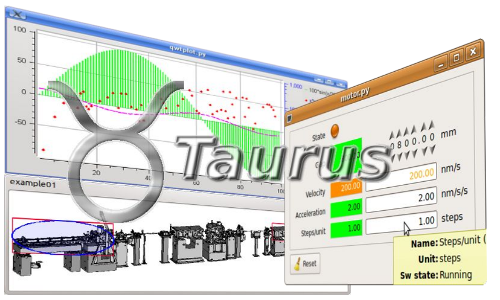
www.cells.es

TAURUS Workshop, 12-13 May 2026



Taurus developer introduction

Oriol Vallcorba on behalf of all Taurus team

Content

Taurus core

Polling/Events

Taurus Schemes

Taurus Extensions

Taurus programmatic GUIs

Custom widgets

Taurus project

Contributing to Taurus

Practical session

Taurus code map

Practical Session format

Resources

Recap

APPLICATIONS, TOOLS & WIDGETS

Dashboards

Synoptic Views

Data Analysis

Code / Scripting

Custom Widgets

...

REUSABLE
Common components and behaviors

FOCUS ON WHAT MATTERS
Applications, not communication details

PROVEN & ADOPTED
Used in many facilities and projects

TAURUS
Open framework for building control system applications

UNIFIED ACCESS
One API to talk to many control systems

EXTENSIBLE
Plugins, custom widgets and extensions

COMMUNITY DRIVEN
Open source, collaborative and welcoming

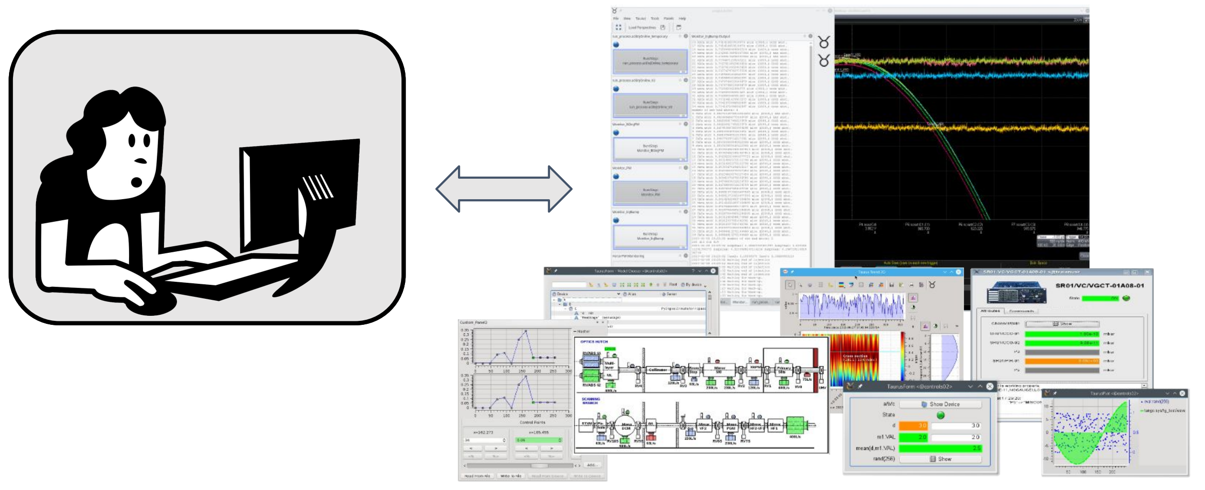
CONTROL SYSTEMS / DATA SOURCES

...

HARDWARE, PLCs, IOCs, ARCHIVERS, FILES, SERVICES, ...

Taurus architecture

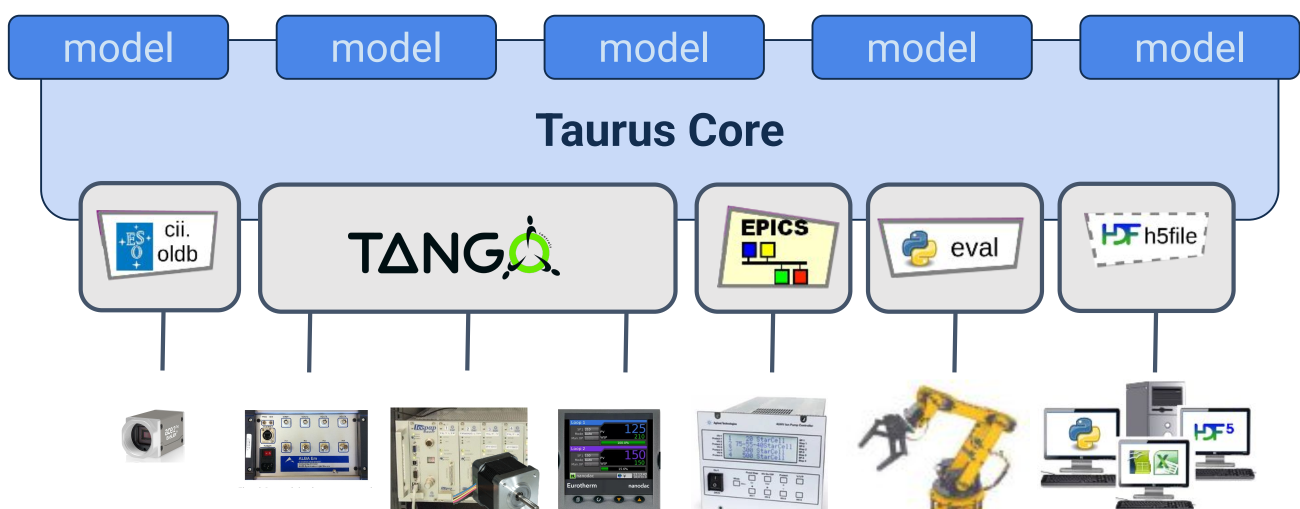
A model-driven abstraction layer for control systems



Taurus GUIs

Taurus Qt Widgets

Everything is a model!



Model Objects
(e.g. Attribute, Device,...)

Schemes

Taurus architecture

- Taurus follows a **model-view-controller** design
- Taurus is **data source agnostic**
- Model objects are **singletons**
- Model names are **URIs** (Uniform Resource Identifiers)
- Models are provided by schemes plugins. Each scheme provides:
  - A model factory for **Authority**, **Device** and **Attribute**
  - Model name validators
- Each type of widget offers a **particular view** on its model(s)
- All functionality is enabled by just **attaching** the model to the widget (i.e. providing its URI)

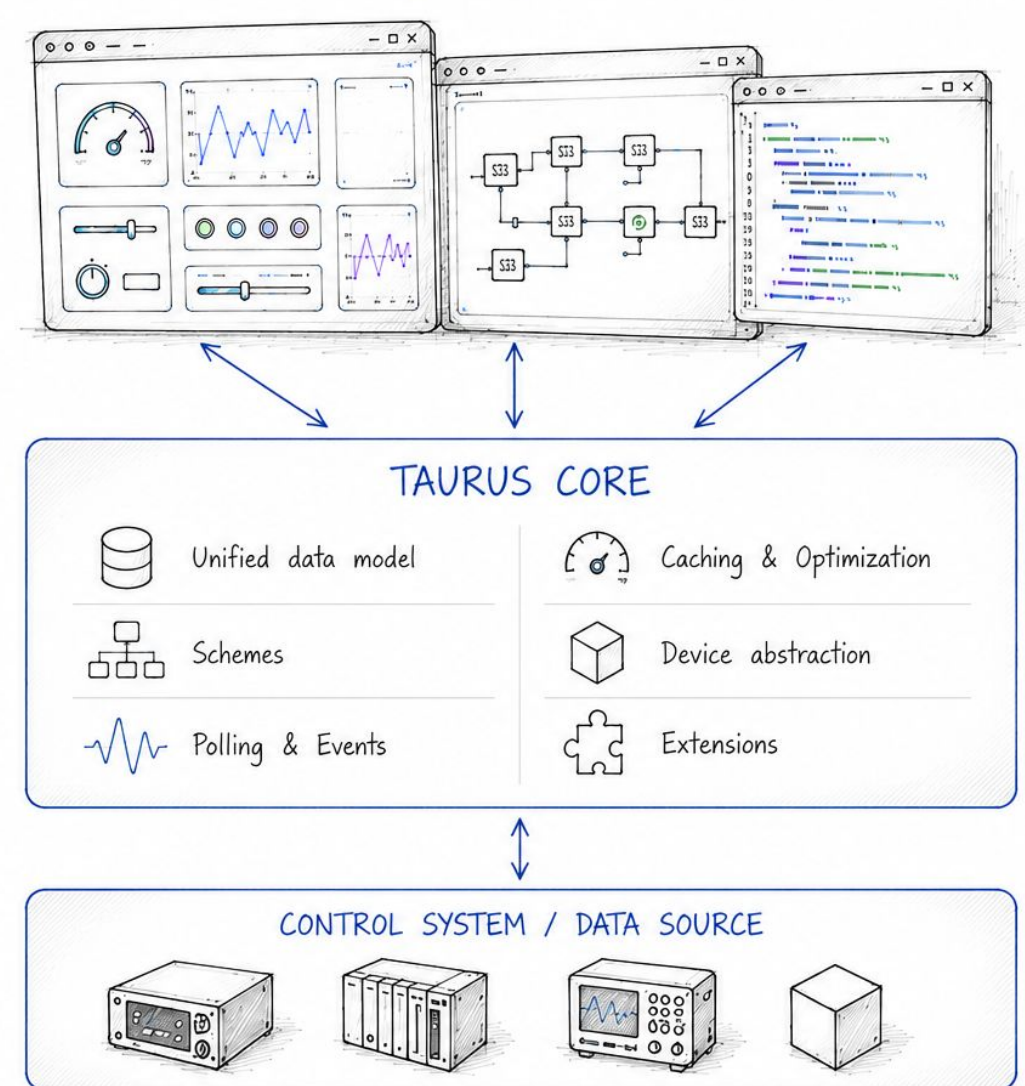
Model names: Uniform Resource Identifier (URIs)

```
scheme:authority/path?query#fragment
```

