1 Karabo GUI: Navigation Panel Folding

The former `Navigation Panel` now has two tab panels hosting the overall `System Topology` and the `Device Topology`.

- The Device Topology fully obeys the `Karabo Naming Convention` (DOMAIN/TYPE/MEMBER)

The topology information is expanded by default. This means that all tree components are being shown in the Navigation Panel resulting in a lengthy tree. The `header` of both panels can be double-clicked to expand/collapse all components.

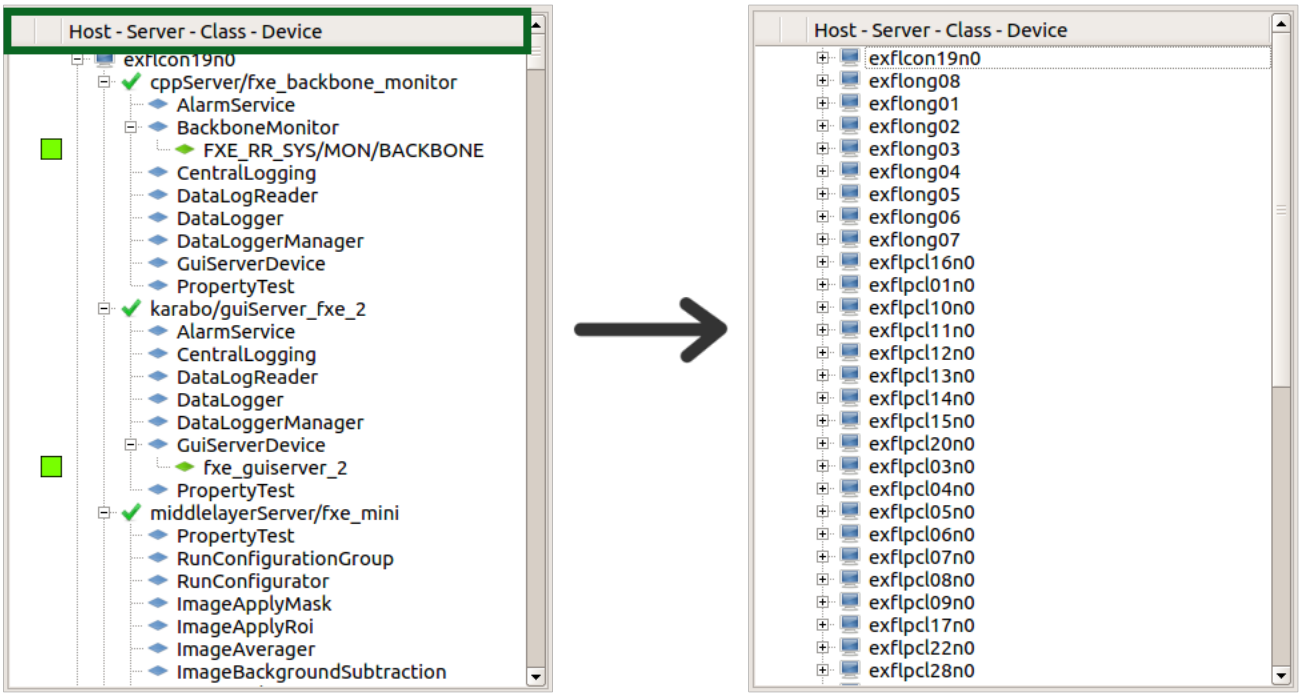


Fig. 1: Double-clicking the header (left) of the `System Topology` will collapse the tree, leaving a list of control hosts (right). Double-clicking it again will expand all tree items.

A tree component can be collapsed with its members being hidden from the list by clicking the arrow icon besides it.

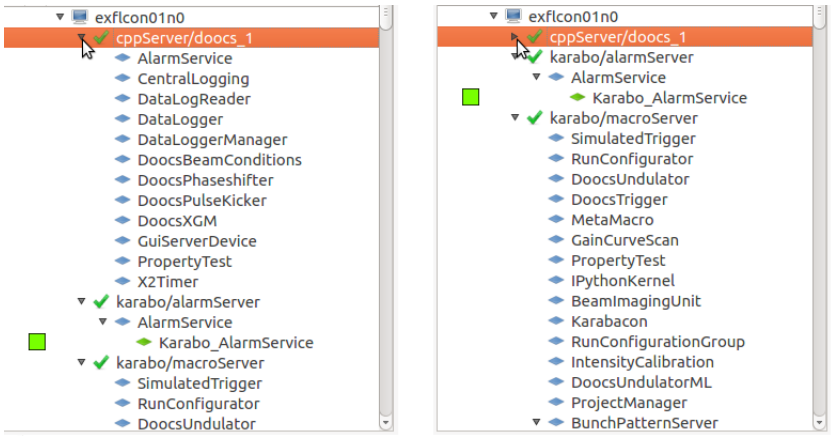


Fig. 2: A Karabo component with members (such as `cppServer/doocs_1`) can be expanded (left) and collapsed (right) by clicking on the arrow icon besides it.

2 #2: TrainId in Configurator

2.1 TrainId in Configurator

The `Configurator Panel` on the right side of the graphical user interface (GUI) can provide essential device information.

- Selecting an **online** device will show the access level dependent configuration
- Selecting an **offline** device will give information about the `class schema`

If you want to check the `TrainId` of a live property, please click on the icon field next to the property (left mouse button) to retrieve a pop-up with more information. The last received value is provided with a `timestamp` and the train Id.

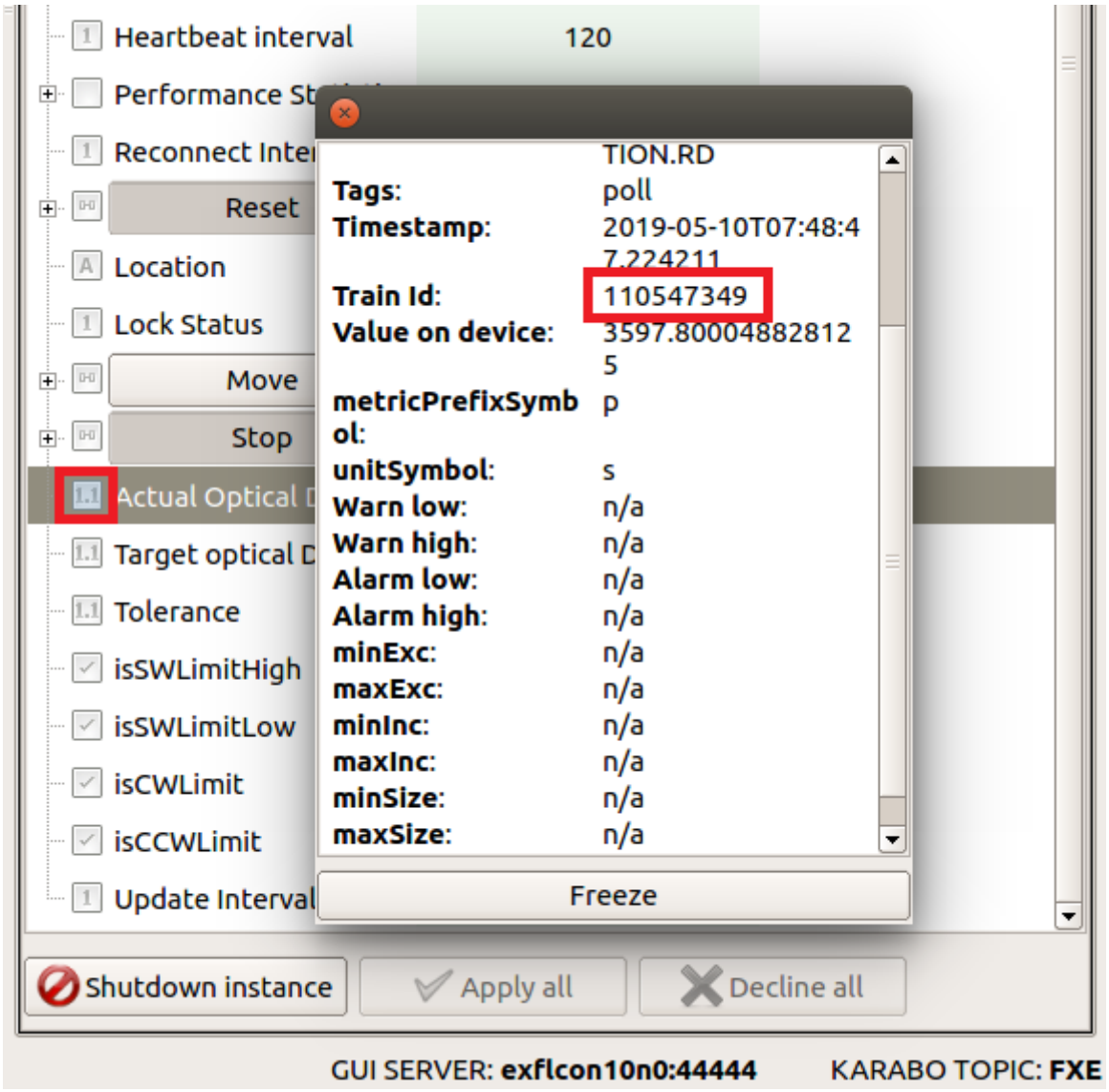


Fig. 3: Pop-Up window for the `actualPosition` of a motor device showing timestamp and train id information as well as additional attributes.

3 #3: Karabo GUI Dependencies

3.1 Karabo GUI Dependencies - Plotting

Major changes in the Karabo GUI have been performed in the last month. We introduced a new plotting library **PyQtGraph** to become the driving plotting library and introduced several plotting widgets. Regarding future steps we plan to remove the currently existing plotting library **Qwt** and make the GUI installation independent from the Framework using **CONDA** (https://docs.conda.io/en/latest/). From **Karabo 2.7** onwards the Karabo GUI can only be installed via CONDA. As an intermediate step we had to replace the C++ based library **PyQwt** with its pure python counterpart **PythonQwt**. Unfortunately, this affects already existing installations and can raise a dependency conflict. We updated our documentation.

In order to remedy the dependency problem please ...

- Linux: Clean install your karabo installation, by removing the previously installed `karabo` folder if necessary.
- Windows: If applicable, perform a clean installation. Alternatively, please try the following lines using the python package installer:

```
* pip install --upgrade guiqwt==3.0.3
* pip uninstall PyQwt
* pip install PythonQwt==0.5.5
```

We recommend using a **clean** installation.

Best wishes, the Karabo GUI Team

4 #4: State Color and Link Widget

4.1 State Color Widget

The state color widget can not only provide the respective state color. It also can provide the **state string** on demand. Use a right-click in edit mode on this widget to change the `Properties` of this widget

- *show-state-string*

Don't forget to save your changes in the project and this widget will show the full state information for every update from now on!

4.2 Link Widgets

The Karabo GUI has three different flavors of so-called `link` widgets.

- Scene Link (grey)
- Device Scene Link (blue)
- Web Link (pink)

All widgets can be distinguished by the cross color on the top left. Both, the **Scene Link** and the **Web Link** can be created from the scene toolbar. The **Device Scene Link** is more special. For this widget, a property named `availableScenes` has to be dragged from a device configuration (**Configurator**) onto the scene. The widgets can then be mutated into the desired link widget.

Note: Generally, the widget configuration offers a background color, font, text etc.

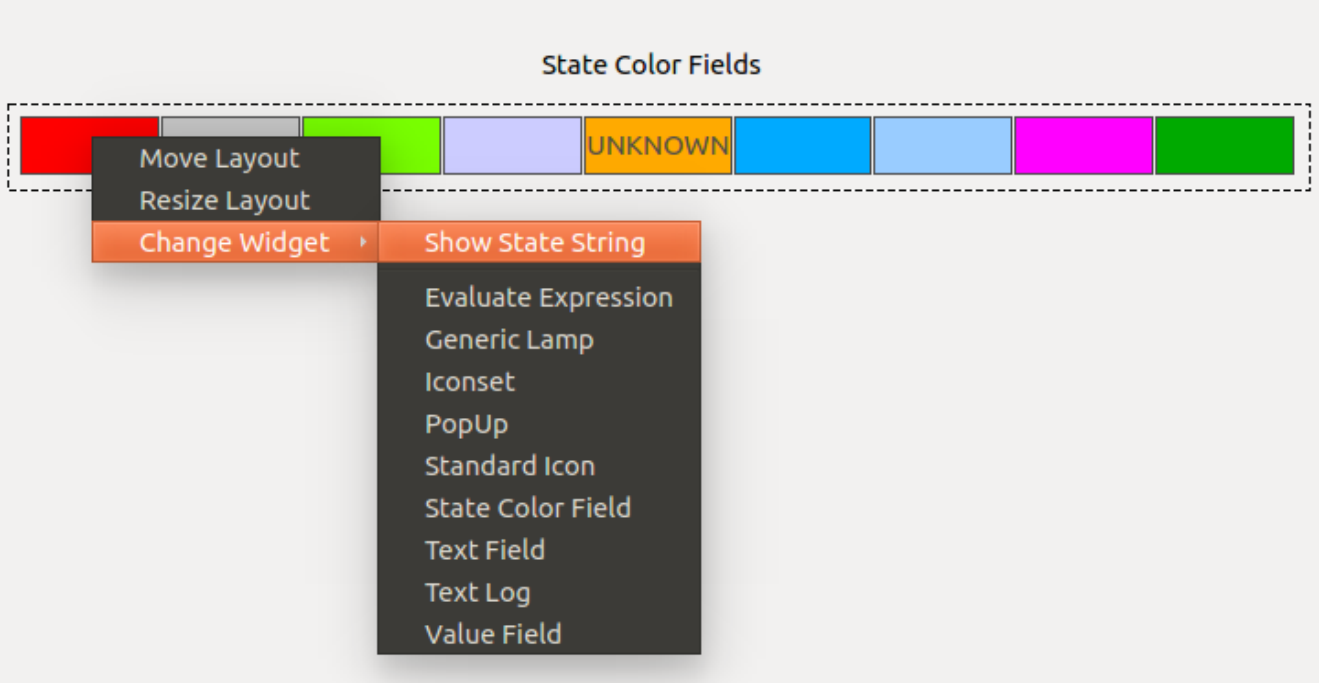


Fig. 4: State color field presentation with state string and without.

5 #5: Fonts in the Karabo GUI

5.1 Fonts in the Karabo GUI

The karabo graphical user interface (GUI) offers a wide potential of **font** settings. Every operator is free to configure his *scene* panels with the desired fonts.

However, not every operating system has every *font* available. Hence, configuring a label with the *print font* `Ubuntu` will affect `Windows` operators, as the operating system will find a substitute. This might not always match the size of the labels in which the text is written. The only cure would be to make the label size larger.

Print fonts that have the capability to render similar on MacOS, Windows and Ubuntu are `Sans Serif`, `Arial` and `Helvetica`.

Therefore, from Karabo 2.7.0 onwards, the default font provided is **Sans Serif**, being the only candidate that was available on all of our OS test machines.

All previous font settings in the karabo projects remain untouched!

Note: The karabo GUI (2.7.0) default *print font* is `Sans Serif` with a point size of 10 (before 11) and the minimum widget height is reduced to 18 pixels (before 20).

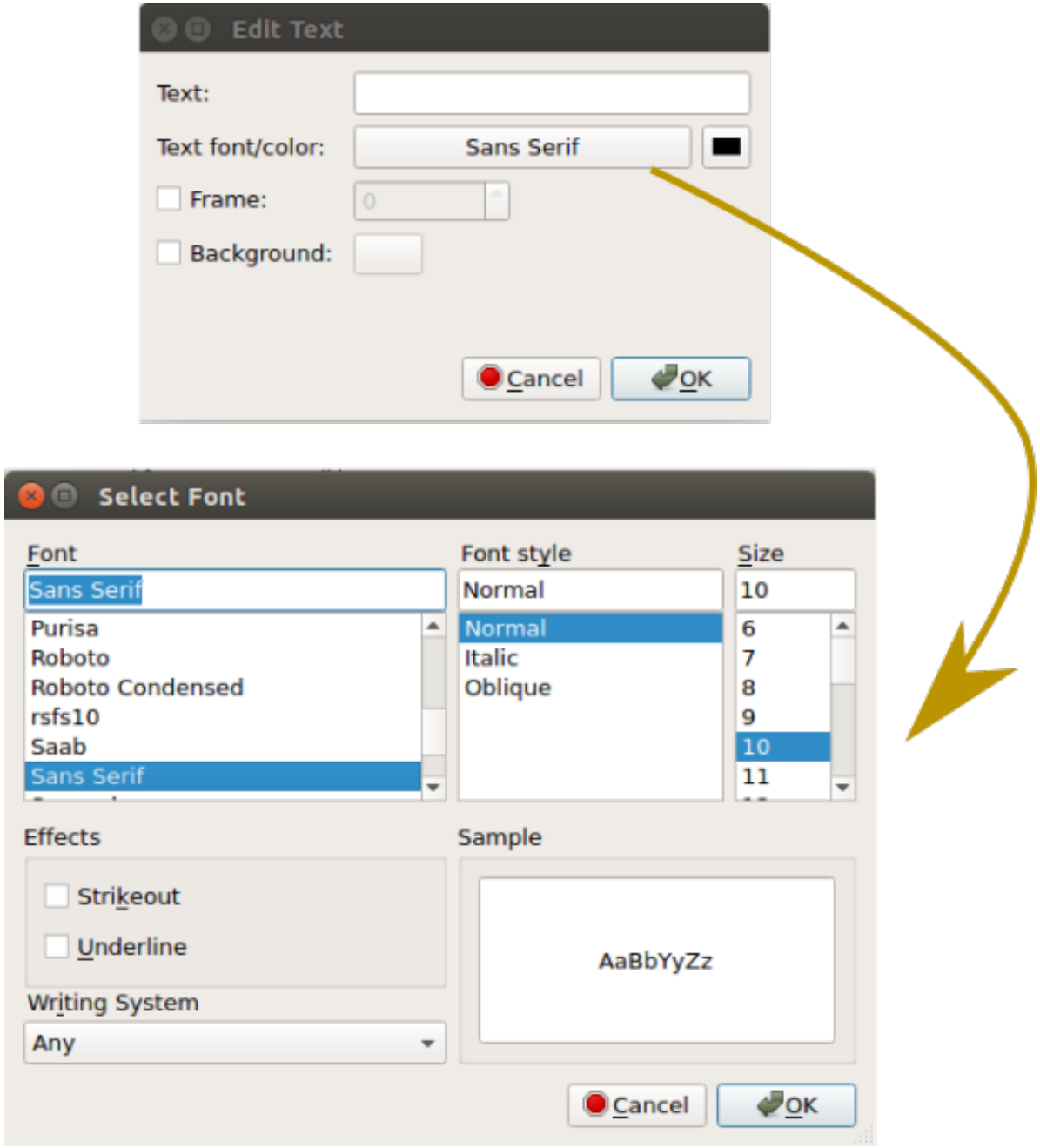


Fig. 5: Font configuration of the label widget.

6 #6 PhaseOut Qwt

6.1 Phasing out Qwt and Matplotlib Widgets from the Karabo GUI Client

The karabo graphical user interface (GUI) currently uses three different plotting libraries, namely **Qwt**, **matplotlib** and **PyQtGraph**. The library **Qwt** was not fulfilling the demands in the past with regard of configuration storage, tailorable tools such as region of interest settings, linked aux plots, etc. Moreover, it uses a C++ backend, which we consider widely as technical debt in our future plans. The library **matplotlib** is not the best choice for fast plotting and interactive features.

For this reason, we implemented each existing **Qwt** plot widget using the library **PyQtGraph** and tailored the features to the feedback we received so far. The `MultiCurve` widget from **matplotlib** will get its graph counterpart in Karabo 2.8.0 in March.

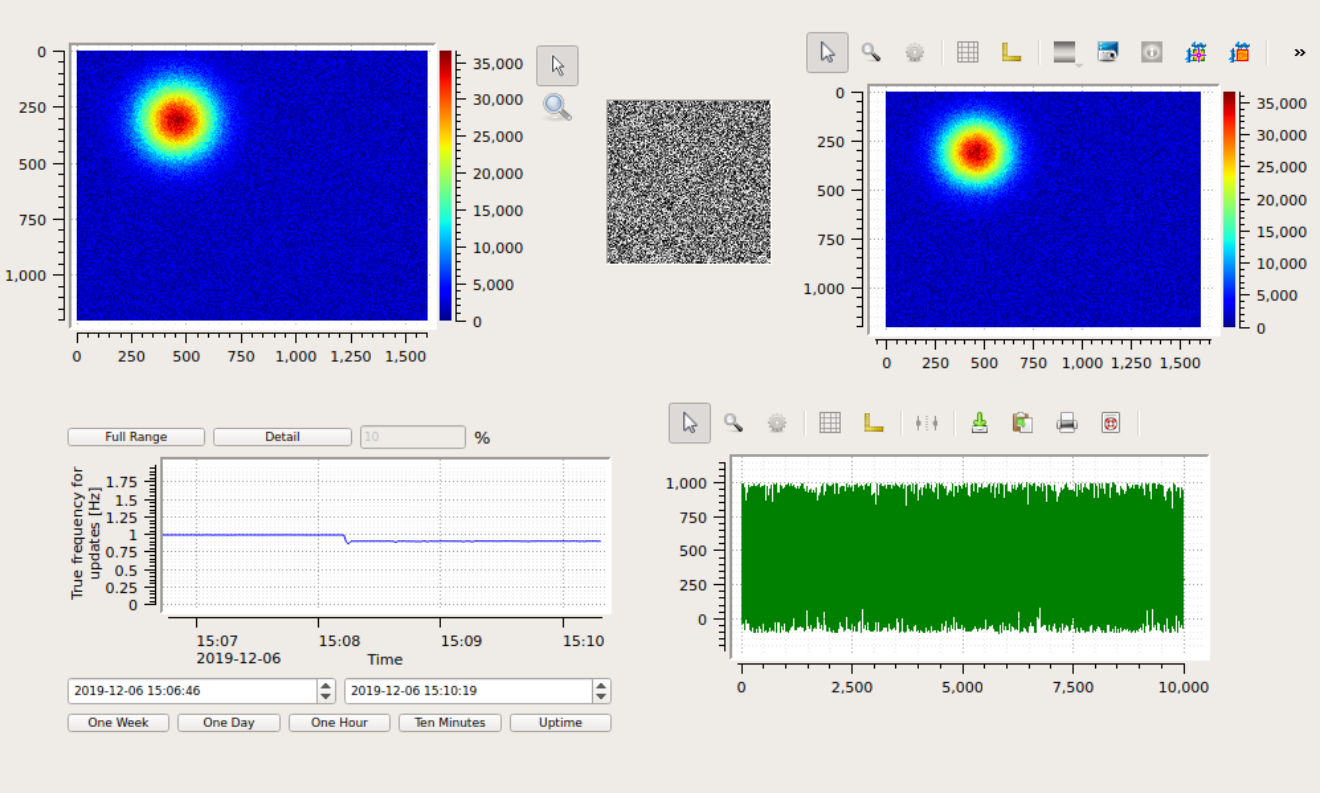


Fig. 6: Old plotting widgets using the library `Qwt`.

