Karabo Developer Workshop: Writing a Middlelayer Device

David Hickin
Controls Group

Grenoble, September 27th, 2024

Python `asyncio` as Foundation of the Karabo Middlelayer API

- This tutorial is about writing a Karabo device using the Middlelayer (MDL) API
  - MDL is written in Python,
  - Largely relying on the advanced **`asyncio`** package – main take away:
    - coroutines, declared with **`async def asyncFunction(..):`**
    - to directly execute them, call with **`await asyncFunction(arguments)`**
      - (for experts: or use **`gather, allCompleted, background`**, etc.)
    - Technically, this allows cooperative multi-tasking in a single thread:
      at any **`await`**, other code can be executed before your coroutine continues

- Note that also macros are based on MDL, but hide the async nature
  - “Synchronised coroutines” (a Karabo feature of many Karabo methods)
    - just work without **`await`** if used in a normal method (i.e. *not* in a coroutine),
    - but if used in a coroutine, the **`await`** has to be added
      (**Caveat** when converting a macro to a device!)

Karabo Middlelayer Device Basics - MDL Device Packages

- Karabo MDL device classes exist in Python packages

- To get started you can modify an existing package or create a new one

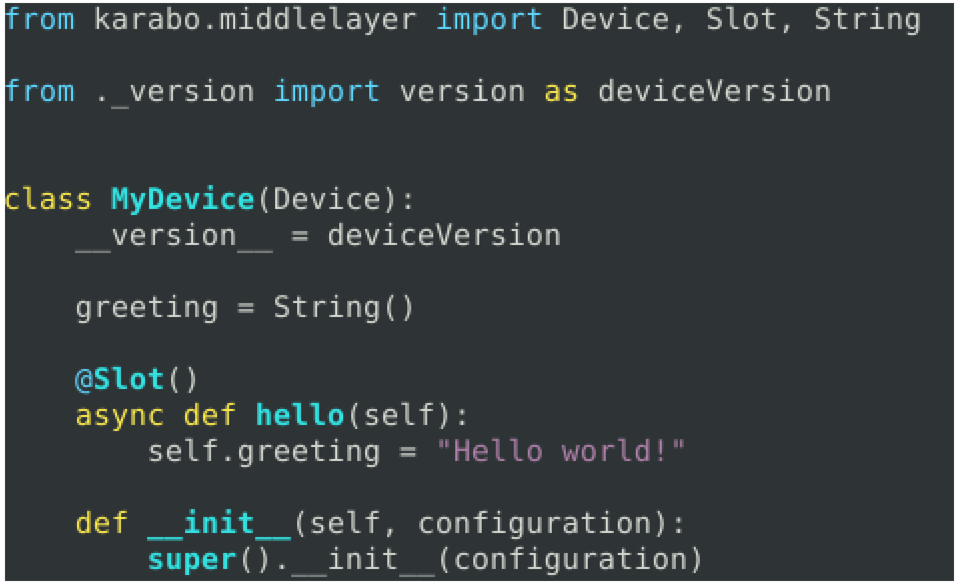
- To create a new MDL device package for a device use “karabo new” with the “middlelayer” argument, e.g.
  - Karabo new myDevice middlelayer

- This will give you an basic package, including:
  - `pyproject.toml` (older projects have setup.py) – used to install device class with pip
  - src/myDevice/MyDevice.py (source including class MyDevice)

- (Today we will modify an existing package for the workshop)

Karabo Middlelayer Device Basics – MDL Device Class

- “karabo new” gives a basic “Hello world” implementation of the device:



