HowToMiddleLayer

Release 1.0

Controls

May 15, 2024

CONTENTS

Contents:

CHAPTER

ONE

KARABO MIDDLELAYER API

The Karabo `middlelayer` API is written in pure Python deriving its name from the main design aspect: controlling and monitoring driver devices while providing state aggregation and additional functionality. The same interface is also used for the macros. Hence, this api exposes an efficient interface for tasks such as interacting with other devices via device proxies to enable monitoring of properties, executing commands and waiting on their completion, either synchronously or asynchronously.

1.1 Start simple: Hello World Device!

Below is the source code of a Hello World! device:

```
from karabo.middlelayer import Device, Slot, String


class HelloWorld(Device):

    __version__ = "2.0"

    greeting = String(
        defaultValue="Hello World!",
        description="Message printed to console.")

    @Slot()
    async def hello(self):
        print(self.greeting)

    async def onInitialization(self):
        """ This method will be called when the device starts.

            Define your actions to be executed after instantiation.
        """
```

The middlelayer device is created by inheriting from the middlelayer's `Device` base class. Below the device class definition are the expected parameters, containing the static schema of the device. A property *__version__* can indicate the lowest Karabo version in which this device is supposed to run. In this device we create a Karabo property *greeting* which is also referred to as `KaraboValue`. This property has an assigned `descriptor` *String* containing the `attributes`. In this example the *defaultValue* and the *description* attributes are defined, which is rendered in the karabo GUI as a text box showing "Hello World!".

Additionally, we create a single `Slot` *hello* by using a decorator. This slot will be rendered in the karabo GUI as a button enabling us to print the greeting string to console.

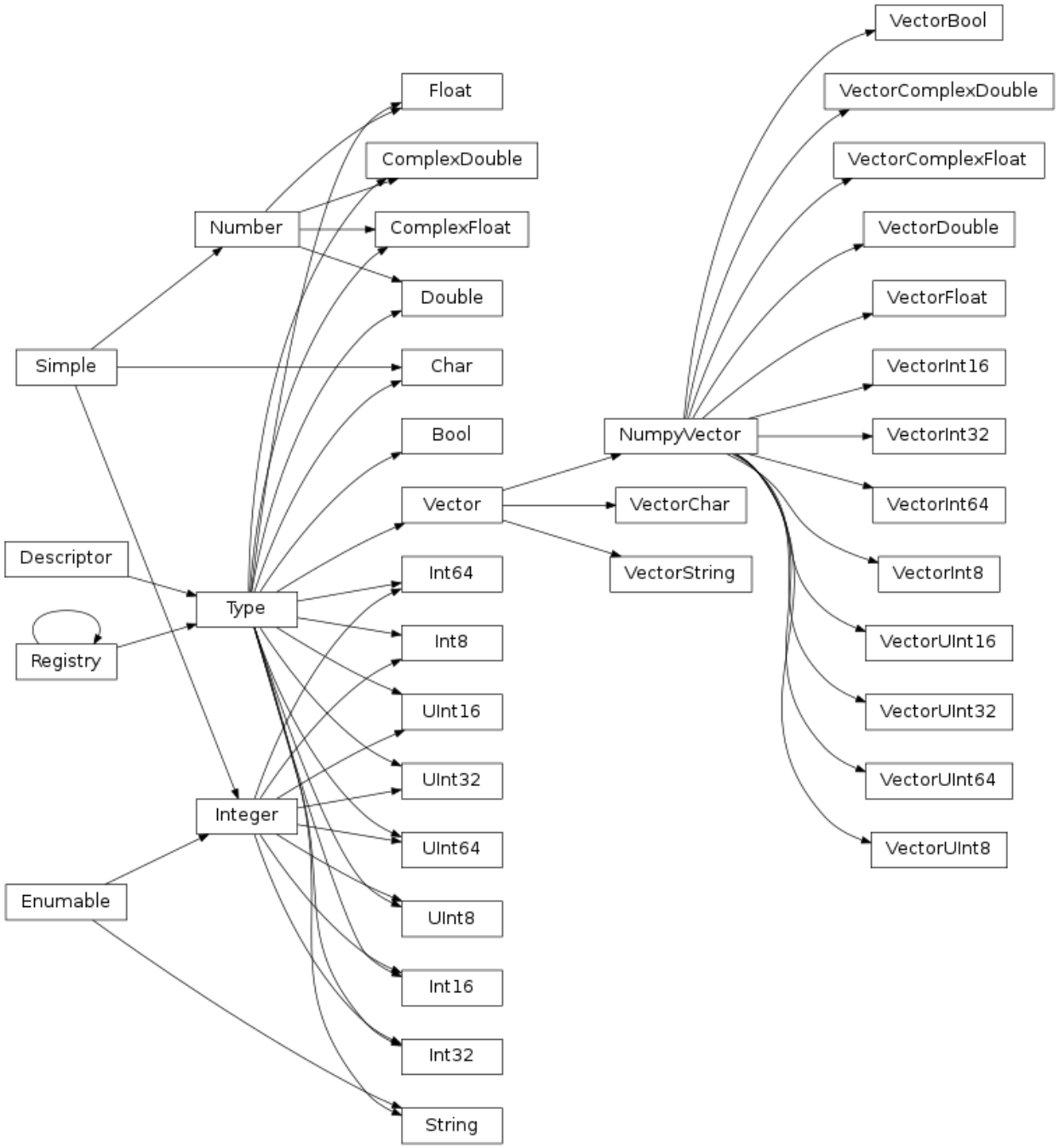
1.1.1 Properties: Karabo Descriptors

As shown by the example, every device has a *schema*, which contains all the details about the expected parameters, its types, comments, units, everything. In the middlelayer this schema is built by so-called **karabo descriptors**. A schema is only broadcasted rarely over the network, typically only during the initial handshake with the device. Once the schema is known, only *configurations*, or even only parts of configurations, are sent over the network in a tree structure called *Hash* (which is not a hash table).

These configurations know nothing anymore about the meaning of the values they contain, yet they are very strictly typed: even different bit sizes of integers are conserved.

Descriptors describe the content of a device property. As shown in the *Hello World* example, The description is done in their attributes, which come from a fixed defined set.

It may be useful to note that instances of this class do not contain any data, instead they are describing which values a device property may take and they are given as keyword arguments upon initialization.


VectorBool
VectorComplexDouble
VectorComplexFloat
VectorDouble
VectorFloat
VectorInt16
VectorInt32
VectorInt64
VectorInt8
VectorUInt16
VectorUInt32
VectorUInt64
VectorUInt8
Float
ComplexDouble
ComplexFloat
Double
Char
Bool
Int64
Int8
UInt16
UInt32
UInt64
UInt8
Int16
Int32
String
Number
Simple
Descriptor
Registry
Type
Integer
Enumable
Vector
VectorChar
VectorString
NumpyVector

1.1.2 Attributes

Attributes of properties may be accessed during runtime as members of the property descriptor. Depending on the type of the property, some attributes might not be accessible.

The **common** descriptor attributes are:

Common Attribute	Example
displayType	e.g. oct, bin, dec, hex, directory
unitSymbol	e.g. `Unit.METER`
metricPrefixSymbol	e.g. `MetricPrefix.MILLI`
accessMode	e.g. `AccessMode.READONLY`
assignment	e.g. `Assignment.OPTIONAL`
defaultValue	the default value or None
requiredAccessLevel	e.g. `AccessLevel.EXPERT`
allowedStates	the list of allowed states
tags	a list of strings as property tags
alias	a string to be used as alias
daqPolicy	e.g. `DaqPolicy.SAVE`

The min and maxSize attributes are only available for **vectors** if they have been set before:

Vector Attribute	Example
minSize	the minimum size of vector
maxSize	the maximum size of vector

Attributes that are available for **simple types** only (*Double*, *Int32*, etc.):

Simple Attribute	Example
minInc	the inclusive-minimum value
minExc	the exclusive-minimum value
maxInc	the inclusive-maximum value
maxExc	the exclusive-maximum value
warnLow	warn threshold low
warnHigh	warn threshold high
alarmLow	alarm threshold low
alarmHigh	alarm threshold high

1.1.3 Handling timestamps

When a user operates on a `KaraboValue`, the timestamp of the result is the newest timestamp of all timestamps that take part in the operation, unless the user explicitly sets a different one. This is in line with the validity intervals described above: if a value is composed from other values, it is valid typically starting from the moment that the last value has become valid (this assumes that all values are still valid at composition time, but this is the responsibility of the user, and is typically already the case).

All properties in Karabo may have timestamps attached. In the middlelayer API they can be accessed from the `timestamp` attribute:

```
self.speed.timestamp
```

They are automatically attached and set to the current time upon assignment of a value that does not have a timestamp:

```
self.steps = 5  # current time as timestamp attached
```

A different timestamp may be attached using `karabo.middlelayer.Timestamp`:

```
self.steps.timestamp = Timestamp("2009-09-01 12:34 UTC")
```

If a value already has a timestamp, it is conserved, even through calculations. If several timestamps are used in a calculation, the newest timestamp is used. In the following code, `self.speed` gets the timestamp of either `self.distance` or `self.times`, whichever is newer:

```
self.speed = 5 * self.distance / self.times[3]
```

Due to this behaviour, using in-place operators, such as += is discouraged, as the timestamp would be conserved:

```
self.speed = 5  # A new timestamp is attached

self.speed += 5  # The timestamp is kept
```

The above effectively is:

```
self.speed = self.speed + 5
```

And whilst the value is 10, we used the newest timestamp available from either component, here the previous one from `self.speed`, and the timestamp never gets incremented! In order to create a new timestamp, the raw value needs to be accessed:

```
self.speed = self.speed.value + 5
```

Since the value and 5 are both integers, no timestamp is available, and a new one is created.

Warning: Developers should be aware that automated timestamp handling defaults to the newest timestamp, i.e. the time at which the last assignment operation on a variable in a calculation occured. Additionally, these timestamps are not synchronized with XFEL's timing system, but with the host's local clock.

When dealing with several inputs, a function can use the `karabo.middlelayer.removeQuantity()` decorator, to ease the readability:

```
from karabo.middlelayer import removeQuantity

steps = Int32()
speed = Int32()
increment = Int32()

@removeQuantity
def _increment_all_parameters(self, steps, speed, increment):
    self.steps = steps + increment
    self.speed = speed + increment

@Slot()
async def incrementAllParameters(self):
    self._increment_all_params(self.steps, self.speed, self.increment)
```

1.2 Karabo Slots

`karabo.middlelayer.Slot` is the way to mark coroutines as actionable from the ecosystem, whether from the GUI, or other Middlelayer devices:

```
from karabo.middlelayer import Slot

 @Slot(displayedName="Start",
       description="Prints an integer",
       allowedStates={State.OFF})
 async def start(self):
     self.state = State.ON

     i = 0
     while True:
         await sleep(1)
         print(i)
         i = i + 1
```

A **golden rule** in a Middlelayer device is that `Slot` are coroutines The slot exits with the last state update and returns once the code is run through.

`Slot` has a number of arguments that are explained in *Karabo Attributes*

1.2.1 Holding a Slot (Return with correct state)

In certain cases, it may be useful to keep a slot, to prevent a user to interfere with current operation, for example. Since slots are asynchronous, some trickery is required. For simplicity, below is an example assuming we read out the motor states in a different task.

```
async def state_monitor():
    # A simple state monitor
    while True:
        await waitUntilNew(self.motor1.state, self.motor2.state)
        state = StateSignifier().returnMostSignificant(
            [self.motor1.state, self.motor2.state])
        if state != self.state:
            self.state = state

@Slot(displayedName="Move",
      description="Move a motor",
      allowedStates={State.ON})
async def move(self):

    # We move to motors at once
    await gather(self.motor1.move(), self.motor2.move())
    # We wait for our own state change here to exit this slot with
    # the expected state, e.g. ERROR or MOVING.
    await waitUntil(lambda: self.state != State.ON)
```

1.3 Karabo Attributes

This section describes how `attributes` can be used in **Properties** and **Slots**.

Slots have two important attributes that regulate their access using states and access modes.

1.3.1 Required Access Level

The `requiredAccessLevel` attribute allows to set at which access level this property may be reconfigured or Slot may be executed. The minimum `requiredAccessLevel` for a reconfigurable property or Slot is at least *USER (level 1)* if not explicitly specified.

Furthermore, this feature can be used to hide features from lower level users and some user interfaces might hide information depending on the access level.

User levels are defined in `karabo.middlelayer.AccessLevel`, and range from *OBSERVER (level 0)* to *ADMIN (level 4)*. Consequently, a user with lower level access, such as *OPERATOR (level 2)*, will have access to less information than *EXPERT (level 3)*.

