HowToBound

Release 1.0

CAS

Sep 27, 2023

CONTENTS

Contents:

CHAPTER

ONE

THE PYTHON API

1.1 Usage Scenarios

The Python *bound* API is to be used if direct hardware interaction is to be implemented in the Python programming language. Additionally, it allows access to Karabo's point-to-point communication interface by means of binding the C++ code. Thus any processing algorithms implemented in Python which need to pass larger amounts of data in a pipelined fashion should be implemented in this API.

1.1.1 Benefits

Depending on the application the Python *bound* API may provide for faster development cycles than the C++ API. In any case, the Python *bound* API is feature-complete with respect to the C++, and frequently uses the same code basis by binding the C++ code.

1.1.2 Limitations

As Python is a dynamically typed language, applications which require close matching to hardware types may be better programmed in statically typed and compile-time checked C++.

Additionally, some libraries may only be available with C or C++ interfaces. In these cases, rather than binding the library to Python it is recommended to use the C++ API.

Finally, Python, being an interpreted language, has performance short-comings with respect to compiled C++ code. For most control aspects these should be negligible, and even for high-performance processing tasks can frequently be mitigated by using e.g. `mod:numpy` or `mod:scipy` routines. European XFEL's calibration suite is an example of high-performance code implemented in Python.

1.2 Programming Policies

While device developers are encouraged to write *pythonic* Python, the API purposely breaks with some conventions to more closely resemble the C++ API calls. This was done so that programmers switching between both languages will not need to learn a separate set of method calls to access the same underlying functionality.

Especially when directly accessing hardware it is considered good practice to be somewhat verbose in coding, rather than aim for the shortest possible implementation in Python. Accordingly, if e.g. a list comprehension significantly obscures the functionality implemented, consider writing a loop instead.

For documentation, it is best practice to follow documentation guidelines set in PEP8, and document using reStructured text syntax. Specifically, you will need to document each device's state diagram, as otherwise it should not be publicly released.

1.3 The “Conveyor” device

Consider the code of our device - ConveyorPy.py:

```
#!/usr/bin/env python

__author__="name.surname@xfel.eu"
__date__ ="November, 2014, 05:26 PM"
__copyright__="Copyright (c) 2010-2014 European XFEL GmbH Hamburg. All rights reserved."

import time

from karabo.bound import (
    BOOL_ELEMENT, DOUBLE_ELEMENT, KARABO_CLASSINFO, OVERWRITE_ELEMENT, SLOT_ELEMENT,
    PythonDevice, State, Unit
)




@KARABO_CLASSINFO("ConveyorPy", "1.3")
class ConveyorPy(PythonDevice):


    @staticmethod
    def expectedParameters(expected):
        """Description of device parameters statically known"""
        OVERWRITE_ELEMENT(expected).key("state")
                .setNewOptions(State.INIT, State.ERROR, State.STARTED, State.STOPPING,␣
↪State.STOPPED, State.STARTING)
                .setNewDefaultValue(State.INIT)
                .commit(),

        # Button definitions
        SLOT_ELEMENT(expected).key("start")
                .displayedName("Start")
                .description("Instructs device to go to started state")
                .allowedStates(State.STOPPED)
                .commit(),

        SLOT_ELEMENT(expected).key("stop")
                .displayedName("Stop")
                .description("Instructs device to go to stopped state")
                .allowedStates(State.STARTED)
                .commit(),

        SLOT_ELEMENT(expected).key("reset")
                .displayedName("Reset")
                .description("Resets in case of an error")
                .allowedStates(State.ERROR)
                .commit(),

        # Other elements
```

(continues on next page)