Widget Table			
Type	Qwt	Matplotlib	PyQtGraph
Vector	Plot	•	Vector Graph
Vector	XY-Plot	•	Vector X-Y Scatter
Vector	•	•	VectorRoll Graph
Vector	•	•	Vector Histogram Graph
Vector	•	•	Vector Bar Graph
Vector	•	•	Vector Fill Graph
Image	Aligned Image View	•	Detector Graph
Image	Image Element	•	WebCam Graph
Image	Image View	•	Image Graph
Image	Webcam Image	•	WebCam Graph
Image	Scientific Image	•	WebCam Graph
Float/Bool/Int	•	XY-Plot	Scatter Graph
Float/Bool/Int	•	MultiCurve	Development . . .
Float/Bool/Int	Trendline	•	Trend Graph

In order to move further, we will finally deprecate and remove the **Qwt** and **matplotlib** widgets with Karabo **2.9.0**, which will be released **15.05.2020**. We are aiming for an automatic procedure with a script working on the project database content to replace the existing Qwt and matplotlib widgets with their PyQtGraph counterpart.

7 #7: Image Graph Transformations

7.1 Image Graph Transformations

The Image Graph provides transformations to further customize the image projection in the plot widget. These options can be accessed by the following actions:

1. Go to design mode
2. Right-click the image widget
3. Hover “Image Graph: Properties”
4. Select “Transformations”

The transformation dialog will then appear:

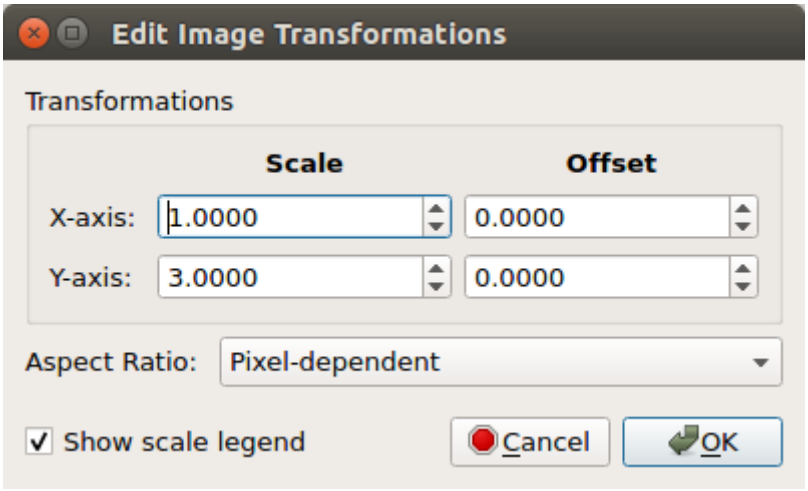


Fig. 7: Transformations Dialog

The following values can be set:

1. **Scale** - dictates the ratio of the pixel dimensions to its plotted value.
2. **Offset** - sets the starting value of the image for each axis
3. **Aspect Ratio** - describes the width and height proportion of the image.
   - “None” will auto-fit the image in the given space
   - “Pixel-dependent” will respect the image dimensions.

     This is the most common aspect ratio as it does not modify the appearance of the received image data.

     For instance, a `100 x 150` image will still have `1:1.5` ratio regardless of input scale.
   - “Scale-dependent” will apply the ratio on the image.

     This is a more advanced aspect ratio as it modifies the appearance of the image heavily with the input scale.

     For instance, a `100 x 150` image will be stretched to satisfy the input scale of `1:3`.

The default value is `1:1` scale with no offsets and uses the pixel-dependent aspect ratio. The scale legend can also be toggled from the dialog.

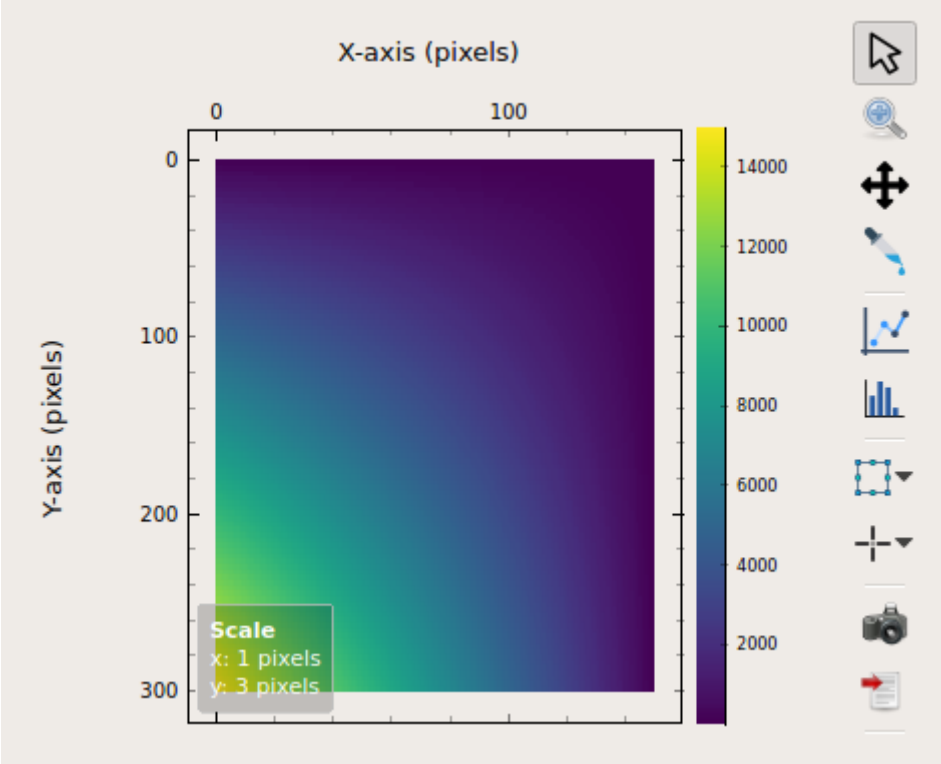


Fig. 8: Aspect Ratio: None

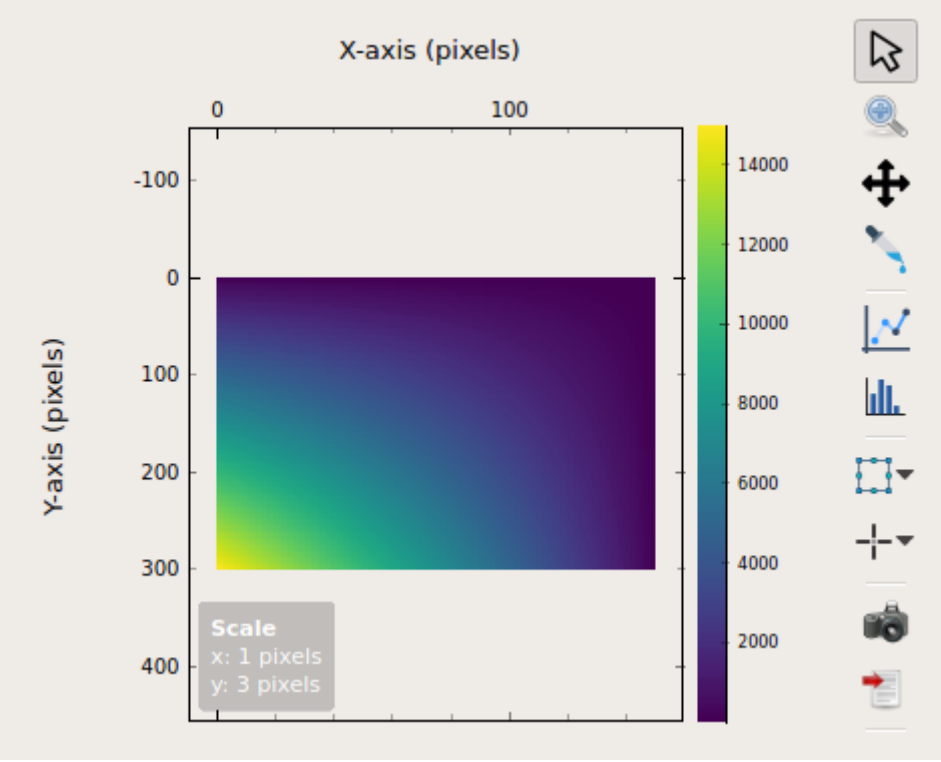


Fig. 9: Aspect Ratio: Pixel-dependent

7.2 Use Case: Changing y-axis to mm-scale

A user wants to change the **y-axis** of an image to **mm-scale**, with a **1 pixel : 3 mm ratio**. He also wants to **adjust the image proportions to the new scale**. Then the following can be done:

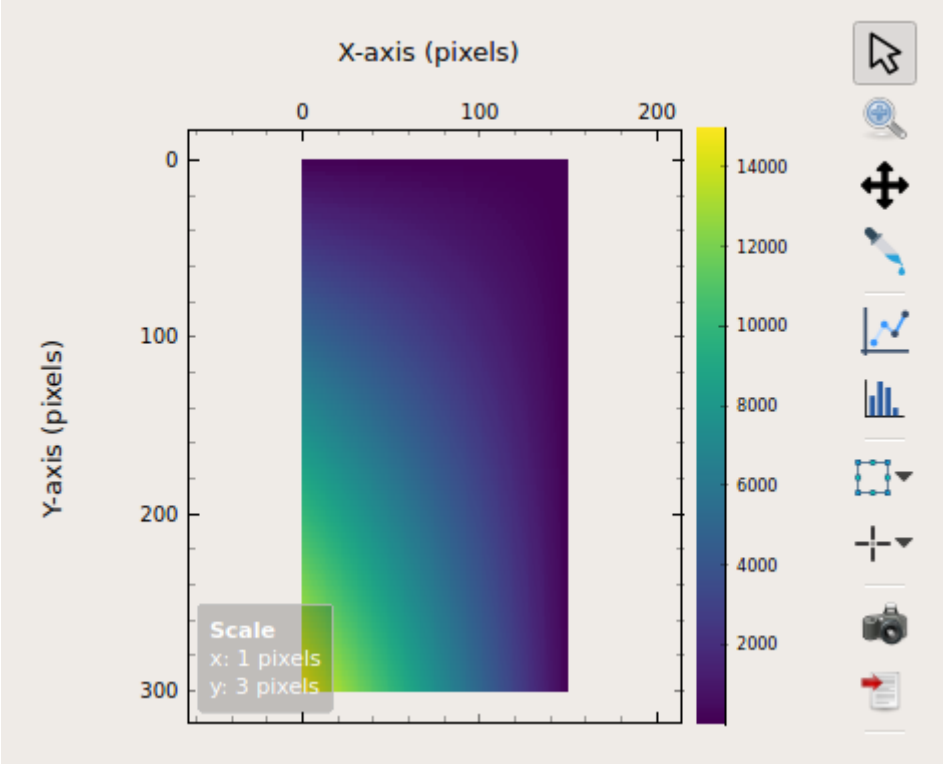


Fig. 10: Aspect Ratio: Scale-dependent

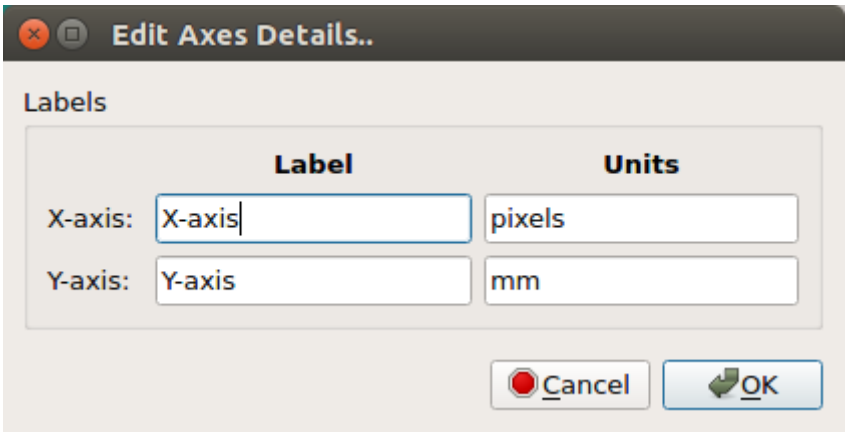
1. Rename the axis to units of interest (in this case, mm)

a. Go to design mode

b. Right-click the image widget

c. Hover “Image Graph: Properties”

d. Select “Axes Labels”



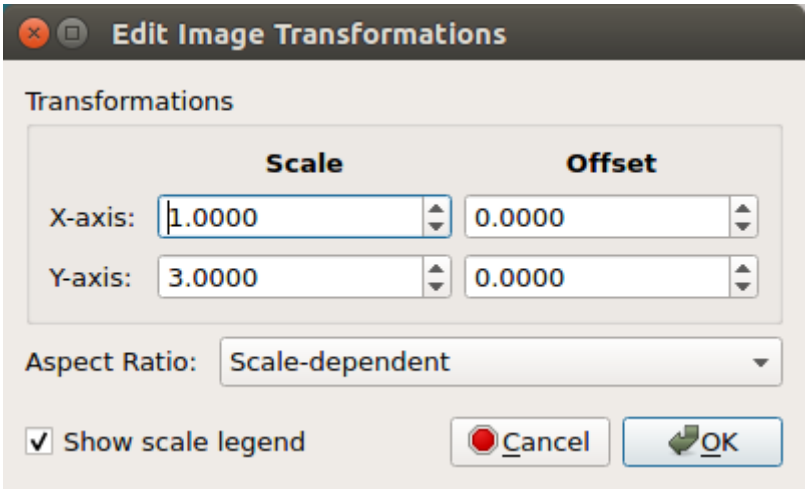
2. Set the transformation factor.

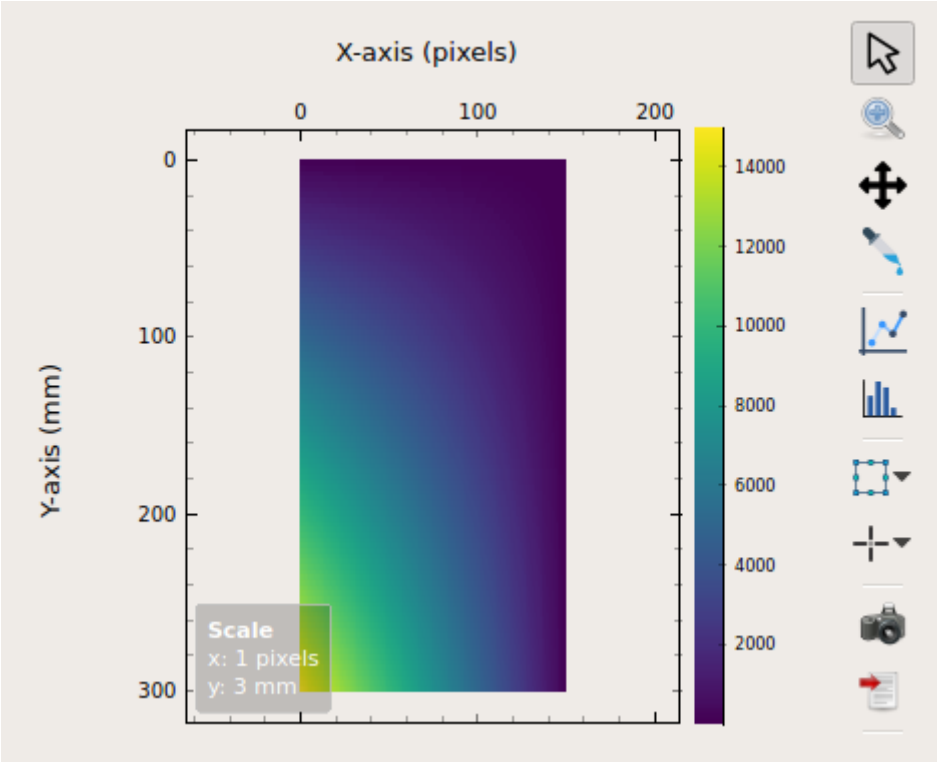
a. Open the “Transformations” dialog.

b. Set the y-axis scale to 3

c. Set the aspect ratio to `Scale-dependent`. This is to adjust the image proportions.

The following image widget is now:





8 #8: Topology Changes

8.1 Topology Changes

With Karabo 2.9.0, the `Topology Tabs` in the `Navigation Panels` are adjusted for an improved usability.

The `System Topology` gets significant changes. The main changes are:

- Classes will not immediately show if there is no device
- It is not possible anymore to instantiate from the `System Topology`
- Classes can be selected, but the class description (schema) cannot be configured

The `Device Topology` is adjusted regarding the names. Instead of *Domain - Type - Member* it will show in the future the full **control name** with **Domain - Type - Name**.

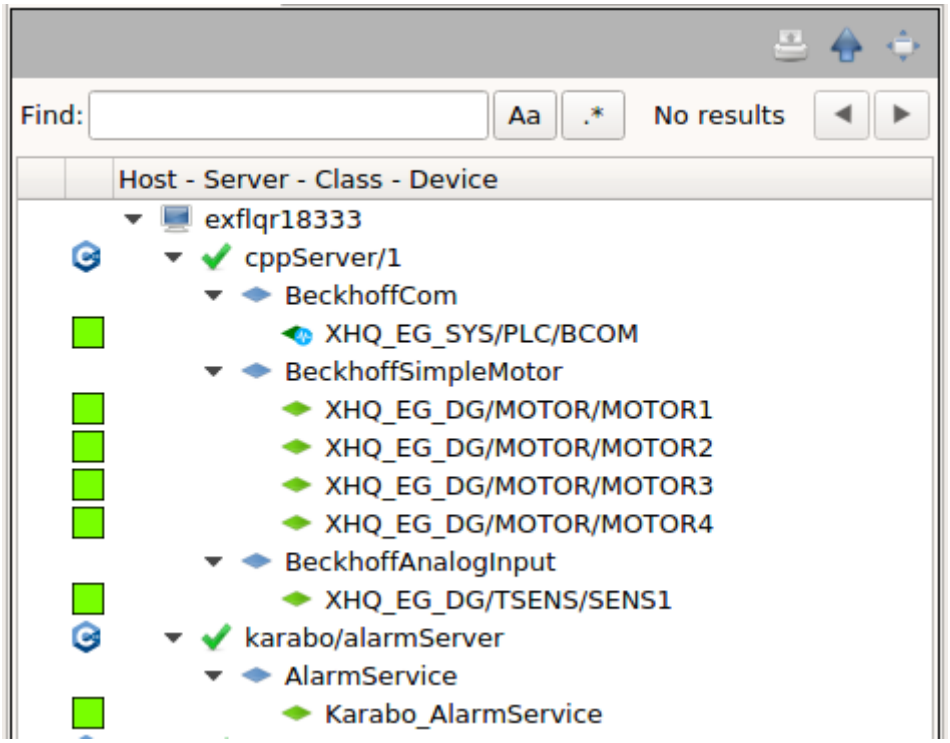


Fig. 11: System Topology with the Hierarchy: Host - Server - Class - Device

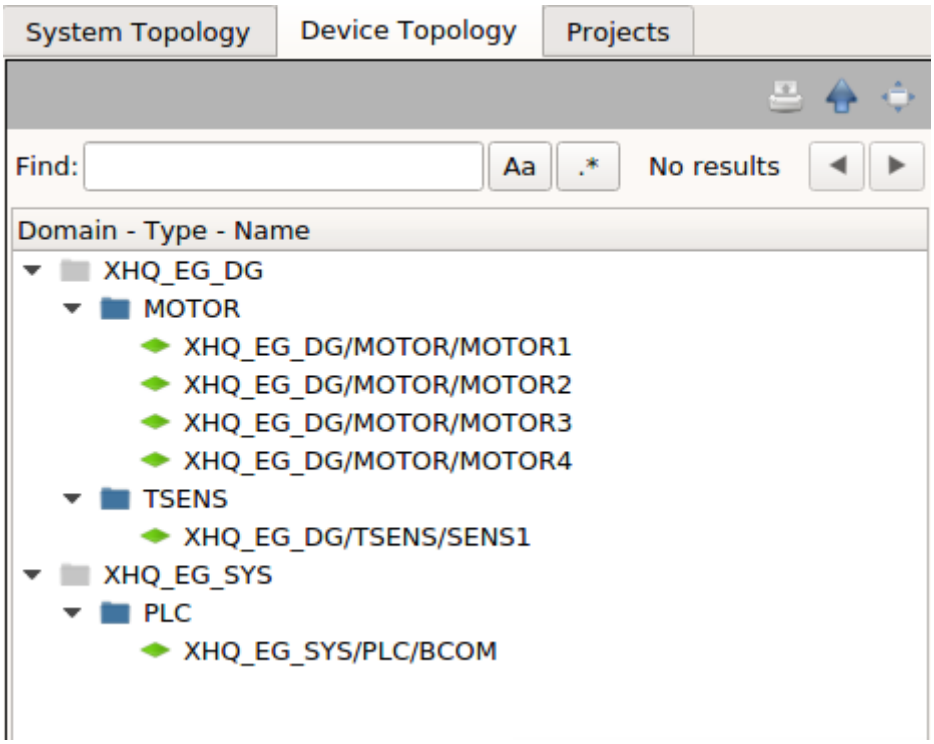


Fig. 12: Device topology with the hierarchy: Domain - Type - Name

9 #9: Phaseout of Qwt 2

9.1 The Phaseout of Qwt and Matplotlib Widgets

Karabo is moving forward and will not support the **Qwt** and **matplotlib** widgets with Karabo **2.9.0**, which will be released **15.05.2020**.

For a rather smooth transition, the new GUI will reinterpret the scene information of Qwt or matplotlib widgets. In other words, a new GUI will open a scene with a Qwt widget, but shows the counterpart listed in the table below. The old widget is still part of the scene, but will be fully replaced once the scene is saved to the project database.

The geometry of the existing widgets are reused on the replacements to preserve the scene layout. There might be cases that these would not be the best sizes for the new widgets, thus we would like to ask the scene developers to check and manually resize the widget to fit your scene. We are sorry for the inconvenience.