First, import `AccessLevel`:

```
from karabo.middlelayer import AccessLevel
```

In the following example, we create a Slot and a property for a voltage controller whose access is limited to the expert level, such that operators or users cannot modify the device. The definition of such a slot is then as follows:

```
targetVoltage = Double(
    defaultValue=20.0
    requiredAccessLevel=AccessLevel.EXPERT)

@Slot(displayedName="Ramp Voltage up",
      requiredAccessLevel=AccessLevel.EXPERT)
async def rampUp(self):
    self.status = "Ramping up voltage"

    ... do something
```

Note: The default requiredAccesslevel is `AccessLevel.OBSERVER` (level 0).

1.3.2 Allowed States

The middlelayer API of Karabo uses a simple state machine to protect slot execution and property reconfiguration. Therefore, it is possible to restrict slot calls to specific states using the *allowedStates* attribute in the `Slot` definition.

States are provided and defined in the Karabo Framework in `karabo.middlelayer.State`

In the example below, the voltage of the controller can only be ramped up if the device is in the state: `State.ON`. In the Slot *rampUp* we also switch the device state to *State.RUNNING*, since a ramp up action will be running after the first call. With this protection, the procedure of ramping up the device can only be executed again after it has finished.

```
targetVoltage = Double(
    defaultValue=20.0
```

(continues on next page)

(continued from previous page)

```
    requiredAccessLevel=AccessLevel.EXPERT,
    allowedStates={State.ON})

@Slot(displayedName="Ramp Voltage up",
      requiredAccessLevel=AccessLevel.EXPERT
      allowedStates={State.ON})
async def rampUp(self):
    self.status = "Ramping up voltage"

    self.state = State.RUNNING
    ... do something
```

It is possible to define an arbitrary quantity of states:

```
allowedStates={State.ON, State.OFF}
```

Note: By default every property and Slot may be reconfigured or executed for **all** device states.

1.3.3 AccessMode

The *accessMode* attribute allows to set if a property in a device is a **READONLY**, **RECONFIGURABLE** or **INITONLY**.

Init only properties can only be modified during before instantiation of the device.

First, import `AccessMode`:

```
from karabo.middlelayer import AccessMode
```

Based on the previous example, we add a read only property for the current voltage of our voltage controller:

```
currentVoltage = Double(
    accessMode=AccessMode.READONLY,
    requiredAccessLevel=AccessLevel.OPERATOR)

targetVoltage = Double(
    defaultValue=20.0
    requiredAccessLevel=AccessLevel.EXPERT)
```

Note: The default *accessMode* is `AccessMode.RECONFIGURABLE`. The read only setting of a property has to be provided explicitly.

1.3.4 Assignment

The *assignment* attribute declares the behavior of the property on instantiation of the device. Its function is coupled to the *accessMode*. It can be either **OPTIONAL**, **MANDATORY** or **INTERNAL**.

Init only properties can only be modified during before instantiation of the device.

These assignments are very import in the configuration management.

- **INTERNAL** assigned properties are always erased from configuration and indicate that they are provided from the device internals on startup. This is made visible to the operator, they cannot be edited for example in the graphical user interface.
- **MANDATORY** assigned properties must be provided on instantiation. They are typically left blank, and the operator must provide a value (e.g. host, ip for a camera).

```
from karabo.middlelayer import AccessMode, Assignment, Double, String

# assignment have no effect
currentVoltage = Double(
    accessMode=AccessMode.READONLY,
    requiredAccessLevel=AccessLevel.OPERATOR)

# default assignment is OPTIONAL
targetVoltage = Double(
    defaultValue=20.0
    requiredAccessLevel=AccessLevel.EXPERT)

# default accessMode is RECONFIGURABLE
# on instantiation, this property is MANDATORY and must be provided
host = String(
    assignment = Assignment.MANDATORY,
    requiredAccessLevel=AccessLevel.EXPERT)

# default accessMode is RECONFIGURABLE
# on instantiation, this property is INTERNAL. In this case it is read
# from hardware, but it can be reconfigured on the online device
targetCurrent = Double(
    assignment = Assignment.INTERNAL,
    requiredAccessLevel=AccessLevel.ADMIN)
```

Note: The default *assignment* is `Assignment.OPTIONAL`.

1.3.5 DAQ Policy

Not every parameter of a device is interesting to record. **As a workaround for a missing DAQ feature**, the policy for each individual property can be set, on a per-class basis.

These are specified using the `karabo.middlelayer.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.

```
from karabo.middlelayer import DaqPolicy

 currentVoltage = Double(
     accessMode=AccessMode.READONLY,
     requiredAccessLevel=AccessLevel.OPERATOR,
     daqPolicy=DaqPolicy.SAVE)
```

1.3.6 Handling Units

You can define a unit for a property, which is then used in the calculations of this property. In the Middlelayer API units are implemented using the `pint` module.

A unit is declared using the `unitSymbol` and further extended with the `metricPrefixSymbol` attribute:

```
distance = Double(
    unitSymbol=Unit.METER,
    metricPrefixSymbol=MetricPrefix.MICRO)
times = VectorDouble(
    unitSymbol=Unit.SECOND,
    metricPrefixSymbol=MetricPrefix.MILLI)
speed = Double(
    unitSymbol=Unit.METER_PER_SECOND)
steps = Double()
```

Once declared, all calculations have correct units:

```
self.speed = self.distance / self.times[3]
```

In this code units are converted automatically. An error is raised if the units don't match up:

```
self.speed = self.distance + self.times[2]  # Ooops! raises error
```

If you need to add a unit to a value which doesn't have one, or remove it, there is the `unit` object which has all relevant units as its attribute:

```
self.speed = self.steps * (unit.meters / unit.seconds)
self.steps = self.distance / (3.5 * unit.meters)
```

Warning: While the Middlelayer API of Karabo in principle allows for automatic unit conversion, developers are strongly discouraged to use this feature for critical applications: the Karabo team simply cannot guarantee that `pint` unit handling is preserved in all scenarios, e.g. that a unit is not silently dropped.

1.4 Device States

Every device has a state, one of these defined in `karabo.middlelayer.State`. They are used to show what the device is currently doing, what it can do, and which actions are not allowed.

For instance, it can be disallowed to call the `start` slot if the device is in `State.STARTED` or `State.ERROR`. Such control can be applied to both slot calls and properties.

The states and their hierarchy are documented in the Framework.

Within the Middlelayer API, the `State` is an enumerable represented as string, with a few specific requirements, as defined in `karabo.middlelayer_api.device.Device`

Although not mandatory, a device can specify which states are used in the `options` attribute:

```
from karabo.middlelayer import Overwrite, State

state = Overwrite(
     defaultValue=State.STOPPED,
     displayedName="State",
     options={State.STOPPED, State.STARTED, State.ERROR})
```

If this is not explicitly implemented, all states are possible.

1.4.1 State Aggregation

If you have several proxies, you can aggregate them together and have a global state matching the most significant. In the example, this is called *trumpState* and makes use of `karabo.middlelayer.StateSignifier()`.

```
from karabo.middlelayer import background, StateSignifier

async def onInitialization(self):
    self.trumpState = StateSignifier()
    monitor_task = background(self.monitor_states())

async def monitor_states(self):
    while True:
        # Here self.devices is a list of proxies
        state_list = [dev.state for dev in self.devices]
        self.state = self.trumpState.returnMostSignificant(state_list)
        await waitUntilNew(*state_list)
```

As well as getting the most significant state, it will attach the newest timestamp to the returned state.

It is also possible to define your own rules, as documented in `karabo.common.states.StateSignifier`

The following shows how to represent and query a remote device's state and integrate it in a device:

```
from karabo.middlelayer import (
    AccessMode, Assignment, background, connectDevice, State, String,
    waitUntilNew)

remoteState = String(
    displayedName="State",
    enum=State,
```

(continues on next page)

(continued from previous page)

```
    displayType="State",  # This type enables color coding in the GUI
    description="The current state the device is in",
    accessMode=AccessMode.READONLY,
    assignment=Assignment.OPTIONAL,
    defaultValue=State.UNKNOWN)

async def onInitialization(self):
    self.remote_device = await connectDevice("some_device")
    self.watch_task = background(self.watchdog())

async def watchdog(self):
   while True:
       await waitUntilNew(self.remote_device)
       state = self.remote_device.state
       self.remoteState != state:
         self.remoteState = state:
```

1.5 Tags and Aliases

It is possible to assign a property with tags and aliases.

- Tags can be multiple per property and can therefore be used to group properties

together. - Aliases are unique and for instance used to map hardware commands to Karabo property names.

These are typically used both together without the need for keeping several lists of parameters and modes.

To begin, mark the properties as desired, here are properties that are polled in a loop, and properties that are read once, at startup, for instance:

```
from karabo middlelayer import AccessMode, Bool, String

isAtTarget = Bool(displayedName="At Target",
                  description="The hardware is on target position",
                  accessMode=AccessMode.READONLY,
                  alias="SEND TARGET",  # The hardware command
                  tags={"once", "poll"})  # The conditions under which to query

hwStatus = String(displayedName="HW status",
                  description="status, as provided by the hardware",
                  accessMode=AccessMode.READONLY,
                  alias="SEND STATUS",  # The hardware command
                  tags={"poll"})  # The conditions under which to query

hwVersion = String(displayedName="HW Version",
                   description="status, as provided by the hardware",
                   accessMode=AccessMode.READONLY,
                   alias="SEND VERSION",  # The hardware command
                   tags={"once"})  # The conditions under which to query
```

Tags of a property can be multiple, and are contained within a set.

Once this is defined, `karabo.middlelayer.Schema.filterByTags()` will return a hash with the keys of all properties having a specific tag:

```
async def onInitialization(self):
    schema = self.getDeviceSchema()

    # Get all properties that are to be queried once
    onces = schema.filterByTags("once")
    # This returns a Hash which is the subset of the current configuration,
    # with the property names that have 'once' as one of their tags.

    # Get the hardware commands, aliases, for each of the properties
    tasks = {prop: self.query_device(schema.getAliasAsString(prop)) for prop in once.
↪keys()}

    # Query
    results = await gather(tasks)

    # Set the result
    for prop, value in results.items():
        setattr(self, prop, value)
```

whilst a background task can poll the other parameters in a loop:

```
from karabo.middlelayer import background, gather

async def onStart(self):
    schema = self.getDeviceSchema()
    # Get all properties that are to be polled
    to_poll = schema.filterByTags("poll")

    # Create a background loop
    self.poll_task = background(self.poll(to_poll))


 async def poll(self, to_poll):
     while True:
         # Get the hardware commands for each of the properties
         tasks = {prop: self.query_device(schema.getAliasAsString(prop)) for prop in to_
↪poll.keys()}

         # Query
         results = await gather(tasks)

         # Set the result
         for prop, value in results.items():
             setattr(self, prop, value)
```

Note: The concepts of background and gather are explained later in chapter 2

- OphirPowerMeter is a device interfacing with a meter over tcp making use of tags and aliases

1.6 Nodes

Nodes allow a device's properties to be organized in a hierarchical tree-like structure: Devices have properties - node properties - which themselves have properties.

If a device has a node property with key *x* and the node has a property of type double with key *y*, then the device will have a property of type double with key *x.y*.

1.6.1 Defining a Node's Properties

To create a device with node properties, first create a class which inherits from `Configurable` with the desired properties for the node. These are created as you would for properties of a device, at class scope, and understand the same attribute arguments.

For example, the following class is used to create a node for a (linear) motor axis with units in mm and actual and target position properties:

```
class LinearAxis(Configurable):
    actualPosition = Double(
        displayedName="Actual Position",
        description="The actual position of the axis.",
        unitSymbol=Unit.METER,
        metricPrefixSymbol=MetricPrefix.MILLI,
        accessMode=AccessMode.READONLY,
        absoluteError=0.01)

    targetPosition = Double(
        displayedName="Target Position",
        description="Position argument for move.",
        unitSymbol=Unit.METER,
        metricPrefixSymbol=MetricPrefix.MILLI,
        absoluteError=0.01)
```

1.6.2 Adding Node Properties to a Device

Nodes are added to a device in the same way as other properties, at class scope, using the `Node` class and understand the same attribute arguments as other properties where these make sense.

So the following creates a device with two node properties for two motor axes, using the `LinearAxis` class above:

```
class MultiAxisController(Device):
    axis1 = Node(
        LinearAxis,
        displayedName="Axis 1",
        description="The first motor axis.")

    axis2 = Node(
        LinearAxis,
        displayedName="Axis 2",
        description="The second motor axis.")
```

The resulting device will have, for example, a node property with key *axis1* and a double property with key *axis2.targetPosition*.

1.6.3 Node: Required Access Level

To be able to access a property, a user must have access rights equal to or above the required level for the property, specified by the *requiredAccessLevel* descriptor. For properties belonging to nodes, the user must have the access rights for the property and all parent nodes above it in the tree structure.