#	Model name (URI)	Scheme	Model type	Represented source of data/control object
1	*tango://foo:1234*	TANGO	Authority	**Tango** database listening to port *1234* of host *foo*
2	*tango://foo:1234/a/b/c*	TANGO	Device	**Tango** Device *a/b/c* registered in database *foo*
3	*tango:a/b/c/state*	TANGO	Attribute	**Tango** attribute *state* of device #2
4	*tango:a/b/c/d#units*	TANGO	Attribute	**Tango** attribute *d* of device #2 (*units* fragment)
5	*ca:XXX:m1.VAL*	EPICS	Attribute	**EPICS** process variable *XXX:m1.VAL*
6	*eval:({tango:a/b/c/d}+{epics:XXX:m1.VAL})*0.5*	eval	Attribute	**Calculated** average of the values of #4 and #5
7	*eval:rand(256)*	eval	Attribute	Random **generated** array of 256 values
8	*msenv://foo:1234/macroserver/bar/1/ScanDir*	[MSenv]	Attribute	*ScanDir* variable from **Sardana's environment**
9	*h5file:/mydir/myfile.hdf5*	HDF5	Device	File in **HDF5** format saved at */mydir/myfile*
10	*h5file:/mydir/myfile.hdf5:data/energy*	HDF5	Attribute	**HDF5** dataset *energy* of group *data* from file #9
11	*ssheet:myfile.ods:Sheet1.A1*		Attribute	Contents of cell A1 of Sheet1 of *myfile.ods* **spreadsheet**

Taurus Core

Polling/Events
Taurus Schemes
Taurus Extensions



Taurus Core

A model-driven abstraction layer for control systems

Taurus model classes

(core abstraction)

```
TaurusModel
  ├── TaurusAuthority
  ├── TaurusDevice
  └── TaurusAttribute
```

Applications interact with Taurus models, not directly with backend APIs.

```
In [1]: from taurus import Device, Attribute

In [2]: dev = Device("sys/tg_test/1")

In [3]: attr = Attribute("sys/tg_test/1/double_scalar")

In [7]: print(dev.state)
TaurusDevState.Ready

In [5]: print(attr.read())
TangoAttrValue{'rvalue': <Quantity(112.223536,
'kelvin')>, 'wvalue': <Quantity(0.0, 'kelvin')>, 'time':
TimeVal(tv_nsec = 346, tv_sec = 1778053201, tv_usec =
407759), 'quality': <AttrQuality.ATTR_VALID: 0>,
…

In [6]: print(attr.read().rvalue)
112.2235359524402 7 K
```

Taurus Core

Key Taurus Core Behaviors

Attribute cache

`attr.read()`
may return cached value

`attr.read(cache=False)`
forces backend read

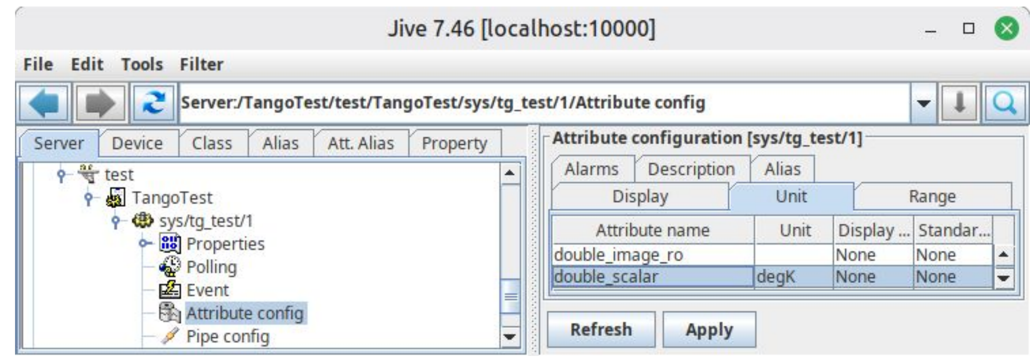
`attr.read(cache=N)`
accept cache if younger than N sec

- Less backend traffic
- Shared values between widgets
- Updated by events/polling

```
In [1]: from taurus import Device, Attribute

In [2]: from datetime import datetime, timedelta

In [3]: attr = Attribute("sys/tg_test/1/long_scalar")

In [4]: value = attr.read()

In [5]: dt = datetime.fromtimestamp(value.time.tv_sec) + timedelta(microsecond
   ...: s=value.time.tv_usec)

In [6]: now = datetime.now()

In [7]: print("Attribute value time:", dt.strftime("%Y-%m-%d %H:%M:%S.%f"))
   ...: print("                Now:", now.strftime("%Y-%m-%d %H:%M:%S.%f"))
Attribute value time: 2026-05-06 10:00:07.055260
                Now: 2026-05-06 10:00:40.262902

In [8]: now-dt
Out[8]: datetime.timedelta(seconds=33, microseconds=207642)

In [9]: datetime.fromtimestamp(attr.read().time.tv_sec)
Out[9]: datetime.datetime(2026, 5, 6, 10, 0, 7)

In [10]: datetime.fromtimestamp(attr.read().time.tv_sec)
Out[10]: datetime.datetime(2026, 5, 6, 10, 0, 7)

In [11]: datetime.fromtimestamp(attr.read(cache=False).time.tv_sec)
Out[11]: datetime.datetime(2026, 5, 6, 10, 1, 30)
```

Taurus Core

Key Taurus Core Behaviors



Working with units

Taurus numeric (float or integer) attributes and their associated properties (such as limits, warning levels, etc.) are `pint.Quantity` objects.

A Quantity is essentially the combination of a **magnitude** and a **unit**.

Features:

- Unit conversions
- Consistent physical values
- Safer scientific applications

```
In [8]: from taurus import Device, Attribute

In [9]: attr = Attribute("sys/tg_test/1/double_scalar")

In [10]: value = attr.read()

In [11]: value.rvalue
Out[11]: <Quantity(-31.1978514, 'kelvin')>

In [12]: value.rvalue.units
Out[12]: <Unit('kelvin')>

In [13]: value.rvalue.to("degC")
Out[13]: <Quantity(-304.347851, 'degree_Celsius')>
```

Taurus Core

Key Taurus Core Behaviors

Object references

- Same model name → same Taurus object
- Centralized event handling
- Shared cache and subscriptions

```
In [26]: a1 =
Attribute("sys/tg_test/1/state")

In [27]: a2 =
Attribute("sys/tg_test/1/state")

In [28]: a1 is a2
Out[28]: True
```

Different from raw backend APIs
(e.g. PyTango usually creates independent proxies)

Default authority / scheme

- Shorter model names
- Configurable environment
- Easier application portability

```
In [17]: from taurus import Authority

In [18]: auth = Authority()

In [19]: auth
Out[19]: TangoAuthority(tango://localhost:10000)
```

Taurus URIs

`scheme:authority/path?query#fragment`

Taurus Core

Key Taurus Core Behaviors

TaurusManager

“central registry and orchestration”

- Scheme discovery
- Object factories
- Model caching
- Event dispatching
- Plugin registration

External plugin registration
https://taurus-scada.org/devel/plugins.html

- Via entry-points
- Define `EXTRA_SCHEME_MODULES` in `tauruscustomsettings`

```
In [20]: from taurus import Manager

In [21]: manager = Manager()

In [22]: manager.default_scheme
Out[22]: 'tango'

In [23]: manager.getPlugins()
Out[23]:
{'tango': taurus.core.tango.tangofactory.TangoFactory,
'tango-nodb': taurus.core.tango.tangofactory.TangoFactory,
'res': taurus.core.resource.resfactory.ResourcesFactory,
'resource': taurus.core.resource.resfactory.ResourcesFactory,
'eval': taurus.core.evaluation.evalfactory.EvaluationFactory,
'evaluation': taurus.core.evaluation.evalfactory.EvaluationFactory,
'ca': taurus.core.epics.epicsfactory.EpicsFactory,
'epics': taurus.core.epics.epicsfactory.EpicsFactory,
'h5file': h5file.h5filefactory.H5fileFactory,
'redis': taurus_redis_scheme.redisfactory.RedisFactory}
```

Taurus dynamically loads schemes, factories, widgets and extensions through plugin registration mechanisms.

Taurus Core

Taurus Listener System (events vs. polling)

Event-driven architecture: You can attach listeners to Taurus models, triggered when a particular event occurs (done automatically when attaching models to widgets)