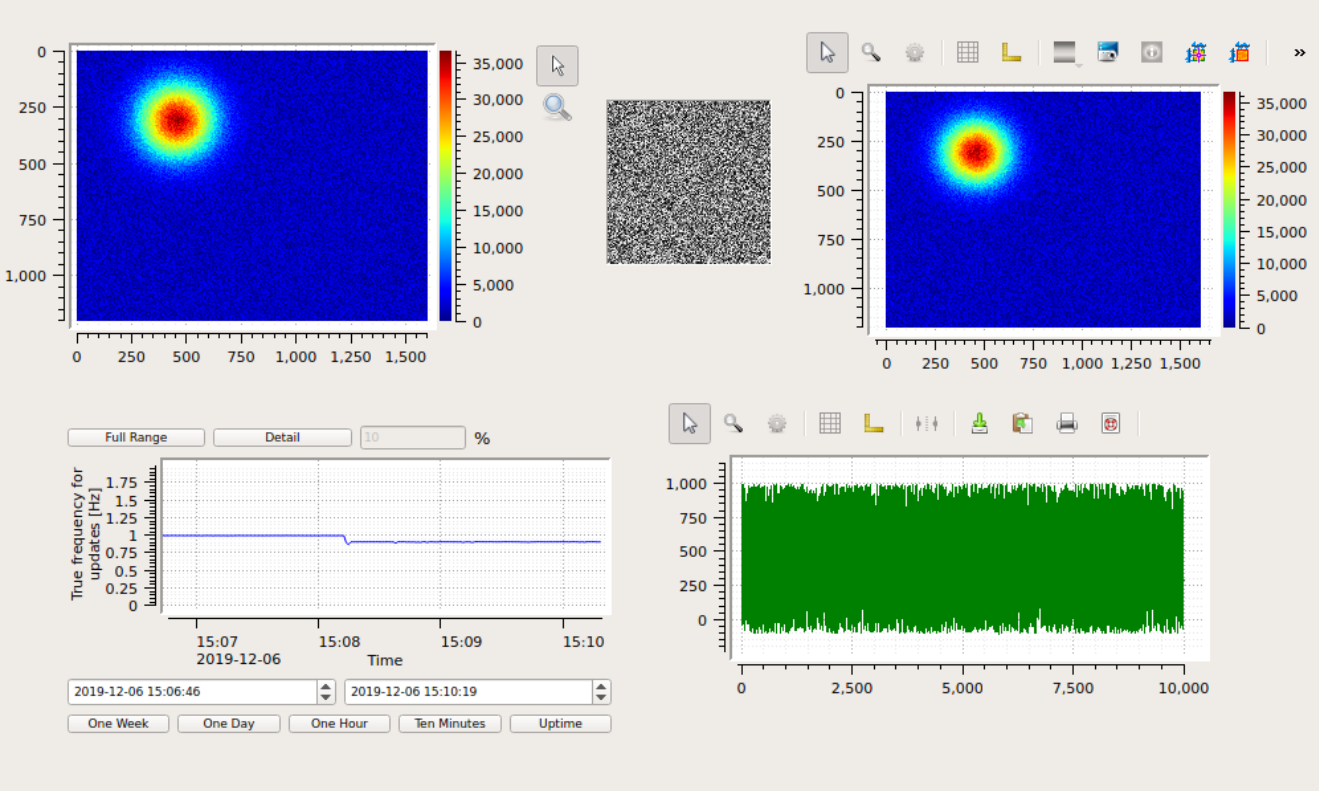


Fig. 13: Old plotting widgets using the library `Qwt` and `matplotlib`.

Widget Table			
Type	Qwt	Matplotlib	PyQtGraph
Vector	Plot	•	Vector Graph
Vector	XY-Plot	•	Vector X-Y Scatter
Image	Aligned Image View	•	Detector Graph
Image	Image Element	•	WebCam Graph
Image	Image View	•	Image Graph
Image	Webcam Image	•	WebCam Graph
Image	Scientific Image	•	WebCam Graph
Float/Bool/Int	•	XY-Plot	Scatter Graph
Float/Bool/Int	•	MultiCurve	MultiCurveGraph
Float/Bool/Int	Trendline	•	Trend Graph

10 #10: Karabo Scene

A number of features have been added in Karabo 2.8.0 to help ease scene development:

1. Scene Grid
2. Selection Indicators
3. Snap to Grid

10.1 Scene Grid

The scene **Design Mode** is now displayed with grid. This can be used as guide for aligning scene items. It has a preset dimension of 10 px x 10 px.

10.2 Selection Indicators

Hovering an item

Hovering the mouse cursor on the scene *Design Mode* will now show an indicator of the current hovered item, which is a blue rectangle that denotes the item boundaries. This aims to make the selection of scene items easier.

Note that this respects the object arrangement, which determines which are in the front or the back. This arrangement can be modified by the *"Bring selected items to front"* and *"Bring selected items to back"* tool buttons.

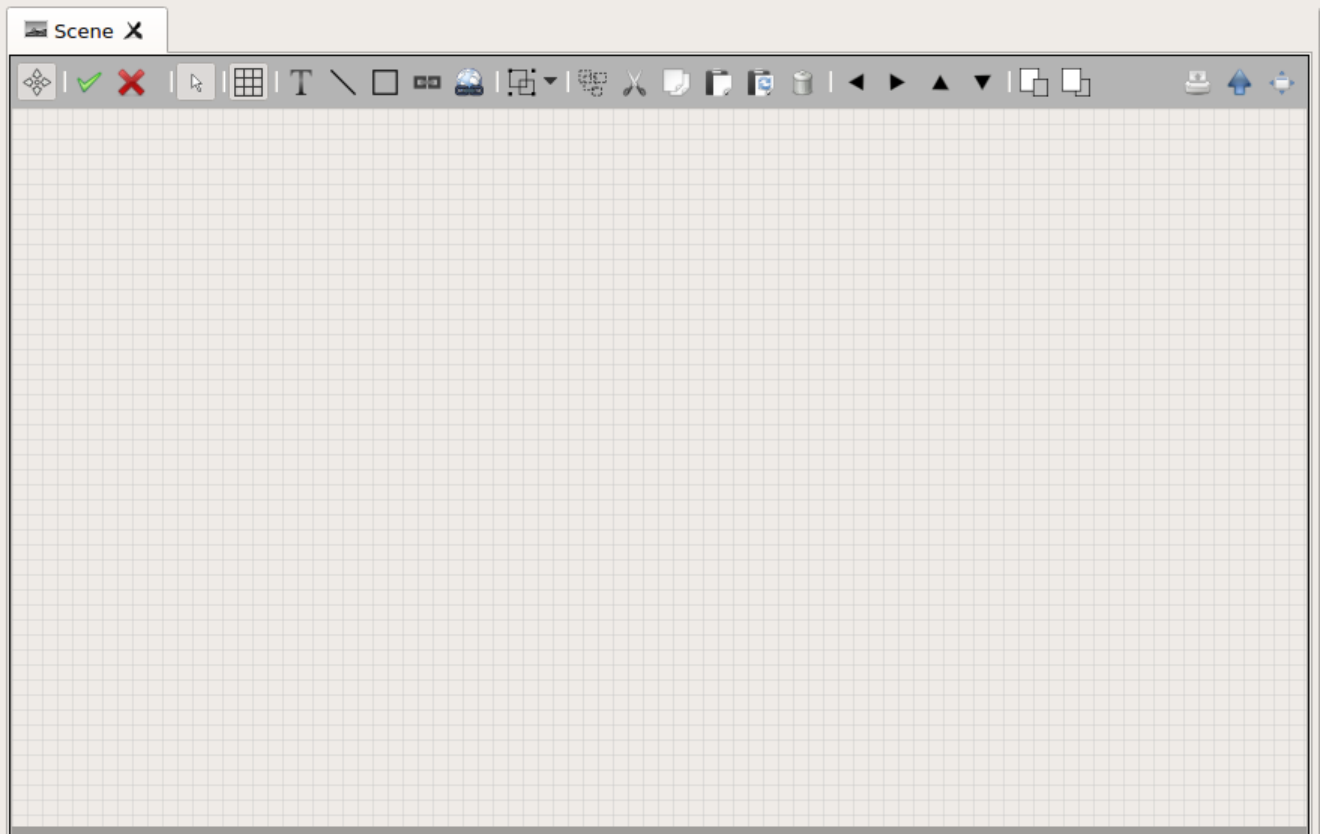


Fig. 14: Scene grid as shown in the Design Mode

Fig. 15: Hovering the Icon Command widgets

Fig. 16: The two icon command widgets are stacked at some area, and hovering on that point will indicate the widget at the front, which is the widget with the “Play” icon.

Selecting a scene item

Selecting a single item can be done by clicking the desired object. The hover rectangle will turn into a selection rectangle, which is displayed as a dashed rectangle with eight handles around it. The selected item can be moved by hovering the mouse inside the selection and dragging to the desired location. It can also be resized by dragging the edge of the selection rectangle to the desired size.

Fig. 17: Selecting, moving, and resizing a widget

Selecting multiple items

Selecting multiple objects can be done by dragging an area which contains the desired items. These items can be moved together by hovering the mouse inside the selection and dragging to the desired location. On the other hand, multiple items cannot be resized together, and is indicated by the grayed out handles of the selection rectangle.

Fig. 18: Selecting and moving multiple widgets at the same time

10.3 Snap to Grid

Scene objects also now snap to the nearest grid location when they are moved or resized via dragging.

Fig. 19: Moving and resizing a widget snaps to the grid

This behavior is enabled by default, but it can be disabled and re-enabled with the *"Snap to Grid"* tool button.

Fig. 20: Toggling the Snap to Grid behavior

11 #11: Layouts and Shape

Two scene elements have been fixed in Karabo 2.9.0. These are:

1. Horizontal and vertical grouping
2. Line shape

11.1 Horizontal and Vertical Grouping

Grouping allows two or more scene items to be collected into a single 'grouped' object. This is useful when arranging scene items in a uniform fashion along the desired orientation, which can be done by the following:

1. Go to design mode
2. Select multiple items by dragging on an area containing the items of interest
3. Click the dropdown button in the "Group" tool button
4. Select **"Group Horizontally" / "Group Vertically"**

The resulting group object can be moved and resized, as if it is a single scene item. As a consequence, all of the contents will be moved and resized at the same time.

Fig. 21: Grouping command widgets horizontally

Fig. 22: Grouping command widgets vertically

When grouping the scene items in this manner, these items will be initially resized to their default geometry. This is to help ensure the uniformity of the group resize.

11.2 Line Shape

Shapes are a great addition when creating unique and appealing scenes. The Line Shape has now been fixed and can now be moved and resized with more ease. Creating horizontal and vertical lines is also now easier with the help of *"Snap to Grid"*.

Fig. 23: Creating, resizing, and moving the line shape

12 #12: ROI Items in Graphs

12.1 ROI Items in the Graph Widgets

The new Graph Widgets offers region of interest (ROI) items for marking and selecting plot regions. These can be hidden, shown, and drawn using the ROI tool button found in the plot toolbar.

Plot Graph and Crosshair ROI

Plot Graphs have Crosshair ROI, which can be used to mark a point in the plot. It can be added with the following steps:

1. Go to **scene control mode** (not design mode)
2. Click the dropdown button next to Crosshair ROI button
3. Select *"Draw Crosshair ROI"*.
4. Click on the desired location in the plot to draw the crosshair

The Crosshair ROI displays its coordinates in a text box right below it.

Image Graph and Rectangle ROI

Image Graphs have Rectangle ROI, in addition to the Crosshair ROI. It can be added in the plot with the following steps:

1. Go to scene control mode (not design mode)
2. Click the dropdown button next to Rectangle ROI button
3. Select "Draw Rectangle ROI".
4. Drag a rectangle in the plot to draw the ROI

The Rectangle ROI displays its geometry in a text box right below it.

The ROIs in the Image graph are capable of automatically slicing the image data and can be used in conjunction with Aux Plots, which is also a provided tool.

The ROI items can also be selected and moved around. They can also be removed from the plot by right-clicking them and select "Remove ROI". Multiple ROIs can also be added, with a bolder line width for the currently selected ROI.

As of Karabo 2.9.0, only the currently selected ROI displays the text box containing its properties. This is to help minimize innate resize problems and to reduce the items inside the plots.

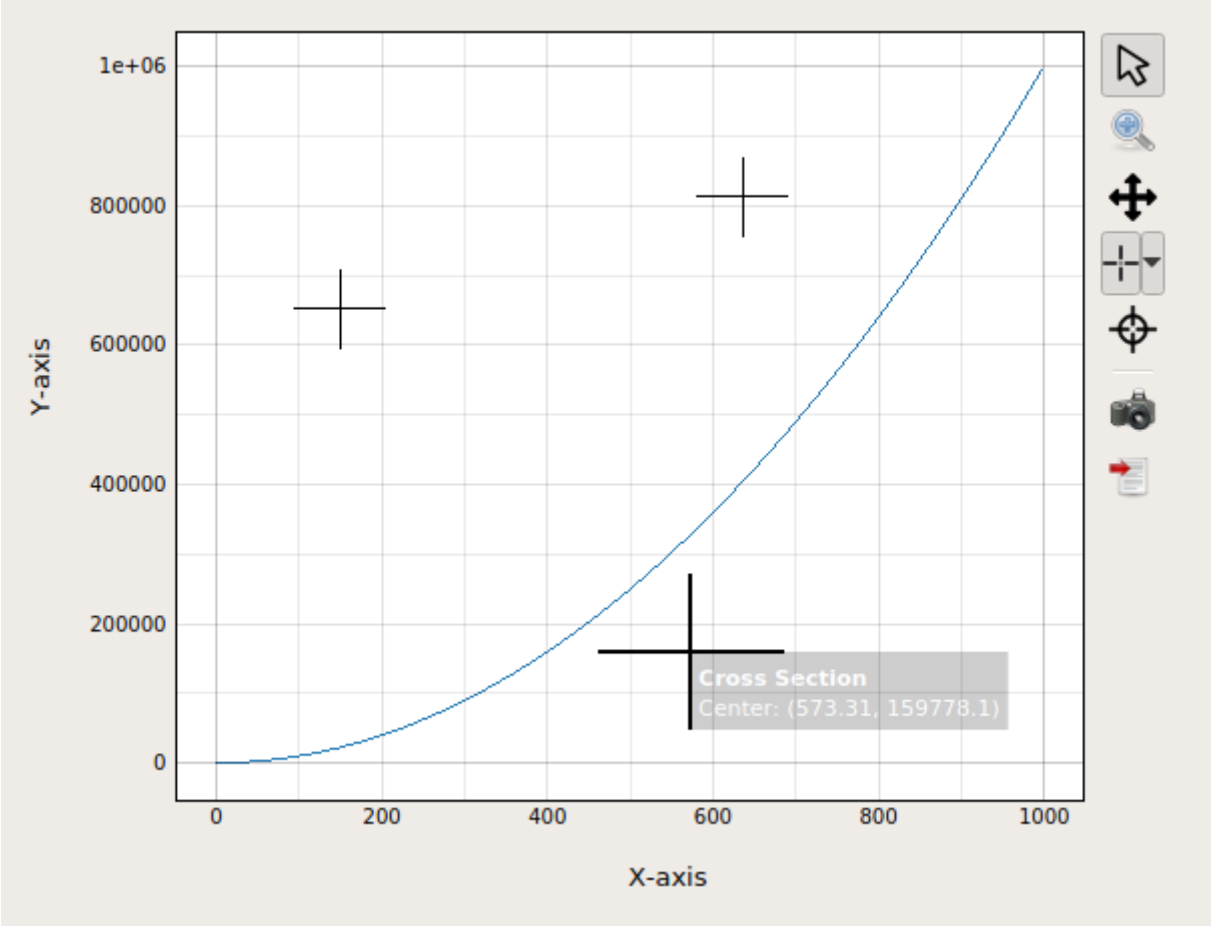


Fig. 24: Crosshair ROIs in a Vector Plot

Saving ROI Items in the Scene

The ROI items can be saved in the widget and can be loaded on the next scene opening.

For the Plot Graphs, this can be done by the following:

1. Go to **scene design mode**
2. Right-click the graph widget
3. Hover *"Vector Plot: Properties"*
4. Select *"Set ROI"*
5. Save the project

This saves the location of the ROI items and their visibility. For instance, if a user added three ROIs and then hid them from the plot before setting them and saving the project, the next scene opening will load the three ROIs, but they are hidden initially. One has to toggle the "Show ROI" to make them visible again.

For the Image Graph, the process is very similar to the one above :

1. Go to **scene design mode**
2. Right-click the graph widget
3. Hover *"Vector Plot: Properties"*
4. Select *"Set ROI and Aux"*
5. Save the project

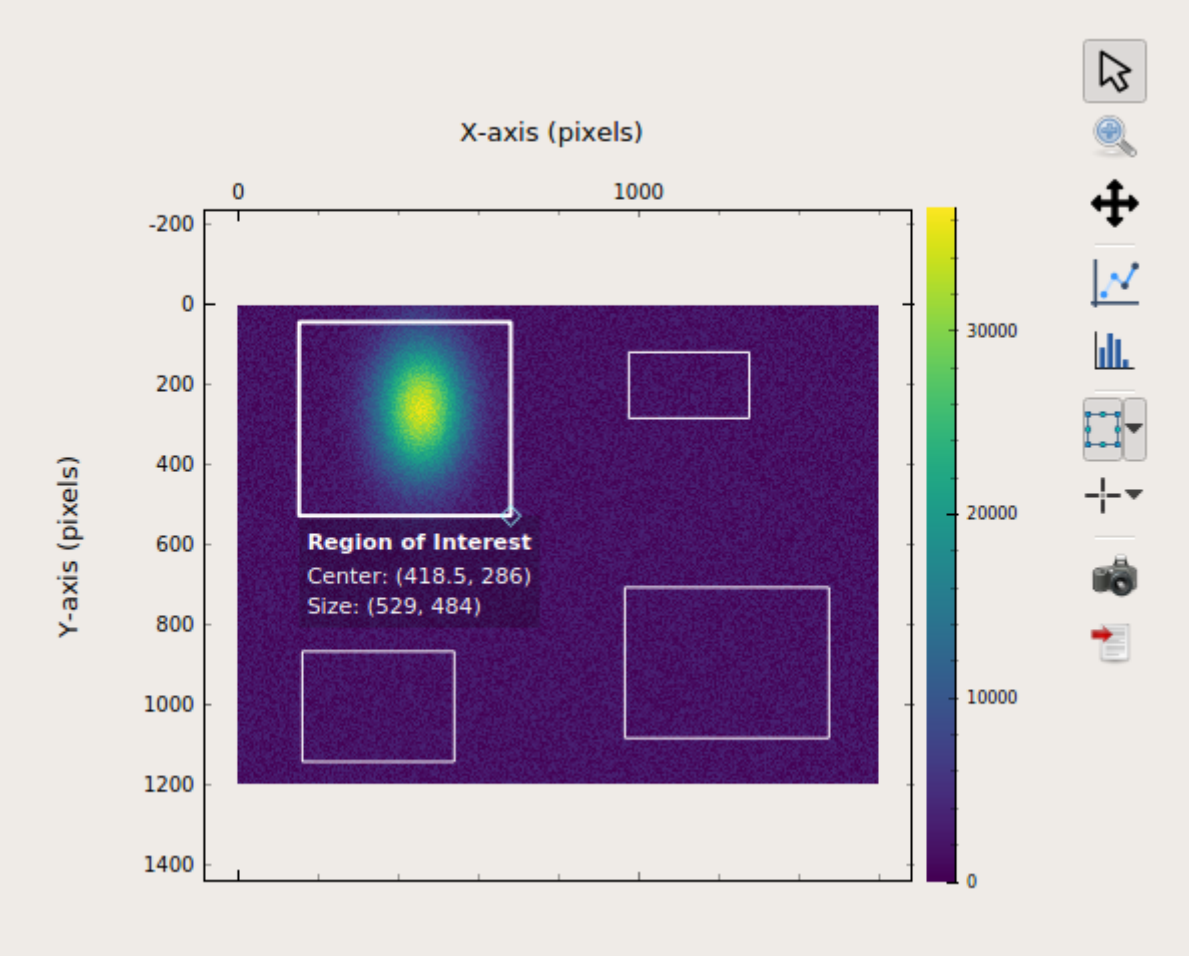


Fig. 25: Rectangle ROIs in an Image Graph

Fig. 26: Aux Plots gets the data to be analyzed from the ROI slices

This saves not only the ROI geometry and visibility for both Rectangle and Crosshair ROIs, but also the Aux Plots visibility.

The two-step saving process ensures that the scene and project are not modified with every ROI modification, as the ROIs are manipulated in the scene control mode and not in design mode.

13 #13: Sticker Widget

13.1 New Scene Tool: Sticker

Karabo **2.9.0** received one more scene feature, the `Sticker Widget`. A sticker can be added via the toolbar of the scene.

Fig. 27: Sticker tool in the Scene toolbar

The Sticker Widget is intended to display text of multiple lines, which would be useful for adding notes and user manuals. This is in contrast with the Label Widget, which is designed for displaying only a few words, such as device names and properties. The features of both widgets are similar, with the Sticker widget excelling at handling long text. Scrollbars are shown when

the widget size does not match the text size, either vertically or horizontally. During editing, the edit dialog will take the same size as the widget on the scene.

Fig. 28: Sticker widget available in Karabo 2.9.0

14 #14: Deprecation of Slider

14.1 The Deprecation of the Slider Widget

Karabo is moving forward and will only support a single `Slider` widget with Karabo **2.10.0**, which will be released **13.11.2020**.

The classic slider widget will be deprecated in favor of the `TickSlider`. For a rather smooth transition, the new GUI will reinterpret the scene information of the classic slider and replace it with the Tick Slider.

The `TickSlider` has more features such as the step size configuration and an option to show the value of the current slider position. Hence, the currently set value on the slider can be seen before applying (e.g. pressing *enter*) on the device.

The geometry of the existing widgets are reused on the replacements to preserve the scene layout.

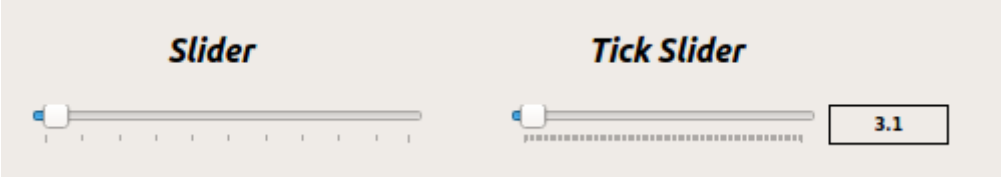


Fig. 29: Slider and TickSlider in Karabo

15 #15: AuxPlots in Image Graphs

15.1 Auxiliary Plots for Image Graphs

The Image Graphs currently have two auxiliary plots (aux plots) that offer quick analysis and visualization on the displayed image data:

1. Beam Profile
2. Histogram

These plots can be toggled from the image widget toolbar:

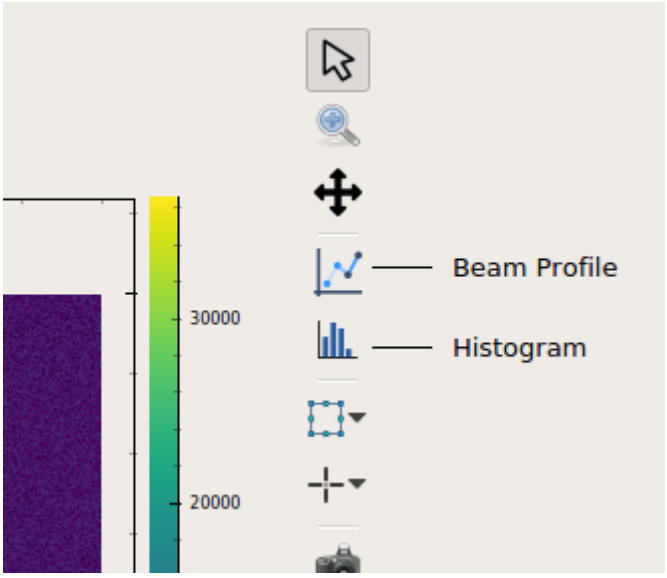


Fig. 30: The aux plots tool buttons in the image widget

Beam Profile

The profile aux plot provides the intensity profile of the image by taking the **average** of the data for each axis. It consists of two plots, which are found on the left and on the top of the image plot for x- and y-axis, respectively.