1.7 Error Handling

Errors happen and When they happen in Python typically an exception is raised. The best way to do error handling is to use the usual Python try-except-finally statements.

There are two types of errors to take care of in Middlelayer API: `CancelledError` and `TimeoutError`

```
from asyncio import CancelledError, TimeoutError, wait_for
from karabo.middlelayer import connectDevice, Slot

@Slot()
async def doSomething(self):
    try:
        # start something here, e.g. move some motor
    except CancelledError:
        # handle error
    finally:
        # something which should always be done, e.g. move the motor
        # back to its original position

@Slot()
async def doOneMoreThing(self):
    try:
        await wait_for(connectDevice("some_device"), timeout=2)
    except TimeoutError:
        # notify we received a timeout error
    finally:
        # reconnect to the device
```

Note: Both `CancelledError` and `TimeoutError` are imported from *asyncio*.

Sometimes, however, an exception may be raised unexpectedly and has no ways of being handled better. `onException()` is a mechanism that can be overwritten for this usage:

```
async def onException(self, slot, exception, traceback):
    """If an exception happens in the device, and not handled elsewhere,
    it can be caught here.
    """
    self.logger.warn(f"An exception occured in {slot.method.__name__} because {exception}
↪")
    await self.abort_action()
```

It is also possible that a user or Middlelayer device will cancel a slot call:

```
async def onCancelled(self, slot):
    """To be called if a user cancels a slot call"""
    tasks = [dev.stop() for dev in self.devices()]
    await allCompleted(*tasks)
```

1.7.1 Don't use *try ... except Exception* pattern

In the middlelayer API so-called tasks are created. Whenever a device is shutdown, all active tasks belonging to this device are cancelled. Tasks might be created by the device developer or are still active *Slots*. If a task is cancelled, an *CancelledError* is thrown and by using a *try ... except Exception* pattern, the exception and underlying action will always be fired. In the bottom case, we want to log an error message. Since the device is already shutting down, the task created by the log message will never be retrieved nor cancelled leaving a remnant on the device server. Subsequently, the server cannot shutdown gracefully.

This changed on **Python 3.8** where *CancelledError* is inherits from *BaseException*.

```
from asyncio import CancelledError, TimeoutError, wait_for
from karabo.middlelayer import connectDevice, Slot

async def dontDoThisTask(self):
    while True:
        try:
            # Some action here
        except Exception:
            self.logger.error("I got cancelled but I cannot log")
            # This will always be fired
```

Warning: Always catch a `CancelledError` explicitly when using a *try ... except Exception* pattern!

1.8 Code Style

While PEP8, PEP20, and PEP257 are stylings to follow, there are a few Karabo-specific details to take care of to improve code quality, yet keep consistency, for better maintainability across Karabo's 3 APIs.

An example of a major exception from PEP8 is that underscores should never be used for public device properties or slots.

1.8.1 Imports

Imports follow *isort* style: they are first in resolution order (built-ins, external libraries, Karabo, project), then in import style (*from imports*, *imports*), then in alphabetical order:

```
import sys
from asyncio import wait_for

import numpy as np

from karabo.middlelayer import (
```

(continues on next page)

(continued from previous page)

```
    connectDevice, Device, Slot, String)

from .scenes import control, default
```

isort can automatically fix imports by calling it with the filename:

```
$ isort Keithley6514.py
Fixing /data/danilevc/Keithley6514/src/Keithley6514/Keithley6514.py
```

1.8.2 Class Definitions

Classes should be *CamelCased*:

```
class MyDevice(Device):
    pass
```

Abbreviations in class names should be capitalised:

```
class JJAttenutator(Device):
    pass

class SA3MirrorsWitch(Device):
    pass
```

1.8.3 Class Properties

Properties part of the device's schema, which are exposed, should be *camelCased*, whereas non-exposed variables should *have_underscores*:

```
name = String(displayedName="Name")

someOtherString = String(
    displayedName="Other string",
    defaultValue="Hello",
    accessMode=AccessMode.READONLY)

valid_ids = ["44eab", "ff64d"]
```

1.8.4 Slots and Methods

Slots are *camelCased*, methods *have_underscores*. Slots must not take arguments, apart from *self*.

```
@Slot(displayedName='Execute')
async def execute(self):
    """This slot is exposed to the system"""
    self.state = State.ACTIVE
    await self.execute_action()

@Slot(displayedName='Abort',
```

(continues on next page)

(continued from previous page)

```
      allowedStates={State.ACTIVE, State.ERROR})
async def abortNow(self):
    self.state = state.STOPPING
    await self.abort_action()

async def execute_action(self):
    """This is not exposed, and therefore PEP8"""
    pass
```

Mutable objects must not be used as default values in method definitions.

1.8.5 Printing and Logging

Logging is the way to share information to developers and maintainers. This allows for your messages to be stored to files for analysis at a later time, as well as being shared with the GUI under certain conditions.

The Middlelayer API has its own *Logger* implemented as a `Configurable`. It is part of the Device class and no imports are required.

Whilst it can be used either as *self.log* or *self.logger*, the preferred style is as follows:

```
from karabo.middlelayer import allCompleted

async def stop_all(self):
    self.logger.info("Stopping all devices")
    tasks = [device.stop() for device in self.devices]
    done, pending, failed = await allCompleted(*tasks)
    if failed:
        self.logger.error("Some devices could not be stopped!")
```

Note: Logging is disabled in the constructor `__init__()`.

1.8.6 Inplace Operators

Inplace operations on Karabo types are discouraged for reasons documented in *Handling timestamps*.

Don't do:

```
speed = Int32(defaultValue=0)

@Slot()
async def speedUp(self):
    self.speed += 5
```

But rather:

```
speed = Int32(defaultValue=0)

@Slot()
async def speedUp(self):
    self.speed = self.speed.value + 5
```

1.8.7 Exceptions

It is preferred to check for conditions to be correct rather than using exceptions. This defensive approach is to ensure that no device would be stuck or affect other devices running on the same server.

Therefore, the following is discouraged:

```
async def execute_action(self):
    try:
        await self.px.move()
    except:
        pass
```

But rather:

```
async def execute_action(self):
    if self.px.state not in {State.ERROR, State.MOVING}:
        await self.px.move()
    else:
        pass
```

If exceptions are a must, then follow the *Error Handling*

1.8.8 Use Double and NOT Float

The middlelayer API supports both *Double* and *Float* properties.

However, behind the scenes a *Float* value is casted as numpy's *float32* type. Casting this value back to float64 may lead to different value. Hence, services on top of the framework might cast this value to a string before casting to the built-in python float of 64 bit to prevent cast errors. Note, that a 32 bit float has a precision of 6, which might be of different expectation for a normal python developer.

Use `karabo.middlelayer.Double` instead of `Float`.

CHAPTER

TWO

MIDDLELAYER DEVICE WITH PROXIES

MonitorMotor is a middle layer device documenting best practice for monitoring a single *remote device*, that is, a device written with either of the C++ or Python API.

In this example, the device will initialise a *connection* with a *remote motor device*, restart the connection if the remote device disappears or resets, and display the *motorPosition* integer property of that device.

This example introduces the concepts of *Device, connectDevice, isAlive, waitUntilNew,* and *wait_for*.

2.1 Start simple: A single proxy

We recapitulate our knowledge and start simple by creating our device class, inheriting from *Device*:

```
from karabo.middlelayer import Device, State

class MonitorMotor(Device):

    def __init__(self, configuration):
        super(MonitorMotor, self).__init__(configuration)

    async def onInitialization(self):
        self.state = State.INIT
```

`Device` is the base class for all middle layer devices. It inherits from `Configurable` and thus you can define expected parameters for it.

2.1.1 Connecting to the Remote Device

To connect to the remote device, we must have its control address. In this example, it is registered as "SA1_XTD9_MONO/MOTOR/X".

We must first import the `connectDevice()` function:

```
from karabo.middlelayer import connectDevice, Device, State

REMOTE_ADDRESS = "SA1_XTD9_MONO/MOTOR/X"
```

Device are typically connected to only once during the initialisation, using `karabo.middlelayer.connectDevice()`:

```
def __init__(self, configuration):
    super(MonitorRemote, self).__init__(configuration)
    self.remoteDevice = None

async def onInitialization(self):
    self.state = State.INIT
    self.status = "Waiting for external device"
    self.remoteDevice = await connectDevice(REMOTE_ADDRESS)
    self.status = "Connection established"
    self.state = State.STOPPED
```

This function keeps the connection open until explicitly closing it. For a more local and temporary usage, `karabo.middlelayer.getDevice()`, can be used:

```
with (await getDevice(REMOTE_ADDRESS)) as remote_device:
    print(remote_device.property)
```

Note: The `async with` statement is supported from **Karabo 2.13.0** onwards.

2.1.2 Continuous Monitoring

You now have a connection to a remote device! You may start awaiting its updates by defining a slot and using the waitUntilNew function

```
from karabo.middlelayer import connectDevice, State, waitUntilNew
...

@Slot(displayedName="Start",
      description="Start monitoring the remote device",
      allowedStates={State.OFF})
async def start(self):
    self.state = State.ON
    while True:
        await waitUntilNew(self.remoteDevice.remoteValue)
        print(self.remoteDevice.remoteValue)
```

By awaiting the `waitUnitNew()` coroutine, a non-blocking wait for the updated value of the property is executed before proceeding to the print statement.

Note: It may happen that the remote device gets reinitialized, e.g. the underlying device of the proxy is gone, such as after a server restart. The proxy will automatically switch the state property to **State.UNKNOWN** once the device is gone and reestablish all connections when it comes back.

2.2 Grow stronger: Several proxies in a device

Now that a device can be remotely monitored, and the connection kept alive, let's see how to connect to several devices at once, and then control them.

In this example, we will build upon the previous chapter and initialise several connections with three *remote motor devices*, get their positions, and set them to a specific position.

The concepts of *gather*, *background* are introduced here.

2.2.1 Multiple Connection Handling

In order to handle several devices, we must make a few changes to the watchdog and reconnection coroutines.

Let us define three motors we want to monitor and control:

By using `asyncio.gather()` and `karabo.middlelayer.background()`, we simultaneously execute all the tasks in *devices_to_connect* and await their outcomes.

2.2.2 Monitoring Multiple Sources

Monitoring multiple resources is done very much the same way as monitoring a single one, passing a list of devices as a starred expression:

```
async def monitorPosition(self):
    while True:
        positions_list = [dev.position for dev in self.devices]
        await waitUntilNew(*positions_list)

        motorPos1 = self.devices[0].position
        motorPos2 = self.devices[1].position
        motorPos3 = self.devices[2].position
```

2.2.3 Controlling Multiple Sources

Setting properties of a device is done directly by assigning the property a value, for instance:

```
self.remoteMotor.targetPosition = 42
```

This guarantees to set the property. It is possible, however, to do a blocking wait, using `setWait()`:

```
await setWait(device, targetPosition=42)
```

It may be desirable to do so, when the parameter needs to be set before further action should be taken. In this example, setting the desired target position is done with setWait such that we proceed to moving the motor *only after* the device has acknowledged the new target position.

As with properties, functions are directly called. To move the motor to the aforementioned position, await the `move()` function:

```
await self.remoteMotor.move()
```

Once the parameters are set, `karabo.middlelayer.background()` can be used to run the task:

```
background(self.remoteMotor.move())
```

This will create a `KaraboFuture` object of which the status can easily be tracked or cancelled.

As with reconnections, expending this methodology to cover several devices is done using `gather()`:

```
async def moveSeveral(self, positions):
    futures = []

    for device, position in zip(self.devices, positions):
        await setWait(device, targetPosition=position)
        futures.append(device.move())

    await gather(*futures)
```

2.2.4 Exception Handling with Multiple Sources

A problem that now arises is handling exception should one of the motors develop an unexpected behaviour or, more commonly, a user cancelling the task. Cancellation raises an `asyncio.CancelledError`, thus extending the above function with a try-except:

```
async def moveSeveral(self, positions):
    futures = []
    for device, position in zip(self.devices, positions):
        await setWait(device, targetPosition=position)
        futures.append(device.move())
    try:
        await gather(*futures)
        await self.guardian_yield(self.devices)
    except CancelledError:
        toCancel = [device.stop() for device in self.devices
                    if device.state == State.MOVING]
        await gather(*toCancel)
```

Note: Note that the appropriate policy to adopt is left to the device developer.

The try-except introduces a `guardian_yield()` function. This is required in order to remain within the `try` statement, such that any cancellation happening whilst executing the futures, will be caught by the `except`.

