SLATE™ Web Editor Instructions

Version 1.2.8

Table of Contents

Introduction

This document describes usage of SLATE Web Editor and its fundamentals*. Document is created in a way that it could be used as a referential manual. It should not be needed to read the document from beginning to end, however a complete overview is recommended. It is believed that the editor user or document reader is a person with some technical background.

Pages created with the editor are intended for displaying and altering data within the SLATE system. Editor output is a HTML page extract containing set of configured widgets, which acts as a regular web page after deploying into the SLATE Base module.

Document is divided into four chapters. First chapter deals with editor installation and editor running. This chapter explains a way how to create a project and how such project is stored in the system. In second chapter are described editor fundamentals, how is editor organized, what are the individual parts for and how they can be used.

Third chapter is focused on individual widgets and their use. Widgets are described in the order how they appear in the list of available widgets. There is a screenshot of each widget for an illustration, often also with its specific properties. First three sections – Numeric, Boolean and Text – contain widgets, which work with value type that correspond to their section name. Section Container contains widgets that are able to include other widgets and work with them as a group. Section Fuel Air Commissioning contains widgets, which work only with the Fuel Air Ratio module. Work with a multimedia content and creation of graphical elements is described in Media section. User authentication and authorization to protected content is manageable with widgets described in Special section.

Last fourth chapter is intended for users who have already become familiar with the editor. Chapter contains explanation of an inheritance concept, which can be utilized for creating reusable pages. There are also guides for adding custom content and for creating custom styles and widgets.

* Implementation details are beyond this document.

1 Running the Editor

1.1 Installation

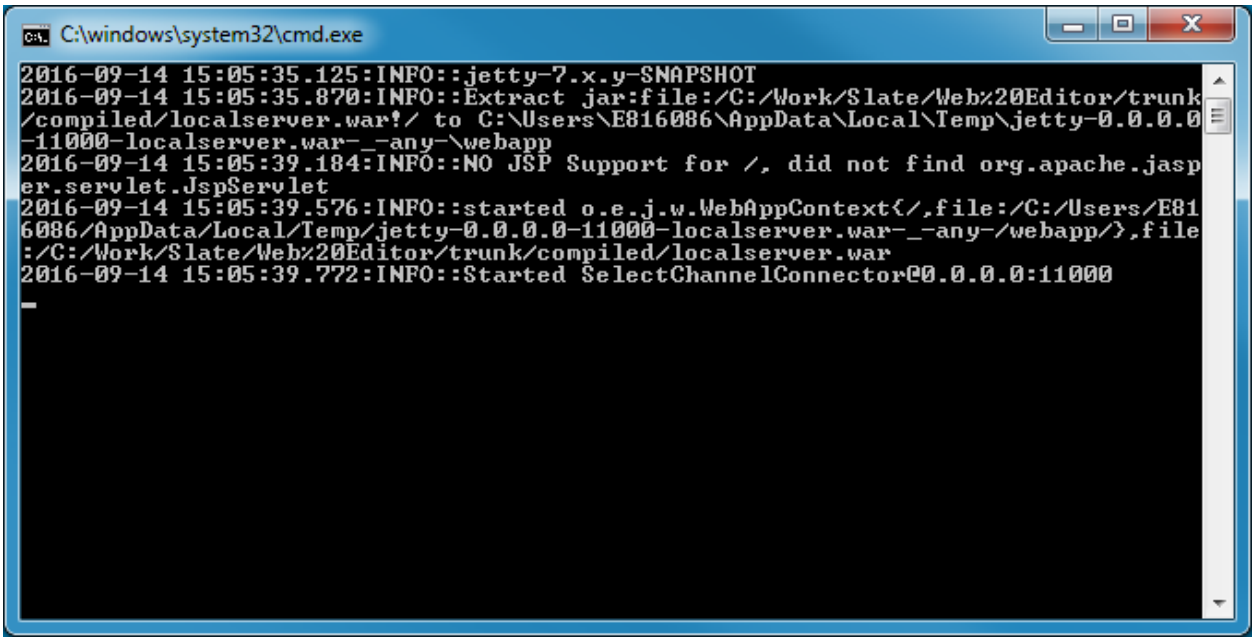
The editor has been created in a way that nothing has to be installed apart from Java Runtime Environment version 1.5 or higher. Most likely this has already been installed on your PC, otherwise go to https://www.java.com/en/download/ and follow the instructions.

1.2 Preparation

Copy the content (war file and bat file) from either B drive [B:\Kettos\Software\Web Editor\] or from subversion [/Web Editor/trunk/compiled/] to a folder on your local disk. If you prefer running the local server on a different port from the default 11000, you can achieve this by modifying start.bat and setting the property `port`.

1.3 Running

Run the start.bat file. It takes several seconds to run. When no problems occur, it should look similar to:



1.4 Accessing the Editor

Open Chrome browser (the editor will not work in Internet Explorer and other browses like Safari, Firefox or Opera) and navigate it to http://localhost:11000/index.

1.5 Generating the Project

If you see Welcome page, all worked as intended. Hit Create a New Web Editor Project and fill in some project name, leave ID untouched and pick a resolution (you can optionally change the project ID, but accepted are only alphanumerical characters, – and _). The project folder structure should be generated and you should now see the link which starts the editor in the context of your project. You can also

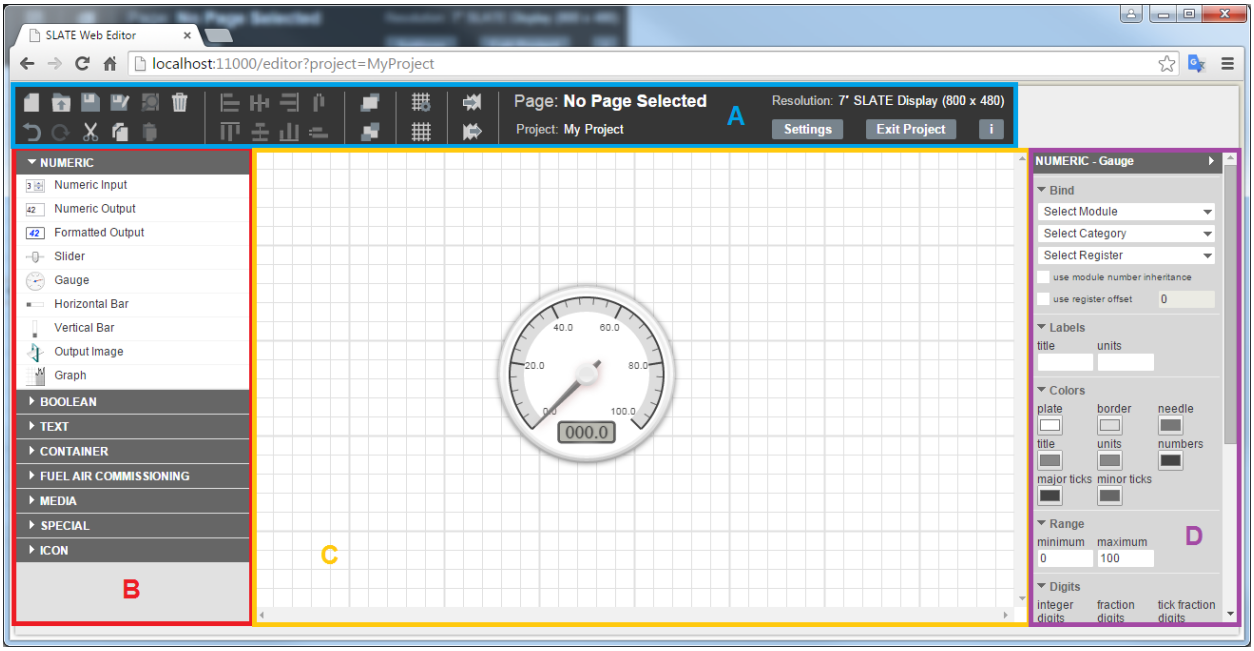
check the folder where you downloaded the war file. You will see <ID of your project>.xml with a default _settings.xml file and also projects/<ID of your project> folder. In this folder you will find all pages created in the editor (in /web/html), but it is also the destination for the XML file that defines the registers used in your design that is generated by the Niagara tool (/niagara), custom JS files (/web/js), custom CSS files (/web/css), custom images (/web/img) and custom data files, such as audio, video, documents etc (/web/data). The destination of global version of the last four sets (i.e. JS, CSS, images and data) is in the folder with war file /global/ following the same folder substructure, see below for more details about adding custom content. Let's go back to the Welcome page and click the project link. This should bring us into the editor.

2 Using the Editor

2.1 Editor Fundamentals

2.1.1 Editor Layout

If all is working properly, the page should display four sections as shown below (section D is shown after some widget has been dragged from B to C or after a mouse double-click on a widget or from widget's Context menu).



- Section A contains the information about edited file, project and buttons for widget operations.
- Section B contains the structured list of available widgets.
- Section C is the main editor area, where user will place desired widgets.
- Section D is the properties area, where widgets can be customized by setting their individual properties. Width of the properties area can be enlarged by clicking on the arrow in upper right corner.

2.1.2 The Basics of the Workflow

If the user wants to create his/her own file, then all is ready. If instead he/she wants to modify existing file, by using button **Load** any previously created file can be loaded. Adding new widgets is performed by choosing a widget from given category of widgets and dragging it anywhere to the editor area. Whenever a single widget is selected, its properties appear next to the editor area and can be modified. Some changes may be immediately visible on the widget (such as color), while some are invisible (such as register mapping). When the user is satisfied with the editor content, he/she can save it via **Save** or **Save As** buttons. Then he/she can also preview it in simulation mode by clicking Preview button (right from Save As button).

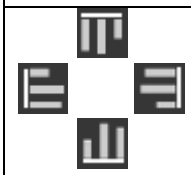
2.1.3 Linking of the Dynamic Widgets

Some widgets are static. They appear exactly the same way at all times. But some widgets are dynamic and reflect values obtained from the Base Module (in the production mode) or from the local server (in the simulation mode). Values are obtained periodically each 0.5s from the Base Module, and each 2s from the local server. The widget is linked to some particular register in some particular module. If it is an output widget (such as Numeric Output, Gauge or Vertical Bar), it periodically asks for the value of this register and accordingly updates its value/image/shape. It is an input widget, it displays the value provided by the server and if the user chooses to change it, the widget propagates this change to the server. As long as the widget is selected, it is not updated, so the user can perform the change.

2.2 Editor Commands

The commands are summarized in the table below. Let's go from top left corner to the right:

Icon	Button	Shortcut	Effect
	New		Clears the editor content and file name.
	Load/Rename/Delete		Opens a dialog window with list of available files (HTML pages). Open action loads the content of the chosen file into the editor area. This dialog window also allows renaming or deleting a file from the list.
	Save	Ctrl+S	Saves the editor content into the previously chosen file, or asks for a file name if none was specified.
	Save As		Asks for a file name and saves the editor content into this file.
	Preview		Opens a new tab with the saved page in the simulation mode.

	Delete	Del	Deletes selected widget(s) from the main editor area.
	Undo	Ctrl+Z	Reverts the last action.
	Redo	Ctrl+Y	Re-performs the last reverted action.
	Cut	Ctrl+X	Copies selected widget(s) into the editor clipboard and removes the widget(s).
	Copy	Ctrl+C	Copies selected widget(s) into the editor clipboard.
	Paste	Ctrl+V	Pastes the widget(s) from the editor clipboard to the editor area.
	Delete	Del	Deletes the selected widgets.
		Ctrl+A	Selects all widgets.
	Align Buttons	Alt+Num Arrows	Aligning all selected widgets in the particular direction: 2 is down, 6 is right, 8 is up, 4 is left.
	Horizontal Align	Alt+H	Align widgets by their center horizontally
	Vertical Align	Alt+V	Align widgets by their center vertically
	Horizontal Distribute	Alt+J	Evenly distribute widgets horizontally
	Vertical Distribute	Alt+B	Evenly distribute widgets vertically
	To Front	Alt+U	Shifts the selected widget(s) to the front (above all other widgets).
	To Back	Alt+D	Shifts the selected widget(s) to the bottom (below all other widgets).
	Grid Options		Opens a dialog window with grid options.
	Toggle Grid		Toggles the grid visibility.

	In Grid Options	Alt+Q	Increase grid (for snapping when shift is held while dragging)
	In Grid Options	Alt+A	Decrease grid
	Save To Scrapbook		Asks for a scrapbook item name and saves the current selection to scrapbook. Scrapbook allows reusing a selection of already defined widgets on multiple pages.
	Paste From Scrapbook		Opens a dialog window with list of scrapbook items. Paste action loads the content of the chosen scrapbook item into the editor area. This dialog window also allows to rename or to delete a scrapbook item from the list.
	Settings		When designing page different from the default project resolution, the width and height can be changed here.
	Exit Project		Navigates the browser back to the welcome page.
	Information		Opens an information window which contains the Web Editor version number, release date, list of included JavaSript and CSS files (if any) and list of available JavaSript and CSS files for the simulation mode (if any).

The remaining operations are described in the following table:

Operation	**Effect**
Right Click	Invokes the context menu (on the widget or on the editor area).
Ctrl + Left Click	Toggles selection for the widget that has been clicked.
Left Click widget + Drag	If a widget was selected in the editor area, this widget is moved.
Left Click resizing handle + Drag	If the right or bottom bound or right bottom corner arrow is clicked, the widget is resized.
Left Click to open area + Drag	A rectangle is shown, which can be used to select all widgets contained in or touched by the rectangle.

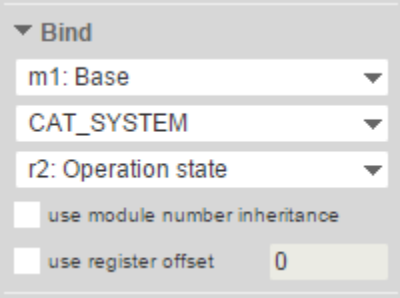
Left Click to open area + Drag + Ctrl	With the control key the selection rectangle behavior is modified to select only those widgets that are completely contained within the rectangle.
Right Click anywhere + Drag	By using a right-click, only a selection rectangle occurs, that is if a widget happens to be at the position of the mouse-down it does not move, whereas a left click would select and move that widget. A rectangle is shown, which can be used to select all widgets contained in or touched by the rectangle.
Right Click anywhere + Drag + Ctrl	With the control key the selection rectangle behavior is modified to select only those widgets that are completely contained within the rectangle.
Ctrl + Arrows	Moves the selected widget(s) by one pixel.
Alt	Removes handles for resizing (to enable dragging of small widgets) and borders (to get a better idea about the final look of the page)
Ctrl + Drag	"Snap to object" feature is active while dragging (can be changed in Grid Options)
Shift + Drag	"Snap to grid" feature is active while dragging (can be changed in Grid Options)
Ctrl + P	Show or hide the properties area. To use this shortcut, some widget(s) must be selected.