Native events path

- No continuous polling
- Lower CPU/network usage

Backend
CHANGE event
TaurusAttribute
CHANGE event
TaurusListener
Qt Widget

TaurusPollingTimer
periodic read()
TaurusAttribute
PERIODIC event

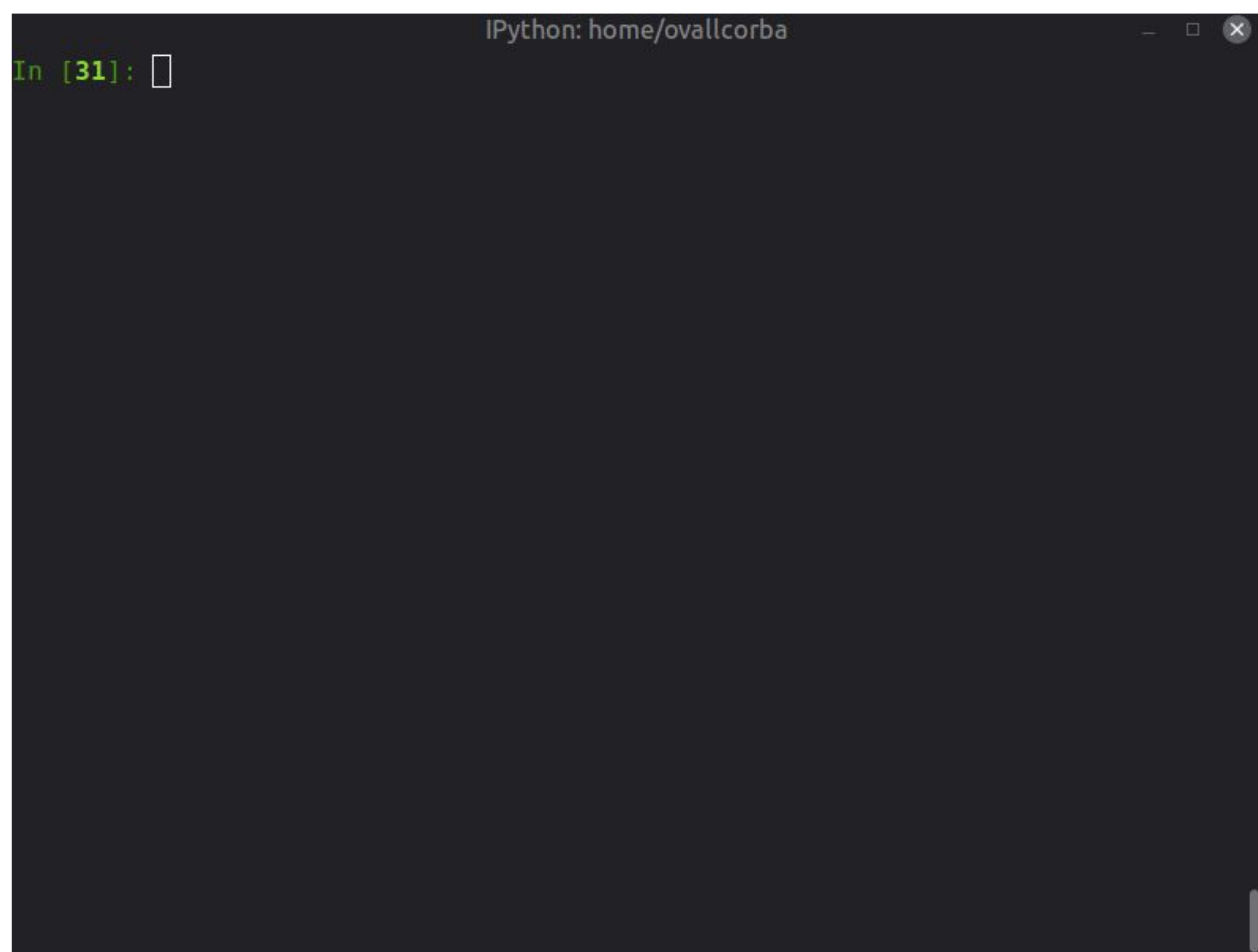
Polling fallback path

- Default polling period: 3s
- One polling thread per process/periodicity
- Shared by all attributes using that timer

Widgets do not need to know if updates come from backend events or Taurus polling.

Taurus Core

Taurus Listener System (events vs. polling)



Attaching a listener to a Taurus model

```
In [39]: from taurus.core.taurusbasetypes import
TaurusEventType

In [40]: TaurusEventType.keys()
Out[40]: ['Change', 'Config', 'Periodic', 'Error']
```

Taurus event types

Taurus Core

Taurus Listener System (events vs. polling)

```
In [32]: attr
Out[32]: TangoAttribute(tango://localhost:10000/sys/tg_test/1/long_scalar)

In [33]: attr.addListener(my_listener)
MainThread     INFO     2026-05-06 12:42:51,805 localhost:10000.sys/tg_test/1.long_scalar: Subscription failed.Activating polling.
Out[33]: True

(TangoAttribute(tango://localhost:10000/sys/tg_test/1/long_scalar), 2, TangoAttrValue{'rvalue': <Quantity(100, 'dimensionless')>, 'wvalue':
<Quantity(0, 'dimensionless')>, 'time': TimeVal(tv_nsec = 181, tv_sec = 1778064174, tv_usec = 818200), 'quality': <AttrQuality.ATTR_VALID: 0>,
'error': None, '_attrRef': <weakproxy at 0x725ddbd7acf0 to TangoAttribute at 0x725d972a8950>, '_TangoAttrValue__attrName':
'tango://localhost:10000/sys/tg_test/1/long_scalar', '_TangoAttrValue__attrType': 0, 'config': <weakproxy at 0x725ddbd7acf0 to TangoAttribute
at 0x725d972a8950>, '_pytango_dev_attr': DeviceAttribute(data_format = tango._tango.AttrDataFormat.SCALAR, dim_x = 1, dim_y = 0, has_failed =
False, is_empty = False, name = 'long_scalar', nb_read = 1, nb_written = 1, quality = tango._tango.AttrQuality.ATTR_VALID, r_dimension =
AttributeDimension(dim_x = 1, dim_y = 0), time = TimeVal(tv_nsec = 181, tv_sec = 1778064174, tv_usec = 818200), type =
tango._tango.CmdArgType.DevLong, value = 100, w_dim_x = 1, w_dim_y = 0, w_dimension = AttributeDimension(dim_x = 1, dim_y = 0), w_value = 0)})
(TangoAttribute(tango://localhost:10000/sys/tg_test/1/long_scalar), 2, TangoAttrValue{'rvalue': <Quantity(107, 'dimensionless')>, 'wvalue':
<Quantity(0, 'dimensionless')>, 'time': TimeVal(tv_nsec = 812, tv_sec = 1778064177, tv_usec = 808157), 'quality': <AttrQuality.ATTR_VALID: 0>,
'error': None, '_attrRef': <weakproxy at 0x725ddbd7acf0 to TangoAttribute at 0x725d972a8950>, '_TangoAttrValue__attrName':
'tango://localhost:10000/sys/tg_test/1/long_scalar', '_TangoAttrValue__attrType': 0, 'config': <weakproxy at 0x725ddbd7acf0 to TangoAttribute
at 0x725d972a8950>, '_pytango_dev_attr': DeviceAttribute(data_format = tango._tango.AttrDataFormat.SCALAR, dim_x = 1, dim_y = 0, has_failed =
False, is_empty = False, name = 'long_scalar', nb_read = 1, nb_written = 1, quality = tango._tango.AttrQuality.ATTR_VALID, r_dimension =
AttributeDimension(dim_x = 1, dim_y = 0), time = TimeVal(tv_nsec = 812, tv_sec = 1778064177, tv_usec = 808157), type =
tango._tango.CmdArgType.DevLong, value = 107, w_dim_x = 1, w_dim_y = 0, w_dimension = AttributeDimension(dim_x = 1, dim_y = 0), w_value = 0)})
```

Periodic event

Taurus Core

Taurus Listener System (events vs. polling)

- `TaurusPollingTimer` is the one calling the listener
- Internally contains a thread which will continuously, with a given periodicity, call the `TaurusDevice.poll()` method in an asynchronous way. For Tango device it means calling `DeviceProxy.read_attributes_asych()` and `DeviceProxy.read_attributes_reply()`
- The default periodicity of the reads is 3s.
- By default, for the whole process, so for all attributes, there is only one thread.
- You can have more than one polling timers, but it must have a different periodicity.
- The polling timer object is owned by the `TaurusFactory`.

Taurus Core

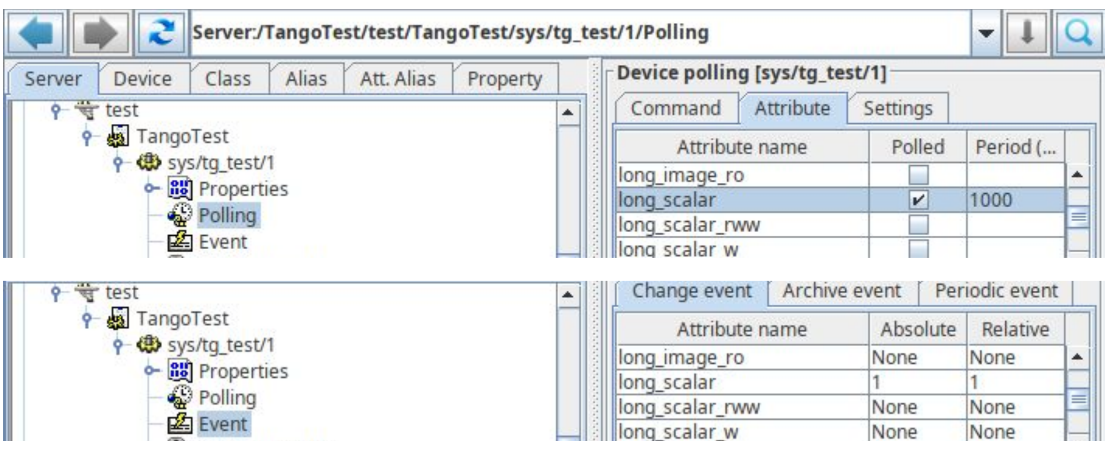
Taurus Listener System (events vs. polling)