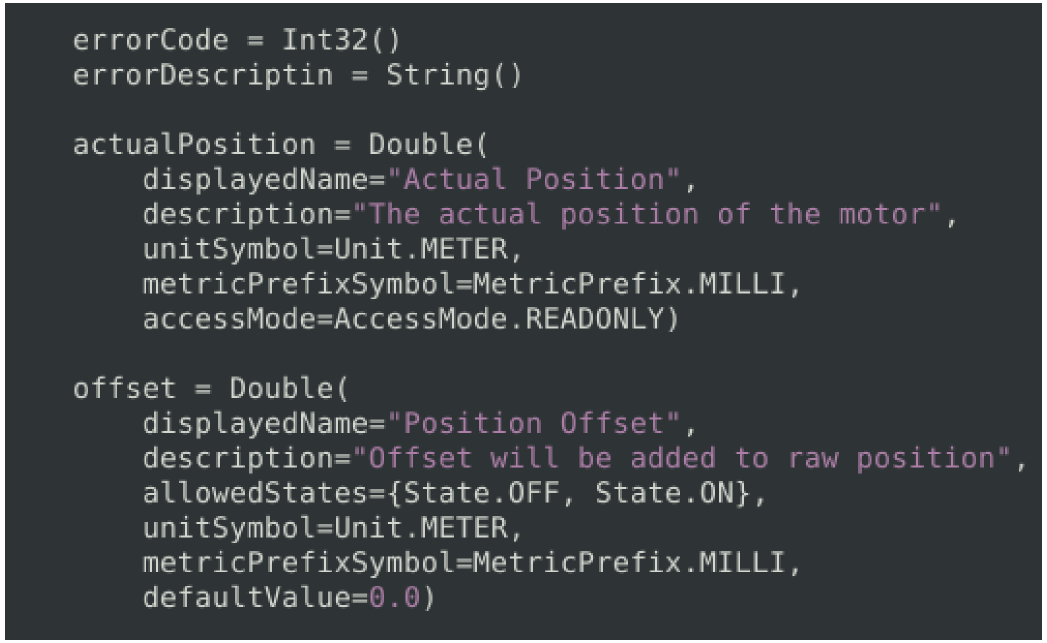
```
from karabo.middlelayer import Device, Slot, String

from ._version import version as deviceVersion


class MyDevice(Device):
    __version__ = deviceVersion

    greeting = String()

    @Slot()
    async def hello(self):
        self.greeting = "Hello world!"

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

- Karabo MDL device classes inherit from **`karabo.middlelayer.Device`**
  - also other MDL classes and function should be imported from **`karabo.middlelayer`**
    - ► Do not mix with **`karabo.bound_api`**! (but bound/MDL devices can live in the same package)

Karabo Middlelayer Device Basics - Properties

- The simplest way to add a property is by adding something like this to the class:
  - **`propertyName = PropertyType(attribute1=aValue, attribute2=anotherValue, …)`**
    - ▶ Property types are e.g. **`String`**, **`Double`**, **`Int64`**, **`VectorBool`**, …
    - ▶ In this tutorial we will touch these attributes: **`displayedName`**, **`description`**, **`defaultValue`**, **`allowedStates`**, **`accessMode`**, **`unitSymbol`**, **`metricPrefixSymbol`**

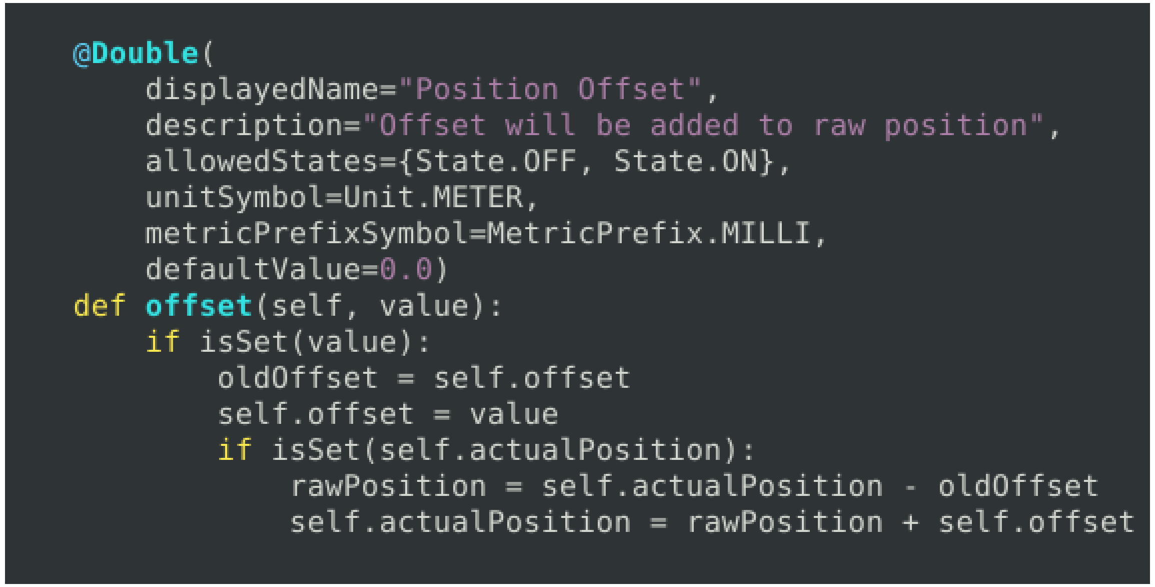
- `Examples:`