2.3 Common Properties

This section covers widget properties which are shared by many widgets, so they can be skipped when individual widgets are described.

2.3.1 Bind

Module, module inheritance, category, register, register inheritance and register offset are all properties defining binding and inheritance, see section 4.1. Example:



2.3.2 Position

Position of the widget (precisely its top left corner) can be specified manually, if dragging fails to do the job. The properties are X Position and Y Position. In this example is a widget that has its top left corner located 200 pixels from left and 100 pixels from top:

2.3.3 Size

Same way, the size can be entered manually, when designer chooses or cannot use resizing. The properties are X (width) and Y (height). Example:

2.3.4 External Binding (Class)

This property allows the designer to assign a class attribute to the widget. This can then be used for CSS styling and for JavaScript referencing. The idea behind this is that no matter how rich styling capabilities are given to the editor, there always might be something missing what the designer wants. This way the designer might use his own styles to achieve whatever he wants. Example for a widget with "`namedClass`" class property:


```
.namedClass * {
  color: red;
}
```

In addition to styles, class is also useful for referencing from JavaScript code, when the designer needs to add some special event handling to the widget. While in case of many widget it doesn't make sense, in case of e.g. Button widget this is the only way how to define custom behavior to be performed after

clicking it. This is how could be used a JavaScript reference for a widget with "`namedClass`" class property:

```
var widgetElement = document.getElementsByClassName("namedClass")[0];
```

Note that some widgets have some class in default (e.g. containers). This has been exposed for two reasons. The designer might want to add some styling to these default classes, and/or the designer might want to remove these default classes and thus remove the styling associated to these classes.

2.3.5 Reference

Two reference types need to be distinguished: the reference to a web page and the reference to a file.

2.3.5.1 Web Page Reference (Hyperlink)

When the designer wants to create a link to some other page (e.g. using widgets Text, Link Button or Hyperlink Area), he has four possibilities:

- Link outside the Slate system – then the link should start with 'http://' or 'https://' (e.g. http://google.com)
- Link inside the Slate system, but outside the pages generated by the editor – then the link should start with '/' (e.g. /text); this will work only in live mode
- Link from the designer realm to the Honeywell realm (i.e. link from the pages created by the designer to pages created by Honeywell) – then the link should start with '~' (e.g. ~FADash); this will work only in live mode
- Link to a local page generated by the editor – then the link should not start with the prefixes above (e.g. myPageName).

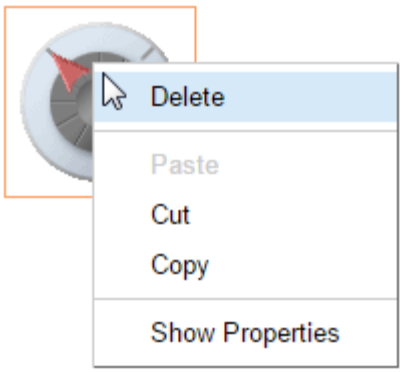
All four possibilities allow passing the parameters to the referenced page. In the last two cases this can be used to influence the inheritance, see section 4.1.

2.3.5.2 File Reference

When the designer wants to reference a file (e.g. using widgets Image, Audio, Video), he has three possibilities:

- File outside the Slate system – starting by 'http://' or 'https://', same as above
- File from the global scope – starting by '/', then it's expected to be found in the global folder, e.g. /global/img for images, /global/data for audio, etc.
- File from the project scope – not starting by prefixes above, then it's expected to be found in the project folder, e.g. /projects/<projectName>/web/img for images, etc.

2.4 Context menu



Context menu is shown after right mouse click on a widget. Typical content of widget's context menu is:

- Delete
- Paste
- Cut
- Copy
- Show Properties

Delete option removes actual widget from editor area. Copy option puts selected widget(s) to clipboard for further use. Cut does the same action as Copy and also removes selected widget(s) from the editor area. Paste adds widget(s) from clipboard to the editor area. These actions can be easily managed using the Undo/Redo functionality. Show Properties option opens the Properties area on the right side of the editor with properties of the selected widget. Same effect can be achieved by mouse double-click on the widget.

When more than one widget is selected and the context menu is shown, last item in context menu will display Common Properties area, which allows changing the position, size and classing properties for all selected widgets at once.

In addition to typical context menu options, container type widgets (see section 3.4) offers another options:

- Copy the container and everything in it
- Copy the container without widgets
- Include surrounded widgets
- Un-include all widgets

3 Widgets

In this section, all widgets will be described in the structure as they appear in the editor widget's list. Frequently used notions:

- Static widget – is a widget that doesn't change after loaded
- Dynamic widget – is a widget that can change after loaded, typically it updates its value (each 0.5s from the Base Module, and each 2s from the local server) according to what it obtained from the server, but it can also act differently
- Read-only widget – is a dynamic widget that can only obtain values from the server
- Write-only widget – is a dynamic widget that can only send values to the server
- Read/write widget – is a dynamic widget that can communicate with the server fully

3.1 NUMERIC

This group contains widgets, which work with numeric values.

3.1.1 Numeric Input

Dynamic read/write widget, which allows changing the numeric values of the particular register. Widget allows to set a count of decimal digits. Decimal digits are always displayed in entered count and a dot as a delimiter is used.

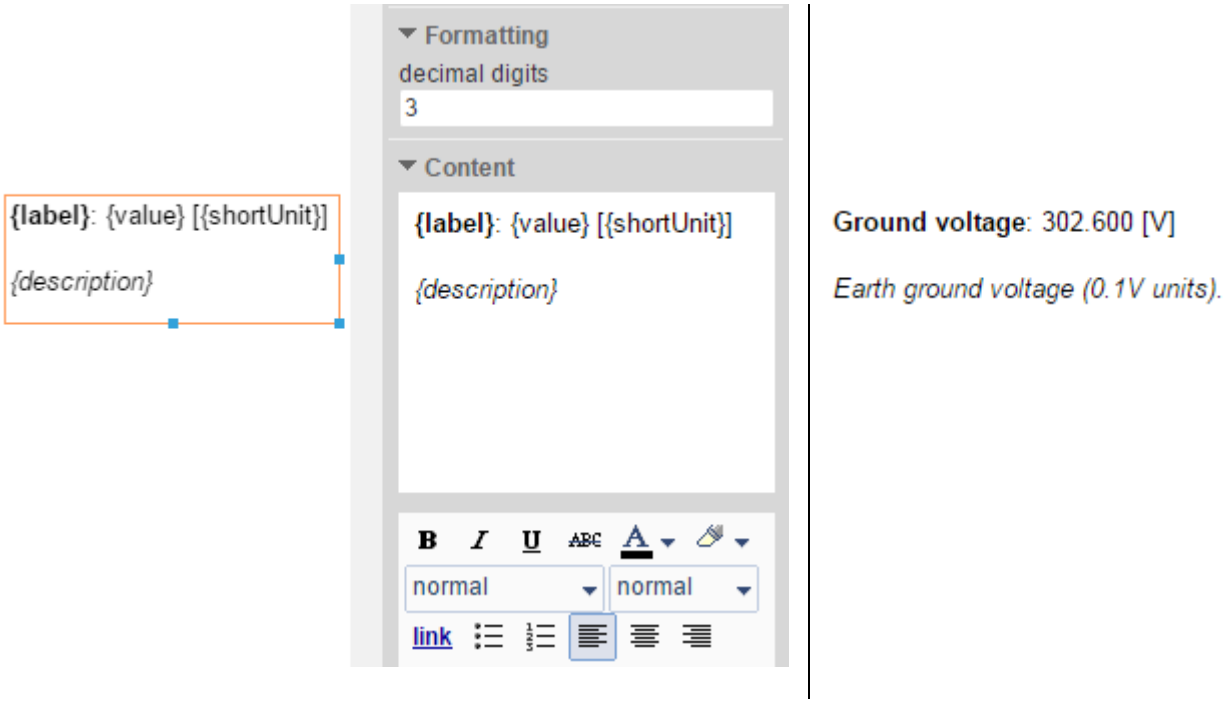
3.1.2 Numeric Output

Dynamic read-only widget, which displays the numeric values of the particular register. Widget allows to set a count of decimal digits. Decimal digits are always displayed in entered count and a dot as a delimiter is used.

3.1.3 Formatted Output

Dynamic read-only widget, which can be used for displaying not only numeric values of the particular register, but also its label, unit or description, or even static text. The widget allows to format the appearance (color, font, size, position, ...). The strings {value}, {label}, {shortUnit}, {longUnit} and {description} are replaced by the values obtained from the server when in live mode. It's possible to remove some of these special values. It's also possible to style (e.g. change color) of these special values, but the value must remain compact. The replacing will not happen if the special value is disrupted by a style, such as: {value}. Widget allows to set a count of decimal digits for the numeric value. Decimal digits are always displayed in entered count and a dot as a delimiter is used.

Example:



3.1.4 Slider

Dynamic read/write widget, which allows changing the numeric values of the particular register using a slider. Widget is able to read and write only values that are within the minimum and maximum limit properties. It is useful to set a width that is proportional to Slider's range.

3.1.5 Gauge

Dynamic read-only widget, which displays the numeric values of the particular register using a gauge. This widget offers rich possibilities of adjusting the appearance, as well as functional parameters (e.g.

amount of ticks or min/max values). Widget is able to read-only values that are within the minimum and maximum range properties.

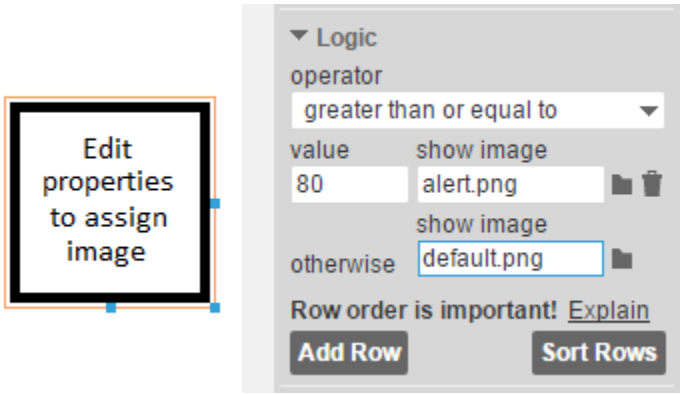
3.1.6 Horizontal Bar

Dynamic read-only widget often called as progression bar, which displays the numeric values of the particular register using a horizontal bar with adjustable appearance. Widget is able to display only values that are within the minimum and maximum range properties.

3.1.7 Vertical Bar

Dynamic read-only widget often called as progression bar, which displays the numeric values of the particular register using a vertical bar with adjustable appearance. Widget is able to display only values that are within the minimum and maximum range properties.

3.1.8 Output Image



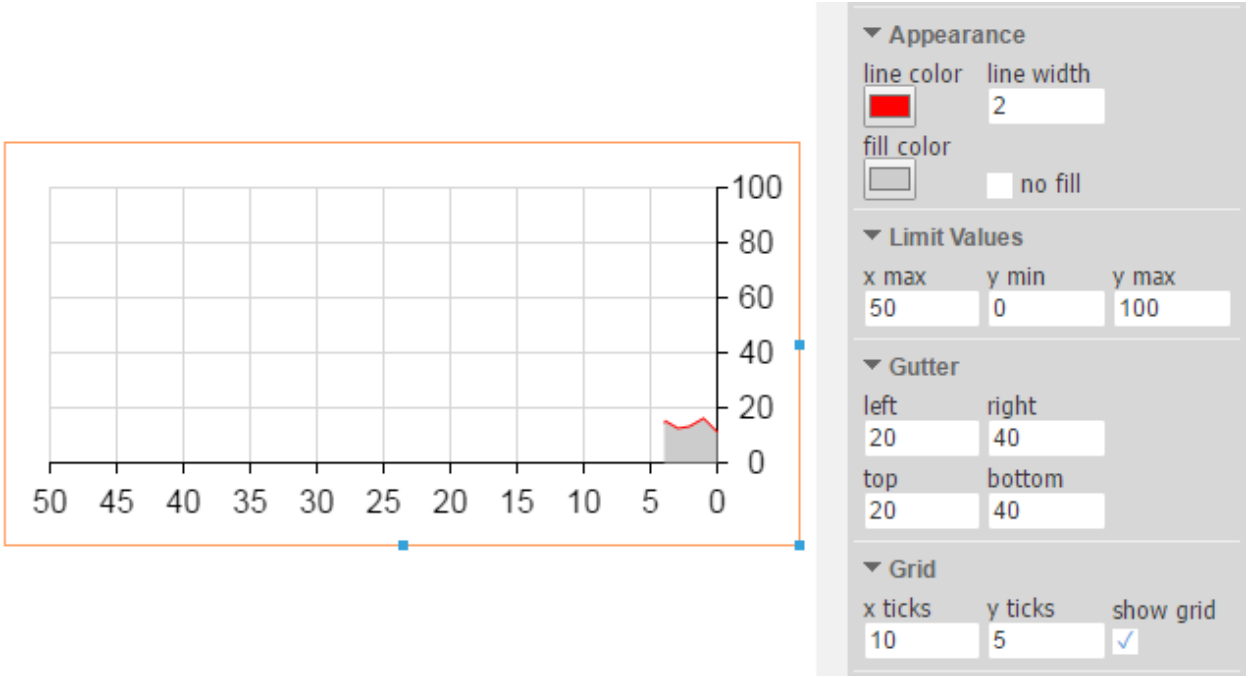
Dynamic read-only widget, which displays the specified image according to the numeric value of the particular register. The widget compares the value to the stored conditions and the displays the image of the first satisfied condition, or the image of the *otherwise* field, if no condition was satisfied. It is possible to change the form of the conditions. Available operators are:

- less than or equal to,
- less than,
- greater than or equal to,
- greater than,
- equal to,
- not equal to.

The amount of specified conditions is not limited and can be changed by button Add Row and button for removing the row ( ). Order of conditions is important because the test selected by the "operator" drop down list is applied to the conditions in top - down order; that is, in the order that the conditions appear in the properties list. The Sort Rows button sorts conditions according to selected operator and values.

Any image used by this widget should be available and placed according the instructions in section 4.2.2. List of available images can be invoked by button. Path to an image has no prefixes for local images, for global images the path starts with "/", see section 2.3.5.1.