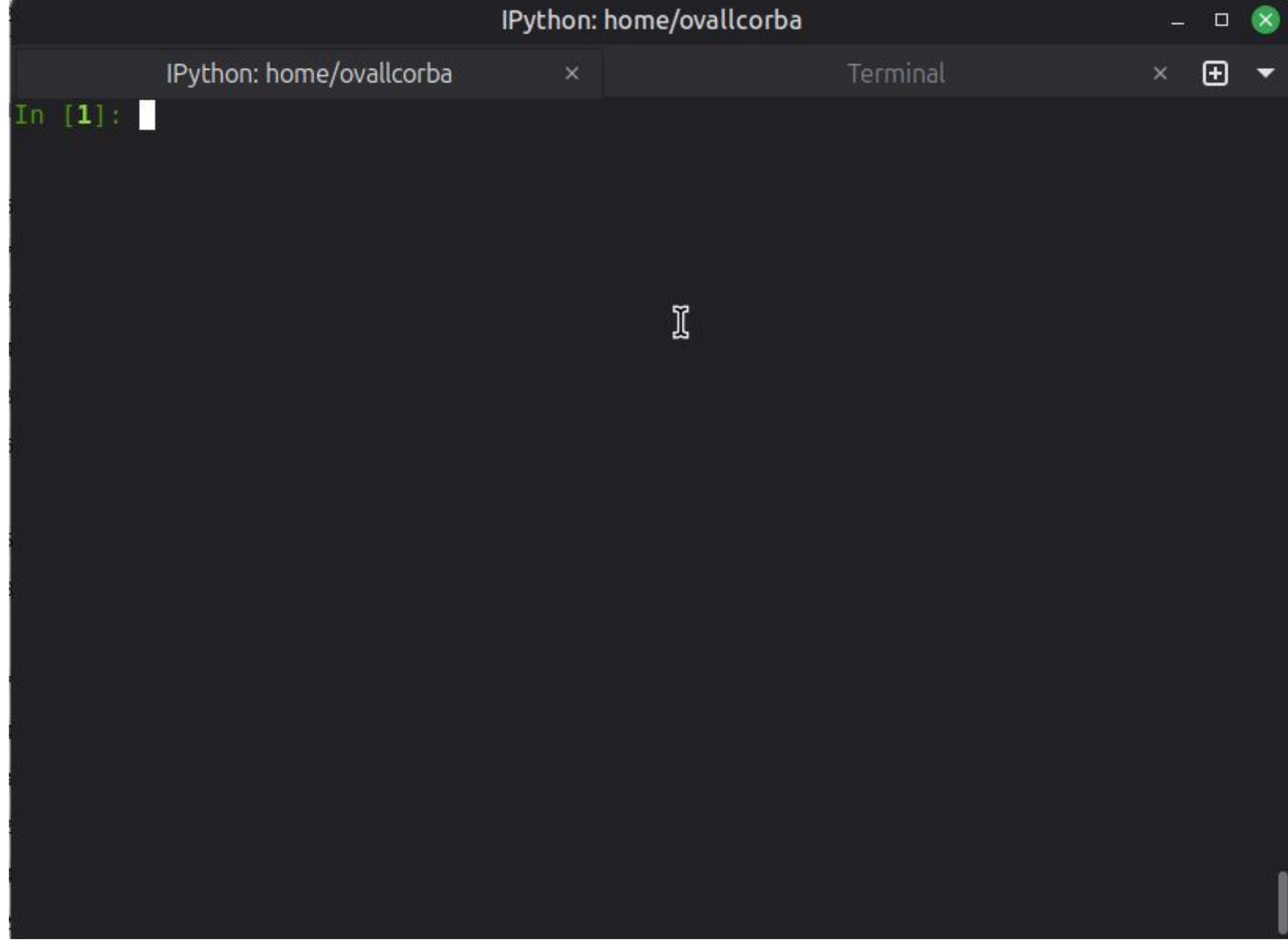
```
In [44]: import taurus

In [45]: factory = taurus.Factory()

In [46]: factory.polling_timers
Out[46]: {}

In [47]: attr.addListener(my_listener)
MainThread     INFO     2026-05-06 12:57:21,307 localhost:10000.sys/tg_test/1.long_scalar: Subscription
failed.Activating polling.
Out[47]: True

In [48]: factory.polling_timers
Out[48]: {3000: <taurus.core.tauruspollingtimer.TaurusPollingTimer at 0x725d85b97190>}

In [49]: factory.polling_timers[3000].dev_dict
Out[49]: {TangoDevice(tango://localhost:10000/sys/tg_test/1): <CaselessWeakValueDict at 0x725d85e68c10>}
```

Taurus Core

Taurus Listener System (events vs. polling)

- Taurus may choose to subscribe to backend events (if available) instead of creating the polling timer.
- This is the case for Tango
- Tango also has special event type `ATTR_CONF_EVENT` that Taurus subscribes by default





Taurus Core

Taurus Listener System (events vs. polling)

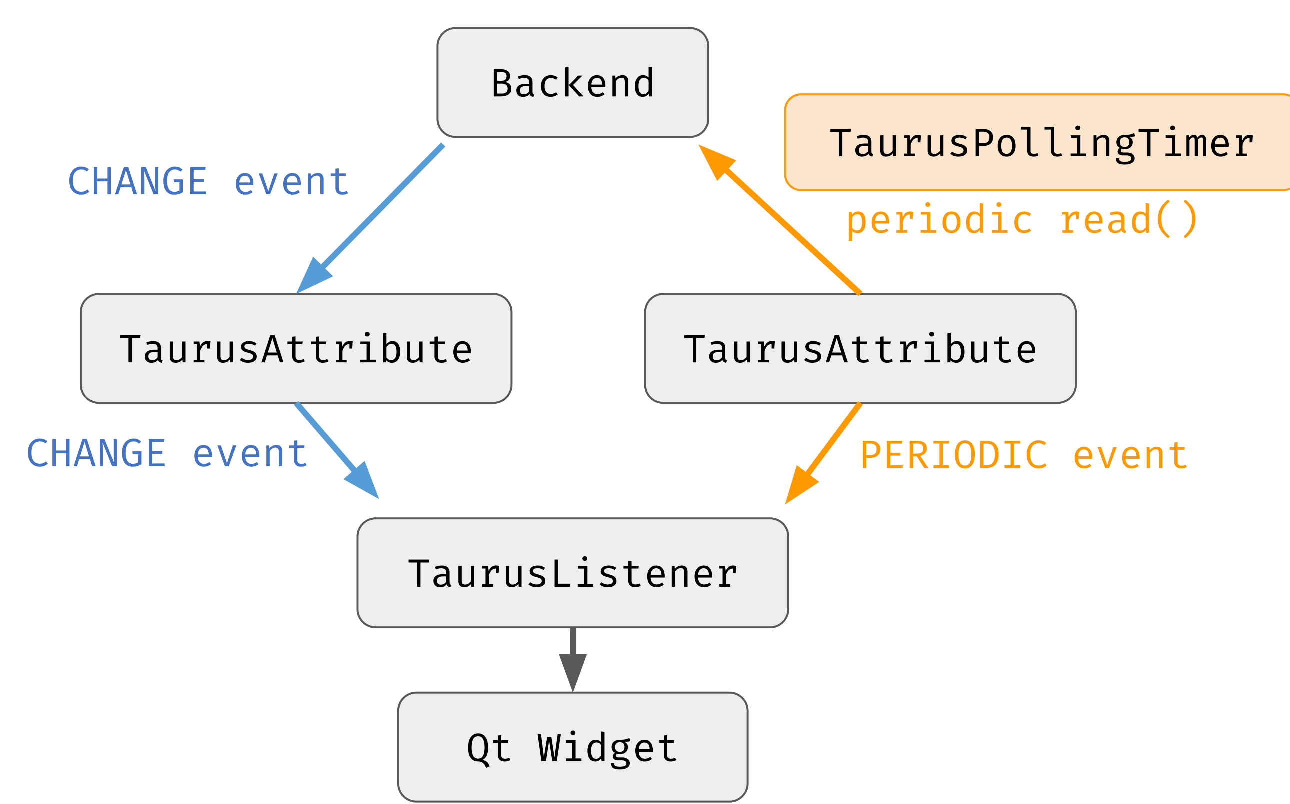
```
In [1]: from taurus import Device, Attribute

In [2]: attr = Attribute("sys/tg_test/1/long_scalar")

In [3]: def my_listener(*args):
   ...:     print(args)

In [4]: attr.addListener(my_listener)
(TangoAttribute(tango://localhost:10000/sys/tg_test/1/long_scalar), 0, TangoAttrValue{'rvalue': <Quantity(26, 'dimensionless')>, 'wvalue': <Quantity(0, 'dimensionless')>, 'time': TimeVal(tv_nsec = 902, tv_sec = 1778075261, tv_usec = 907715), 'quality': <AttrQuality.ATTR_VALID: 0>, 'error': None, '_attrRef': <weakproxy at 0x7258ed373ce0 to TangoAttribute at 0x7258ed40a6d0>, '_TangoAttrValue__attrName': 'tango://localhost:10000/sys/tg_test/1/long_scalar', '_TangoAttrValue__attrType': 0, 'config': <weakproxy at 0x7258ed373ce0 to TangoAttribute at 0x7258ed40a6d0>, '_pytango_dev_attr': DeviceAttribute(data_format = tango._tango.AttrDataFormat.SCALAR, dim_x = 1, dim_y = 0, has_failed = False, is_empty = False, name = 'long_scalar', nb_read = 1, nb_written = 1, quality = tango._tango.AttrQuality.ATTR_VALID, r_dimension = AttributeDimension(dim_x = 1, dim_y = 0), time = TimeVal(tv_nsec = 902, tv_sec = 1778075261, tv_usec = 907715), type = tango._tango.CmdArgType.DevLong, value = 26, w_dim_x = 1, w_dim_y = 0, w_dimension = AttributeDimension(dim_x = 1, dim_y = 0), w_value = 0)})
Out[4]: True

(TangoAttribute(tango://localhost:10000/sys/tg_test/1/long_scalar), 0, TangoAttrValue{'rvalue': <Quantity(144, 'dimensionless')>, 'wvalue': <Quantity(0, 'dimensionless')>, 'time': TimeVal(tv_nsec = 711, tv_sec = 1778075262, tv_usec = 908152), 'quality': <AttrQuality.ATTR_VALID: 0>, 'error': None, '_attrRef': <weakproxy at 0x7258ed373ce0 to TangoAttribute at 0x7258ed40a6d0>, '_TangoAttrValue__attrName': 'tango://localhost:10000/sys/tg_test/1/long_scalar', '_TangoAttrValue__attrType': 0, 'config': <weakproxy at 0x7258ed373ce0 to TangoAttribute at 0x7258ed40a6d0>, '_pytango_dev_attr': DeviceAttribute(data_format = tango._tango.AttrDataFormat.SCALAR, dim_x = 1, dim_y = 0, has_failed = False, is_empty = False, name = 'long_scalar', nb_read = 1, nb_written = 1, quality = tango._tango.AttrQuality.ATTR_VALID, r_dimension = AttributeDimension(dim_x = 1, dim_y = 0), time = TimeVal(tv_nsec = 711, tv_sec = 1778075262, tv_usec = 908152), type = tango._tango.CmdArgType.DevLong, value = 144, w_dim_x = 1, w_dim_y = 0, w_dimension = AttributeDimension(dim_x = 1, dim_y = 0), w_value = 0)})
```

Change event

Taurus Core

Taurus Listener System (events vs. polling)

- Event-driven architecture
- Automatic GUI updates
- Shared subscriptions between widgets
- When backend events are unavailable: Polling fallback
- Widgets do not care if updates come from backend events or Taurus polling.