```
errorCode = Int32()
errorDescriptin = String()

actualPosition = Double(
    displayedName="Actual Position",
    description="The actual position of the motor",
    unitSymbol=Unit.METER,
    metricPrefixSymbol=MetricPrefix.MILLI,
    accessMode=AccessMode.READONLY)

offset = Double(
    displayedName="Position Offset",
    description="Offset will be added to raw position",
    allowedStates={State.OFF, State.ON},
    unitSymbol=Unit.METER,
    metricPrefixSymbol=MetricPrefix.MILLI,
    defaultValue=0.0)
```

Karabo Middlelayer Device Basics - Properties

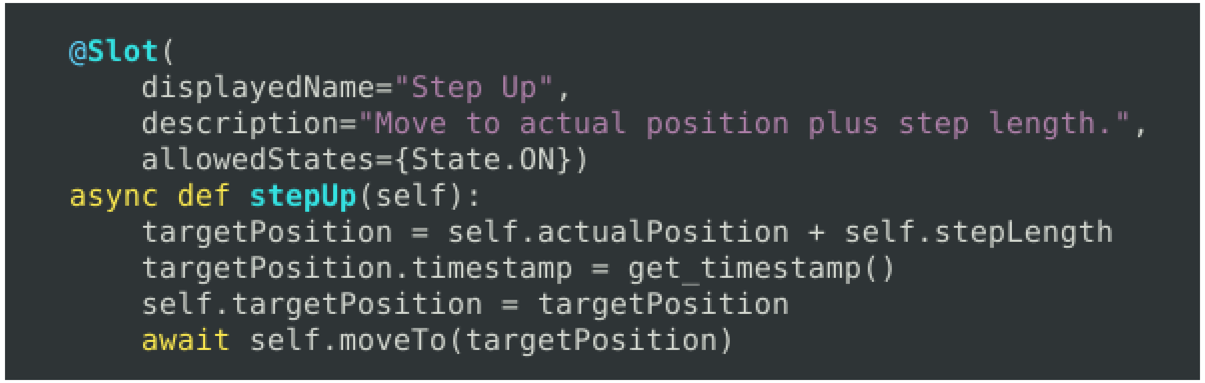
- An alternative syntax using decorators is also possible:



```
@Double(
    displayedName="Position Offset",
    description="Offset will be added to raw position",
    allowedStates={State.OFF, State.ON},
    unitSymbol=Unit.METER,
    metricPrefixSymbol=MetricPrefix.MILLI,
    defaultValue=0.0)
def offset(self, value):
    if isSet(value):
        oldOffset = self.offset
        self.offset = value
        if isSet(self.actualPosition):
            rawPosition = self.actualPosition - oldOffset
            self.actualPosition = rawPosition + self.offset
```

Karabo Middlelayer Device Basics - Slots

- A slot is a coroutine with the `@Slot` decorator – the decorator can take attributes, e.g.