3.1.9 Graph

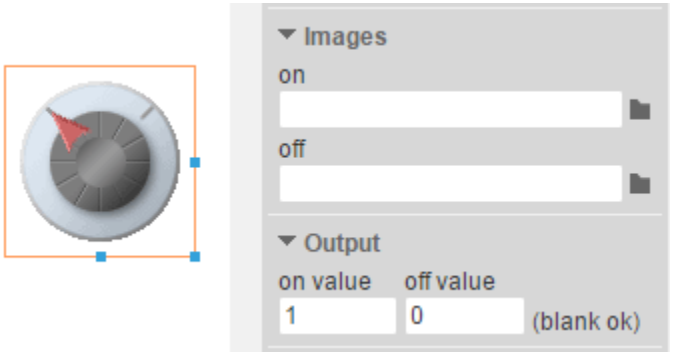


Dynamic read-only widget, which displays the numeric values of the particular register using a graph. In running mode, widget is not offering any user interaction. The widget periodically asks for a value of register and progressively fills the chart with it. Appearance of this widget is adjustable and widget also offers rich possibilities of adjusting the functional parameters.

3.2 BOOLEAN

This group contains widgets, which work with Boolean values.

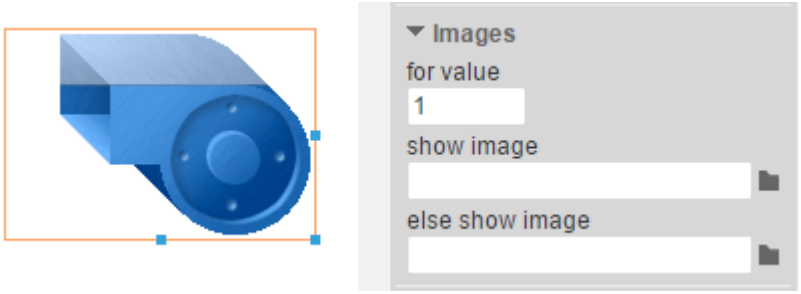
3.2.1 Switch



Dynamic read/write widget working with two states on and off. Its read mode works in a way that value specified by "On Value" property means ON and everything else means OFF. Its write mode (i.e. when user clicks it) switches the state and sends the specified On or Off Value to the server. It is also possible to leave Off Value blank and then nothing is sent to the server, when user turns the switch off. The appearance of the switch can be customized via "On" and "Off" attributes. “Reset Size” button reverts any width or height changes to original size of an image. Widget works only for number values.

Widget uses default on and off images. Any other image used by this widget should be available and placed according the instructions in section 4.2.2. List of available images can be invoked by ■ button. Path to an image has no prefixes for local images, for global images the path starts with "/", see section 2.3.5.1.

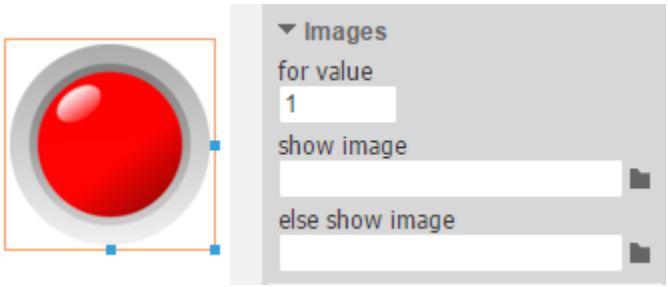
3.2.2 Binary Image



Dynamic read-only version of the Switch widget. It operates only with ON value. If ON value is read from the register, it shows the ON state, otherwise it shows the OFF state. “Reset Size” button reverts any width or height changes to original size of an image. Widget works only for a number value.

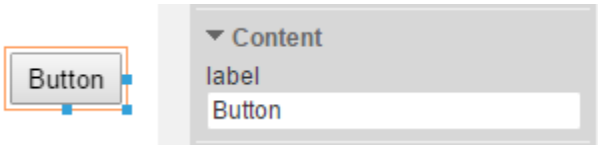
Widget uses default state images. Any other image used by this widget should be available and placed according the instructions in section 4.2.2. List of available images can be invoked by button. Path to an image has no prefixes for local images, for global images the path starts with "/", see section 2.3.5.1.

3.2.3 Binary LED



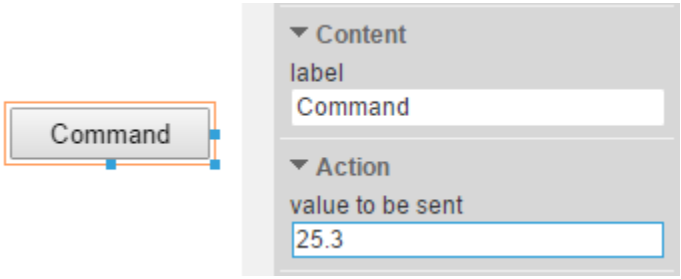
Dynamic read-only version of the Switch widget. Basically the same as the Binary Image widget described in 3.2.3. The only difference is in the default image which symbolizes a LED.

3.2.4 Button



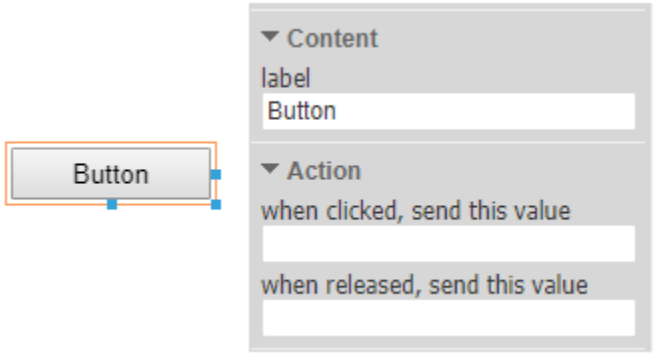
General widget that represents a button. Its functionality can be specified by user-defined JavaScript code and referenced by Class attribute, see section 2.3.4.

3.2.5 Command



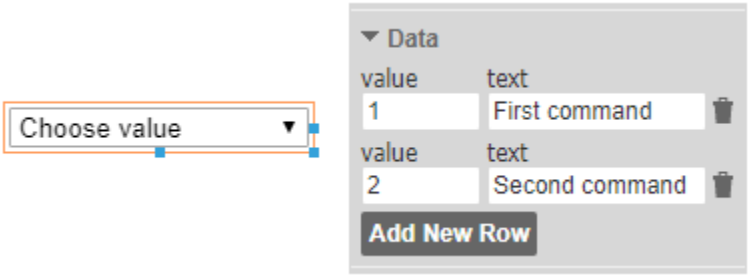
Dynamic write-only widget which sends specified value to a specified register when clicked. Value type depends on the register value type.

3.2.6 Momentary Button



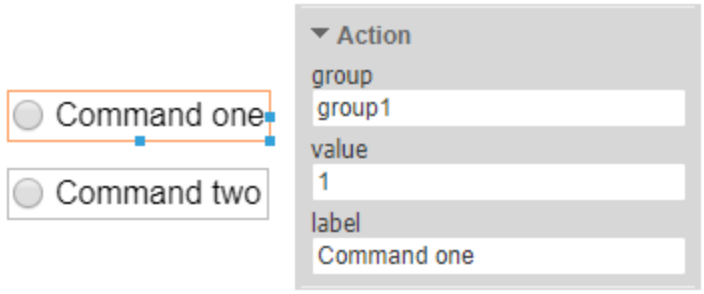
Dynamic write-only widget which sends specified value to a specified register when clicked and another value when released. Value type depends on the register value type.

3.2.7 Select Command



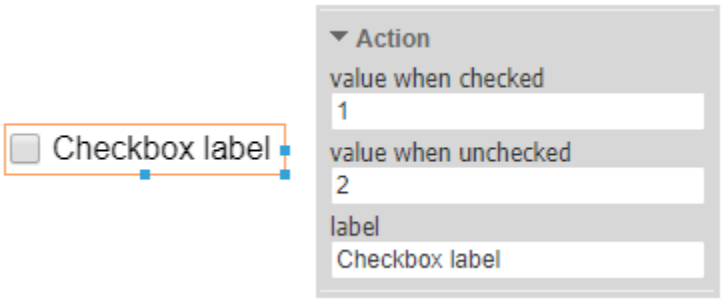
Dynamic write-only widget which sends selected specified value to a specified register when clicked. Value type depends on the register value type.

3.2.8 Radio Button



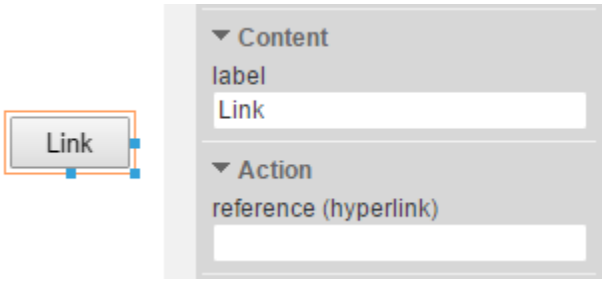
Dynamic write-only widget which sends specified value to a specified register when checked. It can be combined with another Radio Button widgets as a group by specifying the same group name. Value type depends on the register value type.

3.2.9 Checkbox



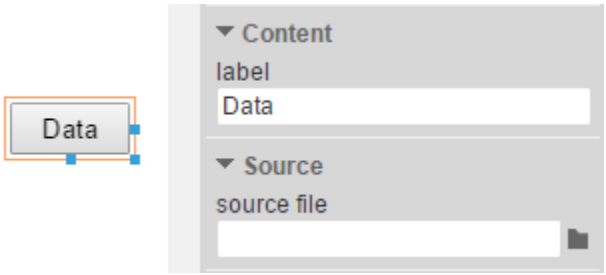
Dynamic write-only widget which sends specified value to a specified register when checked or unchecked. Value type depends on the register value type.

3.2.10 Link Button



Static widget acting as a link. When clicked, the user is navigated to a specified page. For navigation rules see the section 2.3.5.1.

3.2.11 Data Button



Static widget allowing the designer to link any supported data file (such as PDF or other document, video, audio). The button redirects the browser to this file. What the browser does with the file (if he can open and show it) depends only on the browser. Files used by this widget should be available and placed according the instructions in section 4.2.3. List of available files can be invoked by ■ button. Rules for path to a file are in section 2.3.5.1.

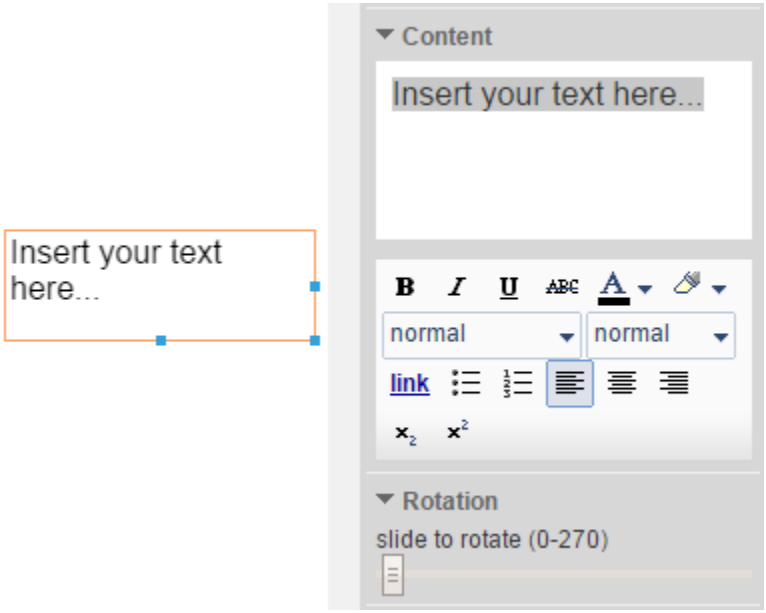
3.2.12 Back Button

Static widget mimicking the “Back to previous page” browser functionality. It redirects the user to previous viewed page in browser, if any.

3.3 TEXT

This group contains widgets, which work with text values.

3.3.1 Text



Static widget that represents rich formatted text. Apart from usual rich text formatting, the links can be added, that follow the same rules as the links in other widgets (see section 2.3.5). Also the widget can be rotated.

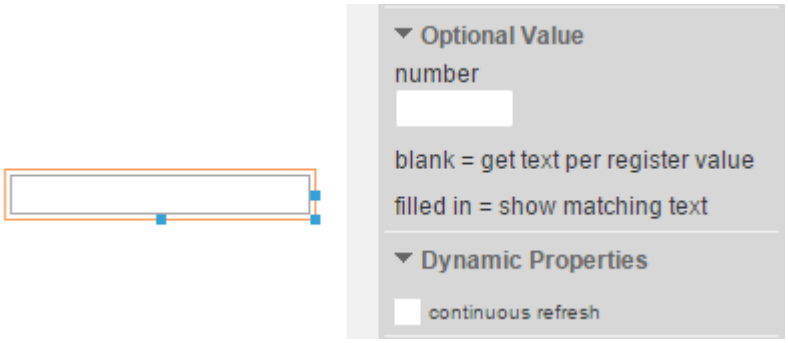
3.3.2 Text Input

Dynamic read/write widget, which allows changing the text values of the particular register. Widget works with numeric values too.

3.3.3 Text Output

Dynamic read-only widget, which displays the text values of the particular register. Widget works with numeric values too.

3.3.4 Enum Text Output



Dynamic read-only widget which works with enumeration values of the register. When page is loaded, this widget retrieves all text values of the particular register and then operates the same way as any other read-only widget, just instead of showing numeric values 1, 2, 3, ..., it shows the text values representing these numeric values. When this widget is bound to the register of type different than enumeration, it will not work.

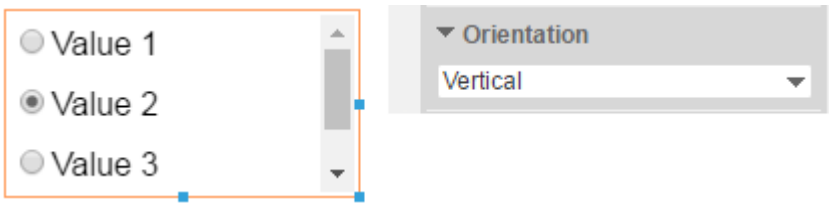
Instead of showing actual enumeration text value, widget allows fixing the displayed value to an optional constant number which represents one of the possible enumeration text values.

Default behavior is a semi-dynamic, when after loading the initial set of enumeration texts widget doesn't contact server. To enable full dynamic behavior, check the continuous refresh property.

3.3.5 Enum Select Input

Dynamic read/write widget which works with enumeration values the same way as Enum Text Output widget above, in addition it allows also changing the value. On the other hand it does not allows fixing the value as a constant and works only in semi-dynamic mode, which means that data are loaded only once during the initialization.