The suggested solution for the guardian yield is to wait until all the device go from their busy state (*State.MOVING*) to their idle (*State.ON*) as follows:

```
async def guardian_yield(self, devices):
    await waitUntil(lambda: all(dev.state == State.ON for dev in devices))
```

2.3 Device Nodes

Next to use `connectDevice()` or `getDevice()` it is possible in middle layer device development to use a `DeviceNode`.

Note: DeviceNodes are **MANDATORY** properties and can only be set before instantiation, e.g. are **INITONLY**!

In contrast to a regular `Access.MANDATORY` definition, a `DeviceNode` also does not accept an empty string. Defining a remote device is done as follows:

```
motor1Proxy = DeviceNode(displayedName="RemoteDevice",
                         description="Remote motor 1")

motor2Proxy = DeviceNode(displayedName="RemoteDevice",
                         description="Remote motor 2")

motor3Proxy = DeviceNode(displayedName="RemoteDevice",
                         description="Remote motor 3")
```

However, a `DeviceNode` must connect within a certain timeout (default is 2 seconds) to the configured remote device, otherwise the device holding a `DeviceNode` shuts itself down.

Accessing, setting, and waiting for updates from `DeviceNode` is similar to what was done previously.

Monitoring the position of these three motors is done so:

```
positions_list = [dev.position for dev in [motor1Proxy,
                                           motor2Proxy,
                                           motor3Proxy]]
await waitUntilNew(*positions_list)
```

Setting a parameter is a simple assignment but calling a function needs to be **awaited**:

```
self.motor1Proxy.targetPosition = 42
await self.motor1Proxy.move()
```

Doing so for several devices is done using `gather()`:

```
async def moveSeveral(self, positions):
    futures = []

    for device, position in zip(self.devices, positions):
        await setWait(device, targetPosition=position)
        futures.append(device.move())

    await gather(*futures)
```

Function and parameter calls are now exactly as they were when using `getDevice()` or `connectDevice()`, but now details regarding the connection to remote devices are left to the middle layer API.

The `DeviceNode` holds a proxy to a remote device. During initialization, the DeviceNodes try to establish a proxy connection. Once connected, the deviceId of the remote device can be represented, but internally the proxy is retrieved. For this, a connection must be established. If this cannot be guaranteed within a certain time, the device holding the device nodes will shut itself down.

A timeout parameter of up to 5 seconds can be provided.

```
motor1Proxy = DeviceNode(displayedName="RemoteDevice",
                         description="Remote motor 1",
                         timeout=4.5)
```

The following devices implement the functionalities described above in a working environment, and can be considered reference implementations:

- **fastValve is a middle layer device interfacing with several remote devices**
  through the use of `DeviceNode`

CHAPTER

THREE

BREAD'N'BUTTER FEATURES

The following section describes the Bread'N'Butter features of the middlelayer api.

3.1 Configurables

Devices can have Nodes with properties. Node structures are, as devices, created with `Configurable` classes.

It might be necessary to access the top-level device within a *Node* and this can be straightforwardly done with *Configurable.get_root()* as shown below.

```
from karabo.middlelayer import (
    Configurable, Device, Double, MetricPrefix, Node, Slot, State, Unit,
    background)


class FilterNode(Configurable):
    filterPosition = Double(
        displayedName="Filter Position",
        unitSymbol=Unit.METER,
        metricPrefixSymbol=MetricPrefix.MILLI,
        absoluteError=0.01)

    @Slot(allowedStates=[State.ON])
    async def move(self):
        # access main device via get_root to notify a state change
        root = self.get_root()
        root.state = State.MOVING
        # Move the filter position in a background task
        background(self._move_filter())

    async def _move_filter(self):
        try:
            pass
            # do something
        finally:
            root = self.get_root()
            root.state = State.ON


class GlobalDevice(Device):
    node = Node(FilterNode, displayedName="Filter")
```

Furthermore, it is possible to do a bulkset of a *Hash* container on a *Configurable* and only consider the changes. Note that, if a single value cannot be validated, the whole *Hash* is not set. In all cases the timestamp information is preserved, either from the value of the Hash element or an eventual *KaraboValue* that is coming from a *Proxy*, e.g. via *connectDevice*.

```
from karabo.middlelayer import (
    Configurable, Device, Double, Hash, Int32, MetricPrefix, Node,
    QuantityValue, Slot, State, Unit, minutesAgo)


class FilterNode(Configurable):
    filterPosition = Double(
        displayedName="Filter Position",
        unitSymbol=Unit.METER,
        metricPrefixSymbol=MetricPrefix.MILLI,
        absoluteError=0.01)


class MotorDevice(Device):

    channel = Int32(
        defaultValue=0)

    targetPosition = Double(
        defaultValue=0.0)

    velocity = Double(
        defaultValue=0.0,
        minInc=0.0,
        maxInc=10)

    node = Node(FilterNode, displayedName="Filter")


    def updateFromExternal(self):
        # get external data, everytime the timestamp is considered, either
        # via Hash attributes or `KaraboValue`
        h = Hash("targetPosition", 4.2,
                 "velocity", QuantityValue(4.2, timestamp=minutesAgo(2)),
                 "node.filterPosition", 2)
        # 1. All values will be applied only if they changed
        self.set(h, only_changes=True)

        # 2. All values will be applied
        self.set(h)

        h = Hash("targetPosition", 4.2,
                 "velocity", QuantityValue(100.2, timestamp=minutesAgo(2)),
                 "node.filterPosition", 12)
        # NO VALUE will be applied as velocity is outside limits!
        self.set(h, only_changes=True)
```

3.2 Util functions

3.2.1 Remove Quantity Values from core functions

A good practice is to work with fixed expectations about units and timestamps and strip them away in critical math operations, especially when relying on the external libray *numpy*. Decorate a function to remove *KaraboValue* input with *removeQuantity*.

This function works as well with async declarations and can be used as

```
@removeQuantity
def calculate(x, y):
    assert not isinstance(x, KaraboValue)
    assert not isinstance(y, KaraboValue)
    return x, y


@removeQuantity
def calculate(boolean_x, boolean_x):
    assert not isinstance(x, KaraboValue)
    assert not isinstance(y, KaraboValue)
    # Identity comparison possible!
    return x is y
```

Note: This decorator does not cast to base units! Retrieving the magnitude of the *KaraboValue* allows for identity comparison.

3.2.2 Maximum and minimum

Typically, values in devices and proxies are a *KaraboValue* which comes with a timestamp and a unit. However, not every simple mathmatical operation with KaraboValues is supported by common packages. In order to take the *maximum* and *minimum* of an iterable please have a look:

```
from karabo.middlelayer import maximum, minimum, QuantityValue, minutesAgo

t1 = minutesAgo(1)
t2 = minutesAgo(10)

a = QuantityValue(3, "m", timestamp=t1)
b = QuantityValue(1000, "mm", timestamp=t2)
m = maximum([a, b])
assert m == 3 * unit.meter
m = minimum([a, b])
assert m == 1000 * millimeter
# Timestamp is the newest one -> 1
assert m.timestamp == t1
```

3.2.3 Get and Set Property

Sometimes in scripting it is very convenient to get properties from devices or proxies in a `getattr` fashion, especially with noded structures. Use *get_property* as equivalent to python's builtin `getattr`. Similarly, the *set_property* function is the equivalent to pythons `setattr`.

```
from karabo.middlelayer import get_property, set_property

    prop = get_property(proxy, "node.subnode.property")
    # This is equivalent to accessing
    prop = proxy.node.subnode.property

    # Set a value
    set_property(proxy, "node.subnode.property", 5)
    proxy.node.subnode.property = 5
```

3.3 Alarms in Devices

Each device is equipped with a `globalAlarmCondition` property which can set if an alarm should be highlighted. All alarm levels are provided by an enum of `AlarmCondition` according to the severity of the alarm level. In the middlelayer API the `globalAlarmCondition` does **NOT** require an acknowledging of the alarm setting.

There are different ways of alarm monitoring depending on the environment. The device developer can write an own alarm monitor as shown below observing the difference of temperatures:

```
from karabo.middlelayer import (
    AlarmCondition, background, connectDevice, Device, waitUntilNew)

class AlarmDevice(Device):

    async def onInitialization(self):
        self.temp_upstream = await connectDevice("REMOTE_UPSTREAM")
        self.temp_downstream = await connectDevice("REMOTE_DOWNSTREAM")
        background(self.monitor())

    async def monitor(self):
        while True:
            await waitUntilNew(self.temp_upstream.value,
                               self.temp_downstream.value)
            diff = abs(self.temp_upstream.value - self.temp_downstream.value)
            if diff > 5:
                level = AlarmCondition.WARN
            elif diff > 10:
                level = AlarmCondition.ALARM
            else:
                level = AlarmCondition.NONE

            # Only set the new value if there is a difference!
            if level != self.globalAlarmCondition:
                self.globalAlarmCondition = level
```

Note: The default value of the `globalAlarmCondition` property is `AlarmCondition.NONE`. Other simple settings

are `AlarmCondition.WARN`, `AlarmCondition.ALARM` and `AlarmCondition.INTERLOCK`.

The alarm monitoring can also be automatically configured within a property with different steps and information.

```
from karabo.middlelayer import (
    AlarmCondition, background, connectDevice, Device, Double, waitUntilNew)

class AlarmDevice(Device):

    temperatureDiff = Double(
        displayedName="Temperature Difference",
        accessMode=AccessMode.READONLY,
        defaultValue=0.0,
        warnLow=-5.0,
        alarmInfo_warnLow="A temperature warnLow",
        alarmNeedsAck_warnLow=True,
        warnHigh=5.0,
        alarmInfo_warnHigh="A temperature warnHigh",
        alarmNeedsAck_warnHigh=True,
        alarmLow=-10.0,
        alarmInfo_alarmLow="A temperature alarmLow",
        alarmNeedsAck_alarmLow=True,
        alarmHigh=10.0,
        alarmInfo_alarmHigh="Alarm: The temperature is critical",
        alarmNeedsAck_alarmHigh=True)


    async def onInitialization(self):
        self.temp_upstream = await connectDevice("REMOTE_UPSTREAM")
        self.temp_downstream = await connectDevice("REMOTE_DOWNSTREAM")
        background(self.monitor())

    async def monitor(self):
        while True:
            await waitUntilNew(self.temp_upstream.value,
                               self.temp_downstream.value)
            diff = self.temp_upstream.value - self.temp_downstream.value
            self.temperatureDiff = diff
```

3.4 Table Element (VectorHash)

Known as `TABLE_ELEMENT` in the bound API, `VectorHash` allows users to specify custom entries, in the form of a table, that are, programmatically, available later in the form of an iterable.

Like other karabo properties, `VectorHash` is initialized by displayedName, description, defaultValue, and accessMode. As well, it has a *rows* field that describes what each row in the table contains.

This *rows* field expects a class that inherits from `Configurable`.

```
class RowSchema(Configurable):
    deviceId = String(
            displayedName="DeviceId",
```

(continues on next page)

(continued from previous page)