(continued from previous page)

```
        DOUBLE_ELEMENT(expected).key("targetSpeed")
                .displayedName("Target Conveyor Speed")
                .description("Configures the speed of the conveyor belt")
                .unit(Unit.METER_PER_SECOND)
                .assignmentOptional().defaultValue(0.8)
                .reconfigurable()
                .commit(),

        DOUBLE_ELEMENT(expected).key("currentSpeed")
                .displayedName("Current Conveyor Speed")
                .description("Shows the current speed of the conveyor")
                .readOnly()
                .commit(),

        BOOL_ELEMENT(expected).key("reverseDirection")
                .displayedName("Reverse Direction")
                .description("Reverses the direction of the conveyor band")
                .assignmentOptional().defaultValue(False)
                .allowedStates(State.STOPPED)
                .reconfigurable()
                .commit(),

        BOOL_ELEMENT(expected).key("injectError")
                .displayedName("Inject Error")
                .description("Does not correctly stop the conveyor, such "
                             "that a Error is triggered during next start")
                .assignmentOptional().defaultValue(False)
                .reconfigurable()
                .expertAccess()
                .commit(),

        )

    def __init__(self, configuration):
        # Always call PythonDevice constructor first!
        super(ConveyorPy, self).__init__(configuration)

        # Register function that will be called first
        self.registerInitialFunction(self.initialize)

        # Register slots
        self.registerSlot(self.start)
        self.registerSlot(self.stop)
        self.registerSlot(self.reset)

    def preReconfigure(self, config):
        """ The preReconfigure hook allows to forward the configuration to some␣
↪connected h/w"""

        try:
            if config.has("targetSpeed"):
                # Simulate setting to h/w
```

(continues on next page)

(continued from previous page)

```
                self.log.INFO("Setting to hardware: targetSpeed -> " + str(config.get(
↪"targetSpeed")))

            if config.has("reverseDirection"):
                # Simulate setting to h/w
                self.log.INFO("Setting to hardware: reverseDirection -> " + str(config.
↪get("reverseDirection")))

        except RuntimeError as e:
            # You may want to indicate that the h/w failed
            self.log.ERROR("'preReconfigure' method failed : {}".format(e))
            self.updateState(State.ERROR)

    def initialize(self):
        """ Initial function called after constructor but with equipped SignalSlotable␣
↪under runEventLoop"""
        try:
            # As the Initializing state is not mentioned in the allowed states
            # nothing else is possible during this state
            self.updateState(State.INIT)

            self.log.INFO("Connecting to conveyer hardware...")

            # Simulate some time it could need to connect and setup
            time.sleep(2.)

            # Automatically go to the Stopped state
            self.stop()
        except RuntimeError as e:
            self.log.ERROR("'initialState' method failed : {}".format(e))
            self.updateState(State.ERROR)

    def start(self):
        try:
            self.updateState(State.STARTING) # set this if long-lasting work follows

            # Retrieve current values from our own device-state
            tgtSpeed = self.get("targetSpeed")
            currentSpeed = self.get("currentSpeed")

            # If we do not stand still here that is an error
            if currentSpeed > 0.0:
                raise ValueError("Conveyer does not stand still at start-up")

            # Separate ramping into 50 steps
            increase = tgtSpeed / 50.0

            # Simulate a slow ramping up of the conveyor
            for i in range(50):
                currentSpeed += increase
                self.set("currentSpeed", currentSpeed);
                time.sleep(0.05)
```

(continues on next page)

(continued from previous page)

```
            # Be sure to finally run with targetSpeed
            self.set("currentSpeed", tgtSpeed)

            self.updateState(State.STARTED) # reached the state "Started"

        except RuntimeError as e:
            self.log.ERROR("'start' method failed : {}".format(e))
            self.updateState(State.ERROR)

    def stop(self):
        try:
            # Retrieve current value from our own device-state
            currentSpeed = self.get("currentSpeed")
            if currentSpeed != 0:
                self.updateState(State.STOPPING) # set this if long-lasting work follows
                # Separate ramping into 50 steps
                decrease = currentSpeed / 50.0

                # Simulate a slow ramping down of the conveyor
                for i in range(50):
                    currentSpeed -= decrease
                    self.set("currentSpeed", currentSpeed)
                    time.sleep(0.05)
                # Be sure to finally stand still
                if self.get("injectError"):
                    self.set("currentSpeed", 0.1)
                else:
                    self.set("currentSpeed", 0.0)

            self.updateState(State.STOPPED) # reached the state "Stopped"
        except RuntimeError as e:
            self.log.ERROR("'stop' method failed : {}".format(e))
            self.updateState(State.ERROR)

    def reset(self):
        self.set("injectError", False)
        self.set("currentSpeed", 0.0)
        self.initialize()
```