```
@Slot(
    displayedName="Step Up",
    description="Move to actual position plus step length.",
    allowedStates={State.ON})
async def stepUp(self):
    targetPosition = self.actualPosition + self.stepLength
    targetPosition.timestamp = get_timestamp()
    self.targetPosition = targetPosition
    await self.moveTo(targetPosition)
```

Selected Middlelayer Device Members

- `def __init__(self, config)`: If implemented, do not forget `super().__init__(config)`

- `async def onInitialization(self)`:
  - called once the device is up and participating in communication
  - use to connect to hardware or remote devices

- `self.state` property: any of `State.UNKNOWN, State.INIT, ...ON,...OFF,...MOVING,...`
  - governs the state machine restrictions, i.e. what can/cannot be done when

- `self.status`, a `String` property to convey information to the operator via Text Log widget of the GUI

- `self.logger.[info|warn|error](“message”)` leaves message with timestamp etc. in log file

- A note on setting properties:
  - `self.property = value` does not immediately publish the update.
    - ► Done at next `await`
  - But even if `self.property` is identical to `value`, a message is sent!
    - ► Often one may not want that, i.e. better check against equality before setting

Selected Tools for Interaction with Other Devices

- **`dev = await connectDevice(remoteDeviceId)`**:
  - an always up-to-date *proxy* to the remote device
  - to access remote device properties: **`remoteValue = dev.remoteProperty`**
  - to set remote properties: **`dev.remoteProperty = newValue`**
    - ► note: again, message to actually set the property is not sent immediately, but at next **`await`**
  - to call remote slot, e.g. **`await dev.move()`**
  - Lighter variant (not always up-to-date): **`await getDevice(remoteDeviceId)`**

- **`await waitUntilNew(dev.state, dev.propertyA, dev.propertyB, …)`**:
  - wait until any of the given properties has a new value

- **`await waitUntil(function)`**:
  - wait until the given function (e.g. lambda) containing remote device properties returns **`True`**
  - E.g. **`await waitUntil(lambda: dev.state == State.ON)`**

Documentation

- If you want a deeper insight into **`coroutines`** and **`await`**: Read Python **`asyncio`** documentation
  - https://docs.python.org/3.11/library/asyncio-task.html

- Middlelayer how-to documentation:
  - https://rtd.xfel.eu/docs/howtomiddlelayer/en/latest/chap1/intro_device.html
  - https://rtd.xfel.eu/docs/howtomiddlelayer/en/latest/chap2/intro_device_proxies.html

Hands-On

Part 1

Developing a Karabo Device: Prerequisites

- What you need:
  - A running Karabo installation
    - ► Not a production installation!
    - ► Best is a local standalone one as in our VISA virtual machine.
  - A running Karabo GUI
  - A command line terminal with a Linux shell
  - An editor (vscode, PyCharm, emacs, gedit, vim, …)
  - For version control, a git installation is needed
    - ► Best with access to our EuXFEL GitLab https://git.xfel.eu/

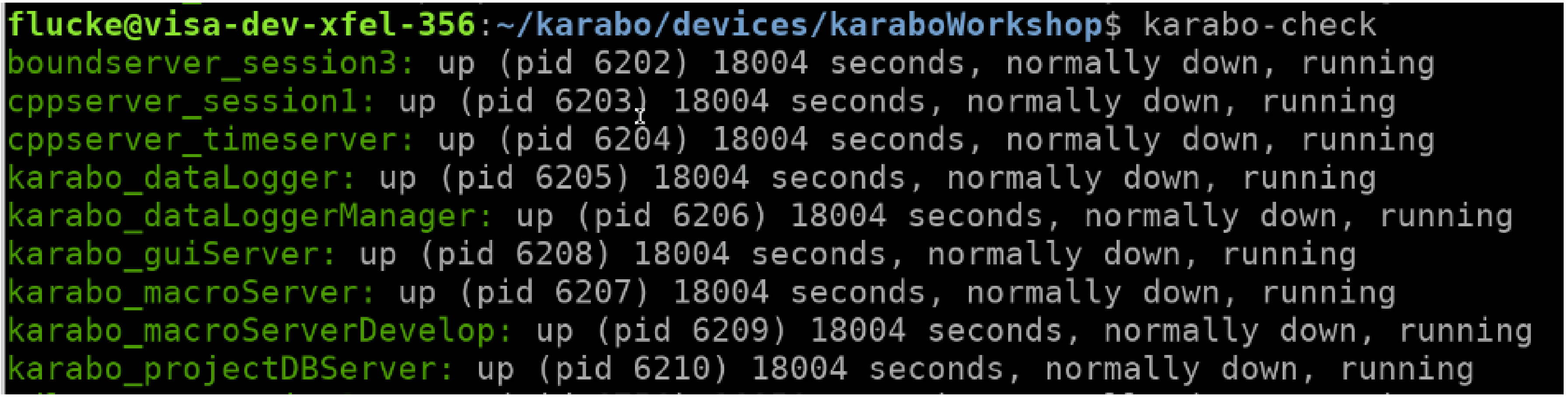
- First steps in terminal (not now!)
  - `source ~/karabo/activate` (in each new shell)
  - `karabo-start` (to start various Karabo servers)
  - Some code to start with
    - ► Create package from scratch *(not now)*: `karabo new thePackageName middlelayer`

Or start from an existing one:

```
karabo -g https://git.xfel.eu develop karaboWorkshop
```

Command Line Tools for an Activated Karabo Environment (Skip!)

- **karabo-check**



```
flucke@visa-dev-xfel-356:~/karabo/devices/karaboWorkshop$ karabo-check
boundserver_session3: up (pid 6202) 18004 seconds, normally down, running
cppserver_session1: up (pid 6203) 18004 seconds, normally down, running
cppserver_timeserver: up (pid 6204) 18004 seconds, normally down, running
karabo_dataLogger: up (pid 6205) 18004 seconds, normally down, running
karabo_dataLoggerManager: up (pid 6206) 18004 seconds, normally down, running
karabo_guiServer: up (pid 6208) 18004 seconds, normally down, running
karabo_macroServer: up (pid 6207) 18004 seconds, normally down, running
karabo_macroServerDevelop: up (pid 6209) 18004 seconds, normally down, running
karabo_projectDBServer: up (pid 6210) 18004 seconds, normally down, running
```

- `karabo-start mdlServer/session2_a` (starts single server)
- `karabo-start` (no argument: acts on all servers)
- `karabo-add-deviceserver mdlServer/session2_c middlelayerserver`
  - ▶ Creates new (middlelayer) server
- Other commands:
  - ▶ `karabo-stop` `(for clean shutdown of all servers)`