Taurus schemes

Taurus URIs `scheme:authority/path?query#fragment`

- **Schemes:** `tango`, `eval`, `h5file`
- Handled by **plugins**
- Map URIs to Taurus Models:
  - `TaurusAttribute`: Represents a scalar or array value
  - `TaurusDevice`: Represents a logical grouping or endpoint
  - `TaurusAuthority`: Represents the source or backend

#	Model name (URI)	Scheme	Model type	Represented source of data/control object
1	*tango://foo:1234*	TANGO	Authority	**Tango** database listening to port *1234* of host *foo*
2	*tango://foo:1234/a/b/c*	TANGO	Device	**Tango** Device *a/b/c* registered in database *foo*
3	*tango:a/b/c/state*	TANGO	Attribute	**Tango** attribute *state* of device #2
4	*tango:a/b/c/d#units*	TANGO	Attribute	**Tango** attribute *d* of device #2 (*units* fragment)
5	*ca:XXX:m1.VAL*	EPICS	Attribute	**EPICS** process variable *XXX:m1.VAL*
6	*eval:({tango:a/b/c/d}+{epics:XXX:m1.VAL})*0.5*	eval	Attribute	**Calculated** average of the values of #4 and #5
7	*eval:rand(256)*	eval	Attribute	Random **generated** array of 256 values
8	*msenv://foo:1234/macroserver/bar/1/ScanDir*	[MSenv]	Attribute	*ScanDir* variable from **Sardana's environment**
9	*h5file:/mydir/myfile.hdf5*	HDF5	Device	File in **HDF5** format saved at */mydir/myfile*
10	*h5file:/mydir/myfile.hdf5:data/energy*	HDF5	Attribute	**HDF5** dataset *energy* of group *data* from file #9
11	*ssheet:myfile.ods:Sheet1.A1*		Attribute	Contents of cell A1 of Sheet1 of *myfile.ods* **spreadsheet**

Taurus schemes

1. URI design
    - `tango://tbl0401:10000/bl04/eh/cryo-01/temperature`
    - `tango:bl04/eh/cryo-01/temperature`
    - `eval:rand(256)`
    - `h5file:/path/to/my/file.hdf5:data/x`
    - `redis://ctbl0401.cells.es:6379/0/bl04:mythen:status`
2. Define **Name validators** for each element type (Authority, Device, Attribute). with the following implementations:
    - `isValid(name)`: to **validate** model names
    - `getUriGroups(name)`: to **parse the URI** and retrieve parts of it (to create the models)
    - `getNames(name)`: to compute the **full**, **normal** and **simple** names
3. Provide a **factory** for the scheme objects (e.g. validators)

Taurus schemes

4. Create the models (Authority, Device and Attribute): How taurus will access the underlying data. For example, Attribute must implement:
    - `decode(self, attr_value)`: Provided a raw value (i.e. a value that came from our data source, converts it to a Taurus-compatible `DataType`
    - `read(self, cache=True)`: Gets the value from the source and returns a `TaurusAttrValue` (it uses the `decode` method)
    - `encode(self, value)`: The opposite of `decode`. Gets a Taurus input and prepares the value to be written in the data source.
    - `write(self, value, with_read=True)`: Sets the value to the data source after encoding.
    - `poll(self)`: Callback function for the Taurus polling mechanism.
    - `isUsingEvents(self)`: Flag if Attribute supports events or not.
5. Register the scheme plugin (taurus entrypoint)

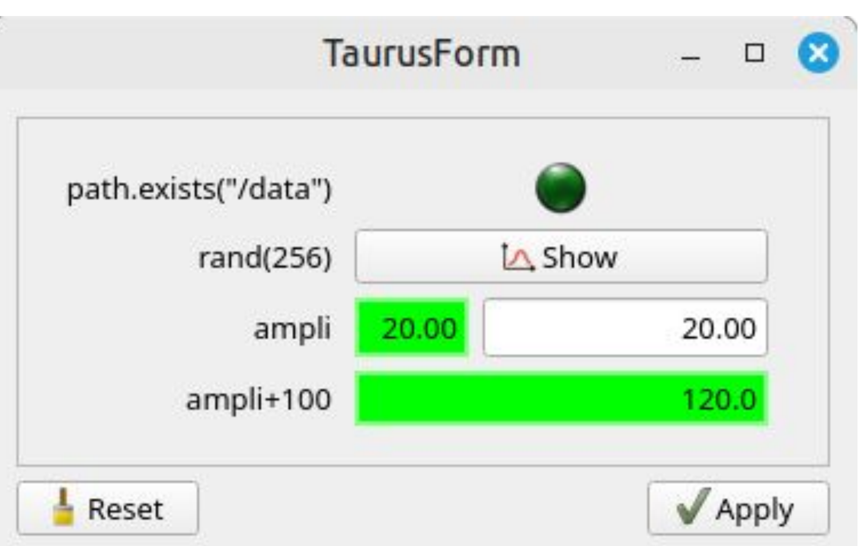
Eval scheme

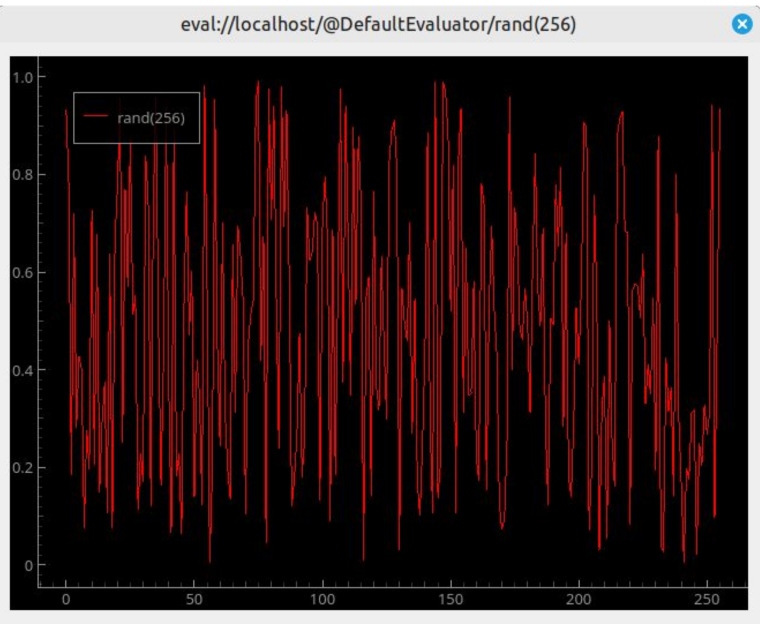
- Extension for taurus core model that provides **evaluation objects** for performing **mathematical evaluations** with data from other source, as well as allowing fast interfacing with **sources of data not supported by specific schemes**.
- `eval:[//<authority>][@<evaluator>/][<subst>;]<expr>`
    - <expr> is a mathematical expression (using python syntax) that may have references to other taurus attributes by enclosing them between { and }. Expressions will be evaluated by the evaluator device to which the attribute is assigned.

Eval scheme

```
taurus form 'eval:@os.*/path.exists("/data")' \
            'eval:rand(256)' \
            tango:sys/tg_test/1/ampli \
            'eval:{tango:sys/tg_test/1/ampli}+100' \
            'eval:@c=mymod.MyClass()/c.foo'
```

```python
class MyClass(object):
    _foo = 0

    def get_foo(self):
        return self._foo

    def set_foo(self, value):
        self._foo = value

    foo = property(get_foo, set_foo)
```





Resource scheme

- The `res:` Scheme
- Provides symbolic model aliases via mapping.
- Decouple applications from real model names
- Better portability and reusability

```
In [1]: import taurus

In [2]: factory = taurus.Factory('res')

In [3]: d = {'my_attr':'tango:sys/tg_test/1/double_scalar'}

In [4]: factory.loadResource(d)
Out[4]: {'my_attr': 'tango:sys/tg_test/1/double_scalar'}

In [5]: my_attr = taurus.Attribute('res:my_attr')

In [6]: my_attr
Out[6]:
TangoAttribute(tango://localhost:10000/sys/tg_test/1/double_
scalar)
```