A Gaussian fit can also be applied to the resulting data to obtain important features such as peak amplitude and position, and full width at half maximum (FWHM). This works best when an evident peak is present.

Fig. 31: The beam profile aux plots with Gaussian fitting enabled

The fitting is disabled by default and can be toggled with the following steps:

1. Go to **scene control mode**
2. Show the *beam profile* aux plot by clicking the beam profile tool button
3. Right-click on one of the profile plots
4. Select *"Gaussian fitting"*

As fitting data is a computation intensive task, enabling this option might make the GUI slow. It is recommended to use it sparingly.

Histogram

The histogram aux plot calculates and visualizes the histogram of the displayed image data. The bin width and count are automatically calculated based on the pixel values and count. It also generates statistics such as the minimum, maximum, mean, mode and standard deviation.

The statistics is enabled by default and can be toggled with the following steps:

1. Go to **scene control mode**
2. Show the *histogram* aux plot by clicking the histogram tool button
3. Right-click the histogram plot
4. Select *"Show statistics"*

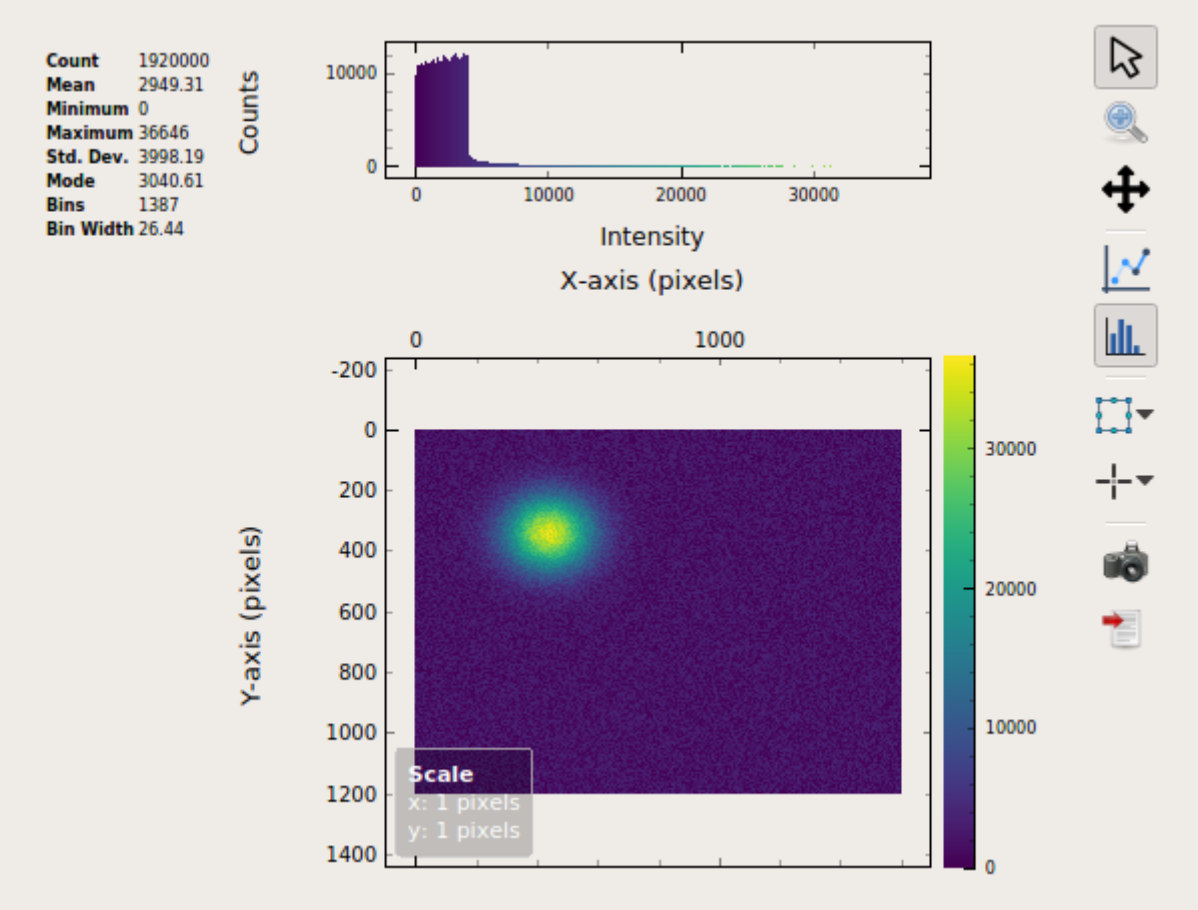


Fig. 32: The histogram aux plots

Saving Aux Plots Visibility

The visibility of aux plots can be saved in the project such that they are shown or hidden by default on the next scene opening. This can be done with the following:

1. Go to **scene control mode**
2. Show or hide the desired aux plot by clicking the consequent tool button
3. Go to **scene design mode**
4. Hover *"Image Graph: Properties"*
5. Select *"Set ROI and Aux"*
6. Save the project

Interaction with ROIs

The region of interest (ROI) tool can be used in conjunction with the aux plots as it is capable of slicing the image data. The aux plots analyzes the sliced data from the selected ROI, when present.

Fig. 33: Aux Plots gets the data to be analyzed from the ROI slices

Fig. 34: Changing the selected ROI also changes the analyzed data by the aux plots

Limitations of Data Analysis in Karabo GUI

Karabo GUI focuses on providing an interface for controls and data acquisition. It uses the computational resources of the machine where it is run. To ensure smooth and reliable feedback, heavier processes are done with Karabo devices. These devices are run on remote servers, which has more available resources.

The image widget offers quick data analysis. As it is considered a computation intensive task, it is not activated by default and the results cannot be stored. It is recommended to use Karabo devices such as `imageProcessor`.

16 #16: GUI Extensions

16.1 The GUI Extensions

The `Karabo GUI` can also be updated regularly with external widgets with an external package.

The external package is referred to as `GUIExtensions` and targetting non-generic and **tailored** controllers for the operators. Examples of tailored widgets such as for the scantool (Karabacon), a scatter position widget with ellipse, a beam position monitor and the adqDigitizer are shown below.

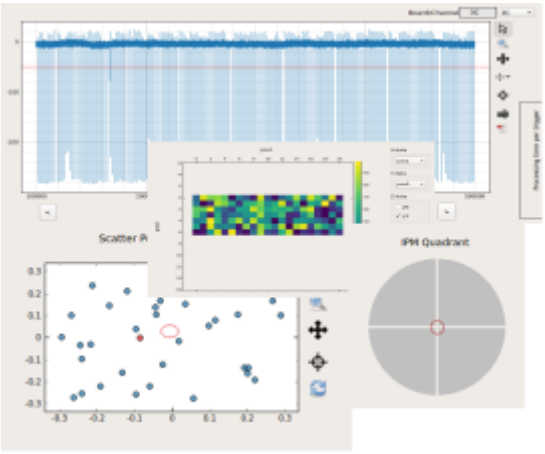


Fig. 35: Overview of the GUI Extensions widgets

The documentation can be found on the European XFEL readthedocs:

https://rtd.xfel.eu/docs/gui-extensions/en/latest/index.html

The development is independent of the Karabo release cycle and can offer quick solutions for the operators on demand. Regarding the client computers in the control room hutches and the Beschleuniger Kontroll Raum *BKR* as well as the *exflgateway* the control deployment system will install the `GUIExtensions`.

The extensions package can be updated via a dialog that can be launched with the *menu bar* of the GUI application.

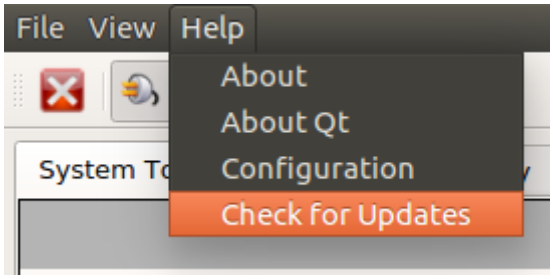


Fig. 36: Check for external updates of the karaboGUI

Using the dialog is illustrated in the video below. The dialog shows the currently installed version and the latest version available. An update procedure will always update to the latest version, which is also the recommended one.

Fig. 37: Updating the karaboGUI Extensions Package

The GUIExtensions package is mandatory for Scantool (Karabacon) usage and highly recommended for the adqDigitizer.

Note: In order to update the package, the operating machine must be within the control or office network.

Another way to install or update the extensions package is to use the command line after activating the conda environment.

```
karabo-update-extensions -l      # Will install the latest available version
karabo-update-extensions 0.4.0   # Specify a tag
```

17 #17: Value Formatting

17.1 Formatted Value Field

The **Value Field** is one of the most extensively used widgets as it is the default for displaying numerical values and strings. In Karabo 2.9, formatting is introduced in this widget to help increase its visibility on larger screens. One can now change its *font size* and *font weight*.

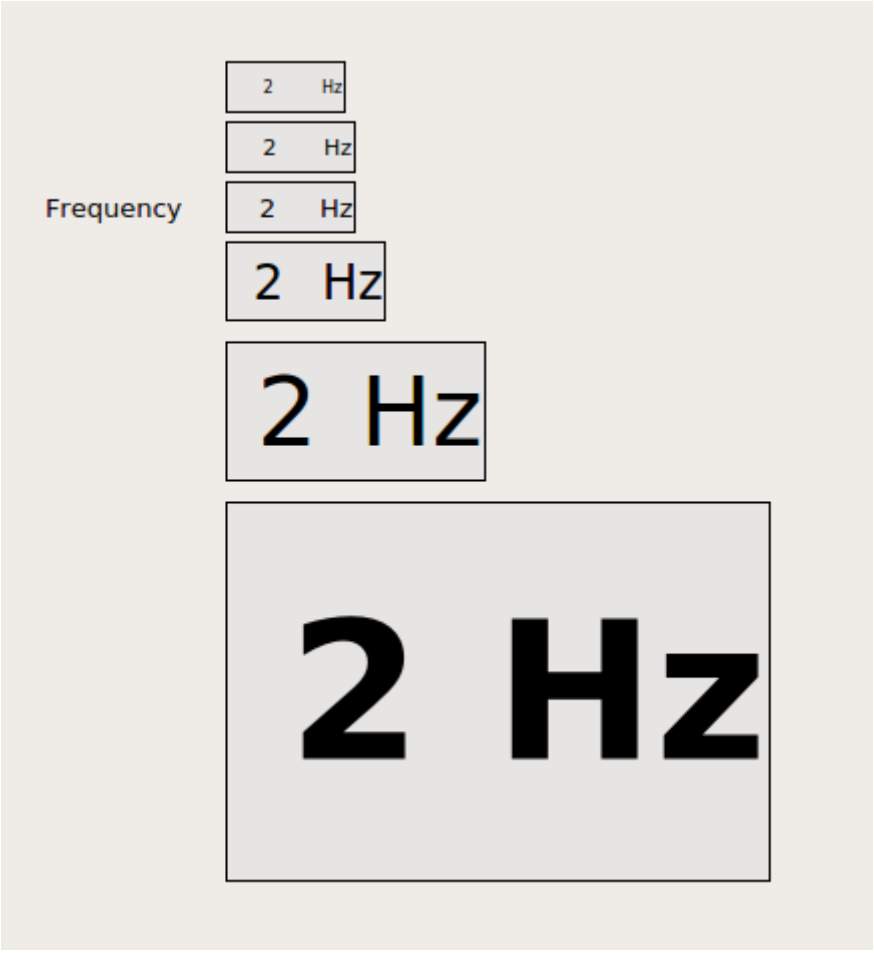


Fig. 38: Value fields with varying font size and weight

This can be down by the following steps:

1. Go to **scene design mode**
2. Right-click the value field widget
3. Hover *"Value Field: Properties"*
4. Select *"Format field.."*
5. Configure the font size and weight as desired

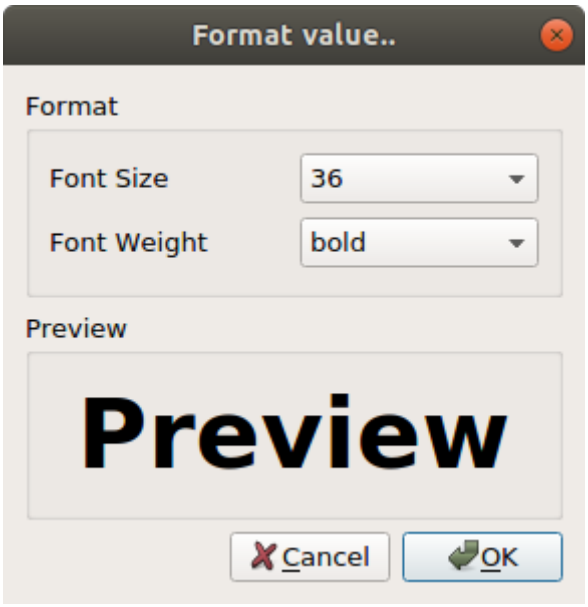


Fig. 39: Configuring the font size and weight

Formatting the value field is optional. It defaults to the application font, which is size `10 pt` and `normal` weight.

18 #18: ViewBox Plot Features

18.1 ViewBox features

The **Plot Graphs** have a few configurable features which are accessible via the so-called `ViewBox` menu which is available by *right-clicking* on the plots widgets.

1. `View All`

Triggering the **View All** option will auto scale the x and y axis of the plot. The auto scale is active until the range of the plot is changed via mouse interaction or configuration.

2. `Reset Range`

The **Reset Range** option will reset the plot range of the plot. This can be either auto scale or a configured range of the plot.

3. `Log X/Y`

If suitable for the plot, a **logarithmic** toggle is provided on the `ViewBox` menu. The setting of this option can be saved in the widget configuration in the project database.

4. `Show data points`

For some plots it can be beneficial to see the **data points**. This is typically the case for the trendline widgets, where this option is activated by default. However, this is an expensive setting as a scatter plot is shown in addition to the line plot consuming cpu processing time. Hence, this option cannot be stored in the widget configuration.

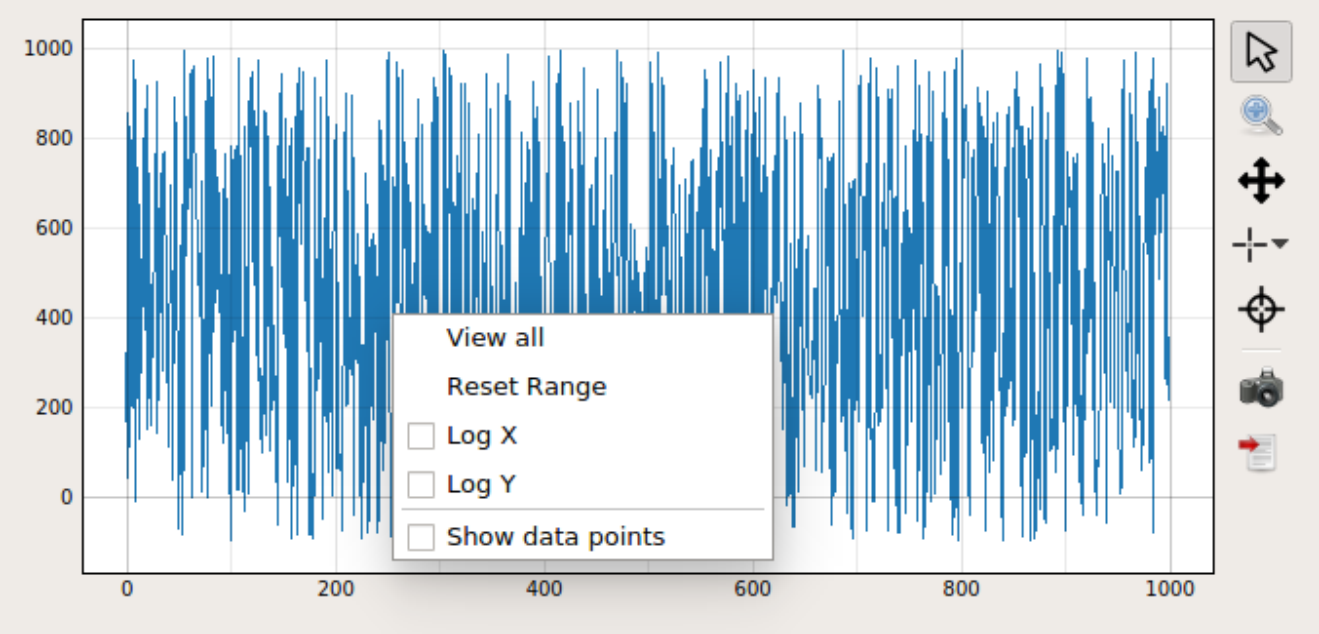


Fig. 40: Accessible **ViewBox** menu of a *VectorGraph*

19 #19: Chooch it!

19.1 Processing lamp

The `Karabo GUI` has a processing lamp in the menu bar on the right side on top of the configurator. It visualizes the difference between the arrival time of a gui server message and the processing of the latter in the GUI client application. In certain circumstances it can happen, that a GUI client receives a lot of messages, e.g. from large project loading. These messages are technically put on a queue in the client application and processed. This is necessary to avoid freezing of the client application. The processing lamp provides a visual indication by color about the processing status.

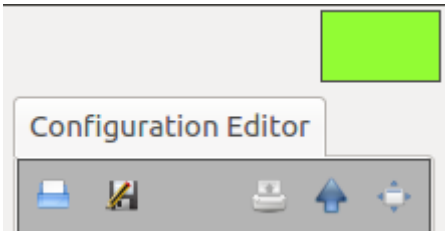


Fig. 41: The processing lamp in the karabo GUI

Color	status	description
	Fine	The Karabo GUI is up to date
	Warning	The updates are at least behind by **2s**
	Alarm	The updates are at least behind by **5s**

19.2 Show processing delay(s)

It is possible to see the exact number of the processing delay.

1. Go to the menu bar on `Help`
2. Open the `About` dialog
3. Click on the dialog and put your mouse cursor in the center of the tunnel picture
4. Type **chooch**

In case you miss typed the cheat, please wait a few seconds a try again. With a succesful *chooch* command, a processing number appears on the processing lamp. **The number (initial as well) will only appear and update when there is activity.**

It is possible to visualize the network and hardware delay of big data. In case this is desired, please follow the steps 1.-3. from above and

4. Type **bigdata**

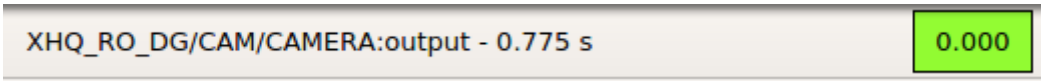


Fig. 42: Visible processing numbers in the karabo GUI

With every visible big data update in the client application, the delay field is updated. The time difference results from the timestamp of the image and the stamp when the image has been processed in the client (a few ms). Hence, this number shows the network and hardware latency.

20 #20: Configurations

20.1 Project Configurations

Karabo stores device configurations in the so-called Karabo *projects*.

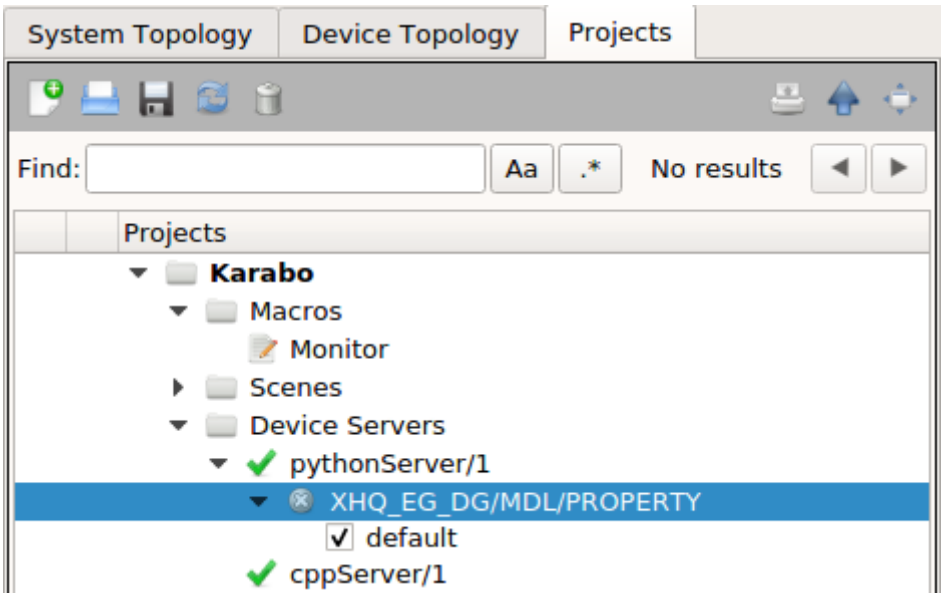


Fig. 43: Project device with a default configuration

The software configurations can be altered **offline** via the **Configurator** before device instantiation. The project will become modified and thus the changes can be saved in the project data base. In reality, it may happen that some devices are rejecting a software configuration if the hardware resident configuration is not editable on instantiation, e.g. PLC devices.

Technical: Why persist OFFLINE configurations?

Many devices in the control system have a dynamic structure (schema). This device structure depends on data unavailable to the offline device when instantiated.

For example:

- A device that builds its own structure from the hardware configuration, or firmware version (e.g. MPOD).
- A device retrieves a system definition from the hardware at connection (e.g. PLC devices).
- A device with a table element for an extensible definition (e.g. Beam Imager). After instantiation the device building itself up from the table element.
- A middlelayer device connects to driver devices to fetch their structure and configurations and then builds itself up.

Online device configurations are likely to conflict with the offline device structure (schema).

Configuration From Past

In Karabo, all devices are archived while they are online. Every property change is send to the datalogger devices and stored (disk and/or database). If the changes are not older than ~a month, they can be retrieved and viewed. Either via the trendline widgets for single properties or via the **Configuration From Past** feature. This can be done via the following steps:

1. Right-click the **Device** object. A context menu will show up.
2. Select **Get Configuration** option. A *Configuration Timepoint* dialog will appear.
3. Supply the **Timepoint** (date-time) of the configuration to be retrieved. For convenience, the user can utilize the calendar on the dropdown menu and/or use the preset time buttons.

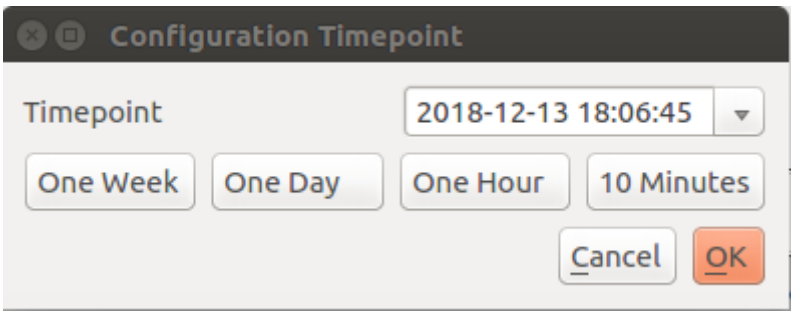


Fig. 44: The configuration from past dialog

After the configuration request a notification window will pop up once the data logger replied. The operator gets a notification about the failure or success. The reply might take a while as the configuration has to be searched. If the configuration request is successful, it will appear in the configurator. For an online device, the respective changes are highlighted with a blue box and need to be applied. In case of an offline device, the retrieved configuration is merged into the offline configuration.