  - ▶ `karabo-kill –t <serverId>` `(for clean shutdown and restart of one)`

    e.g. `karabo-kill –t mdlServer/session2_a`
  - ▶ `karabo-kill –k <serverId>` `(to 'kill -9' a hanging process)`

Hands-on in VISA: Start Our First Device

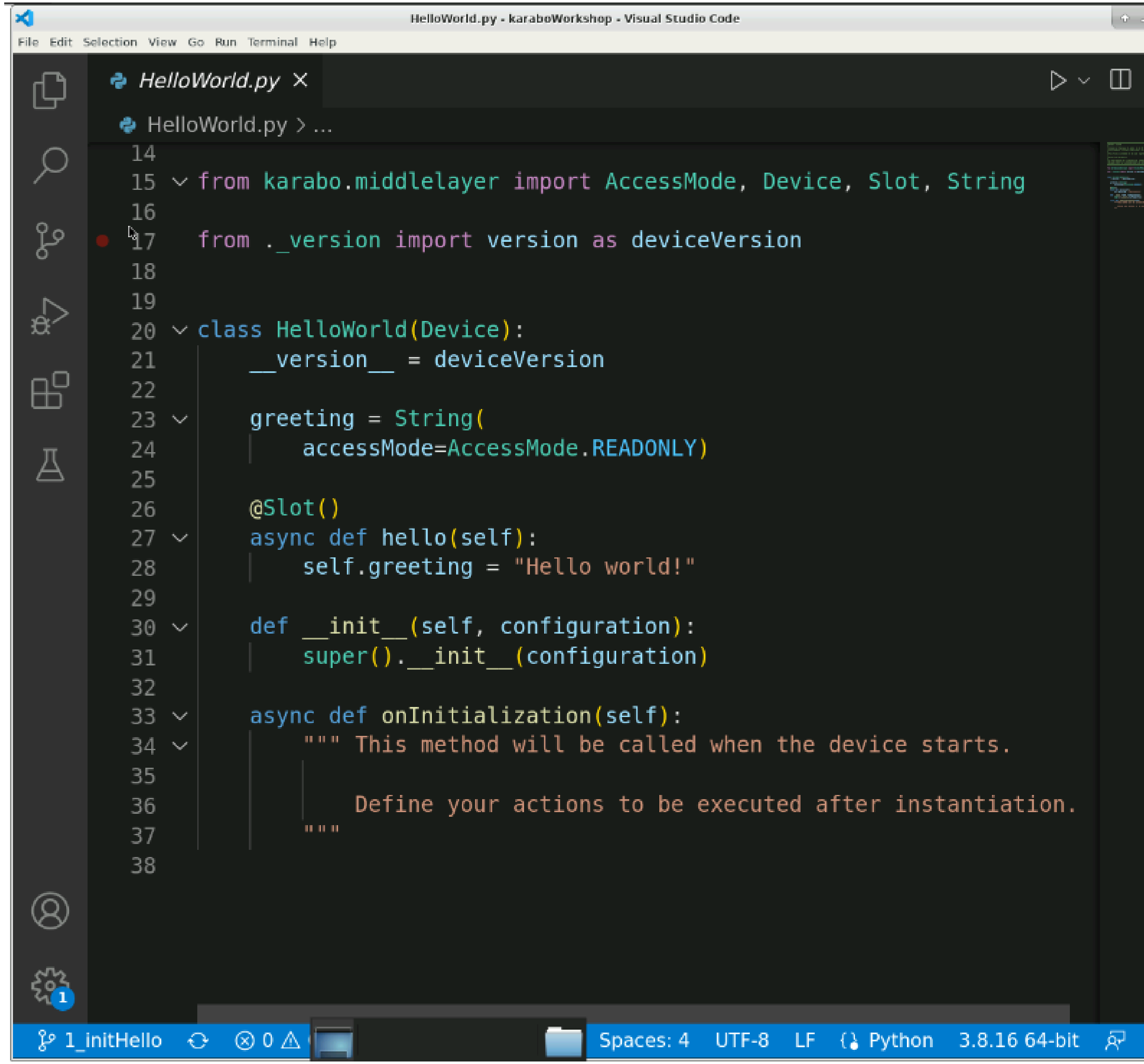
- ***Now*** use GUI:
  - Start GUI from icon, connect to *localhost:44444*
  - Open project **SESSION2** (from database **CAS_INTERNAL**)
  - Start **`KARABO_TEST/MDL/HELLO_WORLD`** from
  - Press execute **`hello`** slot in Configuration editor
    - ► Watch how **`greeting`** property changes

Hands-on: The Device Code

- This is the skeleton – almost as you get it from the templates via **`karabo new helloWorld`** …

- You will now work on your own
  - Extend **`HelloWorld.py`** in three exercises.
  - Then follow more exercises on a **`MotorProcedure.py`**.
  - To get your code changes active, save and shutdown **mdlServer/session2_a**

- Not all code needs to be typed by you:
  - All steps are prepared for you via *tags*
    - ▶ **`git checkout <someTag>`**
    - ▶ if it complaints since you edited changes: first do **`git stash`**
    - ▶ **`git diff <aTag> <nextTag>`**