Consider the main steps of the code above, which are important to mention while writing devices in Python:

1. Import needed pieces from the karabo.bound package:

```
from karabo.bound import (
    KARABO_CLASSINFO, PythonDevice, launchPythonDevice,
    BOOL_ELEMENT, DOUBLE_ELEMENT, OVERWRITE_ELEMENT, SLOT_ELEMENT, Unit, State
)
```

2. Decide whether you want to use an FSM. In our example we don't use it, therefore we have:

```
from karabo.bound import Worker
```

   The current recommendation is to use NoFsm.

3. Place the decorator `KARABO_CLASSINFO` just before class definition. It has two parameters: "classId" and "version" similar to the corresponding C++ macro. In class definition we specify that our class inherits from `PythonDevice` as well as from `NoFsm` (see step 2):

```
@KARABO_CLASSINFO("ConveyorPy", "2.3")
class ConveyorPy(PythonDevice):
```

4. Constructor:

```
def __init__(self, configuration):
    # always call superclass constructor first!
    super(ConveyorPy,self).__init__(configuration)
    # Register function that will be called first
    self.registerInitialFunction(self.initialState)
    # Register slots
    self.registerSlot(self.start)
    self.registerSlot(self.stop)
    self.registerSlot(self.reset)
    self.worker = None
    self.timeout = 1000  # milliseconds
    self.repetition = -1 # forever
```

   In the constructor you always have to call the super class's constructor first.

   Then you need to register the function that will be called when the device is instantiated.

   Finally you have to register all the slots: in the example start, stop and reset.

5. Define the static method `expectedParameters`, where you should describe what properties are available on this device.

6. Define the implementation of initial function (in the example `initialState`) and of the slots. They will have to call `self.updateState(newState)` at the very end, in order to update device's state.

   These functions must be non-blocking: if they need to run some process which takes a long time, they should start it in a separate thread, or even better by using the `Worker` class. See the complete example code for the Worker's usage.

1.4 The "Worker" class

The `Worker` class is suitable for executing periodic tasks. It is defined in the `karabo.bound` module, from which it must be imported,

```
from karabo.bound import Worker
```

It can be instantiated and started like this:

```
self.counter = 0
self.timeout = 1000  # milliseconds
self.repetition = -1  # forever
self.worker = Worker(self.hook, self.timeout, self.repetition).start()
```

The 'repetition' parameter will specify how many times the task has to be executed (-1 means 'forever'), the 'timeout' parameter will set the interval between two calls and `self.hook` is the callback function defined by the user, for example:

```
def hook(self):
    self.counter += 1
    self.log.INFO("*** periodicAction : counter = " + str(self.counter))
```

The worker can then be stopped by overriding `preDestruction` method, like this:

```
def preDestruction(self):
    if self.worker is not None:
        if self.worker.is_running():
            self.worker.stop()
        self.worker.join()
        self.worker = None
```

This method will be automatically called during device shutdown process.

1.5 Usage of "threading" module

Sometimes if you would decide to use multithreading directly for executing periodic tasks in python karabo device like `Klass` you may follow these steps.

First import `threading` module.

```
import threading
```

Add some flag into python constructor (`__init__`) with the class scope…

```
self.running = False
```

Then define class function that will do periodic work as a thread

```
def polling(self):
    self.running = True
    while self.running:
        # do some useful work like
        # polling the hardware ...
        # ...
        time.sleep(1) # define some idle interval (1 sec)
```

Then define method that starts `polling` thread …

```
def start_polling(self):
    # use state here that fits your needs
    self.updateState(State.MONITORING)
    if self.pollingThread is None:
        self.pollingThread  threading.Thread(target=self.polling)
        self.log.INFO("Start polling thread")
        self.pollingThread.start()
```

The `start_polling` may be placed into `initialize` function, into slot of `start` button or another function that should activate the periodic task.