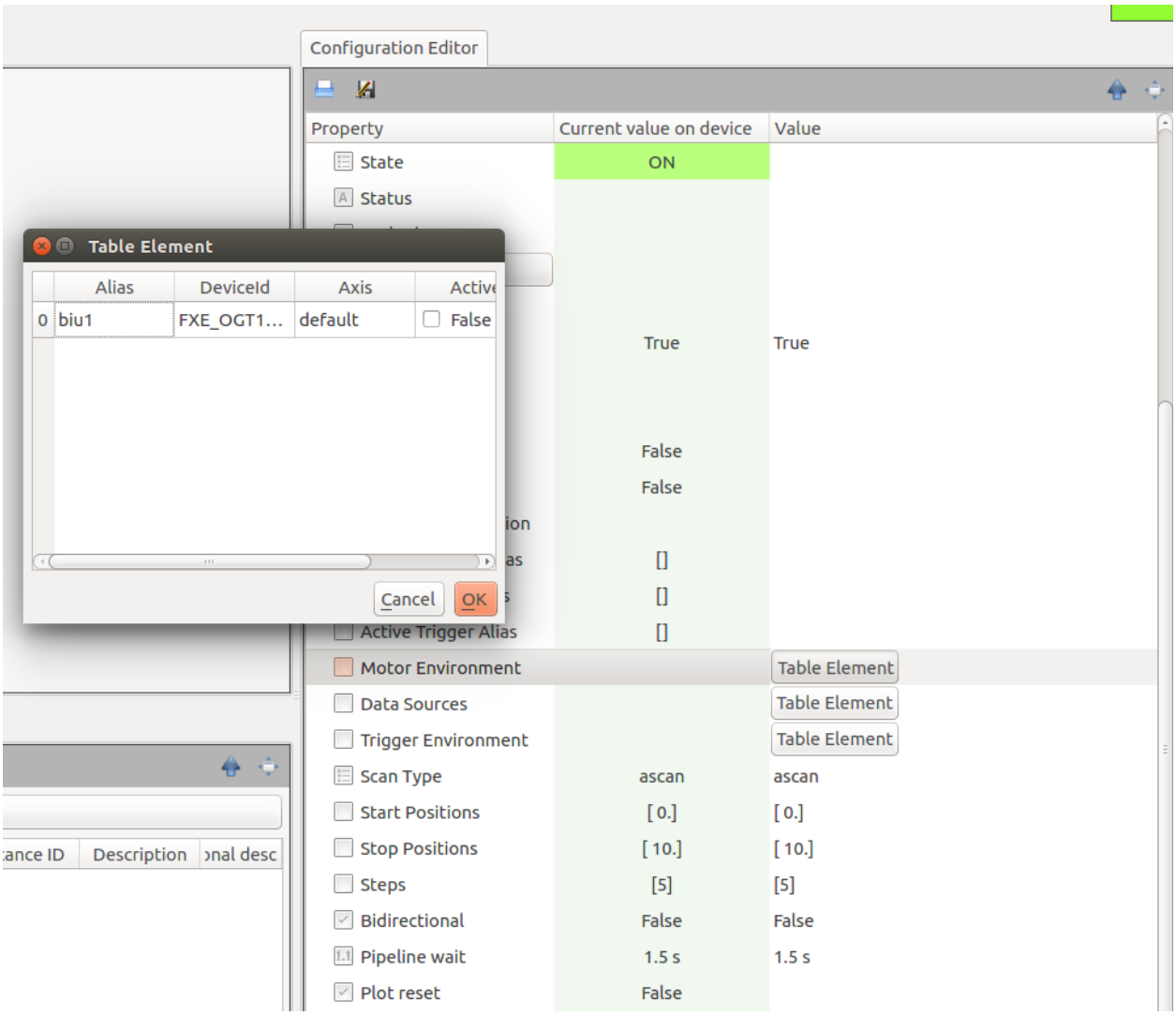
```
            defaultValue="")
    instanceCount = Int(
            displayedName="Count")
```

This class can have as many parameters as desired, and these will be represented as columns in the table.

With `RowSchema`, the definition of the VectorHash is as follows:

```
class MyMLDevice(Device):
    userConfig = VectorHash(
                    rows=RowSchema,
                    displayedName="Hot Initialisation",
                    defaultValue=[],
                    minSize=1, maxSize=4)
```

The user will now be presented with an editable table:



Note that it is possible to provide the user with predefined entries, such as default values or reading a configuration file, by providing a populated array in the `defaultValue` option. The `minSize` and `maxSize` arguments can limit the

table's size if needed.

```
class MyMLDevice(Device):
    userConfig = VectorHash(
                    rows=RowSchema,
                    displayedName="Default Value Example",
                    defaultValue=[Hash("deviceId", "XHQ_EG_DG/MOTOR/1",
                                       "instanceCount", 1),
                                  Hash("deviceId", "XHQ_EG_DG/MOTOR/2",
                                       "instanceCount", 2)],
                    minSize=1, maxSize=4)
```

The Table Element can be a bit customized with the **displayType** attribute. Specifying for example

```
class RowSchema(Configurable):
    deviceId = String(
            displayedName="DeviceId",
            defaultValue="")
    state = String(
            defaultValue="UNKNOWN",
            displayType="State",
            displayedName="State")
```

will have a table schema that describes the state column with a displayType `State`. The graphical user interface can then color the column according to state color.

With **Karabo 2.14.X**, the table offers more customization, e.g.

```
class RowSchema(Configurable):
    progress = Double(
            displayedName="ProgressBar",
            displayType="TableProgressBar"
            defaultValue=0.0,
            minInc=0.0,
            maxInc=100.0)

    stringColor = String(
            defaultValue="anystring",
            displayType="TableColor|default=white&xfel=orange&desy=blue",
            displayedName="stringColor")

    numberColor = Int32(
            displayedName="Number Color",
            displayType="TableColor|0=red&1=orange&2=blue"
            defaultValue=0)
```

Hence, for different displayTypes more options are available.

- A progressbar can be declared with `TableProgressBar` on a number descriptor.
- Background coloring can be provided for strings and numbers with the `TableColor` displayType. The coloring is then appended in an URI scheme (separator *&*) which is append to the displayType after |. Declaration of a default background brush can be set with the *default* setting.

```
class RowSchema(Configurable):
    progress = Bool(
```

(continues on next page)

(continued from previous page)

```
            displayedName="Bool Button",
            displayType="TableBoolButton",
            defaultValue=True)
```

For **read only** table element a button can be declared via `TableBoolButton`. The button is enabled depending on the boolean setting.

Clicking the button will send a Hash to the device via the slot **requestAction**.

The hash contains keys with data:

```
action: TableButton
path: the property key
table: the table data
```

The table data itself is a Hash with:

```
data = Hash(
    "rowData", h,
    "row", row,
    "column", column,
    "header", header)
```

The rowData is a Hash of the row of the table button. The elements *row* and *column* provide the indexes of the button and the header the column string.

Another option (**since Karabo 2.15.X**) for a button can be the `TableStringButton`. Besides access level considerations this button is always enabled to provide an action on a **deviceScene** or **url**.

```
class RowSchema(Configurable):

    description = String(
            displayedName="Description",
            defaultValue="")

    view = String(
            displayedName="View",
            displayType="TableStringButton",
            defaultValue="")
```

The value for both protocols are strings and an example to set a table

```
device_scene = "deviceScene|deviceId=YOURDEVICE&name=YOURSCENENAME"
open_url = "url|www.xfel.eu"

self.table = [Hash("description", "Important device", "view", device_scene),
              Hash("description", "Important device doc", "view", open_url)]
```

Vector handling in tables is significantly increased in **Karabo 2.16.X**. Specifiy a button with `TableVectorButton` to launch a list edit dialog via a button in the table element.

```
class RowSchema(Configurable):

    devices = VectorString(
```

(continues on next page)

(continued from previous page)

```
                displayedName="View",
                displayType="TableVectorButton",
                defaultValue=[])
```

Once the VectorHash has been populated, it is possible to iterate through its rows, which are themselves internally stored as a `TableValue`, which itself encapsulates a *numpy* array. From **Karabo 2.14.0** onwards it is possible to convert the *np.array* value to a list of Hashes with

```
table = self.userConfig.to_hashlist()
```

Moreover, iterating over the encapsulated numpy array can be done like

```
@Slot(displayedName="Do something with table")
async def doSomethingTable(self):
    # This loops over the array (.value)
    for row in self.userConfig.value:
        # do something ..., e.g. check the first column
        first_column = row[0]
```

If an action is required on VectorHash update, e.g. a row is added or removed, then the VectorHash should be defined within a decorator:

```
@VectorHash(rows=RowSchema,
            displayedName="Hot Initialisation",
            defaultValue=[])
async def tableUpdate(self, updatedTable):
    self.userConfig = updatedTable
    # This loops over the array (.value)
    for row in updatedTable.value:
        # do something ...
```

3.5 Overwrite Properties

Classes in Karabo may have default properties. A type of motors may have a default speed, a camera may have a default frame rate, and so forth.

Let's say a base class for a motor has a default max speed of 60 rpms:

```
class MotorTemplate(Configurable):
    maxrpm = Int32(
                displayedName="Max Rotation Per Minutes",
                accessMode=AccessMode.READONLY,
                allowedStates={State.ON},
                unitSymbol=Unit.NUMBER)
    maxrpm = 60

    ...[much more business logic]...
```

All instances of that MotorTemplate will have a fixed maximum rpm 60, that can be seen when the state is State.ON, and is read only.

We now would like to create a custom motor, for a slightly different usage, where users can set this maximum, according to their needs, but all other parameters and functions remain the same.

It is possible to do so by inheriting from `MotorTemplate`, and using the `karabo.middlelayer.Overwrite` element:

```
class CustomMotor(MotorTemplate):
    maxrpm = Overwrite(
                minExc=1,
                accessMode=AccessMode.RECONFIGURABLE,
                allowedStates={State.OFF, State.INIT})
```

Note that only the required fields are modified. Others, such as `displayedName` will retain their original values.

Using `Overwrite` allows inheritance whilst replacing pre-existing parameters, keeping the namespace clean, and avoiding confusion between the local and inherited scope, thus providing a simpler codebase.

3.6 Schema injection

A parameter injection is a modification of the class of an object. It is used to add new parameters to an instantiated device or to update the attributes of already existing parameters. An example for the latter is the change of the size of an array depending on a user specified Karabo parameter.

The following code shows an example for the injection of a string and a node into the class of a device:

```
from karabo.middlelayer import Device

class MyDevice(Device):

    async def onInitialization(self):
        # should it be needed to test that the node is there
        self.my_node = None

    async def inject_something(self):
        # inject a new property into our personal class:
        self.__class__.injected_string = String()
        self.__class__.my_node = Node(MyNode, displayedName="position")
        await self.publishInjectedParameters()

        # use the property as any other property:
        self.injected_string = "whatever"
        # the test that the node is there is superfluous here
        if self.my_node is not None:
            self.my_node.reached = False

class MyNode(Configurable):
    reached = Bool(
        displayedName="On position",
        description="On position flag",
        defaultValue=True,
        accessMode=AccessMode.RECONFIGURABLE
    )
```

Note that calling **:Python:`inject_something`** again resets the values of properties to their defaults.

Middlelayer class based injection differs strongly from C++ and bound api parameter injection, and the following points should be remembered:

- classes can only be injected into the top layer of the empty class and, consequently, of the schema rendition

- the order of injection defines the order in schema rendition
- classes injected can be simple (Double, Bool, etc.) or complex (Node, an entire class hierarchies, etc.)
- later modification of injected class structure is not seen in the schema. Modification can only be achieved by overwriting the top level assignment of the class and calling `publishInjectedParameters()`
- injected classes are not affected by later calls to `publishInjectedParameters()` used to inject other classes
- deleted (del) injected classes are removed from the schema by calling `publishInjectedParameters()`

In the above example, new properties are added to the top level class of the device. Next, we consider the case where we want to update a property inside a child of the top level class.

:Python:`Overwrite` is used to update the attributes of existing Karabo property descriptors. Similar to the description in Chapter *overwrite* for the creation of a class, the mechanism can be used in schema injection.

```
class Configurator(Device):

    beckhoffComs = String(
        displayedName="BeckhoffComs in Topic",
        options=[])

    @Slot("Find BeckhoffComs")
    async def findBeckhoffComs(self):
        # Get a list of beckhoffCom deviceIds
        options = await self.get_beckhoff_coms()
        self.__class__.beckhoffComs = Overwrite(options=options)
        await self.publishInjectedParameters()
```

Karabo Devices are instances of a device class (classId). Hence, all instances of the same class on the same device server share the same **static** schema and a modification to any class object of the schema is propagated to all device children. However, as described above, it is possible to modify the top-layer device class. **This leads to, that a change in a noded structure requires a full reconstruction (using a new class) and injection into the device's top level class under the same property key.**

Below is a **MotorDevice** with two different motor axes, *rotary* and *linear* represented in a *Node*. In the following, we would like to change on runtime the *minInc* and *maxInc* attribute of the *targetPosition* property. However, **Schema injection for runtime attributes in mostly not worth it and should be avoided**. Nevertheless, this example shows the technical possiblity.

```
from karabo.middlelayer import (
    Configurable, Device, Double, isSet, Node, String, Slot, VectorDouble)


def get_axis_schema(key, limits=None):

    class AxisSchema(Configurable):

        node_key = key

        @Slot()
        async def updateLimits(self):
            await self.get_root().updateAxisLimits(
                self.node_key, self.targetLimits.value)

        @VectorDouble(
```

(continues on next page)

(continued from previous page)

```
            defaultValue=None,
            minSize=2, maxSize=2,
            displayedName="Target Limits")
        async def targetLimits(self, value):
            if not isSet(value):
                self.targetLimits = value
                return
            # Setter function always called in initialization
            self.targetLimits = value

        targetPosition = Double(
            displayedName="Value",
            minInc=limits[0] if limits is not None else None,
            maxInc=limits[1] if limits is not None else None)

    return AxisSchema


class MotorDevice(Device):

    # Node's take classes to build up
    rotary = Node(get_axis_schema("rotary", None))
    linear = Node(get_axis_schema("linear", [0, 90]))

    async def updateAxisLimits(self, key, limits):
        # 1. Get the previous configuration from the node under `key`
        h = self.configurationAsHash()[key]
        # 2. Create a new configurable schema for `key` with `limits`
        conf_schema = get_axis_schema(key, limits)
        # 3. Set the new node on `key` and inject with previous
        # configuration `h`
        setattr(self.__class__, key, Node(conf_schema))
        await self.publishInjectedParameters(key, h)
```

The factory function **:Python:`get_limit_schema`** provides each *Node* with a *Configurable* class. During the creation, the *minInc* and *maxInc* attributes can be assigned to the *targetPosition* property. Here, the class itself has a *Slot* that propagates to the *MotorDevice* to inject a new *Axis* under the same *key* with different limits. During a runtime schema injection via the *Slot updateLimits*, we again create a **new** updated *Configurable* class for the Node - and - to make sure that the configuration of the *MotorDevice* is preserved, the existing configuration is passed to the **:Python:`publishInjectedParameters`** method. During the initialization, all eventual setters are called as usual.