21 #21: Plot Graph Transformations!

21.1 Plot Graph Transformations

Vector Graph plots may provide an option or mechanism for the x-axis transformation in case an indexed x-axis is not desired.

These options can be accessed by the following actions:

1. Go to **scene design mode**
2. Select the Vector Graph and *right-click* for the context-menu
3. Hover *"Vector Graph: Properties"*

4. Select *"X-Transformation"*

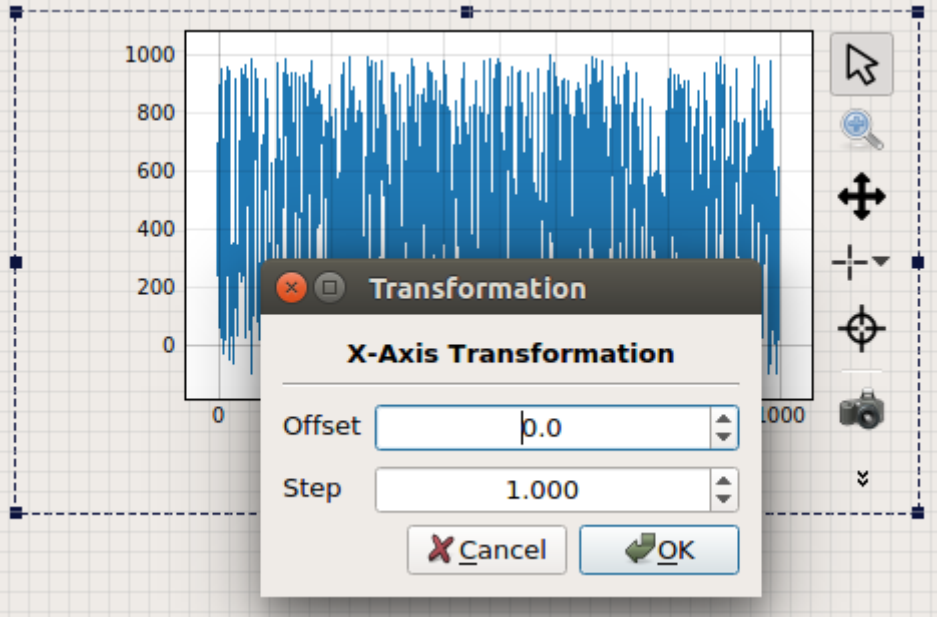


Fig. 45: Transformation dialog of the VectorGraph plot

The x-axis can then be configured with an offset (startpoint) and the binning between indexes.

Another possibility is offered by using the `Vector XY Scatter` or `Vector XY Graph`. If the device provides a vector to build the x-axis, please drag the property from the `Configurator` on the scene and create the controller widget. This will show a blank plot with the dragged property as the x-axis. Afterwards, other vector properties can be dragged in **scene design mode** on the widget to fill the displayed y properties.

22 #22: Plot Graph Improvements

The plot widgets have been improved since the last major release (2.9.0).

1. Decrease in whitespaces around the plots
2. Axis tick spacing enhancement for small plots
3. Log-scale for Vector Bar Graph

These updates are now included from **Karabo GUI 2.9.6**.

22.1 Decrease in whitespaces around the plots

There are cases that there are unintended whitespaces around the plot widget, specifically where the axis labels are located. This has now been removed.

It can be seen that there still have some spaces on the y-axis. This is a buffer for long labels (e.g., `1000000`) and will help minimize widget resize for value updates that involve drastic change in tick label sizes. For instance, a y-axis with range `0 - 10` changing to range `10000 - 50000`, even though differing in number of digits, will not trigger widget resize.

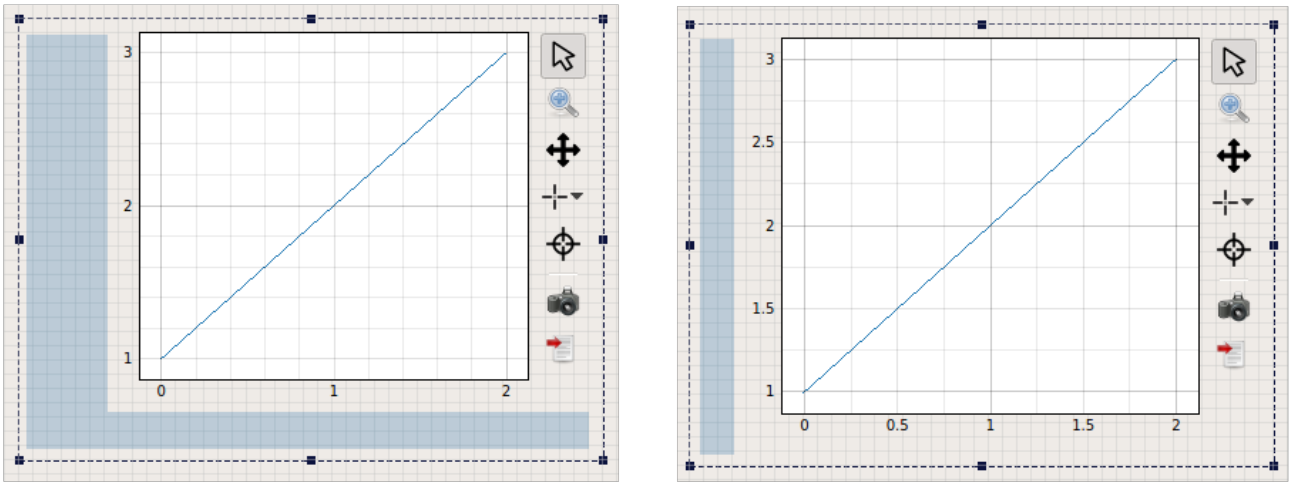


Fig. 46: The plots from previous releases have noticable whitespaces (left). This has now been minimized on the new release (right).

22.2 Axis tick spacing optimization for small plots

Small plots had problems displaying the axis ticks, especially if the labels are going to be too crowded on the available axis space. We have improved the tick spacing calculation such that the labels are shown despite the limited space.

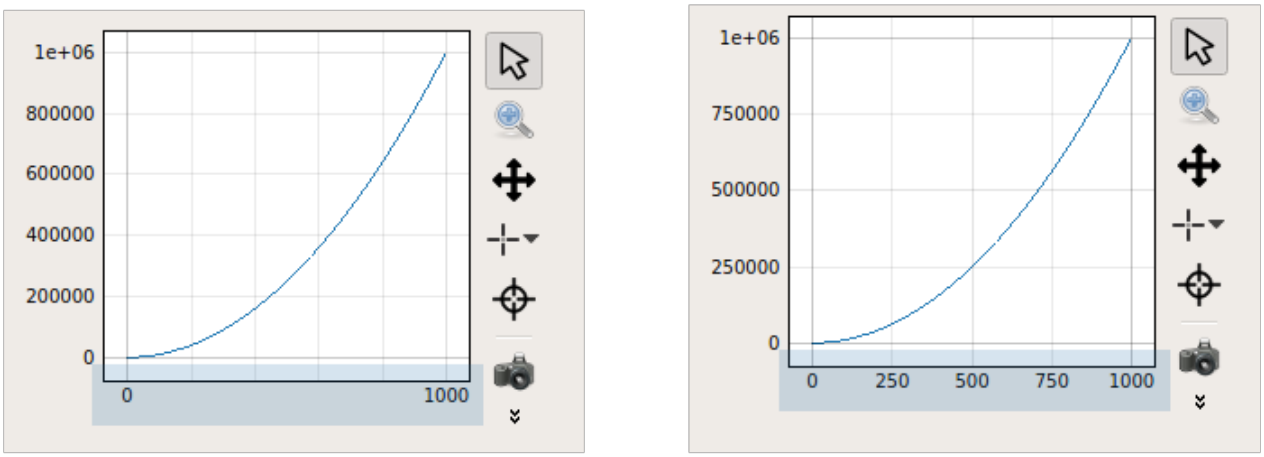


Fig. 47: The axis ticks from previous releases does not display all the axis labels when there is not enough space (left). Calculation of the tick spaces has been improved on the new release to optimally show tick labels.

22.3 Log-scale for Vector Bar Graph

The Vector Bar Graph has been optimized and can now be displayed with a log-scale y-axis.

Please note that no bar will be shown for invalid values `(x <= 0)`

These improvements were made possible by user feedback, so please feel free to let us know what you think! Should you have suggestions or feature requests, send us a mail at controls-integration@xfel.eu.

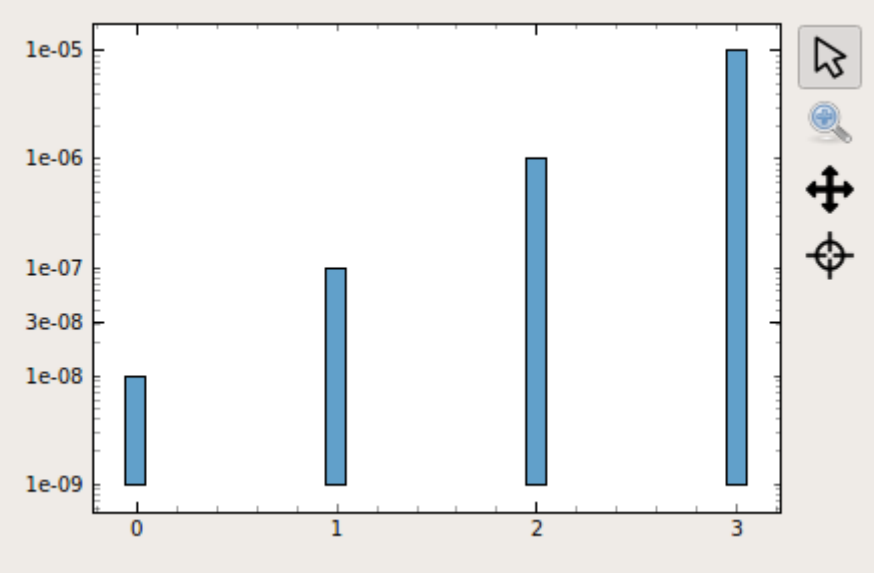


Fig. 48: Vector Bar Graph shows a vector in log-y scale.

23 #23: Karabo GUI Installation on Various Platforms

Did you know that the Karabo GUI can be installed not only in Linux PCs, as those found in the control rooms, but also in Windows and Mac OS PCs? This is possible with `conda`, a cross-platform package and environment system.

One of the benefits of installing Karabo GUI on an own machine rather than using a remote desktop software is the usage of the native user interface. The user interaction will be smoother for most of the cases as it will not be channelled through an internet connection. Only data needed to interact with the control system will be channelled through the internet connection. Files from the GUI will also be stored locally. For instance, saving an image widget snapshot will be done on the local machine and not on the remote machine, thus one now avoid retrieving this file via a remote transfer.

23.1 Installation Instructions

The installation consists of majorly three steps:

1. Downloading `miniconda`.
2. Setting up a `conda` environment and configuring a couple of `conda` settings
3. Install karabogui from the created environment.

The detailed instructions can be found here:

https://rtd.xfel.eu/docs/karabo/en/latest/installation/gui.html

Afterwards, the GUI can now be run with the following commands on the terminal (**Anaconda Prompt** on Windows or **Bash** on Linux or MacOS):

```
conda activate <name of the Karabo environment>
karabo-gui
```

23.2 Connecting to the GUI servers outside of DESY Network

At these time that users are mostly working off-site and are not connected to the DESY network, establishing a connection to GUI servers that are only accessible inside of the network can seem daunting. Such connection can be achieved by the following instructions:

Windows: SSH connections in Windows can be established by using an external tool (`PuTTy`). The instruction can be found here:

https://rtd.xfel.eu/docs/karabo/en/latest/installation/windows.html

Alternatively, one can enable running SSH commands in Windows 10 from the OpenSSH instructions at https://docs.microsoft.com/en-us/windows/terminal/tutorials/ssh. Afterwards, the following steps below can be done instead.

Linux/MacOS:

(These steps can also be done in Windows 10 with an enabled OpenSSH.)

1. Run the following command in the terminal to establish the SSH connection to the desired GUI server (for instance, `SA1`):

```
ssh -L 38080:localhost:38081 <desyusername>@bastion.desy.de -t ssh -L 38081:sa1-br-sys-con-
↪gui1:44444 exflgateway
```

Note that `38080` and `38081` are arbitrary port numbers, so feel free to use an unused port between `1024-49151`.

2. Use the assigned port (in this case, `38080`) in the Karabo GUI.

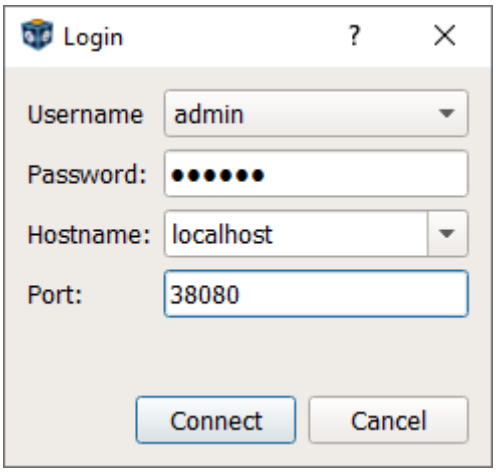


Fig. 49: Using the port `38080` to connect to `SA1`

Our aim is to continuously make the Karabo GUI more accessible. Social distancing practices have forced colleagues to work on laptops and home computers with diverse operating systems. The installation through the `conda` packaging system enables the Karabo GUI installation on Windows, MacOSX (from 2.9.0) and multiple Linux OSs using the x86_64 architecture. Your feedback is greatly appreciated! Please send your feedback to controls-integration@xfel.eu.

24 #24: States and their Colors

24.1 Traffic Control Lights

Most control systems provide many essential information in an abstract property, the so-called **state**. This hardware and software state property aims to describe at best the actual configuration of a device or system for the operator. For example:

- Did a hardware operation encounter an error?
- Is a device able to perform an action, e.g. can a motor move?
- Is currently an operation in progress, e.g. a camera is acquiring?
- ...

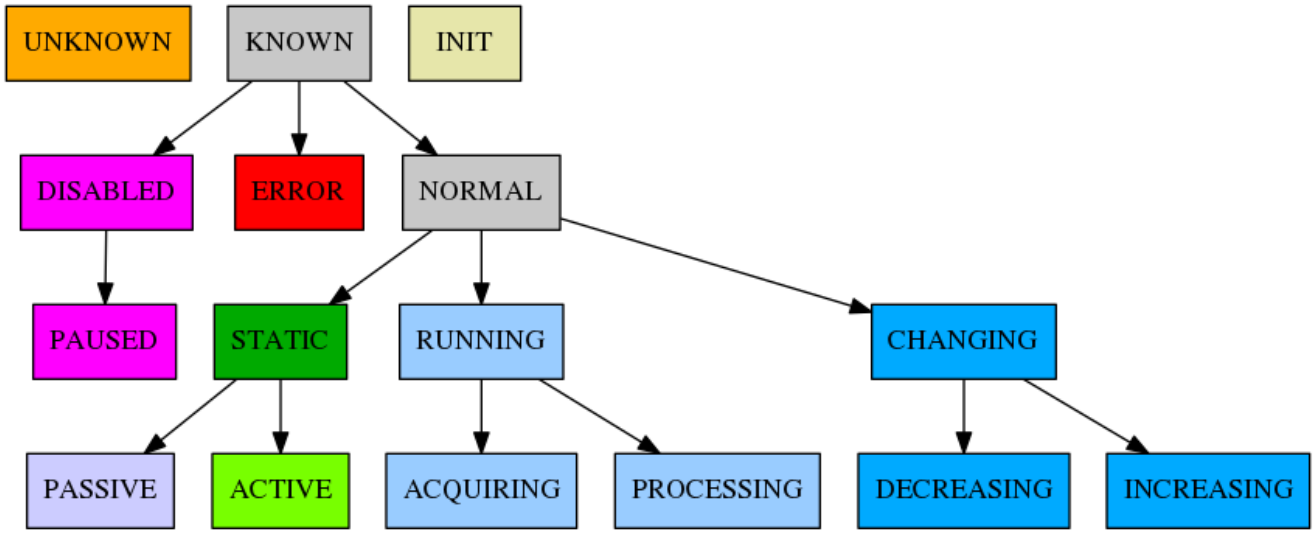


Fig. 50: Karabo states and their families

In the overview above the existing main state families in Karabo are shown.

With a lot of effort Karabo tries to follow a few principles:

- The *ERROR* state is mostly reserved for hardware errors. (*RED*)
- The *UNKNOWN* state describes mostly a missing hardware connection (software error) (*ORANGE*)
- The *CHANGING* state describes the undergoing change of a hardware state. Hence, it is often used as mediator between two states (*BLUE*)
- The *RUNNING* state describes an ongoing operation, e.g. a camera is acquiring (*LIGHT BLUE*).
- The *DISABLED* state describes the situation when a device cannot be operated under the circumstances. This may be an interlock.
- The *ACTIVE* state family describes a state in which operation is enabled (see note on valves below).
- The *PASSIVE* state and its derivatives describes a state in which operation is not enabled.

The state color description follows further:

- The **X-ray beam is on sample** policy leads to having **shutters** and **valves** having a green state when the beam is passing.

The graphical user interface representation offers two widgets. The *Generic Lamp* and the *State color field*.

Although the state colors were chosen with color vision deficiency in mind, sometimes it is difficult to differentiate without a reference. A clearer way to display a state information is using the state color field widget, which offers the option to display the state string.

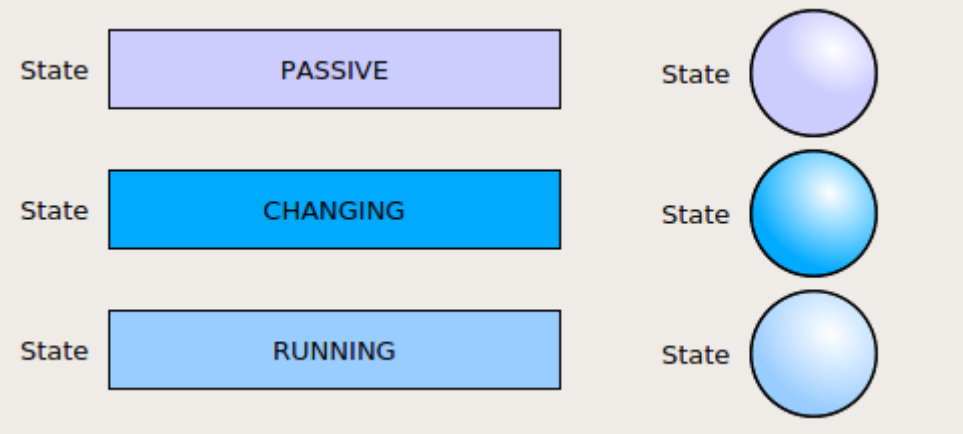


Fig. 51: The *state color field* widget and the *generic lamp* widget

25 #25: Configuration Development

25.1 ConfigurationManager

A new core device, the **Karabo Configuration Manager** is implemented in Karabo **2.10.0**.

The configuration manager device is able to manage configurations of multiple devices. It can save online configurations (keys, value) and the schemas (keys, attributes) of the devices it manages. Hence, a so-called online snapshot with the full device information, e.g. injected properties next to the general static schema, can be stored.

The configuration manager is able to instantiate a device with a given configuration and attributes.

25.2 New Features - Karabo GUI

- Device configurations can be tagged via the device topology and project panels
  - Configuration snapshots **cannot** be deleted or modified
  - Configuration snapshot names are unique for a device

Fig. 52: Get a configuration by a `name` from the configuration dialog

- Device configurations can be retrieved via the device topology and project panels
  - Compare configuration with existing in *Configurator* (as before)
- Configuration dialogs are **not anymore blocking** operator input and can be multiple times open

Fig. 53: Get a configuration by a `time` from the configuration dialog

In certain scenarios devices change over time. Even worse, they might change their schema, e.g. the description of their properties. A bool value might become a string, a table element gets one more column etc. pp. These errors will not be silent anymore but are notified.

Fig. 54: Configuration violations in the Karabo Gui are now notified

Component Management (generic)

The configuration of components consisting of several devices can be managed by so called **Component Manager** devices and their *scenes*.

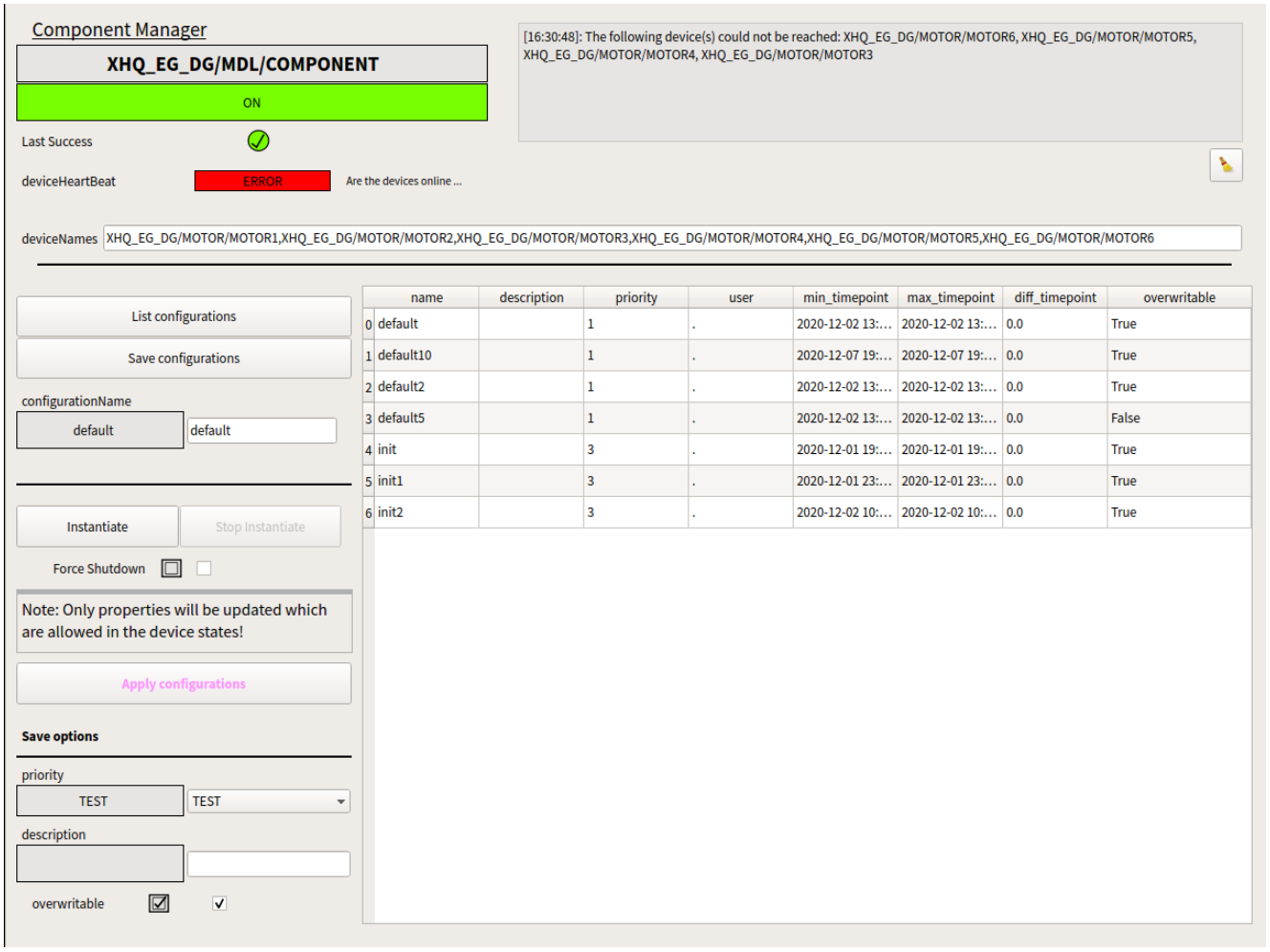


Fig. 55: Scene of a component manager handling a simplified component (six motors)

The component manager is still under active development:

- Quick view if all devices are online
- Configurations can be saved for all devices under same name (snapshot)
- Devices can be instantiated from configuration set by unique *name*
- Configuration set can be applied to all devices as bulk
- … more in the future!