```python

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

from ._version import version as deviceVersion


class HelloWorld(Device):
    __version__ = deviceVersion

    greeting = String(
        accessMode=AccessMode.READONLY)

    @Slot()
    async def hello(self):
        self.greeting = "Hello world!"

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

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

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

```

Hands-on: Property and Slot with Attributes

- It is good practice to
  - make properties and slots appear in GUI as full, capitalized words,
  - add a description,
  - provide defaults where it makes sense.

- Exercise:
  - Add **displayedName**, **description** attributes to property and slots and **defaultValue** for property

- Simple hands-on (just to warm up):
  - **`git checkout 2_decorate`**
  - See how all but **displayedName** for slot **hello** is achieved
    - ► E.g. **`git diff 1_initHello 2_decorate`**
  - Start device again and try out (Do not forget to restart the server!)
  - Edit **`HelloWorld.py`**
  - Add the missing **`displayedName`** (e.g. “Hello Procedure”) for slot **`hello`**
  - In doubt, **`git diff 2_decorate 2_decorate_done`** shows what to do
  - Try out the device after your changes!

Hands-on: State Handling for Slots

- Karabo devices should be in a well defined *state*
  - **`UNKNOWN`** (the default) means: lost contact to resources, e.g. hardware
  - The device base class defines **`self.state`** variable
    - ► Predefined (long…) list: **`State.ON, State.OFF, State.MOVING, State.ERROR,`** …
  - Depending on its state, actions on the device are allowed or not

- Exercise:
  - Set device to **OFF** (or **ON**) in the beginning
  - Add slots **off** and **on** that switch to the corresponding states **OFF** and **ON**
    - ► But define **allowedStates** such that **off** slot can only be called if in **ON** state and vice versa
    - ► Also **hello** slot should only be callable in **ON** state
  - See how that is achieved for all but the **on** slot
    - ► E.g. **`git diff 2_decorate_done 3_states`**

- Hands-on:
  - **`git checkout 3_states`** (if git complains since you edited: **`git stash`** before)
  - Restart device, try out and add the missing **on** slot (Do not forget to restart the server!)
  - In doubt, **`git diff 3_states 3_states_done`** shows what to do

Hands-on: Reconfigurable Properties with State Handling

- So far, our property **`greeting`** could only be set from inside device code (since **`READONLY`**)

- Exercise:
  - explicitly mark **`greeting`** as **`reconfigurable`** at run time
    (that would have been the MDL default…),
  - but only if in state **OFF**

- Hands-on:
  - See how making it reconfigurable is achieved
    ► E.g. **`git diff 3_states_done 4_reconfig`**
  - **`git checkout 4_reconfig`** (**`git stash`** before?)
  - Try out and add the restriction of the **OFF** state
  - In doubt, **`git diff 4_reconfig 4_reconfig_done`** shows what to do

Hands-On

Part 2

More on States and Slots

- So far, our slots did not do much:
  - Their execution did not take long
  - and therefore their success (or failure) was quickly reported to the GUI (or whoever called them)

- A procedure is different by nature:
  - First do something, then another thing, then wait a bit and finally do a third thing, …
  - If all this is directly programmed into a slot, it would time out
    - ► For **`@MacroSlot`** used in macros, the timeout is essentially swallowed

- ➔ Longer procedures (even like simple motor movement) have this pattern in Karabo:
  - A slot only *triggers* the procedure, i.e.
    - ► *switches* to some "***ING**" state (e.g. **MOVING**, **PROCESSING**, **CHANGING**, **STARTING**, …)
    - ► and *triggers* the procedure, i.e. in MDL puts it into the **`background`**
  - If procedure done, leave "***ING**" state again
    - ► Often to the state in which the slot can again be executed

Hands-on: Simple Motor Procedure

- Exercise:
  - A new **`MotorProcedure`** device in **`~/karabo/devices/karaboWorkshop/src/karaboWorkshop/MotorProcedure.py`**
  - Its slot **moveMotor**
    - ► connects to another device,
    - ► sets its **targetPosition**,
    - ► lets it **move**,
    - ► and waits until motor movement is done (i.e. motor not in **`State.MOVING`** anymore)

- Hands-on:
  - Start device **`KARABO_TEST/MOTOR/X`** from project **SESSION_2**
  - Look at source code and the interplay between slot **`moveMotor`** and method **`motor_procedure`**
  - Add the three missing steps and try out (tip: **`await waitUntil(…)`**)
    - ► You may monitor a bit what goes on with the scene **SteerMotor** (not everything on the scene is already available)