3.3.6 Radio Select Input

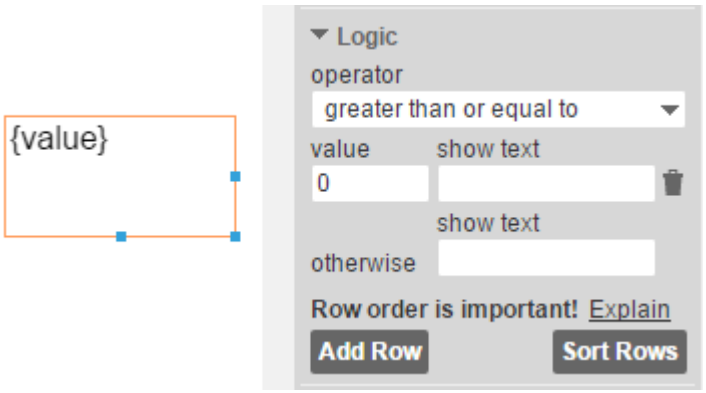


Dynamic read/write widget which works with enumeration values the similarly as the Enum Select Input widget above, but instead of showing values within the select box it shows values as the radio list. Note that if the widget dimensions are smaller than dimensions of the list, a scrollbar will appear as shown on the image. Widget allows showing radios in vertical or horizontal orientation.

3.3.7 Tooltip

Static widget allowing to specify what text is shown after hovering with mouse in the specified area. This widget will have no effect on touch displays.

3.3.8 Conditional Text



Dynamic read-only variant of the Output Image widget (described in 3.1.8) with the difference of handling texts instead of images.

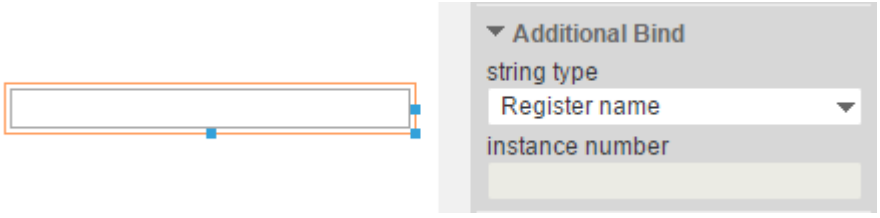
3.3.9 TEXT > LANGUAGE

This subgroup contains widgets, which work with built-in texts to enable their field-editing multi-language modifications.

3.3.9.1 Language Selection

Dynamic widget, which according to selected language allows to change the name of things such as terminal names, register names, register options, register descriptions, etc. The chosen language setting is stored in a cookie in the display/browser and persists as the choice for that display until it is changed. Other displays are not affected. If the display/browser does not contain a language setting cookie, then English is assumed.

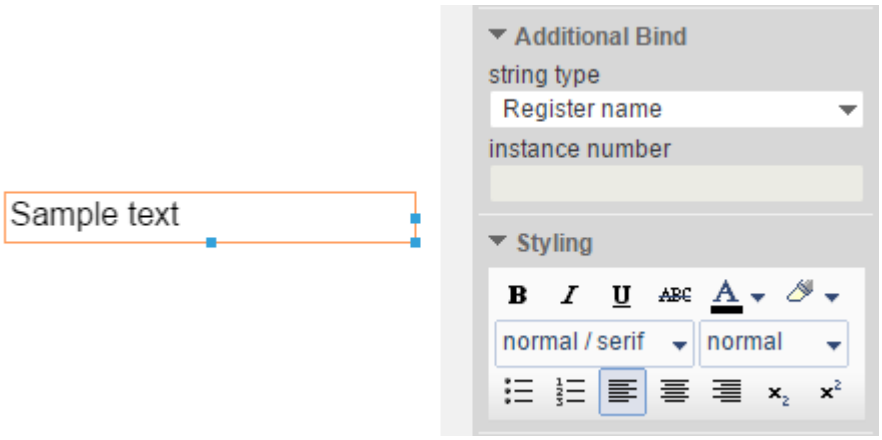
3.3.9.1 Language Text Input



Semi-dynamic read/write widget, which allows changing the text values of chosen register string type. In

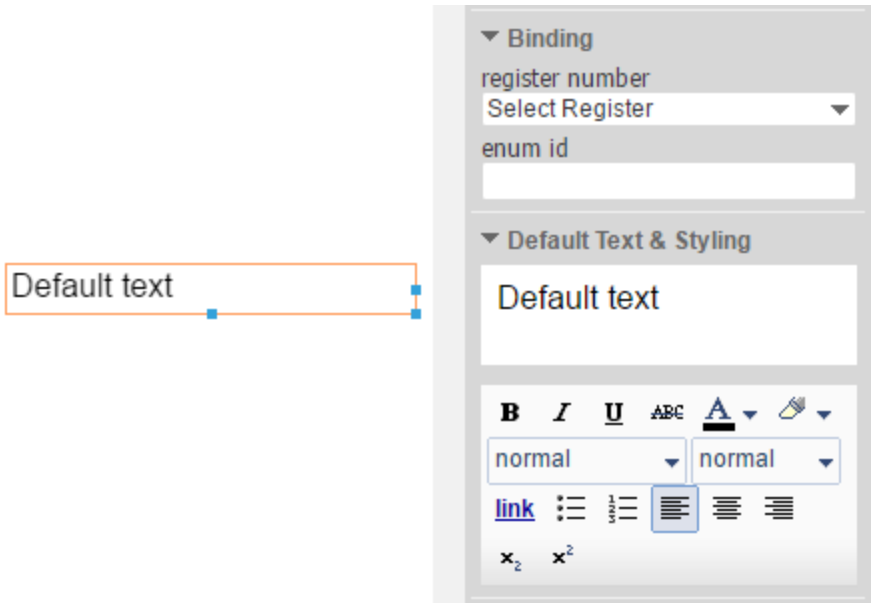
case of enumeration string type, widget works with text based on chosen instance number. Widget's value is loaded only during the initialization.

3.3.9.2 Language Text



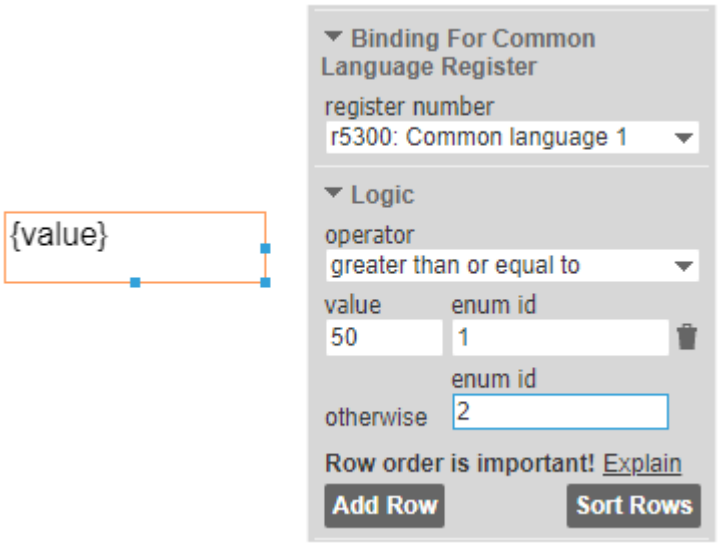
Semi-dynamic read-only widget that displays rich formatted text of chosen register string type. In case of enumeration string type, widget displays text based on chosen instance number. Widget's value is loaded only during the initialization.

3.3.9.3 Common Language Text



Semi-dynamic read-only widget, which automatically points to Base module and System Status category. The developer thus needs to pick Common language register number (range 5300-5399) and an Enum ID. When no text or "undefined" is found, widget displays defined default text. Widget allows displaying a rich formatted text. Widget's value is loaded only during the initialization.

3.3.9.4 Common Language Conditional Text



Dynamic read-only widget, which compares specified register value with a value from Base module and System Status category. The developer thus needs to pick Common language register number (range 5300-5399) and set a logic for specified register value – e.g. when a register value is greater or equal to 50, show the m1r5000 enum id 1 value, otherwise show value for enum id 2. Widget allows displaying a rich formatted text.

3.4 CONTAINER

Containers are type of widgets, into which other widgets can be placed. When moving a widget over a container, the container is highlighted. A container can be placed into a container (and this can be applied recursively without any restriction). When moving a widget over these sets of containers, only the "best" container is highlighted. The best container is the one, which is the most inner, but still contains the widget.

There are two distinct notions:

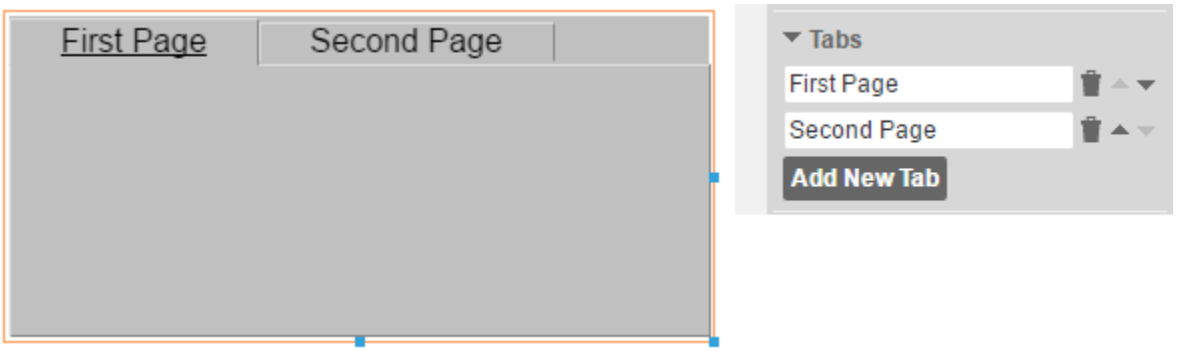
- state when the widget appears to be in the container, i.e. the widget is fully surrounded by the container – let's call this "in the container"
- state when the widget is inside the DOM element of the container, so when container is moved, also widget is moved – let's call this "inside the container"

Normally, these two states overlap. When a widget appears *in the container*, it is *also inside the container*. It is however possible to reach the state, when only one of these states is true. When you drag the widget and put the container on top of it, it looks like it is *in the container*, but is not *inside the container*. On the other hand, if you select two widgets in the container and move them out in a way, that the dragged one still resides in the container, while the other one is already outside, then this second one doesn't appear to be *in the container*, still it is *inside the container*. These cases are usually not desired and should be avoided as they may lead to confusion.

3.4.1 Pane

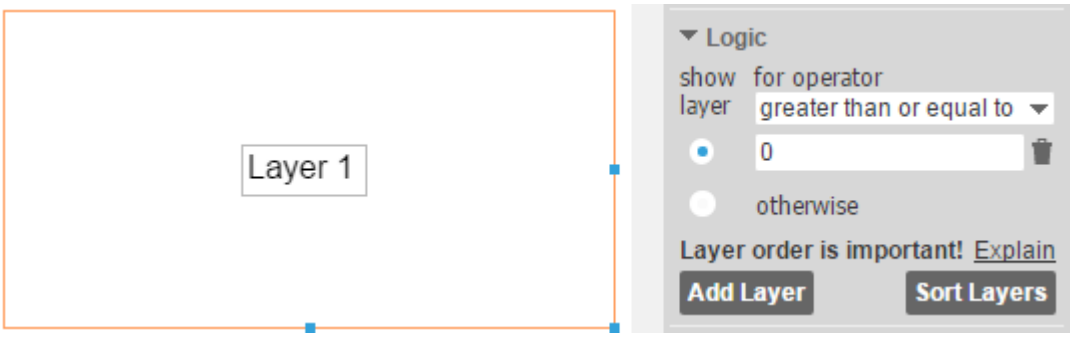
Static widget, which can be used for container inheritance (for details see section 4.1) or styling reasons (for details see section 4.3). In case of no custom styling, it is not drawn in the live mode.

3.4.2 Tab



Static widget useful for dividing the content into its tabs. Tabs can be added and removed or their order can be changed. It is not recommended to use so many tabs, that it fills more than one line. The functionality is not broken by that, but the look suffers.

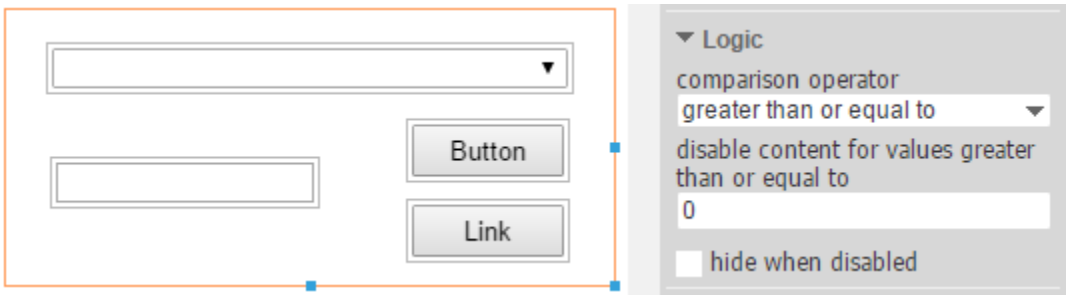
3.4.3 Multi-Display Pane



Dynamic read-only widget, which displays specified layer according to the numeric value of the particular register. The widget compares a value to the stored conditions and the displays the layer of the first satisfied condition, or the layer of the *otherwise* field, if no condition was satisfied. It is possible to change the form of the conditions. Available operators are:

- less than or equal to,
- less than,
- greater than or equal to,
- greater than,
- equal to,
- not equal to.

3.4.4 Disabling Pane



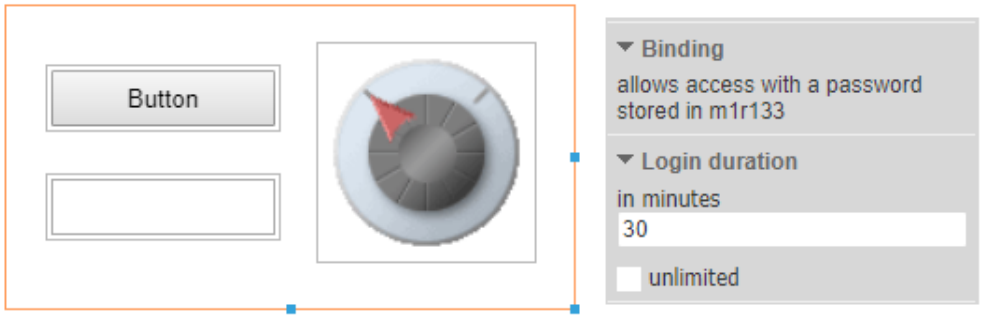
Dynamic read-only widget. It reads given register value and compares it with given value. As long as this is evaluated to true, all widgets inside the disabling pane are disabled. It means they are grayed out and their write behavior is removed. Their read capabilities are untouched though. Once the comparison is evaluated to false, widgets are enabled again. Similar behavior applies for hiding pane's content. If the "hide when disabled" is checked, whole pane's content is hidden instead of disabled when the conditions are met. Available operators are:

- less than or equal to,
- less than,
- greater than or equal to,
- greater than,
- equal to,
- not equal to,
- between,
- not between.

