The device is currently connected.
It subscribed to event channel of the following attribute(s):
- some/device/somewhere/x (CHANGE_EVENT)
- some/other/device/y (PERIODIC_EVENT)
-----
No errors in history since Tue Apr 25 18:26:47 2017 (last initialization).
```

Once properly set up, any event comming from those remote attributes will cause *A* (or *B*) and *C* to be updated. Note that facade devices can easily be chained together since they both publish and subscribe.

It is also possible to apply a conversion to the input data by using *proxy_attribute* as a decorator:

```
@proxy_attribute(
    dtype=float,
    property_name='AAttribute')
```

```
def A(self, a):
    return a * 10
```

Here, the data coming from the event channel is multiplied by 10. Note that the device property can also be a value if the remote attribute doesn't exist:

```
$ python -m tango.test_context --prop "{'AAttribute': 1.0, 'BAttribute': 4.0}" \
  division2.Division2
Ready to accept request
Division2 started on port 8888 with properties {'AAttribute': 1.0, 'BAttribute': 4.0}
Device access: tango://vinmic-t440p:8888/test/nodb/division2#dbase=no
Server access: tango://vinmic-t440p:8888/dserver/Division2/division2#dbase=no
```

Let's check the values:

```
In [2]: d.A = 1
In [3]: d.B = 4
In [4]: d.C
Out[4]: 0.25
```

2.9 Combined attributes

In some cases, it is interesting to access remote attributes in a more dynamic way. The *facadedevice* library does not support dymanic attributes directly, but it provides a *combined_attributes* object that can be used for similar purposes. Let's say we'd like to compute the average of the values of an arbitrary list of attributes:

```
class Average(Facade):

    @combined_attribute(
        dtype=float,
        property_name='AttributesToAverage')
    def average(self, *args):
        return sum(args) / len(args)
```

Here, the *AttributesToAverage* device property is simply the list of all the attributes that should be used for the computation. The attributes may come from the same device, or different devices. If that device property is a single line, it's used a pattern for listing the attributes. For instance, the pattern *a/b/*/x[12]* might yield:

- a/b/c/x1
- a/b/c/x2
- a/b/whatever/x1
- a/b/whatever/x2
- etc.

It includes all the attributes called *x1* or *x2* from any device starting with *a/b/*. Note that the aggregation works the same as for logical attributes.

2.10 Proxy commands

The library also provides an interface for proxy attributes, although it doesn't use of the concepts explained earlier (graph, node, triplets, etc.). It's simply a helper to bind a tango command to a command on a remote device. Consider

the following example:

```
class Commands(Facade):

    reset = proxy_command(
        property_name="ResetCommand")

    echo = proxy_command(
        dtype_in=str,
        dtype_out=str,
        property_name="EchoCommand")

    set_level = proxy_command(
        dtype_in=float,
        property_name="LevelAttribute",
        write_attribute=True)