Taurus extensions

Core extensions

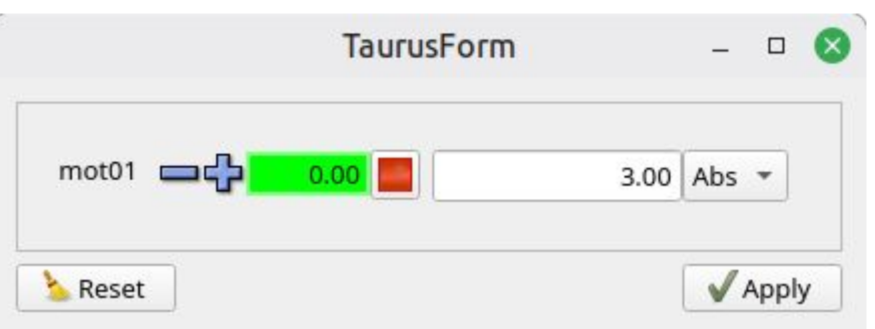
- Taurus allows to map Tango device classes to custom Python classes to customize the Tango device model class
- Useful to add high-level APIs to Tango devices
- Sardana uses this mechanism to enrich the `TangoDevice` class with additional functionalities for their Tango device classes e.g.: `Motor`, `MeasurementGroup`,...

```
In [1]: from taurus.core.tango import TangoDevice
In [2]: import taurus

In [3]: class MyDevice(TangoDevice):
   ...:     def my_method(self):
   ...:         print("Hello from my method!")

In [4]: factory = taurus.Factory()

In [5]: factory.registerDeviceClass("Pool", MyDevice)

In [6]: dev = taurus.Device("pool/pool/1")

In [7]: print(type(dev))
<class '__main__.MyDevice'>

In [8]: dev.my_method()
Hello from my method!
```

Taurus extensions

GUI extensions (item factories)

- Customize how models are displayed in Taurus widgets
- Item factories are functions that receive a taurus model and return a `TaurusValue` (building block of `TaurusForm`) which may contain up to 5 custom/specialized widgets. They can be registered in the entrypoint `taurus.form.item_factories`
- This is also used in Sardana to customize how items are shown in `TaurusForms` (e.g. Motors)




```
class MyMotorWriteWidget(TaurusWidget):
    ...
class MyMotorExtraWidget(TaurusWidget):
    ...

# combine into TaurusValue
class MyMotorTaurusValue(TaurusValue):

   def __init__(self, parent=None):
       super().__init__(parent)
       self.setWriteWidgetClass(MyMotorWriteWidget)
       self.setExtraWidgetClass(MyMotorExtraWidget)

# item factory
def mymotor_item_factory(model):
   dev = model.getParentObj()
   dev_class = dev.getDeviceProxy().info().dev_class
   if dev_class == "MyMotor":
       return MyMotorTaurusValue()
   return None
```

Taurus programmatic GUIs

Code solutions
Custom widgets

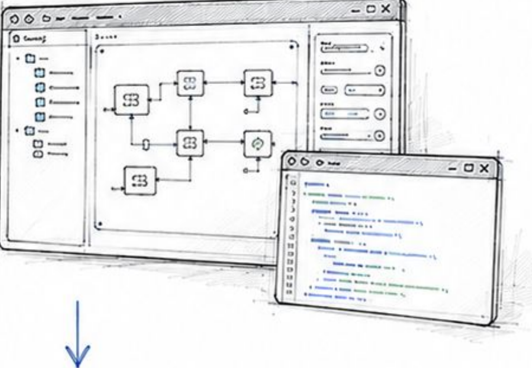
CUSTOMIZATION

ZERO-CODE

- CLI Taurus Widgets
- Taurus GUI

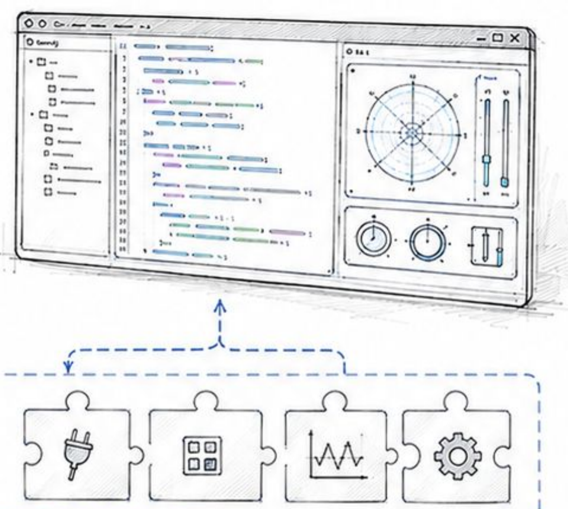
LOW-CODE

- Taurus Designer
- Config files



CODE

- Taurus Application
- Custom widgets



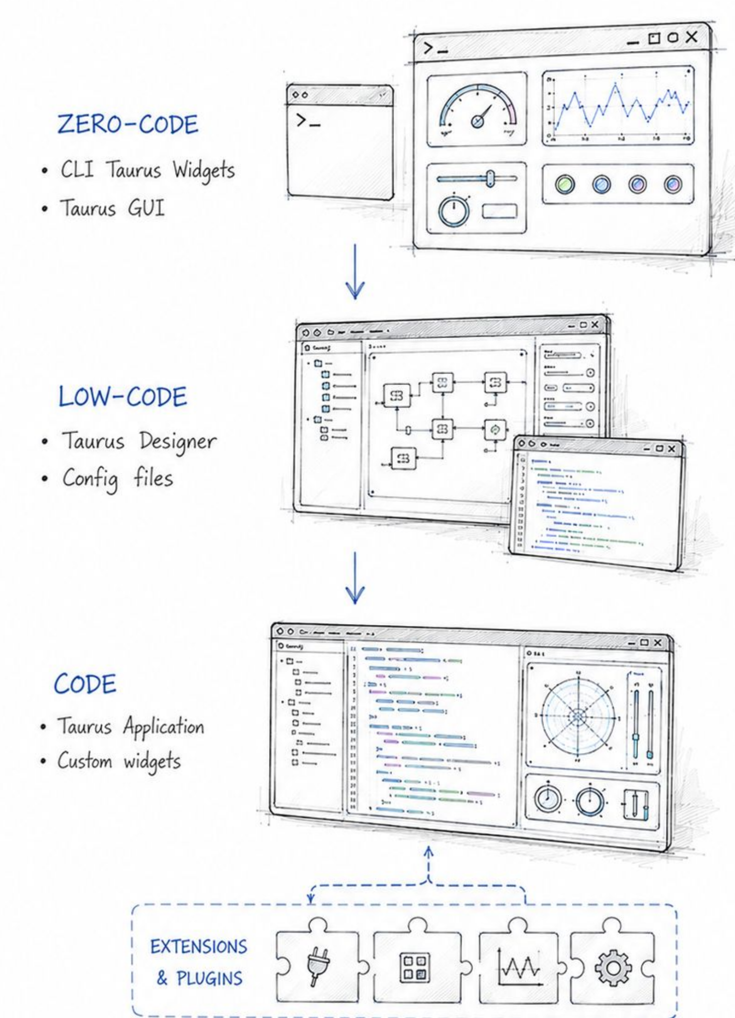
EXTENSIONS & PLUGINS

Using Taurus: from zero-code to custom apps

Three levels of customization

- **Zero-code**: CLI/Form, Taurus GUI
- **Low-code**: Taurus Designer, config files
- **Code**: TaurusApplication + Taurus widgets

Programmatic GUIs



Programmatic GUIs

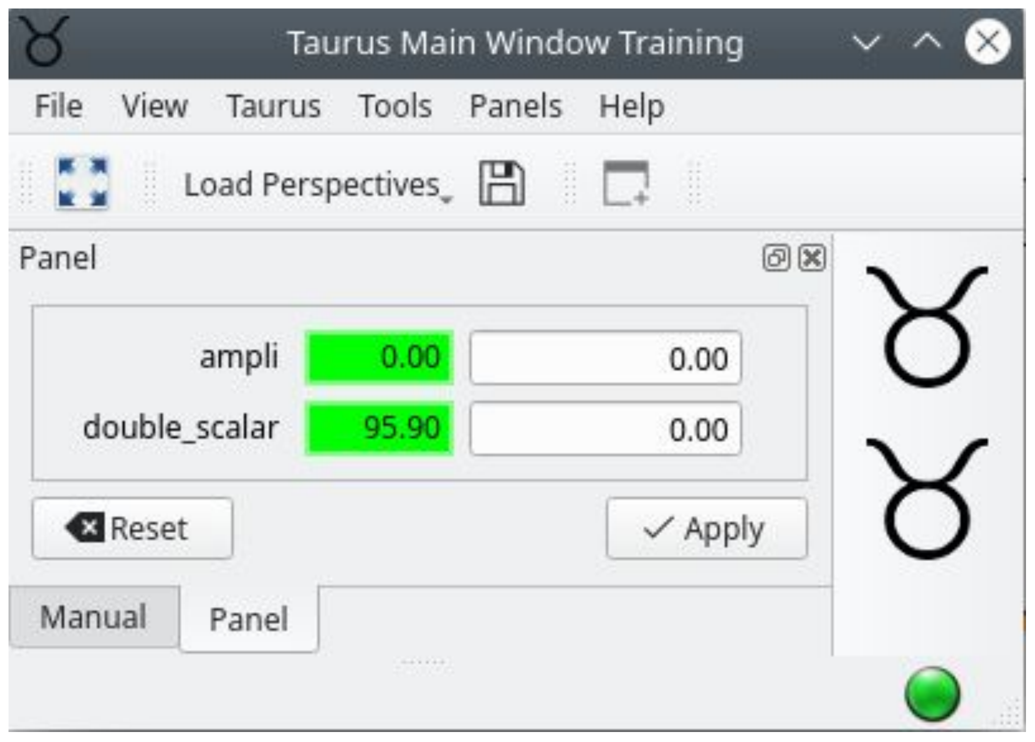
Code: A Taurus GUI can also be created in a *pure* programmatic way following these steps:

1. Create a new `mygui.py` file.
2. Create a `TaurusApplication`, a `TaurusGui` and a widget.
3. Create a panel with the created widget in the `TaurusGui` using `createPanel(widget, name)`.
4. Show the GUI and execute the application.

```
import sys
from taurus.qt.qtgui.panel import TaurusForm
from taurus.qt.qtgui.application import
TaurusApplication
from taurus.qt.qtgui.taurusgui import TaurusGui

app = TaurusApplication(sys.argv, app_name="Taurus
Main Window Training")
taurusgui = TaurusGui()

form = TaurusForm()
form.model = ["sys/tg_test/1/ampli",
"sys/tg_test/1/double_scalar"]
taurusgui.createPanel(form, 'Panel', floating=False,
permanent=True)

taurusgui.show()
sys.exit(app.exec_())
```