It is not recommended to put widgets with inherent enabling/disabling ability (e.g. FA buttons or custom widgets) into the Disabling Pane, as it may lead to flickering, i.e. constant changing from enabled to disabled and back due to conflicting conditions. Disabled state however works directly only for HTML form fields. For other widgets like Image which are not actually HTML form fields, is a possibility to style disabled state with *.paneDisabled* CSS class. This CSS class is automatically added for Disabling Pane widget when disabled.

Widget can be used for container inheritance (for details see section 4.1).

3.4.5 Restricted Area Pane

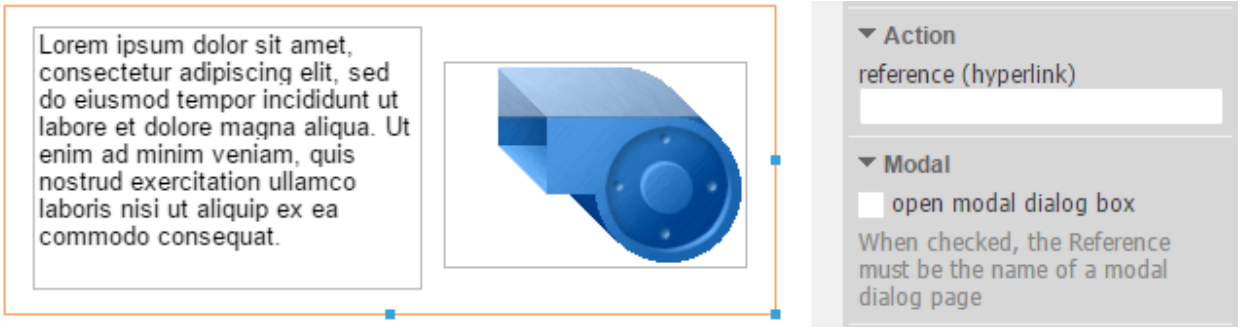


Dynamic read-only widget. Until a correct password is entered which matches with m1r133 register value, all widgets inside the disabling pane are disabled. It means they are grayed out and their write behavior is removed. Their read capabilities are untouched though. Once a correct password is entered, all widgets will be enabled for specified duration in minutes.

Disabled state however works directly only for HTML form fields. For other widgets like Image which are not actually HTML form fields, is a possibility to style disabled state with *.paneDisabled* CSS class.

Widget can be used for container inheritance (for details see section 4.1).

3.4.6 Hyperlink Area



Static widget which allows defining an (invisible) area that is clickable and leads to the specified location after clicking (see section 2.3.5.1 for details about links). If some clickable widget position collides with this widget, the conflict is resolved in the following way:

- If the other widget is *inside in the Hyperlink Area*, then the widget can handle the click event, but can also pass it to the parent, it depends on the widget (widgets reacting to clicking, such as Link Button or Image with reference set will handle the click event by themselves, while widgets not reacting to clicking will pass the click to the Hyperlink Area).
- Otherwise, if the other widget is behind the Hyperlink Area, it cannot be clicked.
- Otherwise, if the other widget is in front of the Hyperlink Area, it can handle the click event and will not pass it to the parent (no matter if it is actually clickable or not).

Widget also allows to be used similarly as the Open Dialog Button widget described in **Error! Reference source not found.**.

3.4.7 Modal Dialog Box

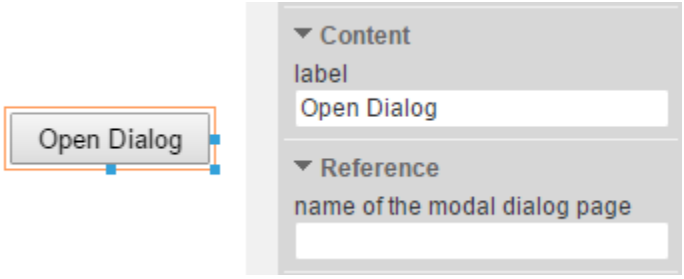
This container widget is tightly linked to the next widget (Open Dialog Button) and these two implement the Modal Dialog support, which is a feature allowing a user to create custom modal dialog windows and buttons to invoke them. The expected scenario is following:

A normal page should NOT contain any Modal Dialog Box widget, but it might contain Open Dialog Button widgets. A page used to model the dialog content should only contain the Modal Dialog Box widget (and its content), which is shown when the particular button is clicked.

When an unexpected state is encountered, the editor recovers in the following way. When a normal page is loaded, all Modal Dialog Box widgets are hidden as well as their content. When a page is invoked via Open Dialog Button widget, everything outside Dialog widgets is ignored and only the first Dialog widget is shown.

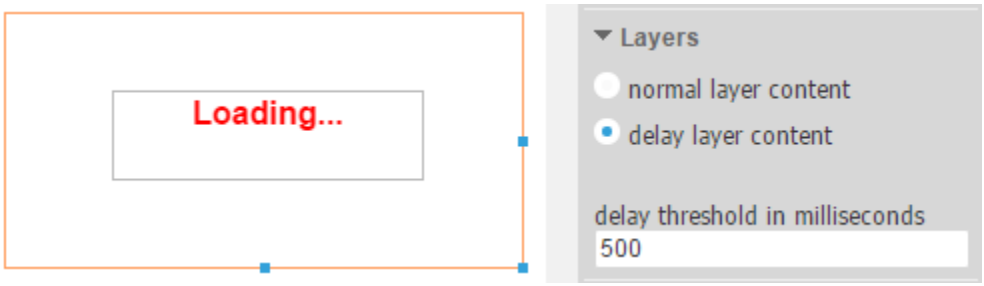
Due to performance reasons, the Modal Dialog concept is recommended only for simple pages (text, buttons, static images).

3.4.8 Open Dialog Button



The only widget that can invoke Modal Dialog windows. The reference is expected to be local (i.e. not starting with http:// or with /) and it should point to the page with the Modal Dialog Box widget (see above, for details about reference options see 2.3.5).

3.4.9 Connection Delay Pane

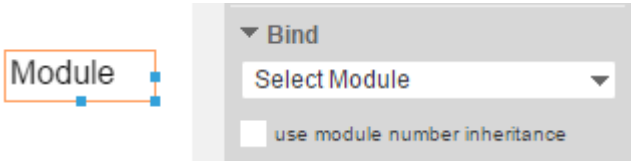


Dynamic read-only widget, which can be used for displaying a delay alert of pending value request. The delay layer is displayed instead of the normal layer after expiring specified delay threshold. When all pending value requests are completed, the normal layer is displayed again.

3.5 FUEL AIR COMMISSIONING

This category contains widgets dedicated to the fuel air commissioning. Their usage doesn't make sense outside the FA scope (i.e. outside the page(s) with FA). In the editor mode all widgets are preloaded with some example values.

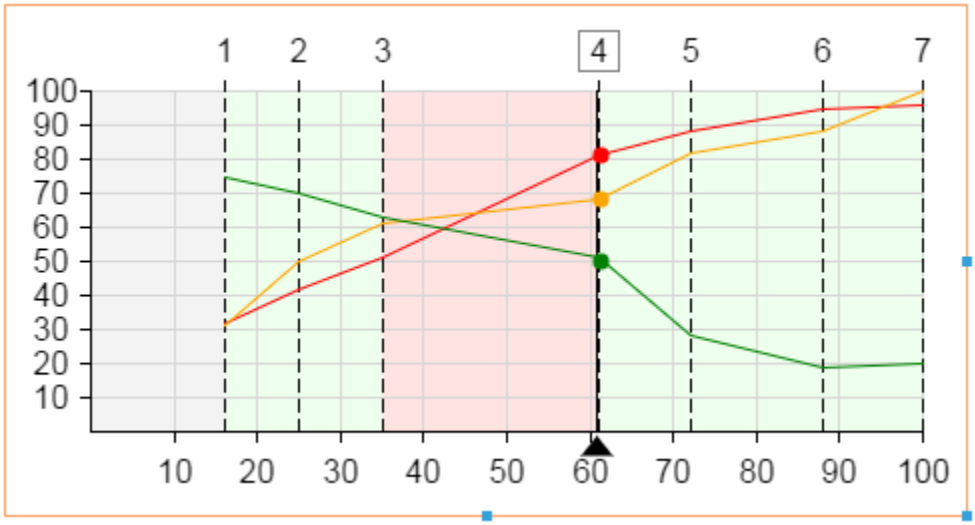
3.5.1 Module Selector



FA page may contain tens of widgets. It is expected that all these widgets will be linked to the same module. Thus, to avoid having to set this module to every single widget, there is this module selector widget which defines the mapping of all FA widgets on the page to the module set in it. All FA widgets require presence of this widget anywhere on the page. This widget is not rendered. Due to technical restrictions, inheritance is supported only partially. It means that this widget processes the inheritance data when the page is loaded, but it ignores the future changes (e.g. done by Module Selector widget, 3.7.1). Thus, when inheritance needs to be supported for FA widgets, passing module identifiers through URL should be used (e.g. Link Button with “FaPage?module=m3” or “FaPage?module=dynamic”, for details see section 4.1).

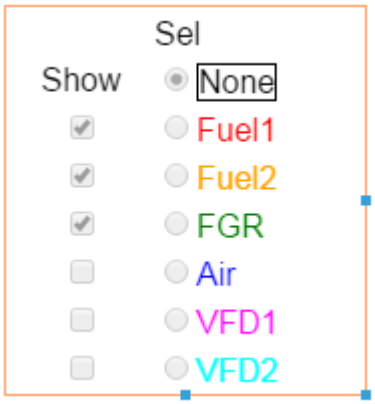
3.5.2 FA > GRAPH

3.5.2.1 Graph



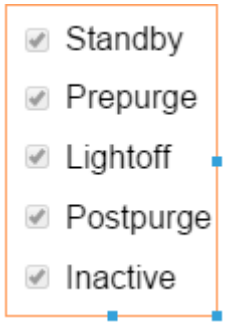
Dynamic read-only widget. It displays the curves, points, throttle position (measured and commanded), presets, status of the throttle, validity of the segments and curve selection. The presets and curve selection are not shown in the editor mode in order to avoid having sample widget too busy.

3.5.2.2 Select Curve



Dynamic read/write widget. The part of selection is the dynamic one (so read from the server, sent to the server), while the part of showing is the state within the scope of the page (it is not sent anywhere, which also means that changes to show/hide state of curves on this page is not reflected on other pages). Showing and hiding curves has effect only on the graph, the table shows always all available curves (columns). It is possible to set default settings of this widget.

3.5.2.3 Select Presets



Static widget. Same way as show/hide part of the select curve widget the state is only within the page scope and sent nowhere.

3.5.3 FA > TABLE

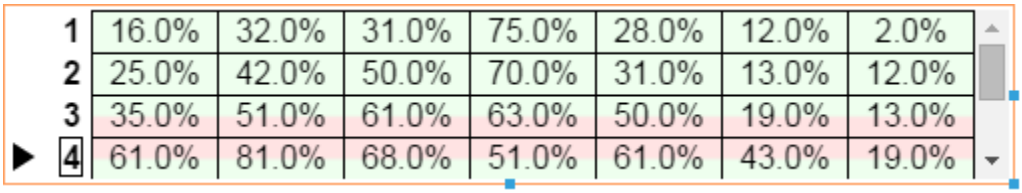
All FA table widgets (except unit selection widget) are designed in a way that they together form a table. The reason of separation them into independent widgets rather than having one table widget (same way as one graph widget) is that the designer may want to change the order of the elements or leave some out. However, it brings some burden to a designer. If he wants to create an entity resembling a table, he needs to pay attention to horizontal alignment and width of all widgets. Both can be set manually (size and x position), the alignment can also be achieved by using a snap-to-grid feature (drag while holding the shift key).

The table widgets always display all available curves in columns. If the designer has knowledge about amount of curves in the target system, he is advised to set this amount to all table widgets, so that he can see how the widgets render and if columns aren't too broad or narrow. This settings is however only cosmetic and has no effect beyond the editor mode, because in the live mode all widgets always display all available columns.

3.5.3.1 Header

Dynamic read-only widget. It displays curve names and colors.

3.5.3.2 Content



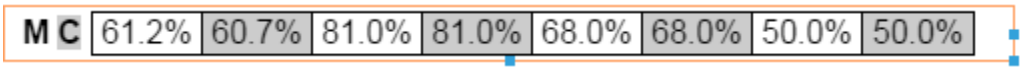
	1	16.0%	32.0%	31.0%	75.0%	28.0%	12.0%	2.0%
	2	25.0%	42.0%	50.0%	70.0%	31.0%	13.0%	12.0%
	3	35.0%	51.0%	61.0%	63.0%	50.0%	19.0%	13.0%
▶	4	61.0%	81.0%	68.0%	51.0%	61.0%	43.0%	19.0%

Dynamic read-only widget. It displays all curve points values, segment validity and throttle position and status.

3.5.3.3 Position Icons

Dynamic read-only widget. It displays actual points statuses (moving, not moving, above curve etc).

3.5.3.4 Footer



Dynamic read-only widget. It displays measured and commanded throttle and actual points.

3.5.3.5 Presets

Dynamic read-only widget. It displays preset values of the first shown preset. Showing and hiding presets is done by Select Presets widget (see 3.5.2.3).

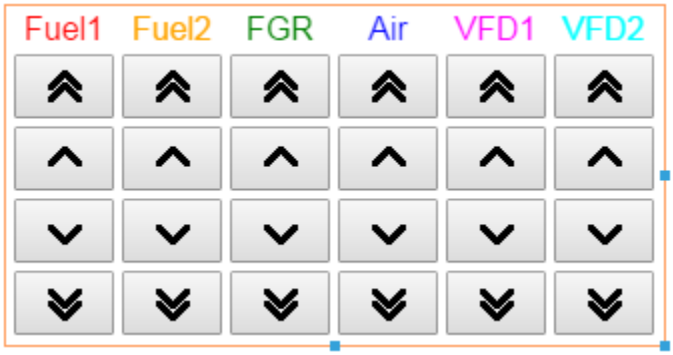
3.5.3.6 Unit Selection

Static widget switching between actual and percentage values of the entire table. Its state is within the page scope and not sent to the server.