If this device is killed the `slotKillDevice` is called, which calls `preDestruction` method. So it is important to override this method to stop running our polling loop above

```
def preDestruction(self):
    self.running = False   # stop the loop
```

Please follow this pattern to allow `karabo-stop` of python server working properly. Otherwise the device and server may hang up!

CHAPTER

TWO

DAQ POLICY

Not every parameter of a device is interesting to record, such as the provided scenes. As such, the policy for each individual property can be set, on a per-instance basis.

These are specified using the `DAQPolicy` enum:

- *OMIT*: will not record the property to file;
- *SAVE*: will record the property to file;
- *UNSPECIFIED*: will adopt the global default DAQ policy. Currently, it is set to record, although this will eventually change to not recorded.

Legacy devices which do not specify a policy will have an *UNSPECIFIED* policy set to all their properties.

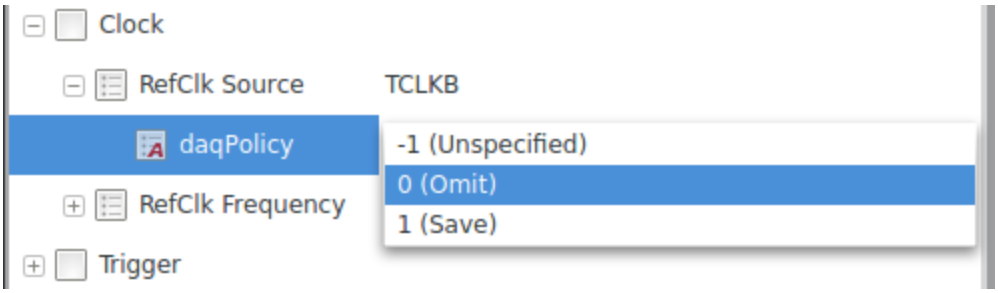
Note: This are applied to leaf properties. Nodes do not have DAQPolicy.

Developers should liaise with users to define which properties should be recorded. These can be set up programmatically:

```
from karabo.bound import DAQPolicy, MetricPrefix, UINT32_ELEMENT

@staticmethod
def expectedParameters(expected):
    (
        UINT32_ELEMENT(expected).key('enzymAssay')
            .displayedName("Assay")
            .descriptio("Enzym Activity")
            .unit(Unit.KATAL)
            .metricPrefix(MetricPrefix.MICRO)
            .daqPolicy(DAQPolicy.SAVE)
            .readOnly().initialValue(0)
            .commit(),
    )
```

The policy can be overwritten from the GUI, before instantiation, and saved in a project:



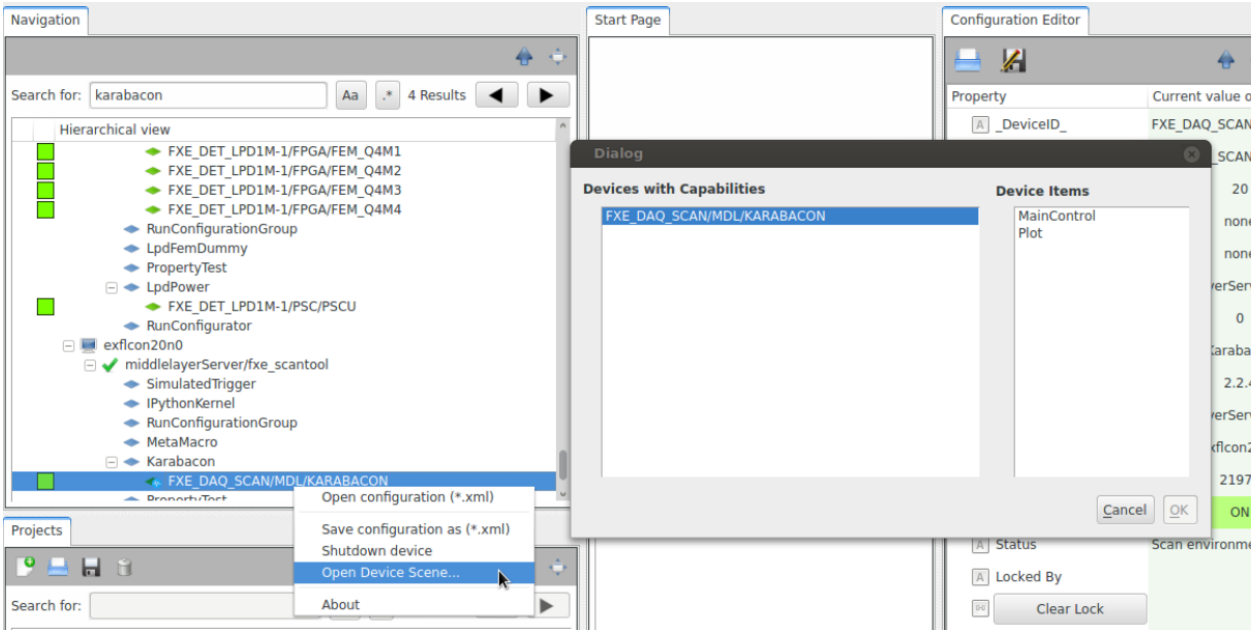
Injected properties, however, can only define their policy at the time of injection.