    @proxy_command(
        dtype_in=int,
        dtype_out=int,
        property_name="EchoCommand")
    def identity(self, subcommand, arg):
        return int(subcommand(str(arg)))
```

The *reset* command here simply delegates to the *ResetCommand* provided in the device properties. It has no input argument, no return value, and the remote command is expected to have the same interface.

The *echo* command delegates to the *EchoCommand* provided in the device properties by passing the input string argument to the remote command and returning its return value. Again, both interfaces are expected to match (otherwise an exception will be raised at runtime).

It is also possible to write a remote attribute instead of running a remote command. The *set_level* command does exactly that by setting *write_attribute=True*. Note that value to write is directly given by the float input argument.

In some cases, we need a finer control over the command behavior. For instance, we might need to apply some conversion before or after running the remote command. It is then possible to use *proxy_command* as a decorator of a method implementing this extra bit of logic.

The *identity* command in the code above is one example of that: the remote command can only handle string, while we'd like our command to work with integers. See how the *identity* method receives the remote command and the input argument, and how it converts the different values to make the types match.

CHAPTER 3

Advanced features

TODO

CHAPTER 4

Limitations

TODO

CHAPTER 5

API Reference

5.1 Simple attributes

class facadedevice.**local_attribute**(*create_attribute=True*, ***kwargs*)
Tango attribute with event support.

Local attributes support the standard attribute keywords.

It can be used as a decorator to set a method providing the default value for the corresponding attribute.

Parameters **create_attribute** (*str*) – Create the corresponding tango attribute. Default is True.

class facadedevice.**logical_attribute**(*bind*, *standard_aggregation=True*, ***kwargs*)
Tango attribute computed from the values of other attributes.

Use it as a decorator to register the function that make this computation. Logical attributes also support the standard attribute keywords.

Parameters

- **bind** (*list of str*) – List of node names to bind to. It has to contain at least one name.
- **standard_aggregation** (*optional, bool*) – Use the default aggregation mecanism. Default is True.
- **create_attribute** (*optional, bool*) – Create the corresponding tango attribute. Default is True.

5.2 Remote attributes

class facadedevice.**proxy_attribute**(*property_name*, *create_property=True*, ***kwargs*)
Tango attribute linked to the attribute of a remote device.

Parameters

- **property_name** (*str*) – Name of the property containing the attribute name.
- **create_property** (*optional, bool*) – Create the corresponding device property. Default is True.
- **standard_aggregation** (*optional, bool*) – Use the default aggregation mecanism. Default is True.
- **create_attribute** (*optional, bool*) – Create the corresponding tango attribute. Default is True.

Also supports the standard attribute keywords.

class facadedevice.**combined_attribute**(*property_name*, *create_property=True*, ***kwargs*)
Tango attribute computed from the values of other remote attributes.

Use it as a decorator to register the function that make this computation. The remote attribute names are provided by a property, either as a list or a pattern.

Parameters

- **property_name** (*str*) – Name of the property containing the attribute names.
- **create_property** (*optional, bool*) – Create the corresponding device property. Default is True.
- **standard_aggregation** (*optional, bool*) – Use the default error aggregation mecanism. Default is True.
- **create_attribute** (*optional, bool*) – Create the corresponding tango attribute. Default is True.

Also supports the standard attribute keywords.

5.3 State handling

class facadedevice.**state_attribute**(*bind=None*, *standard_aggregation=True*)
Tango state attribute with event support.

Parameters

- **bind** (*list of str*) – List of node names to bind to, or None to disable the binding. Default is None.
- **standard_aggregation** (*optional, bool*) – Use the default error aggregation mecanism. Default is True.

5.4 Commands

class facadedevice.**proxy_command**(*property_name*, *create_property=True*, *write_attribute=False*, ***kwargs*)
Command to write an attribute or run a command of a remote device.

It can be used as a decorator to define a more precise behavior. The decorated method takes the subcommand as its firt argument.

Parameters

- **property_name** (*str*) – Name of the property containing the attribute or command name.

- **create_property** (*str*) – Create the corresponding device property. Default is True.
- **write_attribute** (*bool*) – True if the subcommand should an attribute write, False otherwise. Default is false.

Also supports the standard command keywords.

5.5 Facade base classes

class facadedevice.**Facade**(*cl*, *name*)
Base class for facade devices.

It supports the following objects:

- *facadedevice.local_attribute*
- *facadedevice.logical_attribute*
- *facadedevice.proxy_attribute*
- *facadedevice.combined_attribute*
- *facadedevice.state_attribute*
- *facadedevice.proxy_command*

It also provides a few helpers:

- *self.graph*: act as a <*key, node*> dictionnary
- *self.get_combined_results*: return the subresults of a combined attribute

The *init_device* method shouldn't be overridden. It performs specific exception handling. Instead, override *safe_init_device* if you have to add some extra logic. Don't forget to call the parent method since it performs a few useful steps:

- load device properties
- configure and build the graph
- run the connection routine

It also provides an expert command called *GetInfo* that displays useful information such as:

- the connection status
- the list of all event subscriptions
- the exception history

class facadedevice.**TimedFacade**(*cl*, *name*)
Similar to the *facadedevice.Facade* base class with time handling.

In particular, it adds:

- the *UpdateTime* polled command, used trigger updates periodically
- the *Time* local attribute, a float updated at every tick
- the *on_time* method, a callback that runs at every tick

CHAPTER 6

Examples

This section contains a few extra examples.

6.1 Simple example

The following example shows the definition of a rectangle device, getting its width and height from other devices:

```
from facadevice import Facade, proxy_attribute, logical_attribute


class Rectangle(Facade):

    Width = proxy_attribute(
        property_name='WidthAttribute')

    Height = proxy_attribute(
        property_name='HeightAttribute')

    @logical_attribute(
        bind=['Width', 'Height'])
    def Area(width, height):
        return width * height

if __name__ == '__main__':
    Rectangle.run_server()
```

A rectangle device is configured using 2 device properties, e.g.:

- WidthAttribute: *geometry/point/a/x*
- HeightAttribute: *geometry/point/b/y*

The remote attributes are expected to push either change or periodic events.

A rectangle device exposes 3 float attributes:

- Width
- Height
- Area

Those attributes will be updated as soon as a corresponding event is received. They also pushes events, allowing other high-level devices to react to their changes.

6.2 Real-world example

A real-world example of a camera screen device used at MAX-IV:

```
from tango import DevState
from facadedevice import Facade, proxy_command
from facadedevice import proxy_attribute, logical_attribute, state_attribute


class CameraScreen(Facade):

    # Proxy attributes

    StatusIn = proxy_attribute(
        dtype=bool,
        property_name="StatusInAttribute")

    StatusOut = proxy_attribute(
        dtype=bool,
        property_name="StatusOutAttribute")

    # Logical attributes

    @logical_attribute(
        dtype=bool,
        bind=['StatusIn', 'StatusOut'])
    def Error(self, status_in, status_out):
        return status_in and status_out

    @logical_attribute(
        dtype=bool,
        bind=['StatusIn', 'StatusOut'])
    def Moving(self, status_in, status_out):
        return not status_in and not status_out

    # Proxy commands

    @proxy_command(
        property_name="MoveInAttribute",
        write_attribute=True)
    def MoveIn(self, subcommand):
        subcommand(1)

    @proxy_command(
        property_name="MoveOutAttribute",
        write_attribute=True)
    def MoveOut(self, subcommand):
        subcommand(1)
```

```
    # State and status

    @state_attribute(
        bind=['Error', 'Moving', 'StatusIn'])
    def state(self, error, moving, status_in):
        if error:
            return DevState.FAULT, "A conflict has been detected"
        elif moving:
            return DevState.MOVING, "The screen is moving"
        elif status_in:
            return DevState.INSERT, "The screen is inserted"
        else:
            return DevState.EXTRACT, "The screen is exctracted"


if __name__ == '__main__':
    CameraScreen.run_server()
```

Python Module Index

f

Index