3.5.4 FA > MOVEMENT BUTTONS

All widgets contained in this category simply send the movement commands to the server. Vertical movement widgets are completely fixed, horizontal movement widgets send commands to move the throttle in default, but if throttle/point widget (see below) is present, they can be used to move the point too.

3.5.4.1 Matrix Button



Special case of movement buttons is the Button Matrix widget, which is a combination of (Large) Up/Down movement widgets together, where each column represents a curve. If no curve is selected or button in other than selected column is pressed, then the Button Matrix widget at first selects and marks the curve represented by target column. Amount of columns in live mode is determined by the amount of curves in target system. If the amount is less than six, buttons will be stretched in their width to maximum of 50 pixels.

3.5.5 FA > COMMAND BUTTONS

Widgets contained in this category represent buttons sending a fixed value to a fixed register of a module defined by (FA) module selector widget.

3.5.5.1 Create, Delete, Trim, Stop, Update, Confirm Prepurge, Confirm Lightoff

Dynamic write-only widgets sending particular commands.

3.5.5.2 Preset Commands

Dynamic write-only widget consisting of three parts. The drop down menu where any of presets can be chosen, the button performing the setting command and the button performing the go to command. The widget can be oriented horizontally or vertically which can be changed by resizing.

3.5.6 FA > MOVEMENT CONTROL

Widgets having impact on movement.

3.5.6.1 Throttle/Point

Static widget. Changing its state is not sent to the server, it only influences what commands are sent in case of clicking horizontal movement buttons.

3.5.6.2 (Small|Large) (Left/Right|Up/Down) widgets

Dynamic read/write widgets displaying enumeration values of a particular register and allowing to change the value of this register. These widgets control the movement steps of the movement commands.

3.5.7 FA > TRIM

Widgets dedicated to FA Trimming.

3.5.8 FA > TRIM > ACTUATOR TABLE

Similar concept to FA > TABLE part, the table however only displays information about the trimmed actuator. Only Header and Content widgets are contained here.

3.5.9 FA > TRIM > POINT TABLE

Again, Header and Content widgets used to show information about the actual point of the trimmed actuator.

3.5.10 FA > TRIM > SET TRIM BUTTONS

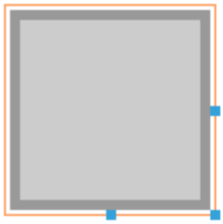
Buttons used for setting the setpoint in various ways.

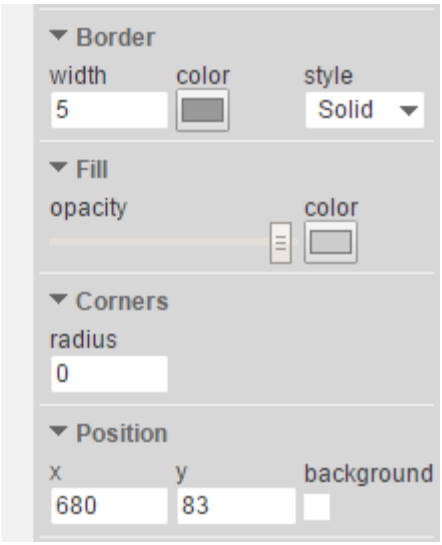
3.5.11 FA > TRIM > MOVEMENT BUTTONS

Buttons allowing the user to move the point in the trimming process.

3.6 MEDIA

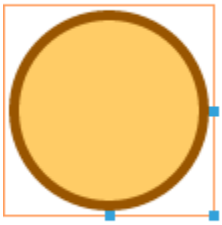
3.6.1 Rectangle

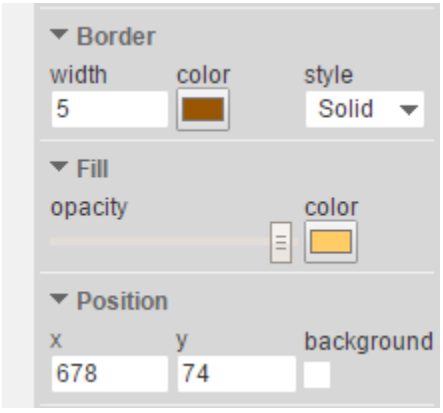




Static widget representing rectangle which can have its border (width, color, style), fill (color, opacity) and radius of corners defined. This widget offers a special background behavior which disables widget handling in editor mode by left mouse button. When the background property is enabled, widget can be handled only by widget's context menu available after clicking on widget by right mouse button. This offers more comfortable handling of other widgets that overlaps the Rectangle widget.

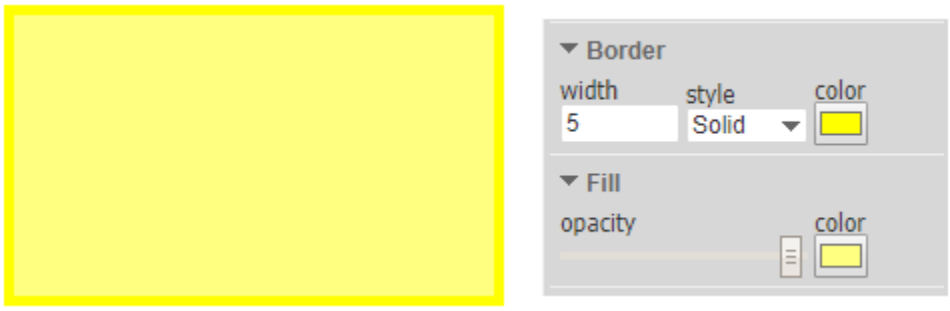
3.6.2 Ellipse





Static widget representing a circle or an ellipse (depending on how it is stretched). Widget offers special background behavior in same manner as the Rectangle widget (see section above 3.6.1).

3.6.3 Background

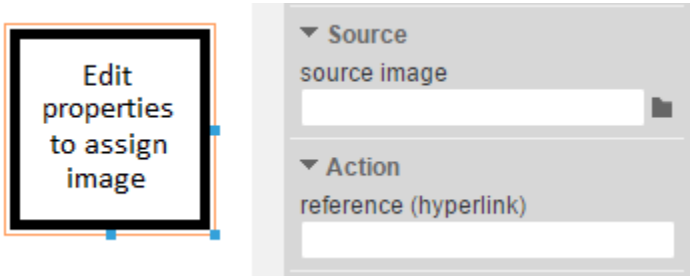


Similar than the Rectangle widget 3.6.1, but this widget uses only the background behavior. This behavior disables widget handling in editor mode by left mouse button. Widget can be handled only by widget's context menu available after clicking on widget by right mouse button. This offers more comfortable handling of other widgets that overlaps this widget.

3.6.4 Line

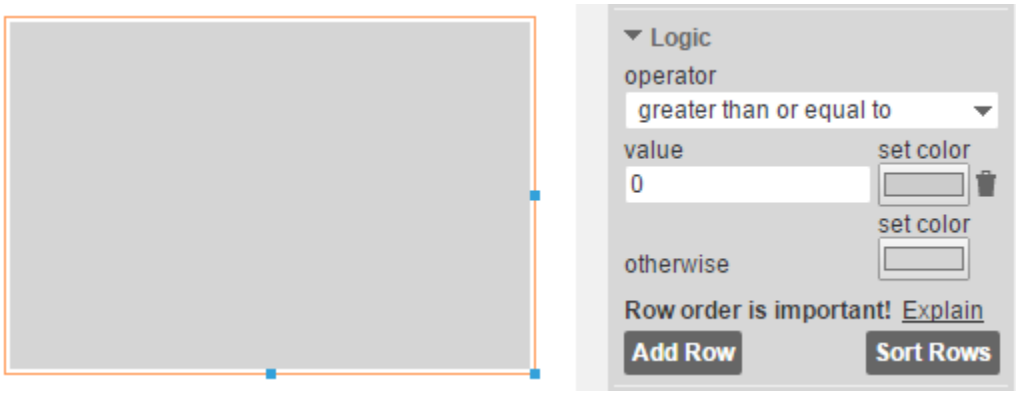
Static widget representing line which can have its width, color and orientation defined.

3.6.5 Image



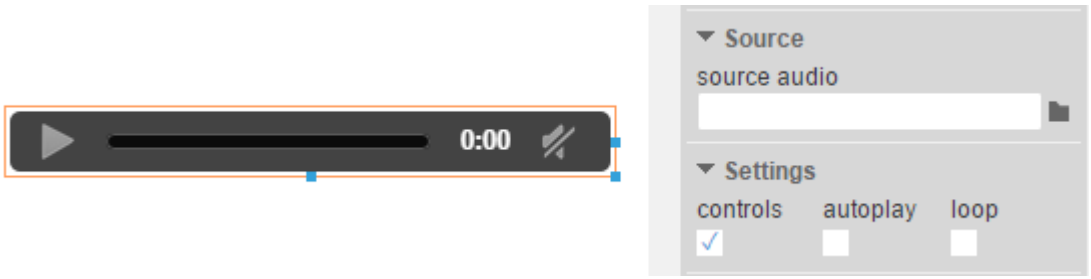
Static widget representing image which can be given a link. For specifying the image, see section 2.3.5.2. For specifying the link, see section 2.3.5.1. Widget offers special background behavior in same manner as the Rectangle widget (see section 3.6.1).

3.6.6 Conditional Color



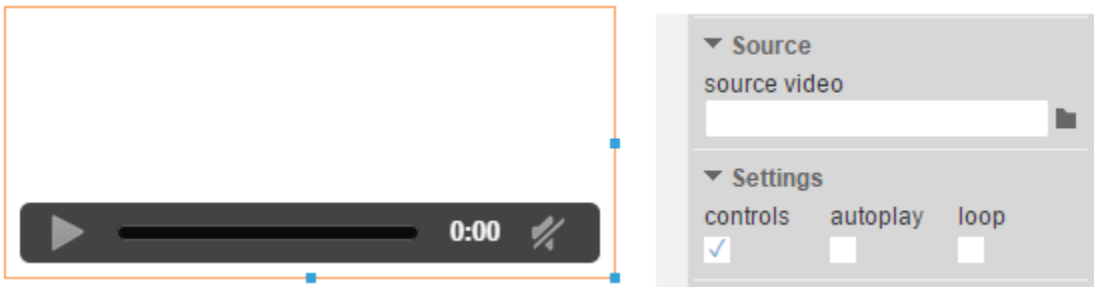
Dynamic widget similar to Output Image widget described in section 3.1.8, with the difference of handling colors instead of images.

3.6.7 Audio



Static widget allowing to play audio. For specifying the audio file, see section 2.3.5.2. Widget offers to show/hide controls, set the autoplay option or the loop option (in infinite loop). It provides a HTML5 standard for playing audio files. Currently, there are 3 supported video formats: MP3, Wav, and Ogg.

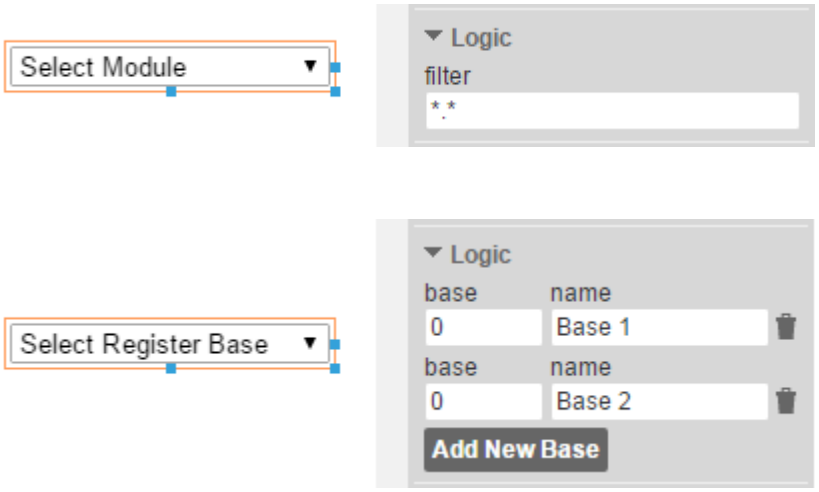
3.6.8 Video



Static widget allowing to play video. For specifying the video file, see section 2.3.5.2. Widget offers to show/hide controls, set the autoplay option or the loop option (in infinite loop). It provides a HTML5 standard for playing video files. Currently, there are 3 supported video formats: MP4, WebM, and Ogg.

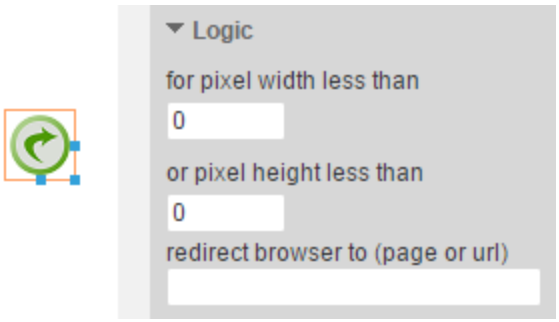
3.7 SPECIAL

3.7.1 Module, Register



Static widgets having great impact on inheritance, for details see section 4.1.

3.7.2 Resolution Redirect



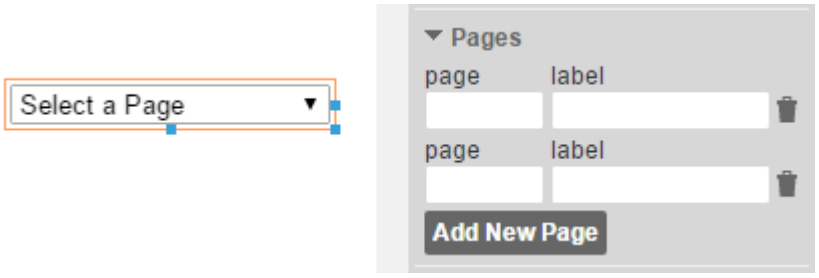
Special widget that can be used to react to different browser resolutions. It is not rendered and after loading the page it has no effect. But during the page loading it compares the size of the browser window with the values specified in this widget and if the browser uses smaller resolution then it is redirected to a specified page, which is expected to be able to handle this small resolution.

These widgets can be cascaded, e.g. for resolution smaller than 801x601 redirect to indexDisplay page, for resolution smaller than 1200x1000 redirect to indexSmall page, otherwise stay on index page. In order to make this to work, the widgets need to be created in the order in which they are required to be evaluated, i.e. smallest resolution first.