CHAPTER

THREE

DEVICE SCENES

Karabo provides a protocol for devices to share predefined scenes. These allows the author of a device to provide what they think are a good starting point. Moreover, these are easily accessible by from the topology panel in the GUI:



A default scene can also be accessed by double-clicking on a device.

This section shows how to enable your device to have builtin scenes.

Implementing this functionality requires the creation of a scene, in the scene editor, conversion to Python, and adding the *requestScene* framework slot.

3.1 From scene to Python

Begin by drawing an adequate scene in the GUI's scene editor, and save it locally on your computer as SVG (right-click on scene -> *Save to File*).

Use the *karabo-scene2py* utility to convert the SVG file to Python code:

```
$ karabo-scene2py scene.svg SA1_XTD2_UND/MDL/GAINCURVE_SCAN > scenes.py
```

The first argument is the scene file, the second is an optional deviceId to be substituted.

As it is generated code, make sure the file is PEP8 compliant. The final result should look more or less like the following:

```
from karabo.common.scenemodel.api import (
    IntLineEditModel, LabelModel, SceneModel, write_scene
 )

def get_scene(deviceId):
    input = IntLineEditModel(height=31.0,
                             keys=['{}.config.movingAverageCount'.format(deviceId)],
                             parent_component='EditableApplyLaterComponent',
                             width=67.0, x=227.0, y=18.0)
    label = LabelModel(font='Ubuntu,11,-1,5,50,0,0,0,0,0', foreground='#000000',
                       height=27.0, parent_component='DisplayComponent',
                       text='Running Average Shot Count',
                       width=206.0, x=16.0, y=15.0)
    scene = SceneModel(height=1017.0, width=1867.0, children=[input, label])

    return write_scene(scene)
```

Add this file to your device source project.

3.2 Providing the scene from your device

Add a read-only *VectorString* property called *availableScenes* to your expected parameters, and implement the *requestScene* framework slot. This is a predefined slot, which allows various actors to understand the scene protocol.

The slot takes a Hash *params* and returns a Hash with the origin, its datatype (deviceScene), and the scene itself:

```
from karabo.bound import DAQPolicy, VECTOR_STRING_ELEMENT
from karabo.common.api import KARABO_SCHEMA_DISPLAY_TYPE_SCENES as DT_SCENES

@staticmethod
def expectedParameters(expected):
    (
        VECTOR_STRING_ELEMENT(expected).key('availableScenes')
            .setSpecialDisplayType(DT_SCENES)
            .daqPolicy(DAQPolicy.OMIT)
            .readOnly().initialValue(['overview'])
            .commit()
    )

def requestScene(self, params):
    name = params.get('name', default='overview')
    payload = Hash('success', True, 'name', name,
                   'data', get_scene(self.getInstanceId()))

    self.reply(Hash('type', 'deviceScene',
                    'origin', self.getInstanceId(),
                    'payload', payload))

self.KARABO_SLOT(self.requestScene)
```