  - In doubt, **`git diff 4_reconfig_done 5_simple_done`** shows what to do

Hands-on: Monitoring Another Device

- A device may want to constantly monitor another device and react on changes

- Exercise:
  - Extend the `MotorProcedure` device:
    - ▶ add a `Double` property **distanceToTarget**
    - ▶ add a coroutine `monitor_task()`
    - ▶ put that into the `background` (in `onInitialization`)
  - `monitor_task` should
    - ▶ connect to the motor device,
    - ▶ whenever its **targetPosition** or **actualPosition** change, assign the difference to **distanceToTarget**

- Hands-on:
  - `git checkout 5_simple_done`
  - Try to implement this (tipp: `await waitUntilNew(..)`)
  - In doubt, `git diff 5_simple_done 6_monitor_done` shows what to do
    - ▶ or just `git checkout 6_monitor_done`, try out and investigate

Hands-on: Extend Motor Procedure to Three Steps

- So far we just moved the motor
  - Could have done using the motor directly.
  - Now let's have more steps in our procedure!

- Exercise:
  - Extend to the **`motor_procedure()`** to
    - ► cache **`actualPosition`** and **`targetVelocity`** of the motor,
    - ► after first movement, **`sleep`** 5 seconds, move back at half speed, reset **`targetVelocity`**
    - ► (Extra: inform operators about what is going on by updating **`self.status`**)

- Hands-on:
  - **`git checkout 6_monitor_done`**
  - Try to implement exercise
  - In doubt, **`git diff 6_monitor_done 7_3steps_done`** shows what to do
    - ► or just **`git checkout 7_3steps_done`**, try out and investigate

Hands-on: Cancel a Procedure

- While a long running procedure executes, you may notice that things go wrong
  - We need something to cancel the procedure!

- Exercise:
  - The **`background`** actually returns a *future* with that one can handle an ongoing procedure
  - Keep track of that in a member variable (e.g. **`self.task`**)
  - Add slot **`cancelMoveMotor`** that
    - ► has the proper **`allowedStates`**
    - ► calls **`cancel()`** of the *future* (and resets the holding the variable)
    - ► resets the state of the **`MotorProcedure`** to **`State.ON`**
    - ► (Extra: inform operators about cancellation by updating **`self.status`**)

- Hands-on:
  - **`git checkout 7_3steps_done`**
  - Try to implement this
  - In doubt, **`git diff 7_3steps_done 8_cancel_done`** shows what to do
    - ► or just **`git checkout 8_cancel_done`**, try out and investigate

Hands-on: Make the Cancel Clean

- Did you notice:
  - When we cancel our procedure while the motor moves, the motor just goes on!
  - If we cancel when moving back at half speed, the **actualVelocity** stays at half speed

- Exercise:
  - Cancelling a future actually injects an **`asyncio.CancelledError`**, so better
    - ► protect the procedure with **try:**,
    - ► use **finally:** to do everything that needs to be cleaned-up (no matter if cancelled or not),
    - ► in **`except CancelledError:`** take care that motor stops
  - Caveat: if cancelled while we sleep, motor cannot be stopped since not moving!

- Hands-on:
  - **`git checkout 8_cancel_done`**
  - Try to implement this
  - In doubt, **`git diff -b 8_cancel_done 9_cancelClean_done`** shows what to do
    - ‘**`-b`**’ ignores changes of whitespace
    - ► or just **`git checkout 9_cancelClean_done`**, try out and investigate

Hands-on: Basic Testing as Good Developer Practice

- A device is something long lived and probably will be developed further
  - How to make sure that a new feature does not break an existing one that you carefully tested?
  - You tested with the current Karabo version (and that of other libraries).
    - ► How to ensure that newer versions do not break your code?

- Exercise:
  - Automated test procedures are needed!
    - ► tests should reside in …/`src/karaboWorkshop/tests`
    - ► We use the **pytest** and the “continuous integration” (CI) of GitLab

- Hands-on:
  - `git checkout 9_cancelClean_done`
  - Have look at …/`tests/test_helloworld.py`
    - ► It is close to what `karabo new` … creates for you
  - `pytest src/karaboWorkshop/tests/`
  - `git checkout 10_withTests_done` and see how …`/tests/test_motorProcedure.py` tests basics of the procedure
    - ► In practice, it is tough to fully test procedurse since interacting with other devices…

Hands-on: Cancellation Still Has Loop Holes

- Did you try to shutdown the motor during the procedure?
  - During the first movement?
  - During the sleep?
  - During the second movement?

- Exercise:
  - Make use of the feature that **`dev.state`** will become **`State.UNKNOWN`** if the device behind proxy **`dev`** shuts down.
  - But since that is also a valid state for a device, check **`isAlive(dev)`** to take care of the device shutdown

- Hands-on:
  - Do on your own now…