New functions: MDL / ikarabo

New *synchronized* (coroutines) functions are available to communicate with the configuration manager:

- **instantiateFromName(device, name=None, classId=None, serverId=None):**
  - device: The mandatory parameter, either deviceId or proxy
  - name: Optional parameter. If no *name* is provided, the latest configuration is retrieved with priority 3 (INIT).
  - classId: Optional parameter for validation of classId
  - **serverId: Optional parameter to overwrite the server where to instantiate**
    (otherwise taken from stored configuration)
- **saveConfigurationFromName(devices, name, description='', priority=1):**
  - The parameter *devices* can be a Karabo *proxy*, a list of deviceIds, a list of proxies or a mixture of them. It can be as well a single deviceId string.
  - The description text is by default empty.
  - The priority default 1 is the lowest available (1-3: TEST - COMMISSIONED - INIT).
- **listConfigurationFromName(device, name_part=''):**
  - Optional: A *name part* can be provided to filter the list of configurations
  - **Returns a list where each configuration item is a Hash containing:**
    * name: the configuration name
    * timepoint: the timepoint the device was taken
    * description: the description of the configuration
    * priority: the priority of the configuration
    * user: place holder, for future user information (planned)
- **getConfigurationFromName(device, name):**
  - Get the configuration *Hash* of a deviceId or proxy with a given *name*
- **getLastConfiguration(device, priority=3):**
  - Get the last configuration *Hash* of a deviceId with given *priority*

25.3 Quick tools - Configuration

Additional quick tools are provided for the command line interface (CLI) to walk through

- **timing functions to facilitate easy timestamping, where *number* can be floating point or integer**
  - daysAgo(number)
  - hoursAgo(number)
  - minutesAgo(number)
- **compareDeviceConfiguration(device_a, device_b):**
  - Compare device configurations (key, values) of two devices
  - The changes are provided in a list for comparison:

```
-> h = compareDeviceConfiguration(device_a, device_b)
-> h
-> <disableEpsilonFeedback{}: [True, False]>
```

- **compareDeviceWithPast(device, timepoint):**

- *timepoint*: The timepoint to compare
- Compare device configuration (key, values) between present and past
- The changes are provided in a Hash of lists for comparison:

```
changes = [PRESENT | PAST]

-> h = compareDeviceWithPast(device, minutesAgo(2))
-> h
-> <disableEpsilonFeedback{}: [True, False]>
```

25.4 Configuration Contract Changes (Developers)

- A karabo configuration consists of *keys* and *values*. A *schema* consists of *keys* and *attributes* (mininum, maxmimum, units, etc.). The karaboGUI was doing a rather difficult work to merge both configuration and schema together as configuration. In the future, the karabo GUI configuration will contain only *keys* and *values*. This full change is aimed for end of 2021, hence, there is a 1 year notification period, together with the backward compatiblity period of 1 release (6 months), there is a total removal period of ~ 1 1/2 years. In the future, certain *attributes* must be exposed as device property if they are supposed to be changed. Since this feature is rarely used, no big effect is expected after a survey. In contrast, important attributes are visible in general and not changeable for devices were they are not supposed to be changed (validation). They are by default writable which is a safety concern. With Karabo **2.10** the *attributes* **minInc, maxInc, minExc, maxExc, absoluteError and relativeError** cannot be set anymore from the GUI client. With Karabo **2.11** the *attributes* **unitSymbol** and **metricPrefixSymbol** are not configurable.
- A Table element must specify default values in their columns. In the future, if the table changes by adding a column and no default value is provided, the whole previous configuration is declared invalid, as the GUI is not able to reconstruct a proper table configuration!
- An **assignment internal** declaration of a device property should only be used in combination with access mode **init_only** . It provides the information that this value cannot be set from outside, but maybe set from internal. These assignment internal properties will also NOT appear in any applied configuration. Examples of assignment internal properties are *__deviceId__* and *__serverId__*. Those two properties are set by a device server as part of the device instantiation process.

26 #26: Daemon Services

26.1 Technical Background

The Karabo control system uses **daemon tools encore**, which is a widely used collection of tools for managing UNIX services. In more detail, karabo *device-servers* are managed by daemon tools and they are typically restarted by the daemon supervisor if shutdown.

The daemon services that run Karabo device servers can be managed through web interfaces independent of the Karabo communication.

The *KaraboDaemonManager* is a device that interfaces this service management with Karabo and the KaraboGUI.

26.2 Daemon Control - Karabo GUI

If the *KaraboDaemonManager* is online, an extra button *Service Manager* will appear in the *System Topology* panel.

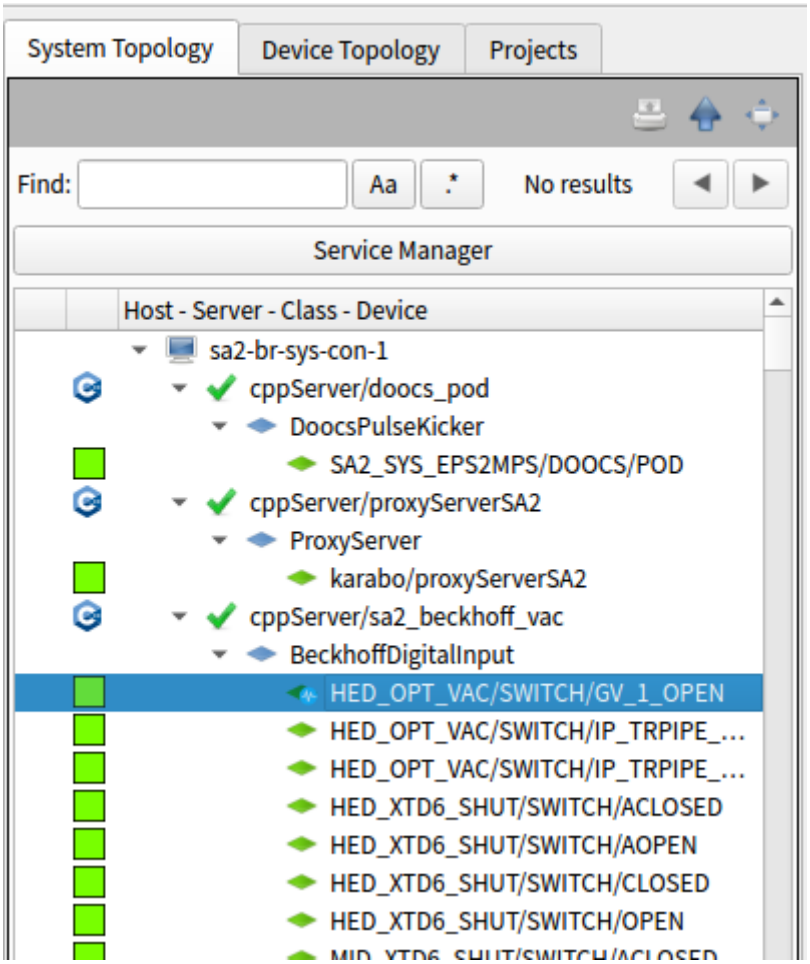


Fig. 56: Retrieve the scene of the *KaraboDaemonManager* by clicking the *Service Manager* button

Clicking on the button will request an overview scene of the *KaraboDaemonManager* device.

The retrieved scene will be shown in an undocked view and provides a table element with all karabo device servers found in the Karabo Topic.

The status of the device servers is highlighted with **green** and **red** colors and the following actions can be requested.

- **Start**: Start a device server. If a device server is already started, nothing happens.
- **Stop**: Stop a device server. The server is asked gently to stop. It will not restart, but remain offline. Press *Start* to start the server again.
- **Kill**: This will force the server to be killed, even if it does not react on *Stop* anymore. The device server will commonly **restart** and come back online. If the server does not restart on its own, you may use the *Start* button.

Requesting a command multiple times does not hurt the system and killing (restarting) multiple times after each other is fine as well, don't hesitate to try this.

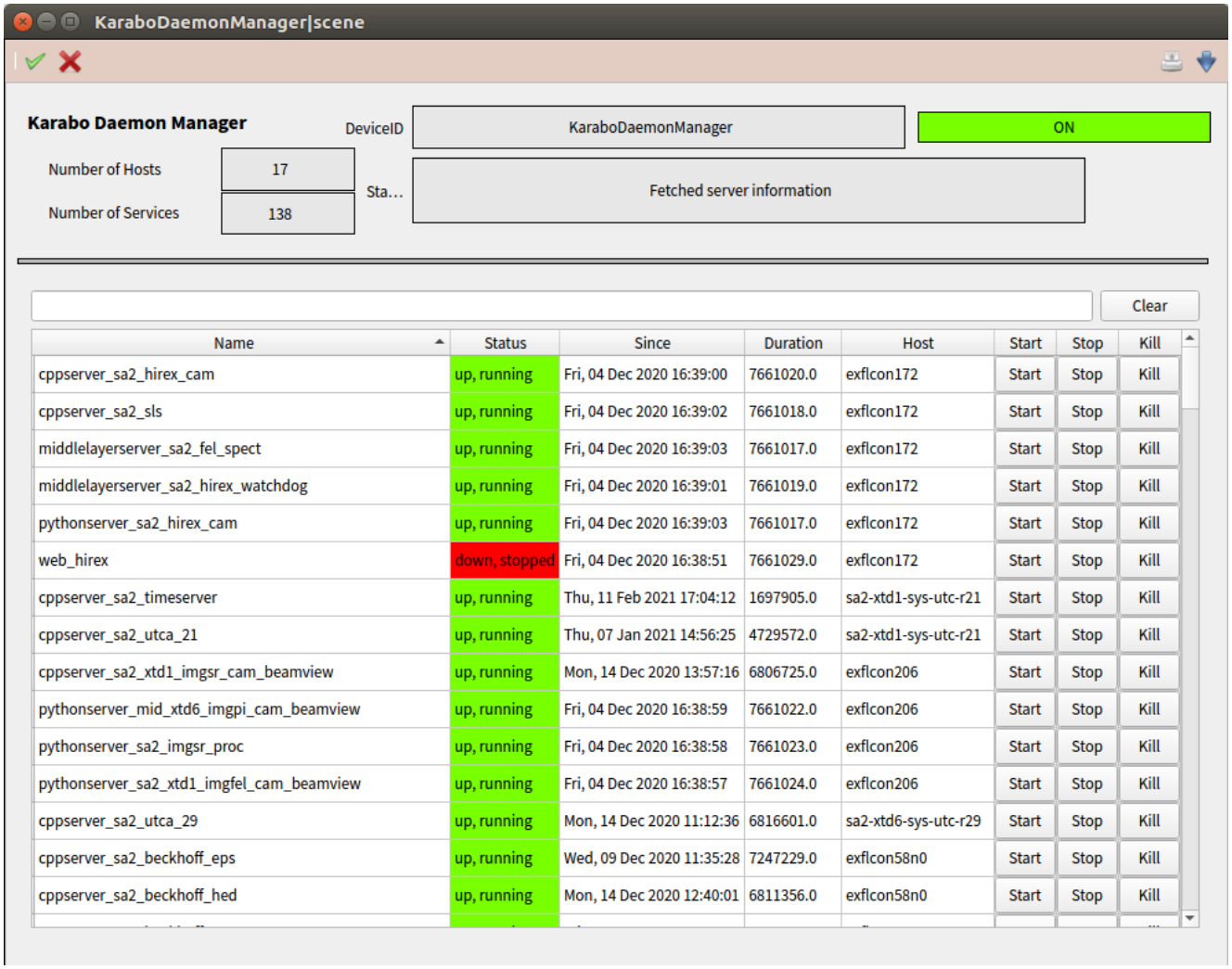


Fig. 57: The scene of the *KaraboDaemonManager* providing operator control of the services

27 #27: Karabo URI Scheme

27.1 Karabo GUI: How to start the Karabo GUI from a link

The Karabo GUI, since version 2.10.2 and on Windows and Linux, provides a Uniform Resource Identifier (URI) scheme handler.

A scheme handler is a standard practice that some applications use to start up an application with a specific configuration without using a specific file launcher. This standard is similar to how web addresses are formatted and is independent from the operating system. For example, The URI scheme handling is used by Zoom to open the client when clicking on a link in a browser or a mail client using the protocol *zoommtg* (often mediated by a web link). and the mail client is connected to the protocol *mailto*.

The Karabo GUI uses a similar approach to implement a unified launching scheme for different operating systems. In all production systems, the Karabo GUI will be registered as the application capable of reading the URI scheme. In user installed system the user is required to register karabo as the application, to use this feature. The registration can be triggered by clicking on the menu bar the `Settings` menu, and `Register Application` within this menu. Alternatively the registration can be done with the `karabo-register-protocol` command from terminal. Please note that this operation should be executed in after each update if the name of the conda environment changes from the previous installation.

To create such a link, one needs to righ-click on a scene, and follow the wizards instructions.

A wizard dialog will appear and guide the user through the procedure. In the first step the user can select if the splash-screen

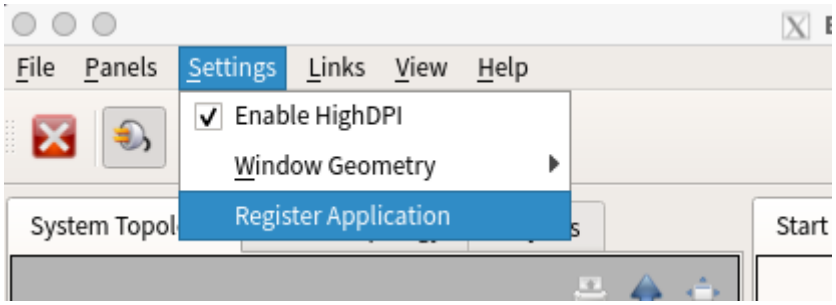


Fig. 58: How to register the current version as a scheme handler

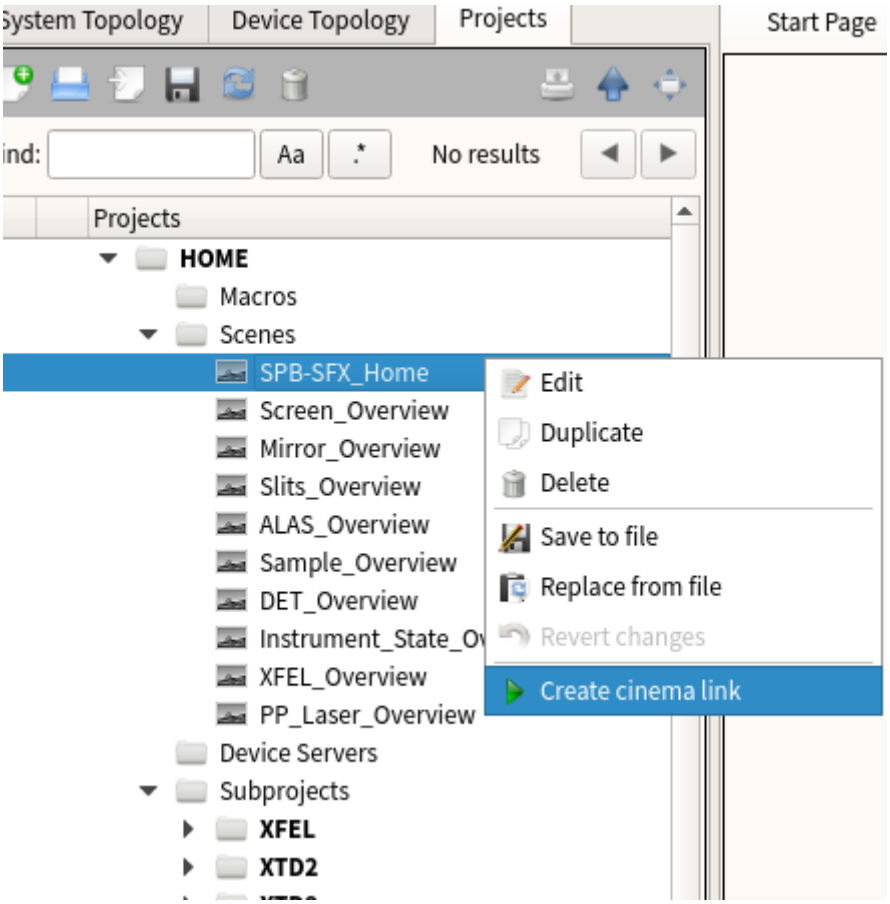


Fig. 59: right-clicking on a scene, on a project will show the `Create cinema link` option.

is shown and ifthe initial login dialog is shown or the hostname, port and user of the current session will be sent automatically.

The Karabo GUI currently needs network access to the port the GUI server is running, in applications within the same network (e.g. a link meant to be opened within the control network, the `show login dialog` option should be deselected/unticked to save some seconds. For a link meant to be shared outside the control network, were each user might have different port redirection solutions, the `show login dialog` option should be selected/ticked).

The next step will allow saving the URI into the clipboard as copied content.

The user can select either the URL form or the HTML tag depending on the use case.

This link can be copied into wiki pages or saved as a link according to the user preference.

The examples below show how one can include these links in different applications. Please be aware that the Karabo GUI application is not responsible for these external behaviours which are operating system dependent. The Karabo GUI application might not be able to guarantee external applications' behaviour.

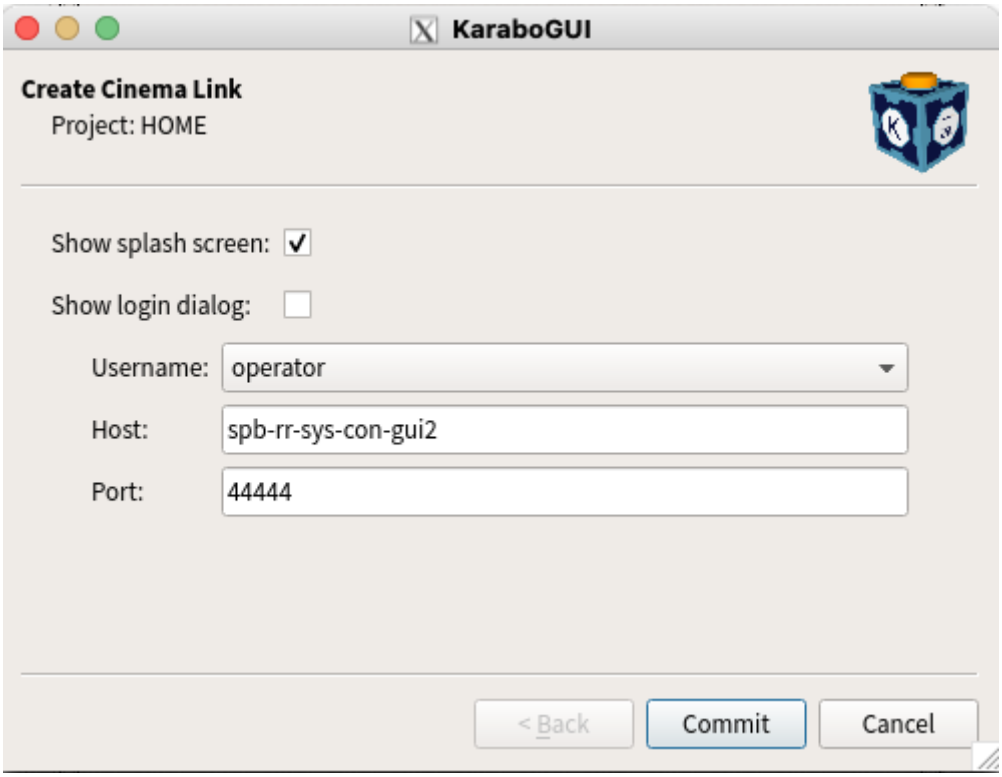


Fig. 60: The first step of the wizard.

Known Issues

This launching system is currently not supported on MacOS and on WSL installations (Ubuntu installation within a Windows System).

Use Example: Redmine Wiki

The link created in the previous step can be copied in a Wiki link in Redmine with the redmine markdown format:

```
"Link to SASE1 overview - no host":karabo://cinema?domain=SA1&scene_uuid=9b9708ab-dd85-4e02-
↪8dd5-ad8424794e63
```

This will appear as an hypertext link, but will open a karabo cinema link pointing the the `uuid` defined by the link.

Use Example: Grafana

The link created in the previous step can be pasted into a Grafana panel using the HTML format.

This solution allows complex Grafana panels to be interfaced easily with the corresponding Karabo scenes in case direct intervention is necessary.

One has to select the `Text` visualization option, within this mode, the `HTML` content mode, and insert the content as `HTML`.

In the example below one can see how the HTML link created by the wizard can be used in practice.