Alternative way of cascading, which is safer but takes longer when viewed, is to add one Resolution Redirect widget per page and create chain of cascading across the pages, i.e. widget on the index.html page would redirect browser to indexSmall.html, if smaller than 1200x1000, and the widget **on the indexSmall page** would redirect browser to indexDisplay page, if smaller than 801x601.

For redirection URL rules see the section 2.3.5.1.

3.7.3 Page Select

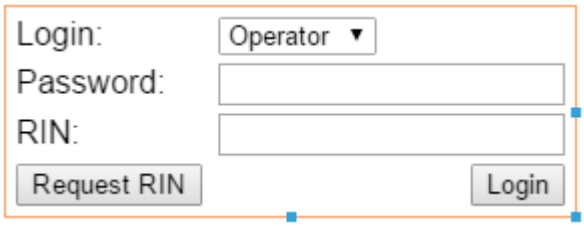


This widget allows to create a navigation between pages. Pages are defined by their name and the label property allows better recognition (determines what will be displayed in the drop box). Creating such navigation only once and using it on multiple pages can be easily done with the Scrapbook functionality. For page navigation rules see the section 2.3.5.1.

3.7.4 SPECIAL > AUTHENTICATION

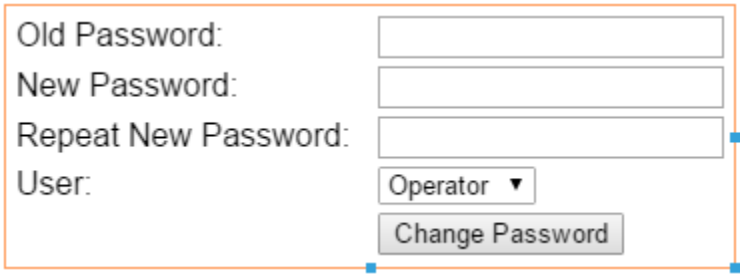
Widgets contained in this section are tightly related to the user authentication, logging in and out, password management and widgets reacting to the logged user.

3.7.4.1 Login



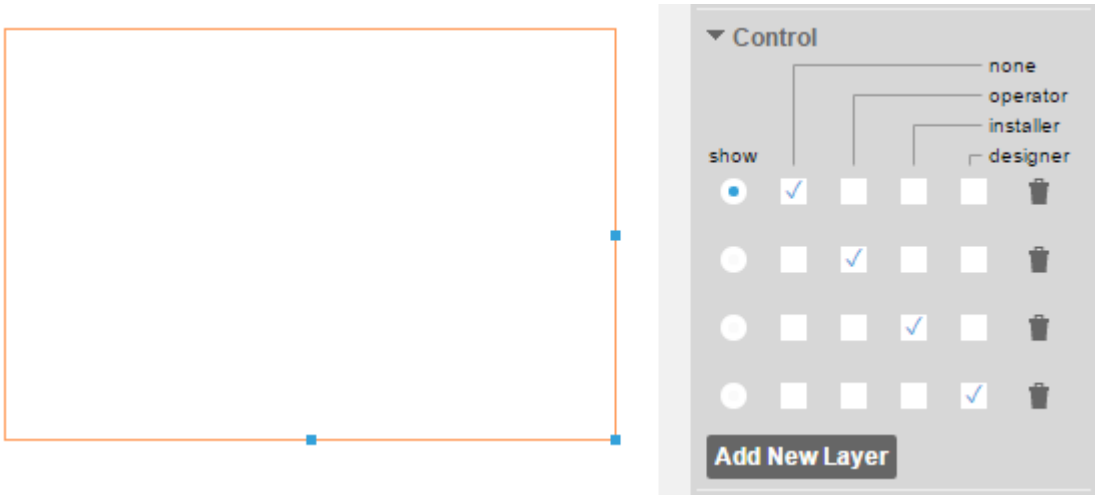
The widget allowing the user to log in. Each of three accounts are protected by the password and also the presence of the user is verified by the RIN concept. In the simulation mode the RIN is ignored and the passwords of the users Operator, Installer and Designer are o, i and d, respectively.

3.7.4.2 Change Password



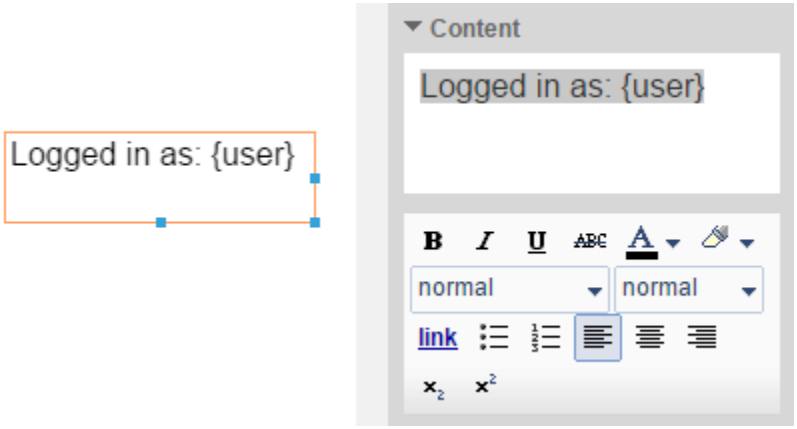
The widget for changing the password. The logged user can change password to himself/herself or to users with lower privileges. In the simulation mode this command is only printed out with no other effect.

3.7.4.3 User Specific Pane



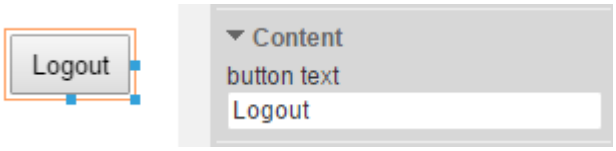
A container widget which reacts to the user who is logged in. It resembles the Tab widget, but the visibility of tabs is not changed by clicking, but according to the fact which user is logged in. The pane has 1-n layers that all behave like containers (so other widgets can be placed into them). The user can specify conditions under which this layer (or more precisely: the widgets in this layer) is visible. The conditions consist of 4 true/false values corresponding to the logged user to be None, Installer, Designer or Operator. The widget will learn about logging in or out event on the same page or on a different browser tab, but it won't learn about expiration of the session unless at least one dynamic widget is present in at least one of browser tabs.

3.7.4.4 Logged As



A simple widget serving as a message about who is logged in. The text of the widget can be modified. Same mechanism of learning changes of the logged user as in the User Specific Pane applies.

3.7.4.5 Logout



Widget for logging out.

3.8 ICON

All kinds of various static HVAC related icon widgets. Various image operations like vertical/horizontal flip, resize and reset size can be performed with them.

4 Advanced

This section covers all areas which are more complicated and thus aimed only for users who have already become familiar with the editor.

4.1 Inheritance

In order to facilitate creating reusable pages or parts of pages, the concept of inheritance has been implemented. It's probably the most complicated part of the editor.

Let's show it in action on an example. The designer creates a page that contains various data about module m2, which is a burner control. He also knows that there is one more burner control m5 in the system and he needs to display same data about it as well. Without inheritance he would have to copy the page and hardcode the module number into each one, which makes it inconvenient when maintaining the pages. Using inheritance he can adjust this page, so that it can be mapped to both burner controls. He can then create a link (on this page or on some other page) which directs the user to this burner control info page linked to m2 and another link pointing to page linked to m5. Alternatively, it can be a dynamic page that contains module control, where the user can switch between mapping to m2 and m5 on the fly. The same applies not only for pages, but also contents of containers. And not only for modules, but also for registers (in a slightly different way).

Let's take a closer look on how the module inheritance works. Each dynamic widget is required to have its own module mapping defined. Then it might have module inheritance on or off. If it's off, it will be always linked to this defined module. If it's on, it might inherit the module value from its parent elements. If the parent element (e.g. Pane) contains the module value, then this value overrides the value defined in the widget. Moreover, the parent element might contain a special widget "Module" and its value overrides the value defined in that parent. However, if the parent element has the module inheritance set to on, then it might also have its value overridden by the parent elements in exactly the same manner, recursively. Finally, the page itself behaves like a container with the inheritance off and the module value equal to "module" parameter in URL (undefined if missing). The only exception in the order of priorities is that the parameter in URL has higher priority than Module widget (and if present, it additionally hides the Module widget), while in containers it's vice-versa.

Register inheritance flow works exactly the same way. The mechanism is extended by the concept of bases and offsets. Module inheritance only overrides modules, but register inheritance working in the same way would not be very useful (container overriding all its widgets to some register R). Instead, its goal is to facilitate work with register clusters, which can be understood as sequences of registers that are repeated in the module. Let's consider that some module has register cluster 100-109, which is repeated also as 110-119 and 120-129 (i.e. same register types, same register semantics). The widget with inheritance on defines not only its default register (which still needs to be present), but also offset within the register cluster (in this example this should be 0-9). Parent containers or Register widgets then might define the register base (which is expected to be 100, 110 or 120 in this case). The result is the sum of the inherited base and the widget offset.

Note, that the module or register in the URL can be specified in a static (page?module=m2) or dynamic (page?module=dynamic) way. In the latter case, the module is specified on the fly according to the parents of that particular widget using the same mechanism as inheritance works for dynamic widgets. When no parent element that could define the value (module in this case) is found, the module property is omitted in the referenced page.

In general, containers can be nested infinitely. All can be set to various variations of module and inheritance with present or absent Module widgets, which yields huge amount of combinations. The result is always deterministic, but not always obvious. However, in practice the nesting should be rare, so let's go through some simple examples, which should cover 99% of inheritance cases.

Example 1: Simple Module Inheritance

Page contains many widgets, all linked to m2 and inheritance on. It also contains Module widget. No containers. When this page is loaded, all widgets are linked to their default, i.e. to m2. Module widget shows "Select module". If module m2 is selected, nothing changes. If a different module is selected, all widgets become linked to this new module.

Example 2: Simple Register Inheritance

Let's consider the register clusters described above. Page then contains widgets, linked to registers 100-109, offset 0-9 and inheritance on. It also contains Register widget with defined bases 100, 110 and 120. No containers. When this page is loaded, all widgets are linked to their default, i.e. to the range 100-109. Module widget shows "Select register". If base 100 is selected, nothing changes. If the base 110 is selected, all widgets are remapped to 110-119 range.

Example 3: Container Module Inheritance

Page contains a pane widget with module set to m2 and inheritance off, and some widgets in the pane, all linked to m2 with inheritance on. It also might contain Module widget outside the pane, but it will not make the difference. When this page is loaded, all widgets in the pane are mapped to the module of their parent widget, which is equal to their default module. Changing module in the module widget has no effect on them. What is this scenario good for? Once the designer created this pane, he might copy & paste it to the same or to the different page. Then by simply changing the module of the pane, it will change mapping of all the widgets to this new value. It would work the same way if the pane contained module widget, just the pane mapping would not be static (m2 or m5), but dynamic according to what's set in the module widget.

Example 4: Module Inheritance with Default Value

As already shown in example 1, all widgets are required to have a default value, which is used when inheritance is off, or when no value is found among parent elements. Let's consider a more complicate case here though. Let's have a page from the example 1. All widgets linked to m2 unless changed by the module widget. The designer either wants to modify this page or create a new one, where everything is the same, but the default value of all register is set to m5. He can achieve this by duplicating the page

and then changing default value of each widget to m5, which can be painful. Instead, he can combine the approach from example 1 with the approach from example 3. The result is the page from the example 3, but now the pane has inheritance on. What's the result? When the page is loaded, all widgets are linked to the module specified in the pane (set to m2), but this can be overridden by the module widget. Changing the default module for all widgets in the pane can be achieved by simply changing the module of the pane.

Example 5: Module Inheritance with the Link

Let's enhance page from the example 3. Designer creates a pane with widgets inside. These widgets have inheritance set to on, so their module is overridden by the pane module. Apart from dynamic widgets, the designer wants to put there also a link (button, image or link in the text) that would take the user to a page with more details about that particular module. He can achieve this by using dynamic link, i.e. 'detailedPage?module=dynamic'. Then he can copy whole pane to same or different pages and change the pane module as he desires. Not only the widgets, but also the link will then take the module value from the parent pane, so when user clicks the link, the browser is navigated to the detailed page with module defined in that particular pane, where the link was clicked. Technically, the link is rewritten e.g. to 'detailedPage?module=m2'. It would work exactly the same way, if pane contained module widget. The link would redirect the user to the detailed page with the module chosen in the module widget.

4.2 Adding Custom Content

In addition to using content provided by Honeywell, it might be convenient or even necessary to be able to add own content, i.e. html (web pages), images, general data, css (styles) or js (JavaScript code). Let's start with the easiest.

4.2.1 HTML – Web Pages

When the designer wants to add his own web pages, he can do so without involving editor. He can use the editor only to reference these pages and he can also add references to the custom pages navigating the web user back to the pages created by editor. For page reference options see section 2.3.5.1.

4.2.2 Images

When the designer wants to add his own images, he has two options. If the images are expected to be used across all his projects (e.g. logo, banner, some generic images), he can place these into the /global/img folder. If the images are expected to be used only within project ABC (specific tailored images for this particular projects), he can place these into the /projects/ABC/web/img folder. This prevents the images to be copied to all other projects. If the designer works only with one project, he can choose either of these destinations. To reference images placed in the project or global folder, use direct name (localImage.png) or name prefixed by slash (/globalImage.png), respectively. For image reference details see section 2.3.5.2.

4.2.3 General Data

Same as in 4.2.2 applies for other data (audio, video, documents, ...), just the folder is /data.

4.2.4 CSS – Cascade Style Sheets

When the designer wants to add his own CSS files, here comes one extra distinction. In addition to storing these locally or globally, the designer might also choose *when* the code will be used. When it should be run only in the editor mode (e.g. debug styles of the widgets), the files need to be prefixed by the *'editor'* prefix (e.g. editorStyle.css or editor.css). When it should be run only in the live mode (e.g. styles distracting the designer), the files need to be prefixed by the *'live'* prefix. Eventually, when the code should be executed in both modes (probably the most common case), it needs to be prefixed by the *'global'* prefix.

4.2.5 JS – JavaScript Code

The same rule with prefixes as above in 4.2.4 applies for JS files.

4.3 Creating Custom Styles (CSS API)

Custom styling is straightforward. Two possibilities of styling are supported: selective and global styling. Note, the ways of adding custom files were described in the previous section.