Slots are decorating functions. If you want to add a Slot, or change the function it is bound to (decorating), the following will do the trick:

```
async def very_private(self):
    self.log.INFO("This very private function is now exposed!!")

@Slot("Inject a slot")
async def inject_slot(self):
    # Inject a new slot in our schema
    self.__class__.injectedSlot = Slot(displayedName="Injected Slot")
    self.__class__.injectedSlot.__call__(type(self).very_private)
    await self.publishInjectedParameters()
```

Note: The key to that slot will not be *very_private* but instead *injectedSlot* So yes, cool that we can change the behaviour of a slot on the fly by changing the function the slot calls, but the key won't reflect that.

If you do change the functions that are called, do put in a log message.

Warning: Consider instead injecting a node with a proper Slot definition.

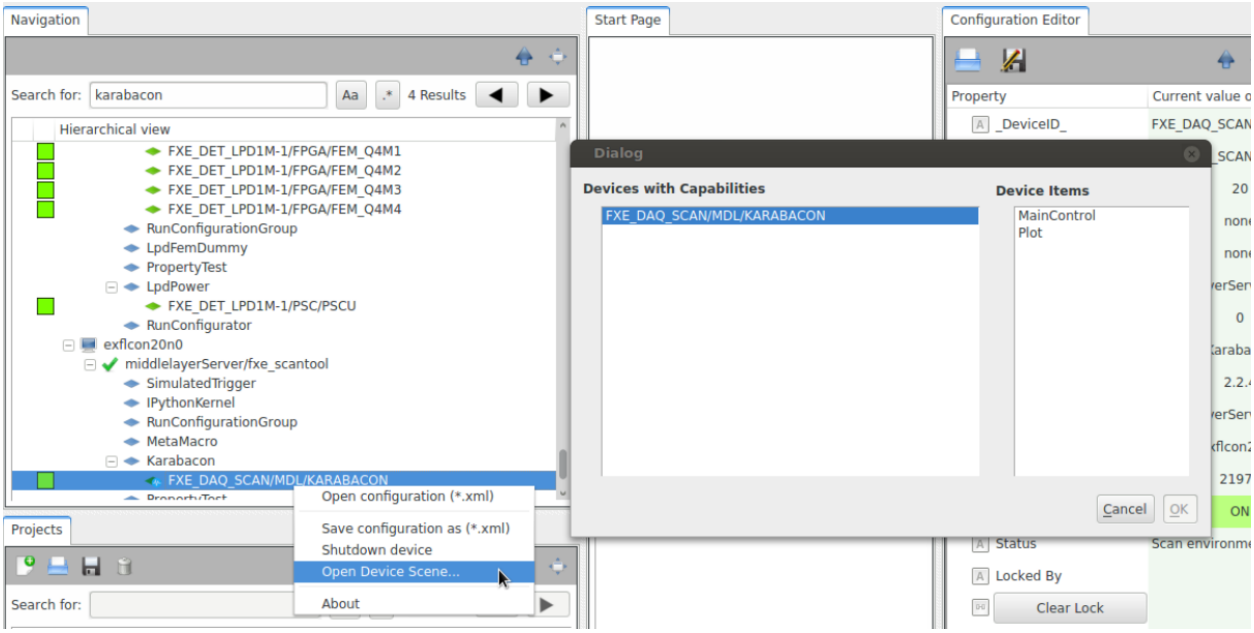
Injected Properties and the DAQ need some ground rules in order to record these properties correctly.

In order for the DAQ to record injected properties, the DAQ needs to request the updated schema again, using the Run Controller's `applyConfiguration()` slot.

This can be prone to operator errors, and therefore it is recommended that only properties injected at instantiation to be recorded.

3.7 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.

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 project.

In your device, add a read-only *VectorString* property called *availableScenes*, 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.middlelayer import AccessMode, DaqPolicy, VectoString, slot
from .scenes import get_scene


availableScenes = VectorString(
    displayedName="Available Scenes",
    displayType="Scenes",
    accessMode=AccessMode.READONLY,
    defaultValue=["overview"],
    daqPolicy=DaqPolicy.OMIT)

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

    return Hash('type', 'deviceScene',
                'origin', self.deviceId,
                'payload', payload)
```

Note: Note that we use here *slot*, and not *Slot()*. These are two different functions. *slot* provides framework-level slots, whereas *Slot* are device-level.

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

```
from karabo.middlelayer import AccessMode, DaqPolicy, VectoString, slot
import .scenes

availableScenes = VectorString(
    displayedName="Available Scenes",
    displayType="Scenes",
    accessMode=AccessMode.READONLY,
    defaultValue=["overview", "controls"],
    daqPolicy=DaqPolicy.OMIT)

@slot
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.deviceId))

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

    return Hash('type', 'deviceScene',
                'origin', self.deviceId,
                'payload', payload)
```

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

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
)
```

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',
        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)
```

Note: *remoteDeviceId* is merely the deviceId, here. If you have a proxy, you may want to rethink the arguments to *overview* and pass it *self* or the proxy object. Then you can find out exactly what scenes are available there, e.g.:

target = 'controls' if 'controls' in px.availableScenes else 'scene' keys=['{}.availableScenes'.format(px.deviceId)], target=target,

---

GainCurveScan: provides a single default scene

Karabacon: provides several scenes

KEP21: definition of the scene protocol

CHAPTER

FOUR

ADVANCED FEATURES

The following section describes the advanced features of the middlelayer api.

4.1 Complete Futures

Synchronizing to single karabo-future completion within a set of karabo-futures, or completion of all karabo-futures of the set is supported by *firstCompleted*, *firstException*, and *allCompleted*. As the set may contain only a single karabo-future it should not be surprising that all functions have the same argument and return signatures. Necessary arguments are the karabo-futures as position or keyword arguments and optional two keyword arguments (timeout and not cancel_pending) to modify function behaviour. The functions return three dictionaries (done, pending, and error) specifying the status of the karabo-futures when the function returns.

The first example shows a timeout wait for connections to be established to devices specified in a clients list of arbitrary length. Note that all exceptions are already caught in the function *allCompleted*.

```
async def setup(self):
    self.state = State.STARTING
    self.devices = []

    done, pending, error = await allCompleted(
        *[connectDevice(deviceId) for deviceId in self.clients],
        cancel_pending=False,
        timeout=10)

    if pending:
        self.logger.warning('Missing clients {}'.format(
              ', '.join([self.clients[k] for k in pending])))
        for k, v in pending.items():
            v.cancel()

    elif error:
        self.logger.warning('Error creating client proxy {}'.format(
              ', '.join([self.clients[k] for k in error])))
        except_entry = [(k, v.__class__.__name__)
                        for k, v in error.items() if v is not None]
        pending_entry = [k for k, v in error.items() if v is None]

    else:
        for k, v in done.items():
            self.devices.append(v)
```

(continues on next page)

(continued from previous page)

```
        return

    self.state = State.UNKNOWN


from collection import ChainMap

async def setup(self):
    self.state = State.STARTING

    fut = {deviceId: connectDevice(deviceId) for deviceId in self.clients}
    fut["timeout"] = 5
    devices, *fails = await allCompleted(**fut, cancel_pending=True)
    chain = ChainMap(*fails)
    if chain:
        status = (f"Not all proxies could be connected {chain.keys()}.")
        self.state = State.UNKNOWN
    else:
        self.devices = list(devices.values())
```

The second example waits indefinitely for left, right and back motor device nodes to reach new target positions.

```
@Slot(displayedName="Move",
       description="Move to absolute position specified by targetPosition.",
       allowedStates={State.STOPPED})
async def move(self):
    left, right, back = virtualToActual(
        self.Y.targetPosition,
        self.Pitch.targetPosition,
        self.Roll.targetPosition,
        self.Length,
        self.Width)

    self.left.targetPosition = left
    self.right.targetPosition = right
    self.back.targetPosition = back

    await allCompleted(moveL=self.left.move(),
                       moveR=self.right.move(),
                       moveB=self.back.move())
```

The following points should be remembered when using the support functions

- Functions wait indefinitely for their completion criteria unless the *timeout* keyword argument is set with the required timeout seconds, when the function will return (a *TimeoutError* is not thrown).
- The returned done, pending and error dictionaries contain k,v pairs, where the key is the enumeration number (e.g. 0, 1, 2 and 3 in example 1) when the karabo-futures are specified as positional arguements, or as user specified values (e.g. “moveL”, “moveR” and “moveB” in example 2) when specified as keyword arguments.
- The order of karabo-futures in done, pending and error returned dictionaries **maintains the order** of the karabo-futures of the calling arguments.
- By default functions cancel any pending karabo-futures (*cancel_pending*) and append the corresponding k,v (with v = None) entry into error, before returning.

- Karabo-futures which raise an exception have their k,v (v = Exception) entries returned in error. Example 1 shows how to build lists of exception and cancelled Karabo-futures in error.

4.2 Async Timer

There are different ways in the middlelayer to continous or repeated checks and procedures. Since **Karabo 2.16.X** the **AsyncTimer** can be used for repeating tasks.

A striking possibility is to bunch device status updates. Not only does the middlelayer bunch updates but also the Karabo Gui Server device is throttling device updates. In the code snippet below, two examples of status throttling are depicted utilizing the **AsyncTimer**.

```
from karabo.middlelayer import AsyncTimer, Device

STATUS_THROTTLE = 0.5
STATUS_THROTTLE_MAX = 2


class StackedStatus(Device):
    """This is a device that has a lot of status updates.

    The `status` property is in the base device and commonly used to provide
    information to the operator what is happening.

    In this example the status updates are concatenated and send out after
    a maximum snooze time of the timer `STATUS_THROTTLE_MAX`.

    The timer is a single shot timer.
    """

    async def onInitialization(self):
        self.stacked_status = []
        # Single shot timer example
        self.status_timer = AsyncTimer(
            self._timer_callback, timeout=STATUS_THROTTLE,
            flush_interval=STATUS_THROTTLE_MAX, single_shot=True)

    def post_status_update(self, status):
        """Cache a status and start the async timer"""
        self.stacked_status.append(status)
        # Start the timer, it will postpone by another `STATUS_THROTTLE`
        # if started already.
        self.status_timer.start()

    async def _timer_callback(self):
        self.status = "\n".join(self.stacked_status)
        self.stacked_status = []
        self.update()

    async def onDestruction(self):
        self.status_timer.stop()
```

(continues on next page)

(continued from previous page)

```
class QueueStatus(Device):
    """This is a device that has a lot of status updates.

    In this example, a status list is continously emptied with a status
    throttle time of `STATUS_THROTTLE`
    """

    async def onInitialization(self):
        self.status_queue = []
        self.status_timer = AsyncTimer(
            self._timer_callback, timeout=STATUS_THROTTLE)
        # Start the timer to continously check the queue.
        self.status_timer.start()

    def queue_status_update(self, status):
        """Queue a status update for the device"""
        self.status_queue.append(status)

    async def _timer_callback(self):
        if self.status_queue:
            self.status = self.status_queue.pop(0)
            # Potential check for status changes before setting
            self.update()

    async def onDestruction(self):
        self.status_timer.stop()
```

Note: All *AsyncTimers* must be stopped before destruction of the device. A typical method is utilizing *onDestruction*!

4.3 Pipelining Channels

Fast or big data in Karabo is typically shared using Pipeline Channel connections. This section explores how to share data in such fashion.

The data is sent on output channels and received on input channels. An output channel can send data to several input channels on several devices, and likewise an input channel can receive data from many outputs.

4.3.1 Output Channels

Firstly, import the required classes:

```
from karabo.middlelayer import (
    AccessMode, Configurable, DaqDataType, Double, InputChannel,
    Node, OutputChannel, Type)
```

Then, define an output channel in your device:

```
output = OutputChannel(ChannelNode,
                       displayedName="Output",
                       description="Pipeline Output channel")
```

You'll notice that we referenced **ChannelNode**. This is the schema of our output channel that defines what data we send and permits other devices to manage their expectations.

To define that schema, create a class that inherits from *Configurable*:

```
class DataNode(Configurable):
    daqDataType = DaqDataType.TRAIN

    doubleProperty = Double(
        defaultValue=0.0,
        accessMode=AccessMode.READONLY)

class ChannelNode(Configurable):
    data = Node(DataNode)
```

Notice that this class has a variable *daqDataType* defined. This is to enable the DAQ to triage the data. The type can be of either PULSE or TRAIN resolution and has to be encapsulated in the Node.

Now that the schema is defined, here's how to send data over the output channel:

```
@Slot(displayedName="Send Pipeline Data")
async def sendPipeline(self):
    self.output.schema.data.doubleProperty = 3.5
    await self.output.writeData()
```

Alternatively, we can send a Hash without setting the property on the device:

```
@Slot(displayedName="Send Pipeline Raw Data")
async def sendPipelineRaw(self):
    await self.output.writeRawData(Hash('data.doubleProperty', 3.5))
```

The output channel can notify all the clients with an end of stream message. Typically, this information is used to change the state of the input processing device.