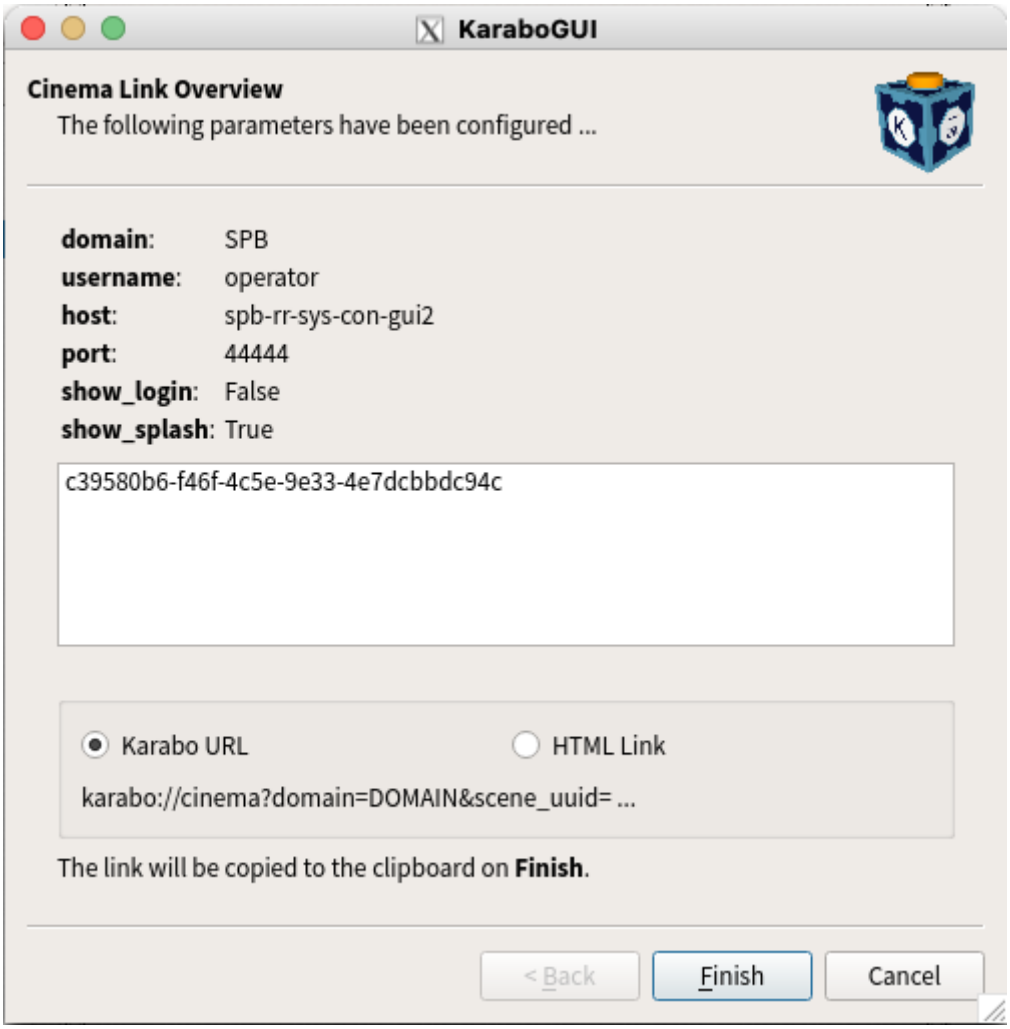


Fig. 61: The second and last step of the wizard.

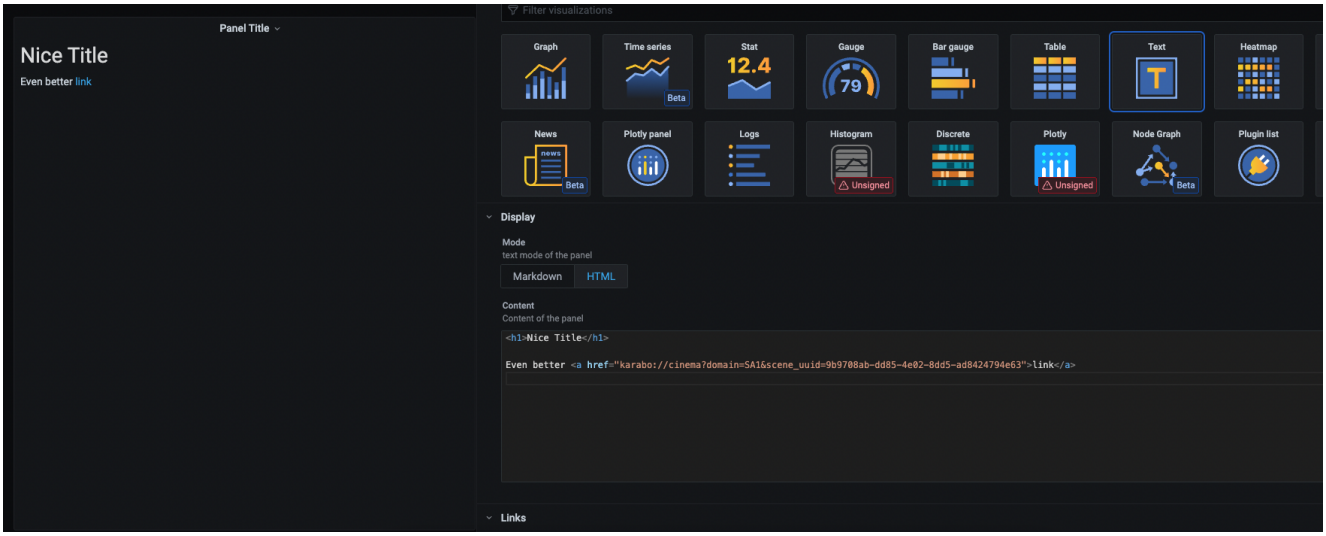


Fig. 62: An example of how a URI can be exposed in Grafana as a link.

Use Example: Ubuntu desktop files

As an ulterior exable, one can open an URI link from a file with a `.desktop` extension. One has to create a text file called `mylink.desktop` on the `Desktop` and include the following text:

```
[Desktop Entry]
Name=SA1Overview
Type=Application
Exec=xdg-open 'karabo://cinema?domain=SA1&scene_uuid=5c646610-0290-4de2-af7f-a87510ecda1b&
↪username=operator&host=sa1-br-sys-con-gui2&port=44444'
```

Where the `karabo://....` is the link copied by the wizard. **Note that for xdg-open the whole link must be defined as a single string.**

This will appear in the Desktop as a link, one needs to enable the execution for the current user and "allow launching". Follow these general instructions for Ubuntu 20 here: https://linuxconfig.org/how-to-create-desktop-shortcut-launcher-on-ubuntu-20-04-focal-fossa-linux

Please refer to the online documentation of your operating system for details.

Multiple scenes

A similar dialog can be used to generate a link able to open multiple scenes when invoked from the `Scenes` folder of a project. Multiple scenes can be selected or excluded in the second step of the wizard.

27.2 URI Format Details

The URI is currently supporting 3 applications: The Karabo GUI `karabo-gui`, the `karabo-cinena` and the `karabo-theatre`. Currently only the `karabo-cinema` option has a wizard to help link generation.

Wikipedia currently has a well written page on URIs, https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

According to the IETF RFC 2396 definitions, the Karabo GUI follows these specs:

- `protocol` : `karabo`
- `path` : `gui`, `cinema`, `theatre`
- `query` : options varies depending on the `path`

karabo-cinema format

If a URI's format follows this pattern:

```
karabo://cinema?domain=TOPIC1&scene_uuid=this-is-not-an-uuid
```

will open the Karabo Cinema with `domain` set to `TOPIC1` and `scene_uuid` set to `this-is-not-an-uuid`.

For the cinema, the options are:

- `domain`: The project domain of the scene to be opened (Mandatory).
- `scene_uuid`: The uuid of the scene to be opened (Mandatory).
- `username`: The username to be selected, i.e. the access level (Optional).
- `host`: The GUI server host (Optional).
- `port`: The GUI server port (Optional).

karabo-theatre format

In the `theatre` path, in most cases, the url will be encoded. For example, this link:

```
karabo://theatre?scene_id=COMPONENT%2FTYPE%2FMEMBER%7CSCENEname
```

Will open a `karabo-theatre` pointing to the device generated scene: `SCENEname` of the device `COMPONENT/TYPE/MEMBER`.

The options for theatre are:

- **`scene_id`: The identifier of the scene from a device (Mandatory).**

  note: `/` and `|` symbols need to be urlencoded, for example `COMPONENT/TYPE/MEMBER|SCENEname` encodes to `COMPONENT%2FTYPE%2FMEMBER%7CSCENEname`
  use this online helper as a sandbox https://meyerweb.com/eric/tools/dencoder/

- `username`: The username to be selected, i.e. the access level (Optional).
- `host`: The GUI server host (Optional).
- `port`: The GUI server port (Optional).

karabo-gui format

If a URI follows this format:

```
karabo://gui
```

The link will open the registered GUI client if the application is registered. It currently has no optional arguments.

28 #28: Karabo Kai - The MDL Dojo

The Karabo middlelayer is a high level framework that does a lot of procedures implicitly. The following sections are provided to introduce a common code style and highlight a few pitfalls and shows best practices using the asyncio python library according to the motto of this dojo:

- coroutine first
- task hard
- no threads

What does it mean? Firstly, use as much awaitables as possible, e.g. prefer the async implementation over a synchronous one. If code execution is taking long (io operations, slot calls) move procedures to a background task.

28.1 Code Style

In general PEP8, PEP20, and PEP257 are stylings to follow, there are a few Karabo-specific details to take care of to improve code quality.

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

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 (
    connectDevice, Device, Slot, String)

from .scenes import control, default
```

As a rule of thumb, *isort -m4* can solve your problems quickly.

Note: Don't do star (*) imports! Your readers don't know which parts you will be using.

Note: Don't do submodule imports, e.g. (*karabo.middlelayer.**). The underlying API might change and your device won't work anymore.

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
```

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"]
```

Slots and Methods

Slots are *camelCased*, methods *have_underscores*. Public slots don't have arguments.

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

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

28.2 Use Doubles, not Floats

Python floats have double precision. Hence, always use the **Double** declaration in a device instead of a **Float** whenever possible. Otherwise you might lose precision and have casting differences in top layer applications.

```
import numpy as np

Do = Double()
Dont = Float()


value = np.float32(0.1234)

>>> value
0.1234
>>> float(x)
0.1234000027179718
```

28.3 Karabo Values - Units and Timestamps

Setting a value on a device property automatically converts the value to a *KaraboValue*. A KaraboValue can have a unit and a timestamp. A timestamp is automatically assigned if the new value does not have one assigned.

```
from karabo.middlelayer import Unit, MetricPrefix

value = Double(
    unitSymbol=Unit.SECOND,
    metricPrefixSymbol=MetricPrefix.MILLI)

@Slot()
def some_function(self):
    self.value = 5
    print(self.value)
    print(self.value.timestamp)
```

(continues on next page)

(continued from previous page)

```
>>> some_function()
5.0 ms
2022-04-26T12:16:26.423016
```

Timestamps are preserved under the hood when calculating.

```
from karabo.middlelayer import unit

def calculate_new(self):
    value = self.value
    print(value)
    print(self.value.timestamp)

    # Calculate a new value
    new_value = value + 20 * unit.ms
    print(self.value)
    print(self.value.timestamp)

>>> some_function()
5.0 ms
2022-04-26T12:16:26.423016
25.0 ms
2022-04-26T12:16:26.423016
```

With more KaraboValues, the newest timestamp is taken.

```
def calculate_new(self):
    encoder_position = self.encoderPosition
    print(encoder_position)
    print(encoder_position.timestamp)

    offset = self.offset
    print(offset)
    print(offset.timestamp)

    # Calculate the total
    total = encoder_position + offset
    print(total)
    print(total.timestamp)

>>> some_function()
5.0 mm
2022-04-26T12:16:26.423016
17.3 mm
2022-04-26T12:17:22.423016
22.3 mm
2022-04-26T12:17:22.423016
```

Warning: If the units of the values cannot be casted together, there will be an exception.

28.4 Karabo Values - Comparison

Dealing with *KaraboValues* is an important fact to always remember. A very often realized pitfall is the comparison by identity (**is**), that fails. *KaraboValues* cannot be compared so singleton values by identity (*None*, *True*, *False*).

```
from karabo.middlelayer import isSet

isLocked = Bool(
    defaultValue=True)

position = Double()

def check(self):
    print(self.isLocked is True)
    print(self.position is None)
    print(self.isLocked.value is True)
    print(isSet(position))

>>> check()
False
False
True
True
```

28.5 Exception handling

Exception handling in Karabo devices is important. Of course, ideally only expected or no exceptions are occuring. Please don't use bare ``try ... except` coding pattern. If an exception is triggered in a slot or reconfiguration, an error message is propagated to the gui client and a pop up is shown. If an exception is happening in a background procedure, it is visible in the logging system. Only this way errors do not propagate without knowledge.

```
@Slot()
async def dontDo(self):
    try:
        # This is a bad example
    except Exception:
        self.state = State.ERROR

@Slot()
async def slotItLikeAnEngineer(self):
    # Work with expected exceptions
    try:
        position = self.positions[self.index]
    except IndexError:
        # do something

@Slot()
async def workingWithWaitFor(self):
    # Slots are not allowed to take time
    await wait_for(setWait("DEVICENOTTHERE", position=5), timeout=2)

@Slot()
async def timeoutSlot(self):
    try:
        await wait_for(setWait("DEVICENOTTHERE", position=5), timeout=2)
    except TimeoutError:
        self.status = "Device not available"
```

Note: The `TimeoutError` must be imported from *asyncio*.

28.6 Numpy and KaraboValues

Numpy is an extensively used package for data analysis routines. When using *KaraboValues*, not every function is supported.

The package that supports the units is *pint*. In order to make our newest timestamp and unit algorithm work, the KaraboValues rely on element wise math operations. With regard to numpy, we are talking about so-called ufuncs (universal functions).

As an example, *numpy.mean* and *numpy.std* are not universal functions and won't work properly with *KaraboValues*. Also *pint* is not yet supporting every universal function and we have found:

- `numpy.positive`
- `numpy.divmod`
- `numpy.heaviside`
- `numpy.gcd`
- `numpy.lcm`
- `numpy.bitwise_and`
- `numpy.bitwise_xor`
- `numpy.bitwise_or`
- `numpy.invert`
- `numpy.left_shift`
- `numpy.right_shift`
- `numpy.logical_and`
- `numpy.logical_or`
- `numpy.logical_xor`
- `numpy.logical_not`
- `numpy.spacing`

Ideally mathmatical operations are always factored out in an own module and unit tested.

Working with *KaraboValues* and *numpy* is **discouraged** and it is the developers responsibility to backup his device code with unit-tests. Numpy, pint and python dependencies are continously upgraded! However, it is possible to use the magnitudes of *KaraboValues* with *.value* in *numpy* functions.

28.7 Slot Calls: Reply with correct State

Devices come with expectations. If we call a motor to move, we expect it after the slot (move) call to be in a *MOVING* state.

Similar scenarios are there for all devices. Hence, we must reply with the correct State, but also very quick!

```
@Slot()
async def onlyAllowedInMacro(self):
    self.state = State.ACTIVE
    await sleep(20)
    self.state = State.PASSIVE

@Slot(allowedStates=[State.PASSIVE])
async def move(self):
```

(continues on next page)

(continued from previous page)

```
    self.state = State.ACTIVE
    background(self._long_operation())
    # The Slot method ends, the reply to the user happens without error
    # and the device is in ACTIVE State.

async def _long_operation(self):
    await sleep(20)
    self.state = State.PASSIVE
```

28.8 Setter knowhow and instantiation workflow

Devices initialize firstly their Schema (Pipeline Channel and Device Nodes) and then enter the *onInitialization* function. The device state by default is *State.UNKNOWN*.

```
class MyDevice(Device):
    is_connected = False

    @Double()
    def targetPosition(self, value):
        if value is None:
            return
        if self.is_connected:
            self.send_target_hardware(value)
        self.targetPosition = value

    async def onInitialization(self):
        # We enter as State.UNKNOWN (default)
        await self.connect_hardware()
        self.is_connected = True
        self.state = State.ON
```

28.9 State Monitor: StateSignifier and background

A typical task of a middlelayer device is to merge a lot of device states to a single one. For this task always use the *StateSignifier*. This example shows how to build a state monitor. Note in this case the extra attributes *assignment* and *accessMode*.

```
from karabo.middlelayer import (StateSignifier, Assignment, AccessMode,
                                background, connectDevice, ...)

class MyDevice(Device):

    mirrorXId = String(
        accessMode=AccessMode.INITONLY,
        assignment=Assignment.MANDATORY)

    mirrorYId = String(
        accessMode=AccessMode.INITONLY,
        assignment=Assignment.MANDATORY)

    def __init__(self, configuration):
        super().__init__(configuration)
        self.signifier = StateSignifier()

    async def onInitialization(self):
```

(continues on next page)

(continued from previous page)

```
        devs = [connectDevice(self.mirrorXId),
                connectDevice(self.mirrorYId)]
        self.devices = await gather(*devs)
        background(self._monitor_state())

    async def _monitor_state(self):
        while True:
            props = [dev.state for dev in self.devices]
            # The state signifier also provides the newest timestamp
            state = self.signifier.returnMostSignificant(props)
            if state != self.state:
                # Only set a different state if necessary
                self.state = state
            await waitUntilNew(*props)
```

28.10 Pipelining: InputChannel and proxies

A typical task of a middlelayer device can also be to combine input channel data (fast data, pipeline data) and proxy data (slow control data). Important to know is that *InputChannel* connections are created in the background to not block the device instantiation.

```
from karabo.middlelayer import connectDevice, InputChannel

class MyDevice(Device):
    is_connected = False

    mirrorXId = String(
        accessMode=AccessMode.INITONLY,
        assignment=Assignment.MANDATORY)

    mirrorYId = String(
        accessMode=AccessMode.INITONLY,
        assignment=Assignment.MANDATORY)

    def __init__(self, configuration):
        super().__init__(configuration)
        self.signifier = StateSignifier()

    async def onInitialization(self):
        # InputChannels might be already alive but we don't have the proxies
        # yet
        devs = [connectDevice(self.mirrorXId),
                connectDevice(self.mirrorYId)]
        self.devices = await gather(*devs)
        self.is_connected = True

    @InputChannel()
    async def input(self, data, meta):
        if not self.is_connected:
            return
        # Do something with proxy and input data
        timestamp = meta.timestamp.timestamp
        train_id = timestamp.tid
        position = self.devices[0].actualPosition
```

(continues on next page)

(continued from previous page)