Programmatic GUIs

Command: `python mygui.py`




```python
import sys
from taurus.qt.qtgui.panel import TaurusForm
from taurus.qt.qtgui.application import
TaurusApplication
from taurus.qt.qtgui.taurusgui import TaurusGui

app = TaurusApplication(sys.argv, app_name="Taurus
Main Window Training")
taurusgui = TaurusGui()

form = TaurusForm()
form.model = ["sys/tg_test/1/ampli",
"sys/tg_test/1/double_scalar"]
taurusgui.createPanel(form, 'Panel', floating=False,
permanent=True)

taurusgui.show()
sys.exit(app.exec_())
```

Programmatic GUIs

Standalone widgets

- Widget functionality only

```
import sys
from taurus.qt.qtgui.application import TaurusApplication
from taurus.qt.qtgui.panel import TaurusForm

app = TaurusApplication(sys.argv)
form = TaurusForm()
form.model = ['sys/tg_test/1/ampli', 'sys/tg_test/1/double_scalar']

form.show()
sys.exit(app.exec_())
```

Programmatic GUIs

Standalone widgets

- Widget functionality only

TaurusMainWindow

- Geometry and state
- Perspectives and “factory settings"
- Customizable splashScreen
- Remote consoles/debugging
- Full-screen mode
- Launchers, StatusBar, heart-beat LED, menus

```
import sys
from taurus.qt.qtgui.panel import TaurusForm
from taurus.qt.qtgui.application import TaurusApplication
from taurus.qt.qtgui.container import TaurusMainWindow

app = TaurusApplication(sys.argv, app_name="Taurus Main Window")
mainwindow = TaurusMainWindow()

form = TaurusForm()
form.model = ["sys/tg_test/1/ampli", "sys/tg_test/1/double_scalar"]
mainwindow.setCentralWidget(form)

mainwindow.show()
sys.exit(app.exec_())
```

Programmatic GUIs

Standalone widgets

- Widget functionality only

TaurusMainWindow

- Geometry and state
- Perspectives and “factory settings"
- Customizable splashScreen
- Remote consoles/debugging
- Full-screen mode
- Launchers, StatusBar, heart-beat LED, menus

TaurusGUI

- Panel handling
- Configuration files

```
import sys
from taurus.qt.qtgui.panel import TaurusForm
from taurus.qt.qtgui.application import TaurusApplication
from taurus.qt.qtgui.container import TaurusGui

app = TaurusApplication(sys.argv, app_name="Taurus GUI")
taurusgui = TaurusGui()

form = TaurusForm()
form.model = ["sys/tg_test/1/ampli", "sys/tg_test/1/double_scalar"]
taurusgui.createPanel(form, 'Panel', floating=False, permanent=True)

taurusgui.show()
sys.exit(app.exec_())
```

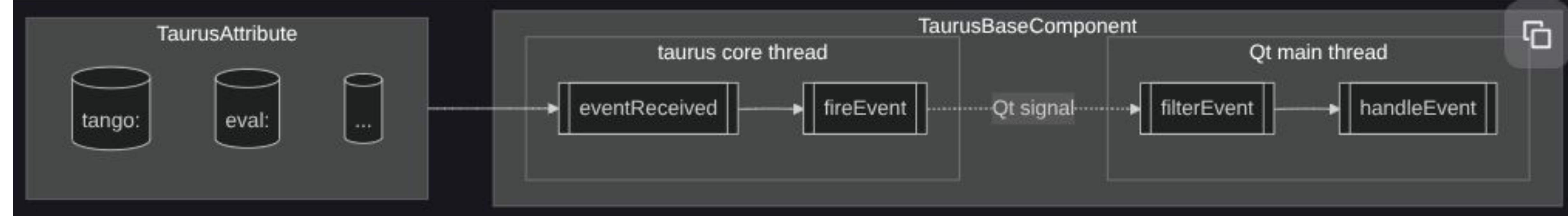
Custom Widgets

Connecting Taurus Core with Taurus Qt

- `TaurusBaseComponent` implements the `setModel()` method. This class should be used as a mixin class with the `QWidget` class. Taurus Qt widgets consume Taurus models directly: `widget.setModel( … )`
- `setModel()` method adds the `TaurusBaseComponent` (`self`) as a listener of the model object.
- `eventReceived()` (listener) method allows to filter the events in the Python thread (in case of the Tango models, it is the event consumer thread from the cppTango library, unless `TANGO_SERIALIZATION_MODE` is set to a value different than `TangoSerial`).
- If the event is not filtered out at this level, then a the `fireEvent()` method is called emitting the `taurusEvent` Qt signal with the event information (source, type and value).
- `TaurusBaseComponent` implements the `filterEvent()` slot which allows to filter the events in the Qt thread.
- If the event is not filtered out at this level, then the `handleEvent()` method is called. This is the method a widget developer should implement to update the widget state.

Custom Widgets

Connecting Taurus Core with Taurus Qt



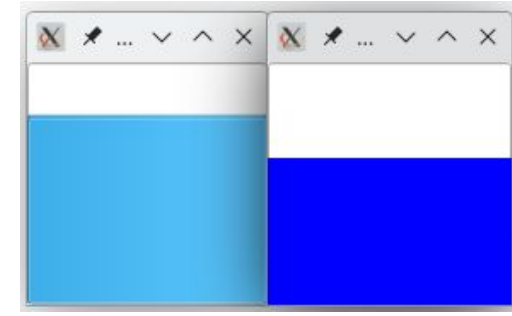
- One typically just needs to implement the high-level `handleEvent()` method. Behind the scenes `TaurusBaseComponent` takes care of auto-subscribing to tango events, filtering them, and decoupling the core event thread (e.g. Tango event thread) from the Qt main thread
- Benefits
  - Minimal GUI logic
  - Automatic synchronization
  - Shared subscriptions
  - Backend-independent widgets

Taurify widgets

Inheritance from `QWidget` and `TaurusBaseComponent`

Only need to implement `handleEvent` method

Model support API
Configuration API
Logger API
Formatter API



```
from taurus.external.qt import Qt
from taurus.qt.qtgui.base import TaurusBaseComponent
from taurus.qt.qtgui.application import TaurusApplication

You, hace 16 meses | 1 author (You)
class PowerMeter(Qt.QProgressBar, TaurusBaseComponent):

    """A Taurus-ified QProgressBar"""
    # setFormat() defined by both TaurusBaseComponent and QProgressBar. Rename.
    setFormat = TaurusBaseComponent.setFormat
    setBarFormat = Qt.QProgressBar.setFormat
      You, hace 16 meses • Taurify widgets …
    def __init__(self, parent=None, value_range=(0, 100)):
        super(PowerMeter, self).__init__(parent=parent)
        self.setOrientation(Qt.Qt.Vertical)
        self.setRange(*value_range)
        self.setTextVisible(False)

    def handleEvent(self, evt_src, evt_type, evt_value):
        """reimplemented from TaurusBaseComponent"""
        try:
            self.setValue(int(evt_value.rvalue.m))
        except Exception as e:
            self.info("Skipping event. Reason: %s", e)
```

```
if __name__ == "__main__":

    import sys      Miquel Navarro, hace 16 meses • Programatic GUI development a
    app = TaurusApplication()

    if len(sys.argv) > 1 and sys.argv[1] == "--single-model":
        ### Single model
        w = PowerMeter()
        w.setModel("eval:Q(60+20*rand())")
    else:
        ### Multi model
        w = PowerMeter2()
        w.setModel("eval:Q(60+20*rand())")  # implicit use of  key="power"
        w.setModel("eval:['green','red','blue'][randint(3)]", key="color")

    w.show()
    sys.exit(app.exec_())
```

Taurus Project

Contributing to Taurus
Taurus codebase map

REPOSITORY
ISSUES & MERGE REQUESTS
DOCUMENTATION
RELEASES
COMMUNITY RESOURCES
INSTALLATION
POLICIES / CONTRIBUTION

Taurus project

GitLab Taurus Organization (**https://gitlab.com/taurus-org/**) contains the source code of Taurus (LGPL v3+ license) and other projects that complement taurus.

The Taurus repository uses nvie's branching model, known as GitFlow.
In this model, there are two long-lived branches:

- **stable**: used for official releases. Contributors should not need to care about it
- **develop**: reflects the latest integrated changes for the next release. This is the one that should be used as the base for developing new features or fixing bugs.

Taurus project **Development workflow**

For the contributions, we use the Fork & Pull Model:

- Fork Taurus and create a branch
- Make atomic, well-scoped commits to a branch based on `develop`
- Ensure tests pass locally
- Run `ruff` (linting and formatting)
- Submit a **Merge Request** (MR) against the `develop` branch of the official taurus repository with description and link/mention to related issue if it is the case (e.g. `Fixes #N`, where N is the number of the issue). This will automatically close the related issue after merge.
- Feedback & iterate. Anybody interested may review and comment on the MR, and suggest changes to it. At this point more changes can be committed on the requestor's branch until the result is satisfactory.
- Once the proposed code is considered ready, a Taurus maintainer merges the MR into `develop`.