3.3 Providing several scenes from your device

Would you want to provide several scenes (e.g., overview and control scene), you can define several functions in *scenes.py*, and modify *requestScene* to check *params['name']*:

```
from karabo.bound import DAQPolicy, VECTOR_STRING_ELEMENT
from karabo.common.api import KARABO_SCHEMA_DISPLAY_TYPE_SCENES as DT_SCENES

import .scenes

@staticmethod
def expectedParameters(expected):
    (
        VECTOR_STRING_ELEMENT(expected).key('availableScenes')
            .setSepcialDisplayType(DT_SCENES)
            .daqPolicy(DAQPolicy.OMIT)
            .readOnly().initialValue(['overview', 'controls'])
            .commit()
    )

def requestScene(self, params):
    payload = Hash('success', False)
    name = params.get('name', default='overview')

    if name == 'overview':
        payload.set('success', True)
        payload.set('name', name)
        payload.set('data', scenes.overview(self.getInstanceId()))

    elif name == 'controls':
        payload.set('success', True)
        payload.set('name', name)
        payload.set('data', scenes.controls(self.getInstanceId()))

    self.reply(Hash('type', 'deviceScene',
                    'origin', self.getInstanceId(),
                    'payload', payload))

self.KARABO_SLOT(self.requestScene)
```

Note: There is the convention that the default scene (of your choice) should be first in the *availableScenes* list.

3.4 Providing Table Elements

As described in table-element, table elements are vectors of hash, the schema is specified as Hash serialized to XML, (which *karabo-scene2py* takes care of).

In this case, it's fine to break the PEP8 80 characters limit. A table element looks like:

```
 table = TableElementModel(
     column_schema='TriggerRow:<root KRB_Artificial="">CONTENT</root>',
     height=196.0, keys=['{}.triggerEnv'.format(deviceId)],
     klass='DisplayTableElement',
     parent_component='DisplayComponent',
     width=436.0, x=19.0, y=484.0
)
```

3.5 Linking To Other Devices Scenes

The following applies whether you want to link to another of your scenes or to another device's scene.

Let's say that you want to add links in your *overview* scene to your *controls* scene.

The `DeviceSceneLinkModel` allows you to specify links to other dynamically provided scenes.

In your *scenes.py*, import *DeviceSceneLinkModel* and *SceneTargetWindow* from *karabo.common.scenemodel.api* and extend `overview(deviceId)()`:

```
from karabo.common.scenemodel.api import DeviceSceneLinkModel, SceneTargetWindow

 def overview(deviceId):
    # remaining scene stays the same

     link_to_controls = DeviceSceneLinkModel(
         height=40.0, width=314.0, x=114.0, y=227.0,
         parent_component='DisplayComponent',
         keys=['{}.availableScenes'.format(deviceId)], target='controls',
         text='Controls scene',
         target_window=SceneTargetWindow.Dialog)

     children = [label, input, link_to_controls]
     scene = SceneModel(height=1017.0, width=1867.0, children=children)

    return write_scene(scene)
```

If you want to link to another device, make `overview()` accept another *remoteDeviceId* parameter, and point the link to that device:

```
def overview(deviceId, remoteDeviceId):
    # remaining scene stays the same

    link_to_remote = DeviceSceneLinkModel(
        height=40.0, width=314.0, x=114.0, y=267.0,
        parent_component='DisplayComponent',
        text='Link to other device',
```

(continues on next page)

(continued from previous page)

```
        keys=['{}.availableScenes'.format(remoteDeviceId)], target='scene',
        target_window=SceneTargetWindow.Dialog
    )

    children = [label, input, link_to_controls, link_to_remote]
    scene = SceneModel(height=1017.0, width=1867.0, children=children)

  return write_scene(scene)
```

3.6 Reference Implementations

LimaCameras: provides a default scene with an image view in its `LimaDevice` KEP21: definition of the scene protocol

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search