```
...
```

29 #29: Magic Shortcuts

The KaraboGUI has magic shortcuts to provide various actions, either by key or mouse event, or a combination of both. In the following we provide a brief overview of all shortcuts that are valid in version **2.16.5**.

Note: MacOS clients have to use the **COMMAND** key instead of **CTRL** and **OPTION** key instead of **ALT**.

29.1 Scene (Control Mode)

- Double-click left mouse button on property widget: Derive a controller widget for historical view of data (trendline)
- ALT + click left mouse button on property widget: Select the device belonging to this property in the *Configurator Panel*
- ALT + double click left mouse button on property widget: Pop up the main window with the *Configuration Panel*. If in *cinema* or *concert* mode, a *Configuration Editor* is provided

29.2 Scene (Edit Mode)

- CTRL + A: Select all items on the scene
- CTRL + G: Group selected items on the scene
- CTRL + SHIFT + G: Ungroup selected items on the scene
- DEL: Delete selected items from the scene
- ARROW KEYS: Move the selected items on the scene with grid increment (10 px)

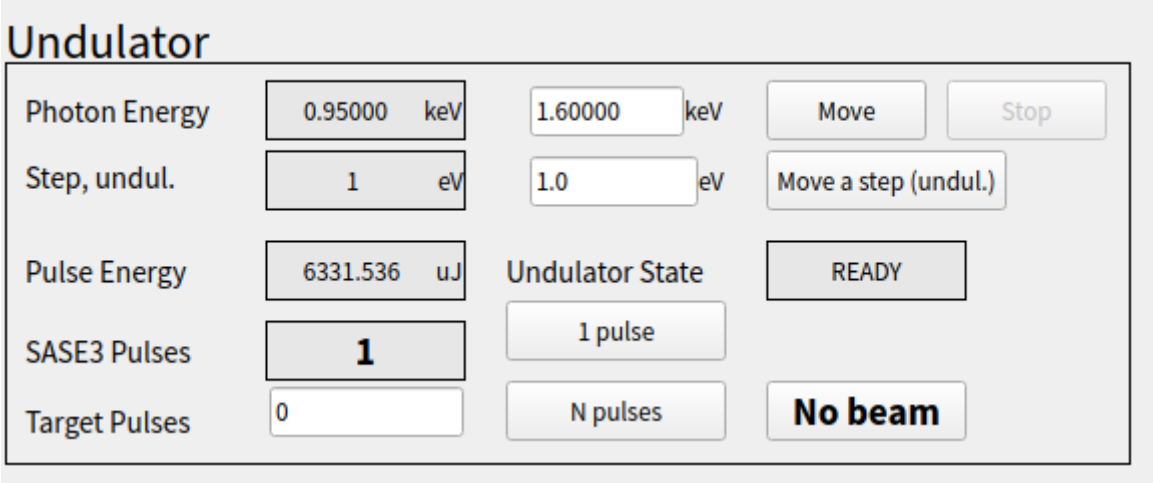


Fig. 63: Scene with widget controllers

29.3 Project & Navigation Panel

- Double-click left mouse button on device: Derives the default device provided scene
- Double-click left mouse button on server: Derives the latest cached log messages of the server

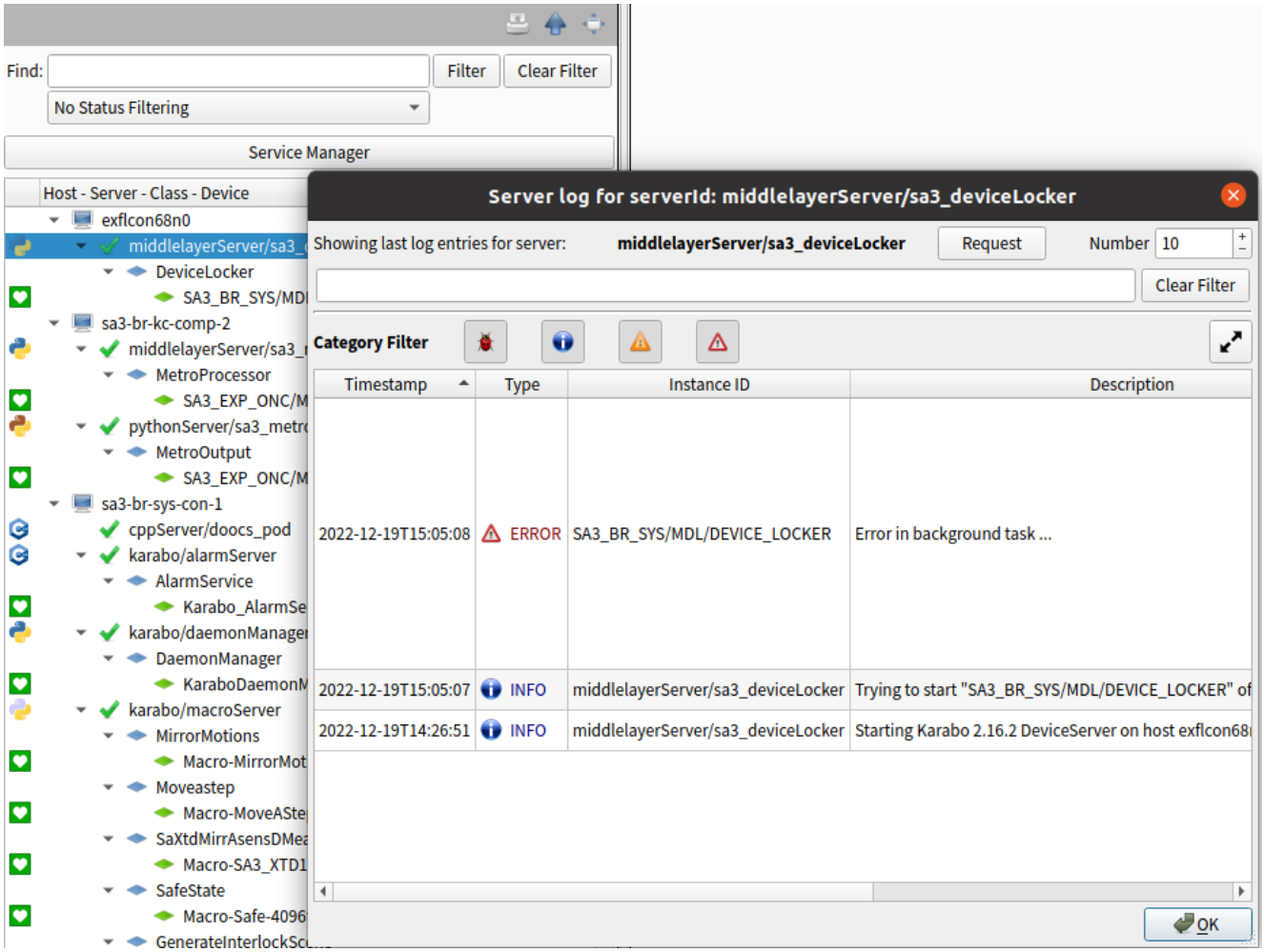


Fig. 64: Panel with log widget

29.4 Macro Panel

- CTRL + F: Find functionality
- CTRL + R: Replace functionality

29.5 Table Widget

- CTRL + N: New row
- DEL: Remove row
- SHIFT + UP: Move selected row up
- SHIFT + DOWN: Move selected row down

Fig. 65: Macro Panel with find and replace toolbar

29.6 Configurator

- Double-click left mouse button on read only property: Derive a controller widget for historical view of data (trendline) or show a VectorGraph (vectors) or WebCamGraph (images)

Conditional Trigger Mask	0b1111	
Use Conditional Trigger	False	False
EnableOutput	True	True
Invert Trigger	False	False
Actual Delay	17537000.0 ns	
Target Delay	0.0 ns	0.0 ns

Fig. 66: Configurator Properties

30 #30: Karabo LogBook

We are excited to announce the new addition to the Karabo GUI 2.19 - the Karabo LogBook system. This new feature seamlessly integrates with Zulip-based logbook, allowing users to post data from the Karabo GUI to the Logbook.

30.1 LogBook Device

The LogBook Karabo device acts as backend and a strong component of this system. It is a bidirectional interface and retrieves the logbook information for the active proposals for the instruments, as well as posts data from the Karabo GUI to the logbook.

30.2 LogBook GUI

Karabo GUI has new dialog “Logbook:Preview” designed for previewing the data destined for the logbook. The dialog is accessible through the new tool button on the top right of all the panels in the Karabo GUI.

Fig. 67: Logbook tool button

30.3 Key Features of the LogBook GUI

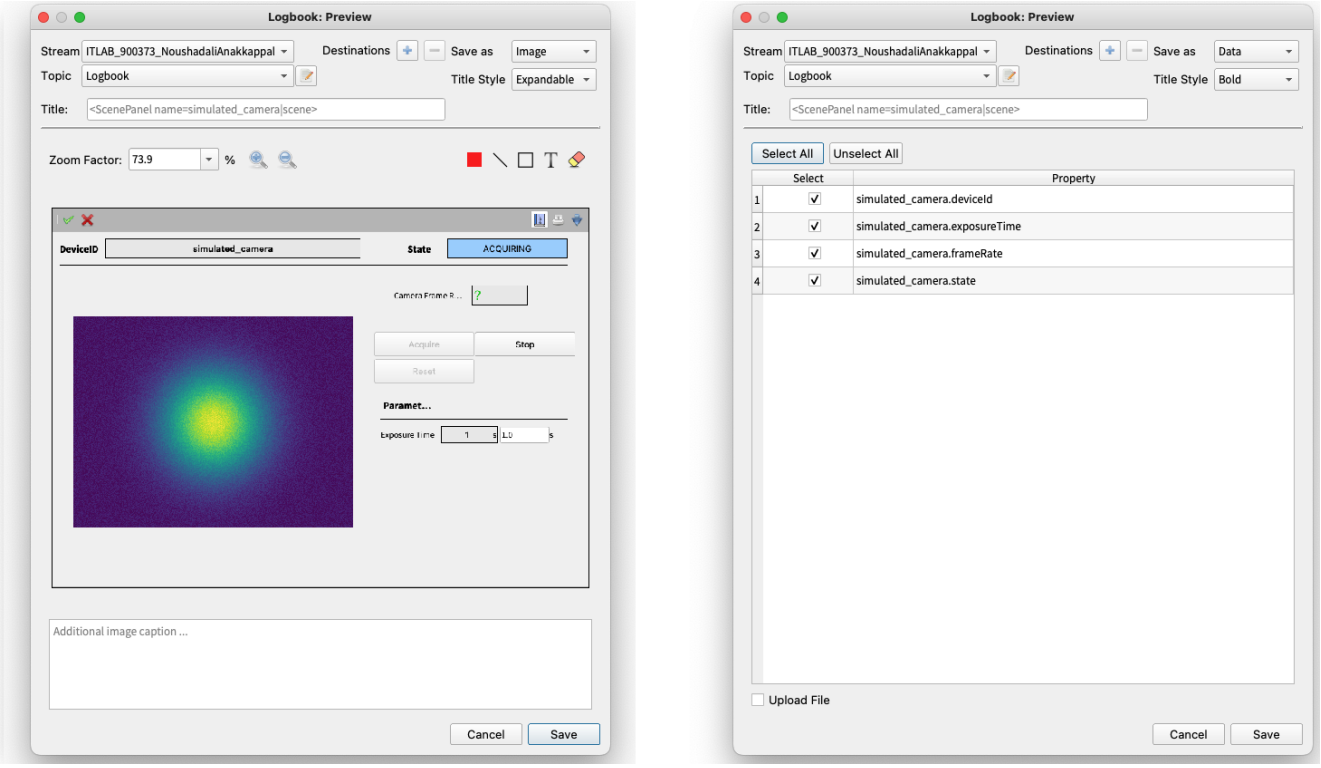


Fig. 68: Logbook GUI with Image preview and Table preview

Logbook Selection

The dialog lists the available logbooks as the *Stream* and *Topic* name for the proposals that are currently active for the Instrument. A user can also optionally choose more than one logbook, by the "Add Destination" option (the "+" button). Creating a new topic within the chosen stream can be done with the edit button next to the "Topic" drop-down options. The dialog remembers the first selected logbook (*Stream* and *Topic* name) across the Karabo GUI sessions.

Data Type Selection

The data from the Karabo can be posted as either image or as a table. User can choose this from the *"Save as"* widget.

Customisable Post Headers

The dialog provides an option to customize the header of the post in the Zulip with the following three options.

- No Title.
- Title as bold text
- Collapsible title as Zulip Spoiler.

The header type will be restored across the Karabo GUI sessions.

By default, the title is set as the panel name and the GUI topic.

Image Preview

The image preview comes with some practical functionalities

- Adjustable zoom for better image examination
- Annotating images with lines, rectangle and text with a customizable color.
- Easy erasure of annotations
- Additional comments to the image.

Table Preview

For the table data, the GUI offers the following convenient features.

- Selection of a subset properties
- Buttons to select *All* or *No* properties.
- Additionally upload the table as a CSV file

31 #31: Authentication

We are excited to introduce the new test phase of Authentication in **SA1**.

Access to a GuiServer will now require logging in with a facility account (*LDAP*). One key update is that functional accounts will be limited to *OPERATOR* privileges. However, users can temporarily elevate their access level during a GUI session by logging in with a personal LDAP account. Each login to the GuiServer will generate a session-identifying *token*, which will be logged. All relevant Gui client actions will also be recorded, along with the *token*, in an *audit* file on the GuiServer machine. The link between the token and the user will be stored in an external database for a specified time period.

Please note that during this testing phase, we are focusing solely on user experience and interface, and no user login information will be stored.

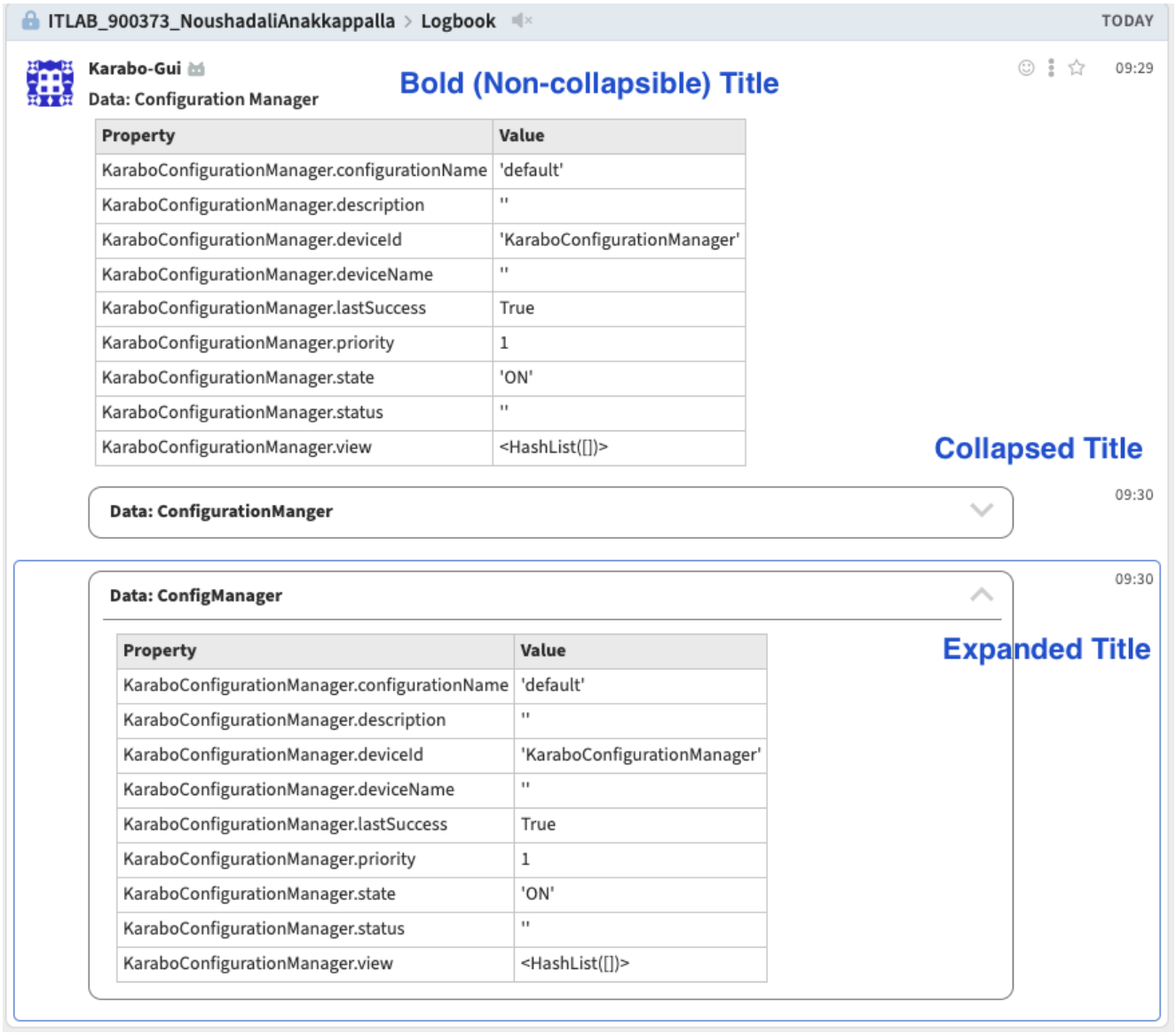


Fig. 69: Headers - Bold, Collapsed and Expanded.

31.1 Authentication Server Workflow

The Karabo *Authentication Server* login form is available under the following domain .

The server is accessible from both the office and control network.

When launching KaraboGui (Version 2.20.X or higher) and attempting to connect to a GuiServer device with authentication enabled, an authentication login panel will appear. By clicking on Open Access Form, the user is directed to the login page (1). Successfully logging in with LDAP credentials (2) generates an **access code** (3), which can then be entered in the GUI login panel (4) to establish a connection to the GuiServer.

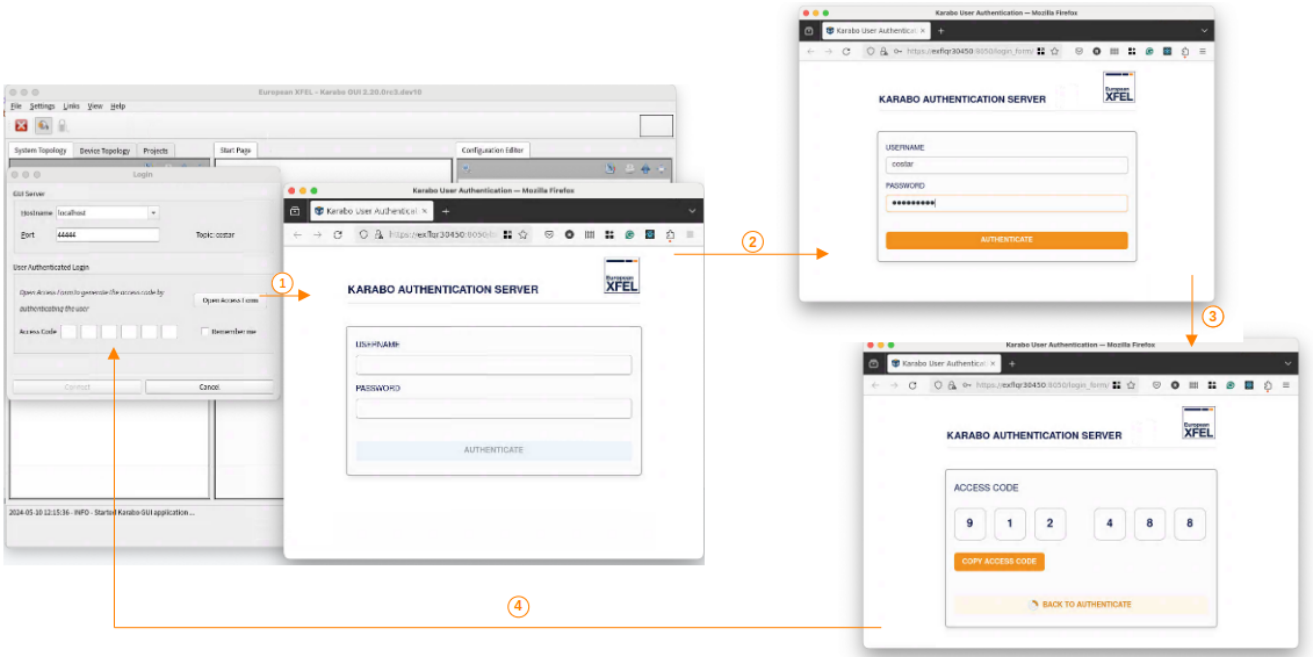


Fig. 70: Authentication workflow

31.2 Frequently Asked Questions

How many access levels does Karabo have?

For historic reasons Karabo has 5 different access levels

- OBSERVER
- USER
- OPERATOR
- EXPERT
- ADMIN

The *OBSERVER* level corresponds to read-only access. Any reconfigurable parameter or Slot (Button) in Karabo should require at least *USER* access. Privileges increase progressively from top to bottom in the list.

Why do we have the access level USER?

The *USER* access level is the first level that permits reconfiguration of parameters. Historically, it was intended for users or operators from external facilities who were assigned the *USER* access level. However, this level may no longer serve a significant purpose and could be phased out in the future.

Remember Me - What is happening?

The Remember Me feature helps eliminate the need for repeated logins. Designed specifically for Instrument Control Hutches or BKR, this functionality allows each client computer to authenticate just once, as tokens are shared and refreshed automatically across clients. An authenticated client PC can then be used on any GuiServer device that requires authentication.

How can I remove my stored information?

It is possible to remove user login information with the menu bar: File -> Clear User Login

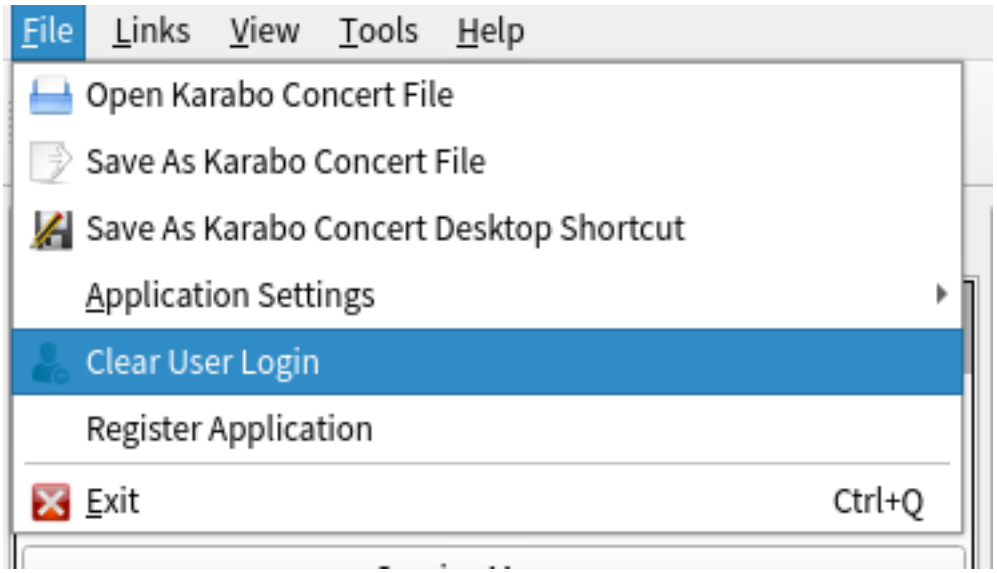


Fig. 71: Clear Login Action

Why are group accounts limited to OPERATOR?

Instrument group accounts remain permanently logged in for daily operations. Since control hutches or the BKR may be open and accessible, these accounts should be restricted to the *OPERATOR* access level. If commissioning parameters (requiring *EXPERT* or *ADMIN* access) need to be modified, a temporary escalation with a properly authenticated personal account is possible.

Which priviliges do I have?

Access levels are determined by group memberships and the *KARABO_TOPIC* of the GuiServer device. In the initial setup, instrument group memberships inherit privileges within their respective SASE.

Examples:

- If your account belongs to the *fxedata* or *exfl_fxe* group, you have *ADMIN* rights in FXE, SA1, and LA1, and *OPERATOR* rights in SPB, etc.
- If your account belongs to the *spbdata* or *exfl_spb* group, you have *ADMIN* rights in SPB, SA1, and LA1, and *OPERATOR* rights in FXE, etc.

- If your account belongs to the *la1data* or *exfl_la1* group, you have *ADMIN* rights in SA1 and LA1.
- If your account belongs to *fxedata* or *la1data*, your rights are the same as the *fxedata* case above.
- If your account belongs to the *sa1data* or *exfl_sa1* group, you have *ADMIN* rights in SPB, FXE, SA1, and LA1.
- If your account belongs to the *exfl_vac* group, you have *ADMIN* rights in SPB, FXE, SA1, and LA1.
- ...

Temporary Session - Duration

The maximum duration of a temporary session is configurable on the gui server, with a default of 1 hour. Once this time expires, the temporary session ends, and the client reverts to their previous maximum access level.

What user information is stored in the future?

Every client login to the GuiServer device is logged in a rotating file log, where only the access token is stored. From that point on, all client actions are logged using the access token in a separate *AUDIT* logging mechanism. The *username* associated with the access token is stored in an external database. This is not the case for the the test phase.

Additionally, each Karabo device has a *lastCommand* property to track an action and its source, e.g. whether the action was triggered by the GuiServer or another device. This information is stored in the *Influx* database.

Can I see or have access to personal login data?

No.

Who decides which Device Properties have which access level?

The device developer determines the access levels for each parameter. Ideally, commissioning parameters that are rarely used should have a higher access level. On the other hand, operational parameters that are needed for daily routines should not require *EXPERT* access or higher. However, there are currently no formal guidelines in place.

By default, read-only properties are assigned an *OBSERVER* access level, while reconfigurable properties are set to *USER*.

Which access roles are available in KaraboGui

The following access roles are available in the *karaboGui*. Since this is fairly new, they are sometimes not fully enforced, future versions might be more restrictive.

- *AccessRole.SCENE_EDIT*: *EXPERT*,
- *AccessRole.MACRO_EDIT*: *OPERATOR*,
- *AccessRole.PROJECT_EDIT*: *OPERATOR*
- *AccessRole.SERVICE_EDIT*: *OPERATOR*
- *AccessRole.INSTANCE_CONTROL*: *OPERATOR*

The core idea is that various editing functions, as well as instance or service control, require an *OPERATOR* access level.

Currently, an exception to this rule is scene management, such as activating the design mode of a scene for modification, which is restricted to clients with *EXPERT* access. In the future, features like project editing, macro editing, and service (device servers) and device control, such as shutdowns, should also be restricted to the OPERATOR access level.

32 #32: KaraboGui installation: Miniforge

In order to use Conda, three basic steps are needed:

1. Install Conda (recommended conda installer). We use **Miniforge**, here.
2. Configure channels.
3. Install the KaraboGUI.

32.1 Install Conda

Download and install **Miniforge** (Python version >= 3)

1. Download the installer script/executable.
   - **Linux:**
     Download and install by running

     ```
     curl -L -O "https://github.com/conda-forge/miniforge/releases/latest/download/
     ↪Miniforge3-$(uname)-$(uname -m).sh"
     bash Miniforge3-$(uname)-$(uname -m).sh
     ```

   - **Mac :**
     Download the script and install by running

     ```
     bash Miniforge3-MacOSX-x86_64.sh
     ```

   - **Windows:**
     Download the installer

     Double-click on `Miniforge3-Windows-x86_64.exe` to run the installer.
2. Follow the installation prompt. On Windows, remember to keep the "Create start menu shortcuts" option selected (by default, selected), in the final step.
3. Open your terminal (Miniforge3 PowerShell on Windows or Bash on Linux and MacOS)
4. If conda is in your path, you should be able to run

   ```
   conda --version
   ```

   If it is not, you need to activate conda first

   - Linux/MacOS:

     ```
     source <miniforge_path>/etc/profile.d/conda.sh
     ```

   - Win:

     ```
     CALL <miniforge_path_path>/condabin/activate.bat
     ```

By default *miniforge_path* is on your home directory - `~/miniforge3` on Linux/Mac and on `C:\Users\<YourUsername>\miniforge3` (or `C:\Users\<YourUsername>\AppData\Local\miniforge3`) on Windows.

32.2 Channel Configuration

If you have already installed karabogui on your machine using conda before, you could skip to the next step, as you may have already configured your channels correctly. Please ensure to remove the **defaults** channels.

```
conda config --remove channels defaults
```

Also ensure that karabo channels are added.

```
conda config --add channels http://exflctrl01.desy.de/karabo/channel
conda config --add channels http://exflctrl01.desy.de/karabo/channel/mirror/conda-
↪forge
```

You can verify that the channels are added correctly, by running

```
conda config --show channels
```

and you will get

```
channels:
   - http://exflctrl01.desy.de/karabo/channel/mirror/conda-forge
   - http://exflctrl01.desy.de/karabo/channel
   - conda-forge
```

For more details for Karabo channels, refer to karabo documentation

32.3 Install KaraboGUI

To install the latest karabogui (recommended)

```
conda create -n karabogui_latest karabogui --yes
```

or install a specific version

```
conda create -n karabogui<version> karabogui=<version> --yes
```

For example:

```
conda create -n karabogui_2_20_7 karabogui=2.20.7 --yes
```