Taurus project Development workflow

- Core Maintainers are currently ALBA, ESO, MAXIV, SOLARIS and DESY. Besides integrating MRs, they are (or should be) the responsible of managing releases and roadmap.
- If you plan to work on a specific issue, assign it to yourself to inform the other developers/maintainers. Discussion can be done in the same issue or associated MR when created.
- For specific topics or questions, we may consider using instant messaging (Mattermost) or even pair-programming remote sessions if it may help in the issue resolution but it is good to keep the comments in the GitLab issue/MR for knowledge sharing.
- Contribution guide: https://gitlab.com/taurus-org/taurus/-/blob/develop/CONTRIBUTING.md

Taurus project Code Quality and Testing

- Follow PEP8 and naming conventions
- Linting and formatting using `ruff`
- Unit tests using `pytest` and `Unittest` with a `DeviceTestContext`. New features must include tests
- All public API code should be documented (modules, classes and public API) using Sphinx extension to reStructuredText
- Every python module file should contain license information. The preferred license is the LGPL. If you need/want to use a different one, it should be compatible with the LGPL v3+.
- Python version support:
  - Those required and in use by the participating facilities.
  - (ideally) Those supported in the NEP29, and in practice those supported by our dependencies in conda-forge

Taurus project Code Quality and Testing

- taurus-docker repository contains the dockerfiles to create the docker images for testing Taurus, and images are available in its container registry. These are used by the Gitlab-CI pipeline for testing and linting.
- Qt programming:
  - Avoid importing PyQt5/6 or PySide2/6 directly. Use `taurus.external.qt` instead.
  - Use of `taurus.qt.qtgui.application.TaurusApplication` instead of `QApplication` is recommended (it takes care of various initialization and exit tasks that are convenient).
- Coding Guide: https://www.taurus-scada.org/devel/coding_guide.html

Taurus project Taurus Enhancement Proposals (TEPs)

- TEPs are formal documents for major changes inspired by Python's PEP system. The documents should include: Motivation, Design, Alternatives, Impact. It is reviewed by maintainers and community
- When to write a TEP?
  - Major new features
  - API changes
  - Deprecations
  - Structural refactors
- TEP Index: https://taurus-scada.org/tep/index.html

Taurus project Release model

- Since 2013, taurus has aimed to do **semi-annually officially-tested releases** (roughly in July and January). The official releases involve merging to the stable branch and adding a version tag. They also require passing both the automated testsuite and a checklist of **manual tests** on selected architectures.
- **PyPI packages**: Released automatically on version tag pushes
- **Conda packages**: MR for the build is created automatically in taurus and taurus_pyqtgraph conda-forge feedstock on new entries in pypi.
- **Debian (or other OS) packages**: Build via Debian-salsa science-team repository pipeline. It may also be built internally by specific facilities (e.g. ALBA) and available on-demand.

Taurus installation

- conda

  e.g. `conda env create -n taurus taurus pytango pyqt taurus_pyqtgraph`
  (specify preferred python, pytango and pyqt versions if needed)
- PyPI
- Debian package
- Pixi

  Check this environment used for trainings and demos:
  https://gitlab.com/alba-synchrotron/controls-section/icalepcs2025-workshop
- Site customization: Taurus can be adapted to each facility using configuration settings in a taurus.ini file (e.g. default scheme/authority, custom plugins, resources, widgets...)
-

TaurusCustomSettings

- Documentation: https://taurus-scada.org/devel/tauruscustomsettings.html
- User-specific Taurus configuration (`taurus.ini` file in user config folder)
- Typical use cases
  - Set default scheme / authority
  - Set default QT API (PyQt, PySide)
  - Register extra scheme modules
  - Register widget alternatives
  - Configure resource files
  - Customize polling/event behavior
  - Define site-specific conventions (e.g. formatters)

```
[taurus]
DEFAULT_SCHEME = "tango"
TANGO_HOST = "tangodb:10000"
EXTRA_SCHEME_MODULES = ["taurus_redis_scheme"]
TANGO_EVENT_SUB_MODE= 'SYNCREAD'
```

Practical session

Contributing to Taurus

Taurus code map
Practical session format
Resources

Taurus code map

```
taurus/
├── core/        Model layer, factories, events, polling, schemes
├── qt/          Qt widgets, panels, TaurusGui, model/view bridge
├── cli/         taurus command-line interface
├── external/    Qt compatibility / abstraction layer
├── tests/       test suite
├── docs/        documentation
└── examples/    examples and snippets
```

Taurus code map

```
taurus/
└── core/                 Taurus model layer
    ├── models            TaurusModel, Attribute, Device, Authority
    ├── factories         object creation, cache, plugins
    ├── events            listeners, polling timers
    └── schemes           tango, epics, eval, res, ...
```

```
taurusmodel.py         Base model abstraction
taurusauthority.py     Control-system authority
taurusdevice.py        Device abstraction
taurusattribute.py     Attribute/value abstraction
taurusfactory.py       Object creation and caching
taurusmanager.py       Plugin discovery and registry
tauruslistener.py      Listener/event callback API
tauruspollingtimer.py  Polling infrastructure
```

Taurus code map

```
taurus/
└── qt/
    ├── qtcore/          Qt-side core utilities
    │                    signals, threading helpers, Qt model integration
    ├── qtgui/           Taurus Qt widgets and applications
    │   ├── base         setModel(), handleEvent(), TaurusBaseComponent, TaurusBaseWidget
    │   ├── model        Qt model adapters (Taurus models → Qt views)
    │   ├── display      read-only widgets: labels, leds, ...
    │   ├── input        write widgets: editors
    │   ├── panel        Composite panels: TaurusForm, TaurusValue, TaurusDevicePanel
    │   ├── container    Main windows and containers: TaurusWidget, TaurusMainWindow
    │   ├── taurusgui    Full TaurusGui framework (panels, perspectives, config)
    │   ├── tree/table   browsers and views
    │   ├── tree/        Tree views and browsers
    │   ├── table/       Table views and models
    │   ├── dialog/      Dialogs and selectors
    │   ├── util/        GUI utilities and helpers
    │   └── extra_*/     e.g. PlotPy integration for image/trend2D widgets
    └── qtdesigner/      Qt Designer integration
                         Taurus widget plugins, designer catalog
```

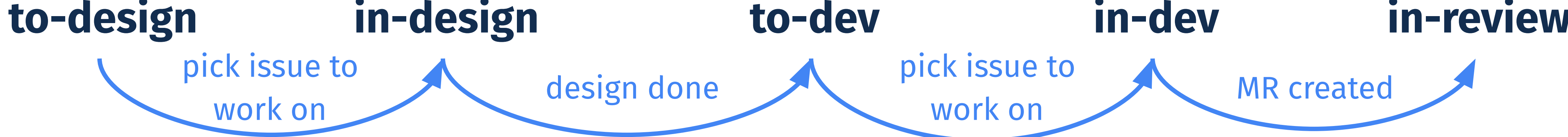
Practical session format

- Prepare a **development environment**: Easiest way may be a conda environment with editable installation of Taurus (see installing taurus section)
- Create **small working groups**. Ideally **2 people** with different degree of expertise in Taurus project.
- The idea is that the groups work on specific issues from the **Taurus issue backlog** or work in **new ideas or needs** that appeared during yesterday's session of the workshop
- Most issues contain **labels** giving hints of difficulty, size, impact and area of Taurus code affected

Practical session format

Development board: https://gitlab.com/taurus-org/taurus/-/boards/11299406

Contain pre-selected issues in different **states**. The flow is:



issue that may need some discussion to agree to a design

When the design is set, coding is ready to start

MR/branch is created and under review (will be closed automatically)

Practical session format

- There are user-focused tutorials in the taurus-training repository https://gitlab.com/taurus-org/taurus-training. Tutorials can be followed in situ and ask questions to participants.
- Apply what you learned in the workshop and from the tutorials to create a GUI for a real use case you may have. Or solve problems in GUIs in development
- Work on small developments:
  - Improve documentation. Unify taurus trainings.
  - Colors improvements (optimal fg color, palettes in custom settings, title bar widget)
  - Light theme for taurus trend in taurus custom settings
  - TaurusDevicePanel favourite commands
  - New schemes (OPCUA)
  - Improve tooltips (better messages, color roles?)
  - ...

Resources

- Taurus documentation: https://taurus-scada.org/
- Taurus repository: https://gitlab.com/taurus-org/taurus
  wiki also contain useful links: https://gitlab.com/taurus-org/taurus/-/wikis/Home
- Taurus PyQtGraph: https://gitlab.com/taurus-org/taurus_pyqtgraph
- Taurus trainings: https://gitlab.com/taurus-org/taurus-training
- Taurus mailing lists
- Monthly follow-up meetings with the community.
  - Meeting minutes: https://gitlab.com/taurus-org/taurus-followup
- Taurus Mattermost channel in Tango-Controls

Mapping issues to code areas

Schemes
(tango, epics, eval, res, external plugins)

values/events

Taurus Core
(models, manager, factories, cache, listeners, polling)

models

Qt integration & event bridge
(models, manager, factories, cache, listeners, polling)

Qt updates

Plotting widgets
(taurus_pyqtgraph plot, trend)

Qt widgets & GUI
(taurus.qt.qtgui display, panel, container, ...)

CLI & Apps

qtdesigner
(Designer plugins)

design mode

Acknowledgements

ALBA: Emilio Morales, Jose Ramos, Sergi Rubio, Zbigniew Reszela, *et.al.*

Big thanks to all contributors (issues, MRs, commits in last 2y, alphabetical order)

Adrian Justyniarski, Adrianna Pytel, Aleix Puiggali, Alexandre Moutardier, Antonio Bartalesi, Arturo Hoffstadt Urrutia, Benjamin Bertrand, Dmitry Egorov, Emilio Morales, Florian Schweiger, Hanno Perrey, Jairo Moldes, Jan Kotanski, Jordi Aguilar, Jose A. Ramos, Konstantin Klementiev, Ludmila Klenov, Lukas Wittenbecher, Maciej Mleczko, Marzena Kaczmarczyk, Mateusz Floras, Michael Huth, Michael Schneider, Michal Piekarski, Mihael Koep, Miquel Navarro, Natxo Vergara, Oriol García, Oriol Vallcorba, Philippe Gauron, Qing Zhang, Raphaël Girardot, Thomas Braun, Valentin Valls, Yury Matveev, Zbigniew Reszela

Thank you for your attention!

TAURUS Workshop, 12-13 May 2026