```
@Slot(displayedName="Send EndOfStream")
async def sendEndOfStream(self):
    await self.output.writeEndOfStream()
```

4.3.2 Input Channels

Receiving data from a Pipeline Channel is done by decorating a function with *InputChannel*:

```
@InputChannel(displayedName="Input")
async def input(self, data, meta):
    print("Data", data)
    print("Meta", meta)
```

The metadata contains information about the data, such as the source, whether the data is timestamped, and a timestamp if so.

If the device developer is interested in the bare Hash of the data, one can set the *raw* option to True:

```
@InputChannel(raw=True, displayedName="Input")
async def input(self, data, meta):
    """ Very Important Processing """
```

For image data it is recommended to use the **raw=False** option, as the middlelayer device will automatically assign an NDArray to the ImageData, accessible via:

```
@InputChannel(displayedName="Input")
async def input(self, data, meta):
    image = data.data.image
```

If it is needed to use the `bare` Hash in the case of ImageData, it can be converted to NDArray as:

```
from karabo.middlelayer import get_image_data

@InputChannel(raw=True, displayedName="Input")
async def input(self, data, meta):
    image = get_image_data(data)
```

It is possible to react on the **endOfStream** or the **close** signal from the output channel via:

```
@input.endOfStream
async def input(self, channel):
    # React on the end of stream of `channel`

@input.close
async def input(self, channel):
    # React on the close of stream of `channel`
```

4.3.3 Policies

Different policies can be set at the device level on the behaviour to adopt when data is arriving too fast on the input channel, or the consumer is too slow on the output channel. The various behaviours are:

- queue: put the data in a queue;
- drop: discard the data;
- wait: create a background task that waits until the data can be sent;
- queueDrop: cycle the data when the limit of the queue is hit

The default mode is *drop* for performance reasons.

The policies are the same on input channels if they are too slow for the fed data rate, but in copy mode only:

```
self.input.onSlowness = "drop"
```

4.3.4 Reference Implementation

A reference implementation can be found in pipelineMDL, where both receiving and sending data is shown.

4.4 Pipeline Proxy Example

In the previous section we learned that devices can have *input* and *output* channels. It is possible to access pipeline data with middlelayer proxies as well.

4.4.1 Output Proxies

Consider a remote device that has an *OutputChannel* with an equal schema as was presented in the last section,

```
class DataNode(Configurable):
    daqDataType = DaqDataType.TRAIN

    doubleProperty = Double(
        defaultValue=0.0,
        accessMode=AccessMode.READONLY)

class ChannelNode(Configurable):
    data = Node(DataNode)


output = OutputChannel(
    ChannelNode,
    displayedName="Output",
    description="Pipeline Output channel")
```

Then it is possible to create a middlelayer proxy and connect to the channel with a channel policy **drop** via:

```
from karabo.middlelayer import connectDevice, waitUntilNew

proxy = await connectDevice("device")
# This is a non-blocking connect in a background task
proxy.output.connect()

# Inspect the data
while True:
    value = proxy.output.schema.data.doubleProperty
    if value > 100:
        break
    await waitUntilNew(proxy.output.schema.data.doubleProperty)

# Disconnect if you are not interested anymore, this saves network traffic
# which other channels can use.
proxy.output.disconnect()
```

As can be seen, it is possible to use *waitUntilNew* on the properties in the output proxy. Additionally, handlers can be registered for convenient use, **before** connecting:

```
from karabo.middlelayer import Hash, connectDevice, get_timestamp

proxy = await connectDevice("deviceId")
# Register handlers (# strong reference)

async def data_handler(data, meta):
    # do something with data hash
    assert isinstance(data, Hash)

    # meta is an object containing source and timestamp information
    source = meta.source # string
    # get the timestamp object from timestamp variable
    timestamp = meta.timestamp.timestamp
    # make sure trainId is corrected
    timestamp = get_timestamp(meta.timestamp.timestamp)
    # do something

async def connect_handler(channel):
   """Connect stream handler of channel"""

async def eos_handler(channel):
   """End of stream handler of channel"""

async def close_handler(channel):
   """Close stream handler of channel"""

proxy.output.setDataHandler(data_handler)
proxy.output.setConnectHandler(connect_handler)
proxy.output.setEndOfStreamHandler(eos_handler)
proxy.output.setCloseHandler(close_handler)
proxy.output.connect()
```

Furthermore, from **Karabo 2.16.X** onwards, it is possible to reassign handlers after disconnecting. Using proxies for pipeline data is a very powerful feature and sometimes it is only needed to get a few context-specific pipeline packages. For this purpose, from **Karabo 2.16.X** onwards, the *PipelineContext* can be used.

4.4.2 Pipeline Context

This context represents a specific input channel connection to a karabo device and is not connected automatically, but may be connected using `async with()` or `with()`

```
channel = PipelineContext("deviceId:output")
async with channel:
    # wait for and retrieve exactly one data, metadata pair
    data, meta = await channel.get_data()
    source = meta.source
    timestamp = get_timestamp(meta.timestamp.timestamp)

with channel:
    await channel.get_data()
```

It is possible to ask for the connection status using `is_alive()` and wait for the pipeline connection awaiting `wait_connected()`

```
async with channel:
    if not channel.is_alive():
        await channel.wait_connected()

# Leaving the context, will disconnect
assert not channel.is_alive()
```

However, awaiting a connection is already implicitly done when waiting for pipeline data to arrive in `get_data()`.

4.5 Images

Karabo has the strong capability of sending `ImageData` via network. The middlelayer API provides this possibility as well.

4.5.1 Image Element

The `Image` element is a helper class to provide `ImageData`

Along the raw pixel values it also stores useful metadata like encoding, bit depth or binning and basic transformations like flip, rotation, ROI.

This special hash `Type` contains an `NDArray` element and is constructed:

```
import numpy as np
from karabo.middlelayer import Configurable, Device, EncodingType

class Device(Configurable):

    image = Image(
        data=ImageData(np.zeros(shape=(10, 10), dtype=np.uint64),
                       encoding=EncodingType.GRAY),
        displayedName="Image")
```

Hence, the *Image* element can be initialized with an *ImageData* KaraboValue.

Alternatively, the *Image* element can be initialized by providing *shape* and *dtype* and the *encoding*:

```
image = Image(
    displayedName="Image"
    shape=(2600, 2000),
    dtype=UInt8,
    encoding=EncodingType.GRAY)
```

The *dtype* can be provided with a simple Karabo descriptor or the numpy dtype, e.g. numpy.uint8.

4.5.2 Image Data

The Karabo `ImageData` is supposed to provide an encapsulated NDArray.

This `KaraboValue` can estimate from the input array the associated attributes of the *ImageData*, such as binning, encoding, etc. The minimum requiremnt to initialize is a numpy array with dtype and shape.

```
import numpy as np
from karabo.middlelayer import ImageData

data = ImageData(np.zeros(shape=(10, 10), dtype=np.uint64))
```

Further attributes can be provided as keyword arguments on object creation, also on runtime. The `ImageData` can be set on runtime on an `Image` element. **However, changing attributes on runtime will not alter the Schema information**

```
class ImageData(KaraboValue):
    def __init__(self, value, *args, binning=None, encoding=None,
                 rotation=None, roiOffsets=None, dimScales=None, dimTypes=None,
                 bitsPerPixel=None, flipX=False, flipY=False, **kwargs):
```

- binning [uint64]: Array or list of the binning of the image, e.g. [0, 0]
- encoding [int32]: The encoding of the image, e.g. EncodingType.GRAY (enum)
- rotation [int32]: The rotation of the image, either 0, 90, 180 or 270
- roiOffsets [uin64]: Array or list of the roiOffset, e.g. [0, 0]
- dimScales [str]: Description of the dim scales
- dimTypes [int32]: The dimension types array or list
- bitsPerPixel: The bits per pixel
- flipX: boolean, either *True* or *False*
- flipY: boolean, either *True* or *False*

4.6 Pipeline Device Example: Images and Output Channel Schema Injection

Karabo has the strong capability of sending `ImageData` via network. Since, sometimes the data type of the image is not know, the corresponding schema has to be injected. Please find below a short example how MDL can use image data for pipelining. It shows a trivial example how schema injection may be utilized on runtime of a device for output channels with **setOutputSchema**.

This is available with **Karabo 2.11.0**

```
from asyncio import sleep

import numpy as np
from karabo.middlelayer import (
    AccessMode, Configurable, DaqDataType, Device, Image, Int32, Node,
    OutputChannel, Slot, State, UInt8, UInt32, background)
```

(continues on next page)

(continued from previous page)

```
def channelSchema(dtype):
    """Return the output channel schema for a `dtype`"""

    class DataNode(Configurable):
        daqDataType = DaqDataType.TRAIN

        image = Image(displayedName="Image",
                      dtype=dtype,
                      shape=(800, 600))

    class ChannelNode(Configurable):
        data = Node(DataNode)

    return ChannelNode


class ImageMDL(Device):
    output = OutputChannel(
        channelSchema(UInt32),
        displayedName="Output")

    frequency = Int32(
        displayedName="Frequency",
        defaultValue=2)

    imageSend = UInt32(
        displayedName="Packets Send",
        defaultValue=0,
        accessMode=AccessMode.READONLY)

    def __init__(self, configuration):
        super(ImageMDL, self).__init__(configuration)
        self._dtype = UInt32

    async def onInitialization(self):
        self.state = State.ON
        self._acquiring = False
        background(self._network_action())

    @Slot(allowedStates=[State.ACQUIRING])
    async def stop(self):
        self.state = State.ON
        self._acquiring = False

    @Slot(allowedStates=[State.ON])
    async def start(self):
        self.state = State.ACQUIRING
        self._acquiring = True

    @Slot(displayedName="Reset Counter")
    async def resetCounter(self):
        self.imageSend = 0
```

(continues on next page)

(continued from previous page)

```
    @Slot(displayedName="Send EndOfStream", allowedStates=[State.ON])
    async def writeEndOfStream(self):
        await self.output.writeEndOfStream()

    @Slot(displayedName="Set Image data dtype")
    async def setImageDtype(self):
        """Example method to show how output channel can be changed on
        runtime"""
        dtype = UInt8 if self._dtype == UInt32 else UInt32
        self._dtype = dtype
        schema = channelSchema(dtype)
        # provide key and new schema
        await self.setOutputSchema("output", schema)

    async def _network_action(self):
        while True:
            if self._acquiring:
                output = self.output.schema.data
                # Descriptor classes have `numpy` property
                dtype = self._dtype.numpy
                image_array = np.random.randint(
                    0, 255, size=(800, 600),
                    dtype=dtype)
                output.image = image_array
                self.imageSend = self.imageSend.value + 1
                await self.output.writeData()

            await sleep(1 / self.frequency.value)
```

4.7 Broker Shortcut

Karabo devices are hosted on device servers. In to communicate, messages are send via the broker to either reconfigure other devices or call device slots.

However, it is possible to directly communicate and control devices on the same devices server via **getLocalDevice**.

This is available with **Karabo 2.15.X**

```
from karabo.middlelayer import Device, Int32, String, waitUntil


class Motor(Device):
    """This is a motor that has shared access to a controller talking
    to hardware"""

    controllerId = String()
    channelId = Int32(defaultValue=1)

    async def onInitialization(self):
        """This method is executed on instantiation"""
```

(continues on next page)

(continued from previous page)

```
        def is_online():
            return self.getLocalDevice(self.controllerId.value) is not None

        await waitUntil(lambda: is_online)
        # A bit less readable example with 2 lambdas
        await waitUntil(lambda: lambda: self.getLocalDevice(
            self.controllerId.value) is not None)
        # Strong reference to the controller device
        controller = self.getLocalDevice(self.controllerId.value)
        # Call a function directly on the device object
        values = await controller.read_hardware_values(self.channelId)
```

4.8 Serialization

4.8.1 What is a Hash

The `Hash` is Karabo's container, across all APIs. All data transferred over network or saved to file by Karabo is in this format. There exists two implementations: in C++, used by the C++ and Bound APIs, and the Middlelayer Python implementation. This document covers serialization details of `karabo.middlelayer.Hash`.

There are various ways to create a `Hash`:

```
from karabo.middlelayer import Hash

value = 'a_string'
h = Hash()
h['key'] = value

h = Hash('key', value)

dict_ = {'key': value}
h = Hash(dict_)
```

`Hash` can be considered as a supercharged `OrderedDict` The big difference to normal Python containers is the dot-access method. The `Hash` has a built-in knowledge about it containing itself. Thus, one can access subhashes by `hash['key.subhash']`.

It also allows to store extra metadata, called *attributes* linked to a datum.

Most commonly, you will find as attribute the trainId, telling when was the datum created. These attributes are also key-value pairs stored in a dictionary:

```
h.setAttribute('key', 'tid', 5)
h['key', 'source'] = 'mdl'

# It is possible to access a single attribute at a time:
h.getAttribute('key', 'tid')
5

h['key', 'source']
```

(continues on next page)

(continued from previous page)

```
'mdl'


# Or all at once:
h.getAttributes('key')
{'tid': 5, 'source': 'mdl'}

h['key', ...]
{'tid': 5, 'source': 'mdl'}
```

Note: There are two ways of accessing and setting attributes. One is *setAttribute* and *getAttribute*, made to respect the C++ implementation, the other consists of using multiple keys and ellipses

With this in mind `h['one', 'b']` accesses the *b* attribute, whereas `h['one.b']` accesses the value *b* of the inner hash *one*

Note: the *tid* attribute is used here on purpose: it is a special attribute representing the trainId, and is always an unit64

4.8.2 XML Serialization

The Middlelayer API offers *saveToFile* and *loadFromFile*, which, given a `Hash` and a file name, will store or load the hash to XML:

```
from karabo.middlelayer import Hash as Mash
from karabo.middlelayer import saveToFile as save_mdl, loadFromFile as load_mdl

save_mdl(h, 'mash.xml')
```

This will result in an XML like the following:

```
<root KRB_Artificial="">
    <key KRB_Type="STRING", tid="KRB_UINT64:5" source="KRB_STRING:mdl">a_string</key>
</root>
```

As shown here, the *tid* and *source* are also stored as xml attributes of *key*. The definition of the entry for key specifies the data type (*KRB_Type*) and any attributes. These types (*KRB_**) are specified using the types as defined in the Framework and have the values separated by a colon, and are the same type accross APIs.

The *root* xml node is there as marker to specify that the information is an encoded `Hash`.

Cross-API

As the format of a Hash is well defined, it is also possible to deserialize a Hash from another API:

```
from karabo.bound import Hash as Bash
from karabo.bound import saveToFile as save_bound, loadFromFile as load_bound

value = 'a_string'
bash = Bash('key', value)
bash.setAttribute('key', 'tid', 5)
bash.setAttribute('key', 'source', 'bound')

save_bound(bash, "bash.xml")

loaded = load_mdl("bash.xml")

type(loaded)
karabo.middlelayer_api.hash.Hash

loaded
Hash([('key', 'a_string')])

loaded[key, ...]
{'tid': 5, 'source': 'bound'}
```

Note: These examples are using both Python APIs, but the behaviour is the same with C++, which also provides saveTo and loadFrom files. These examples work from and to any API.

Note: Although the two Python APIs provide identical functionalities with similar names, their implementation differ greatly, as the Bound API uses C++ whilst the Middlelayer is pure Python, and their usage should not be mixed.

Trying to deserialize a Hash from another API does work, but serialization does not!

4.8.3 Binary Serialization

Binary serialization is used to send data over network. The Framework usually does the serialization, and developers needn't think of it.

The same hash will result in a binary object:

```
0x01 0x00 0x00 0x00 0x03 key 0x1c 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x03
tid 0x12 0x00 0x00 0x00 0x05 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x06 source
0x1c 0x00 0x00 0x00 0x03 0x00 0x00 0x00 mdl 0x08 0x00 0x00 0x00 a_string
```

Which is decomposed as follows:

```
0x01 0x00 0x00 0x00                          # header, indicating how many entries in␣
↪hash, here 1
0x03 key                                     # the first byte define the length of the␣
↪key, here of length 3 (k, e, and y), followed by its value
```

(continues on next page)

(continued from previous page)

```
0x1c 0x00 0x00 0x00                                 # the type of the value for `key`, a string
0x02 0x00 0x00 0x00                                 # 2 attributes!
    0x03 tid                                        # the length of the first attribute key,␣
↪followed by its value
    0x12 0x00 0x00 0x00                             # the type of the `tid` attribute, uint64
    0x05 0x00 0x00 0x00 0x00 0x00 0x00 0x00         # tid, with a value of 5
    0x06 source                                     # the length of the second attribute key,␣
↪followed by its value
    0x1c 0x00 0x00 0x00                             # the type of the `source` attribute
    0x03 0x00 0x00 0x00 mdl                         # the length of the value of `source` and␣
↪the value itself †
0x08 0x00 0x00 0x00                                 # the length of the value for `key`
a_string                                            # the value of the string for the `key` key.
```

†: The reason why the length field of the *mdl* value is an uint32, as opposed to the length field for one of the keys, which are uint8, is that it is a value.

Warning: A `Hash` can contain keys of any length. However, the binary serialization only allowd keys up to 255 bytes. An error will be thrown for longer keys.

Cross-API

As with xml, all APIs understand the binary format:

```
from karabo.bound import BinarySerializerHash, Hash as Bash
from karabo.middlelayer import decodeBinary, encodeBinary

value = 'a_string'
bash = Bash('key', value)
bash.setAttribute('key', 'tid', 5)
bash.setAttribute('key', 'source', 'bound')

serializer = BinarySerializerHash.create(Bash('Bin'))
bound_binary = serializer.save(bash)  # Results in the binary explained above

loaded = decodeBinary(bound_binary)

type(loaded)
karabo.middlelayer_api.hash.Hash

loaded
Hash([('key', 'a_string')])

loaded[key, ...]
{'tid': 5, 'source': 'bound'}
```

Going from Middlelayer to Bound would be:

```
mdl_binary = encodeBinary(h)
loaded = serializer.load(mdl_binary)
```

(continues on next page)

(continued from previous page)

```
type(loaded)
karathon.Hash
```

4.8.4 Table Element

In order to be serialized, a `VectorHash` needs to be put within a hash first. If your device has a table called *table* as one of its properties, then it would be serialized as such:

```
h = Hash()
value, attrs = self.table.descriptor.toDataAndAttrs(self.table)
h['table'] = value
h['table', ...] = attrs
```

Then *h* can be serialized.

To restore it:

```
value = h['table']
attrs = h['table', ...]

table = self.table.descriptor.toKaraboValue(value, attrs)
setattr(self, 'table', table)
```

CHAPTER

FIVE

TESTING FEATURES

Testing is an elemental feature of software engineering. For this reason, typically all karabo devices are shipped with a pep8 code style checker. But there is more that can be done from device developer point of view.

5.1 Device Testing

5.1.1 Device Context

This is only available with Karabo Version >= 2.15.X

The `AsyncDeviceContext` is an asynchronous context manager to handle device instances. If can be fairly straightforward used with **:module:`pytest`**.

```
import uuid

import pytest
import pytest_asyncio
from karabo.middlelayer import Device, Slot, String, connectDevice, isSet
from karabo.middlelayer.testing import (
    AsyncDeviceContext, create_device_server, event_loop)


def create_instanceId():
    return f"test-mdl-{uuid.uuid4()}"


class WW(Device):
    name = String()

    def __init__(self, configuration):
        super().__init__(configuration)
        self.destructed = False

    @Slot()
    async def sayMyName(self):
        self.name = "Heisenberg"

    async def onDestruction(self):
        self.destructed = True
```

(continues on next page)

(continued from previous page)

```
@pytest.mark.timeout(30)
@pytest.mark.asyncio
async def test_example_context(event_loop: event_loop):
    # Make sure to create unique instance id's
    deviceId = create_instanceId()
    device = WW({"_deviceId_": deviceId})
    ctx_deviceId = create_instanceId()
    ctx_device = WW({"_deviceId_": ctx_deviceId})

    # Use the device context class to instantiate devices
    async with AsyncDeviceContext(device=device) as ctx:
        devices = ctx.instances
        assert not isSet(device.name)
        assert not isSet(devices["device"].name)
        proxy = await connectDevice(deviceId)
        await proxy.sayMyName()
        assert device.name == "Heisenberg"
        assert proxy.name == "Heisenberg"
        assert devices["device"].name == "Heisenberg"
        assert len(devices) == 1
        # A new device can always be added to the stack for instantiation
        # and shutdown
        await ctx.device_context(new=ctx_device)
        assert len(devices) == 2
        assert ctx_device.destructed is False
        assert device.destructed is False

    # The context destroys all devices on exit
    assert ctx_device.destructed is True
    assert device.destructed is True


@pytest.mark.timeout(30)
@pytest.mark.asyncio
async def test_example_server_context(event_loop: event_loop):
    """Example how to start and create a server"""
    serverId = create_instanceId()
    # The server can be created with a list of device classes
    server = create_device_server(serverId, [WW])
    async with AsyncDeviceContext(server=server) as ctx:
        server_instance = ctx.instances["server"]
        assert server_instance.serverId == serverId
        # The class name appears in the server plugins and can be used to
        # instantiate devices
        assert "WW" in server_instance.plugins


# It is possible to instantiate multiple devices as fixture for all tests

@pytest_asyncio.fixture(scope="module")
@pytest.mark.asyncio
```

(continues on next page)

(continued from previous page)

```
async def deviceTest(event_loop: event_loop):
    local = WW({"_deviceId_": "local"})
    remote = WW({"_deviceId_": "remote"})
    ctx = AsyncDeviceContext(local=local, remote=remote)
    async with ctx:
        yield ctx


@pytest.mark.timeout(30)
@pytest.mark.asyncio
async def another_local_test(deviceTest):
    """Do something with deviceTest"""
    local = deviceTest["local"]
    assert local is not None


@pytest.mark.timeout(30)
@pytest.mark.asyncio
async def another_remote_test(deviceTest):
    """Do something with deviceTest"""
    remote = deviceTest["remote"]
    assert remote is not None
```

CHAPTER

SIX

MIDDLELAYER SERVER

The following section describes the middlelayer api server features.

6.1 Device server: Broadcast - Unicast

In Karabo messages can be of different type with respect to the target, the so-called **slotInstanceId**. Messages can be either targeted to a single device or all devices. If all devices are targeted, we refer to so-called broadcast messages. In the middlelayer API the device server will subscribe for all broadcast messages and distribute incoming broadcasts to the device children. This removes unnecessary messaging overhead and will give a performance boost.

6.2 Device server: Eventloop

Karabo middlelayer uses Python's asyncio, installing a custom event loop. All devices in a device server share the **same event loop** which is run in a single thread. Hence, every blocking call in any device instance results in blocking every other device in the same device server. However, the device developer is free to start threads using background, which starts a thread if used with a function which is not a coroutine. The event loop thread pool executor can have 200 threads.

6.3 Device server: TimeMiXin

The middlelayer device server automatically connects and reconnects to a TimeServer if the **timeServerId** is provided on initialization. On updates of the TimeServer a singleton-like TimeMixin class is updated with a reference timestamp to calculate the trainId's. Each device will always calculate the trainId automatically if the device server was once connected to a TimeServer.

6.4 HeartBeat Tracking

The heartbeat tracking of devices is by default **disabled** in the middlelayer device server. The heartbeat can be configured via:

```
karabo-middlelayerserver track=True
```

6.5 Autostarting Devices

A server can automatically launch any number of devices upon starting. This is done by specifying the `init` flag:

```
karabo-middlelayerserver init=$INIT_ARGS
```

where `$INIT_ARGS` is a JSON string of the following format:

```
INIT_ARGS='{"DeviceInstanceId": {"classId": "MyDevice", "parameter": 2, "otherParameter
↪": "['a', 'b', 'c']"}}'
```

This will launch a device of type `MyDevice` with the id `DeviceInstanceId`, and other parameters. Mandatory parameters must be specified, else the server will start, but not the device. Note how the dictionary is surrounded by single quotes.

Multiple devices can be added:

```
INIT_ARGS='{"Device1": {"classId": "MyDevice", "parameter": 2, "otherParameter": "['a',
↪'b', 'c']"}
           "GENERATOR": {"classId": "DataGenerator", "autostart": "True"}}'

karabo-middlelayerserver init=$INIT_ARGS
```

The deployment shows examples of middlelayer servers starting the projectDB device.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search