4.3.1 Selective Styling

The designer should pick the selective styling for cases when he desires to define styles non-globally, i.e. for 1 or some (but not all) widgets. The selectivity is achieved by defining the class property (which most of the widgets have) and assigning styles for the given class. Example:

- Any widget (e.g. Button, Image or Gauge) with the class `opaque` will be rendered as opaque.
  ```
  .opaque { opacity: 0.5 }
  ```
- A suitable widget (e.g. Pane) with the class `gradient` will be displayed with blue gradient background.
  ```
  .gradient { background: linear-gradient(135deg, rgba(30,87,153,1) 0%, rgba(125,185,232,0) 100%); }
  ```

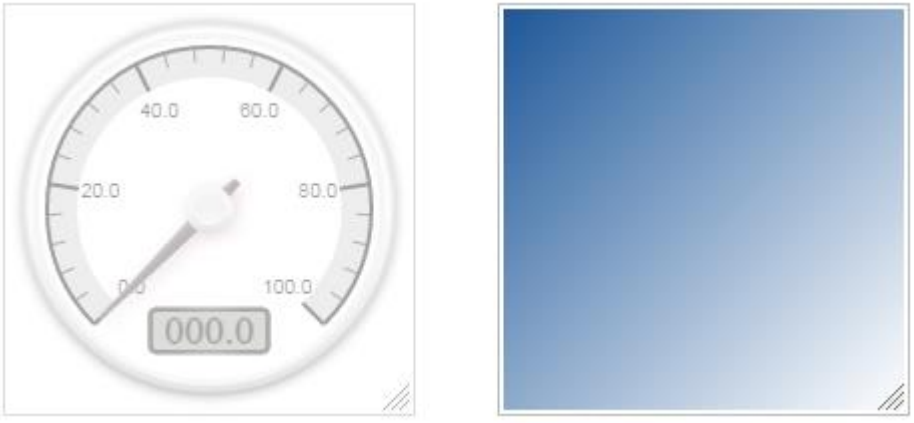
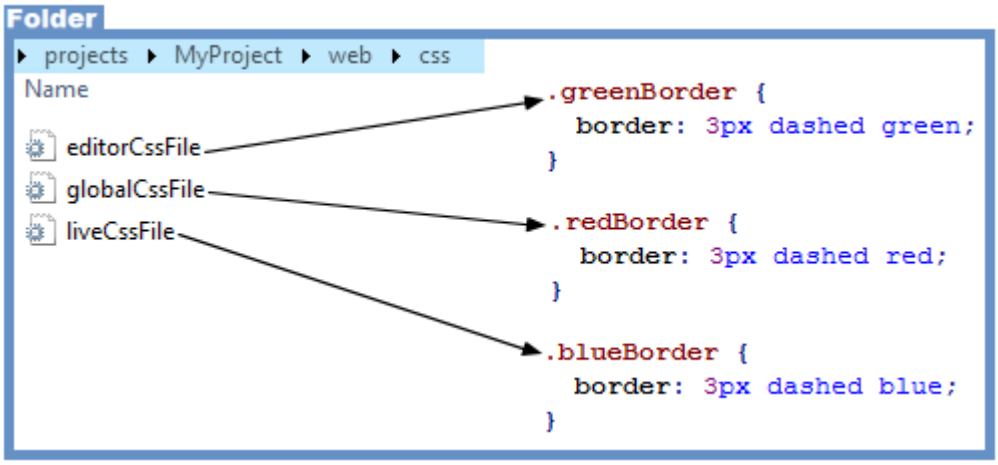


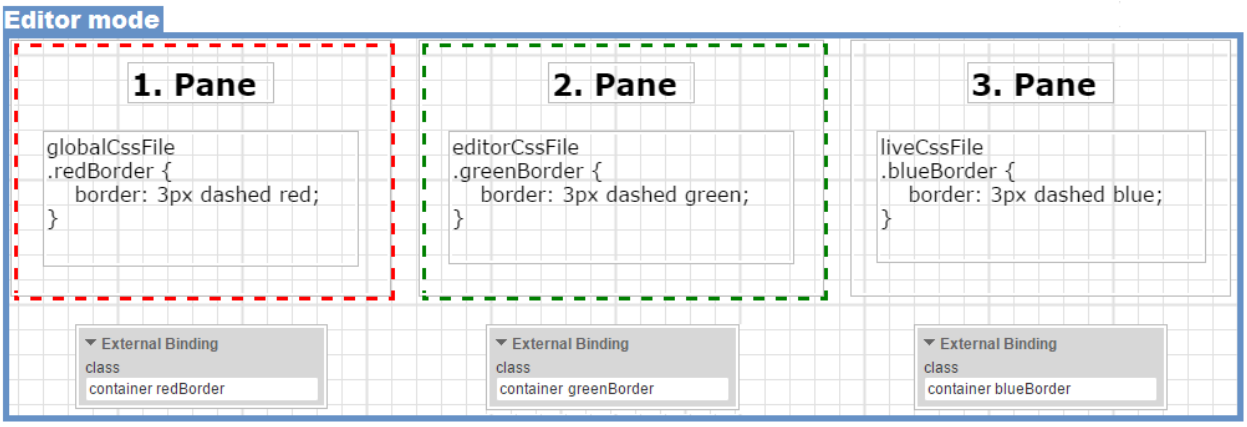
Fig. Examples of selective styling. Opaque gauge on the left, Pane with the blue gradient background on the right.

Example: How to create own CSS style and assign it to some widget?

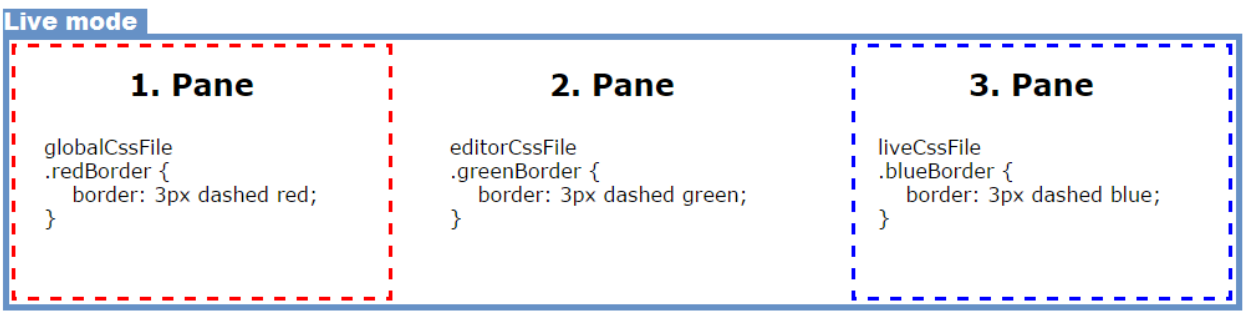
Let's say we have three CSS files in projects/MyProject/web/css folder, which have different prefix (global, editor, live). The rest of the name *CSSFile* is same for all files and it is optional. Each of these three CSS files have a different class defined:



To illustrate different style possibilities, we use the same three Pane widgets and for each of them we assign a different CSS class from different CSS file above. In editor mode, it will look like:



As we can see, the last Pane widget has not a border visible. This is because the *blueBorder* class is defined in CSS file with *live* prefix and therefore it is visible only in the live mode:



On the other side, in the live mode will not be visible a border, which is defined for the second Pane widget – this is because of the *editor* file prefix. To apply a CSS style in both modes we can use the *global* CSS file prefix, as visible for the first Pane widget.

All three Pane widgets use a class property with two classes and one of them is the *container* class. This is a default class used for distinguish that the widget is container-like. We just added another border class. Texts placed in the Pane widgets are here due to better description.

CSS files can be placed in the global/css directory alike, however it should be taken into account that the global styling affects all projects.

4.3.2 Global Styling

The designer should pick the global styling for cases when he desires to change styling of all pages, all widgets or all widgets of a particular type. This way, he can define new styles or remove/redefine default styles. It is a quick way of changing the look of the generated content, but it should be exercised with caution. It is recommended to define the styling at the very beginning. If that is not possible, it is advised to test all the existing pages for possible undesired effects.

Before starting with global styling, it is required that the designer is aware of the widgets' structure, which is uniform in all cases and consists of two wrapper DIV elements.

The outer DIV has class set to `widget`, the reserved attribute `data-type` set to the value which is different for each widget type and inline style defining position and size. For styling of widgets of a particular type, the actual `data-type` value has to be known. It can be found in the generated HTML using the Chrome's Developer Tools (can be invoked by pressing the F12 key). To avoid interfering with the editor design, it is strongly recommended to avoid changing styles to anything outside the class `widget`.

The inner DIV is a container for widget properties other than size and position (i.e. amount of ticks of a Gauge is stored in this element).

The widget specific content is contained in the inner DIV.

Examples:

- Setting color to all widget texts to white

```
.widget * { color: white; }
```

- Changing the font of all Numeric Output widgets

```
.widget[data-type=NumOutput] input { font-family: Verdana; }
```

- Adding shadow to all Image widgets

```
.widget[data-type=Image] img { box-shadow: 10px 10px 5px 0 rgba(0,0,0,0.25);  }
```

Fig. Examples of global styling. White text on the left, changed font in the middle and the image with the shadow on the right.

More detailed overview of the most typical possibilities follows:

Selector:	Styles would be applied to:
`.widget`	all widgets whatsoever
`.widget[data-type="Text"]`	all Text widgets (it might be necessary to inspect the elements in order to find out their type, e.g. Numeric Input widget doesn't have type "Numeric Input", but "NumInput")
`.widget[data-type="Text"] *`	all elements contained in all Text widgets
`.bigLabel`	all widgets, where we put "bigLabel" to the class property
`button`	all buttons (probably not what we want, as this would affect even editor buttons)
`.widget button`	all buttons within all widgets
`.widget input, .widget select`	all inputs and selects within all widgets
`.widget[data-type="FaTableUnit"] input`	both inputs (radio buttons) in the FaTableUnit widget (FA > TABLE > Unit Selection)
`.red .widget[data-type="Button"] button`	all buttons of the Button widget that are in something (e.g. Pane) with a class "red"
`.widget[data-type="ModalDialog"] .container`	the modal dialog container (e.g. for changing the background color)
`#editor, body`	the editor area in the editor mode or to body in the live mode*

* The styles are applied equally in the editor mode as in the live mode, there is just one exception. When it's required to style the `<body>` (i.e. content outside the widgets), the selectors are different in each mode. To solve this, use the selector above, where `#editor` is used in the editor mode and `body` is used in the live mode.

For general information about styles, check any online documentation, such as http://www.w3schools.com/css/ or https://developer.mozilla.org/en-US/docs/Web/CSS.

4.4 Creating Custom Widgets (JavaScript API)

The designer can create its own widgets, which he can use in addition to the ones provided by Honeywell. In order to do so, he needs to be fairly experienced in JavaScript. Knowledge of the Google Closure frameworks helps a lot, but is not necessary. Complete manual with all public methods is beyond this documentation, so for some more complicated widgets the designer will also have to dig in the source code.

The new widget must:

- be an object
- inherit from `kettos.widgets.Base` object
- contain a constructor that calls parent constructor with name and type parameters
- implement `createDragElement` method which is called at the start of dragging and should create the DOM element to be dragged; parent method is advised to be called with size parameters
- implement `getMenuItems` method which is called when widget is selected and the properties menu should be populated; there is whole kettos.menu.item package with prepared elements (input, radio, checkbox, select, slider, button, M/R mapping, RTF, color, class, position and size), so the designer should not need to create his own
- register this widget in the dictionary `kettos.dict.register` passing the constructed widget

The widget also may:

- implement `initAfterDragging` method, if it can't be constructed whole at the start of dragging and needs some processing after the dragging was finished (e.g. in case of expensive widgets: gauge, graph, or in case of widgets that would mess with the dragging: button, checkbox)
- implement `resizeEndCallback` method, if there needs to be some postprocessing after the widget was resized (mainly in case of canvas widgets, where the resizing cannot be continuous e.g. gauge, graph)

Now the widget is ready to be used in the editor. If it has some live behavior, the designer must further implement the `kettos.live.<NewWidgetType>` method which defines the live behavior.

Let's make an example. The designer wants to create a new widget CustomBox that represents some static text styled by CSS.

First, he creates an editorBox.js file where he defines the editor behavior of the widget:

```
1.  kettos.widgets.CustomBox = function() {
2.    // Second parameter is a widget's display name
3.    // Third parameter is a widget type ID and must be equal to widget's object name
4.    goog.base(this, 'Custom Box', 'CustomBox');
5.    this.categories = ['SPECIAL'];
6.  }
7.  goog.inherits(kettos.widgets.CustomBox, kettos.widgets.Base);
8.  kettos.widgets.CustomBox.prototype.createDragElement = function() {
9.    // Parameters: 1 - width, 2 - height, 3 - widget's content
10.   return this.dragElementHelper(300, 50, 'This is a custom widget');
11. }
12.
13. kettos.widgets.CustomBox.prototype.getMenuItems = function(dragger, div) {
14.   var content = new kettos.menu.Content(dragger, div);
15.   // Usual widget menu items
16.   content.addBinding();
17.   content.addClass();
18.   content.addPosition();
19.   content.addSize();
20.
```

```
21.     // Custom menu item example
22.     var customBlock = new kettos.menu.Block('Custom Value');
23.     customBlock.addWideRow(
24.       kettos.menu.item.input(
25.         dragger,
26.         function() {return div.getAttribute('data-custom-value')},
27.         function(value) { div.setAttribute('data-custom-value', value) },
28.         'value description'
29.       )
30.     );
31.     content.add(customBlock);
32.
33.     return content;
34.   }
35.
36.   kettos.dict.register(new kettos.widgets.CustomBox());
```

Constructor is 1-6. Line 4 is call of the parent constructor with name (to be shown in the editor) and type (to be used in HTML, no spaces recommended). Line 5 is adding this widget into SPECIAL category (categories in the left side of the editor). Line 7 makes this object child of the Base object.

Lines 8-11 define what should be created when we click this widget and start dragging. Line 10 is call of the helper method with width 300, height 50 and some content. It creates the wrapper around our widget

Lines 13-34 define the menu attributes. You can use some of the common menu properties or create custom menu attributes. Line 22 is definition of a Custom Value property which will be displayed in live mode. Line 24 is creation of menu's INPUT prepared element with value getter function (line 26), value setter (line 27) function and value description.

Line 36 does the widget registration.

Next, he creates a liveBox.js file, where he places dynamic behavior of this widget. Let's make it simple.

```
1.    kettos.live.CustomBox = function(div) {
2.     var customValue = div.getAttribute('data-custom-value');
3.     div.innerHTML = 'This widget has just become live. Custom value is: ' + customValue;
4.    }
```

The function has parameter div, which references the wrapper of the widget. We can change the text of the div to something else to demonstrate live behavior.