Widget ID

Each user type widget should have a unique identifier within a single controller (ID). Any string can be as ID.

Widget ID is used when installing the widget, appears in its program code and cannot be easily changed after installing the widget in app. Widget ID only appears in UI during installation. To minimize the risk of collisions when using your widgets in various installations, we recommend to use set of characters as ID containing unique part of widget developer (prefix).

Furthermore, some ID are already reserved in system widgets, the app will inform you when attempting to install the widget.

Widget file structure

Each widget contains the following set of files

- script.js , it is required file in the root folder of widget. It contains JavaScript code that describes the widget interface and logic of its work. The script is loaded on the page once regardless of number of widgets of this type that has been created by the user in a particular app.
- style.css, it is optional file in the root folder of widget. It contains CSS styles that are unique for your widget. If this file appears in the widget folder it will be automatically loaded when refreshing the page.
- Any other graphical, JavaScript, CSS files that will be used in your widget. You need to load them in one of mentioned above files.

We recommend to use the following folder structure for widget:

- script.js
- style.css
- images
    - graphical files
- js
    - additional JaScript file and libraries.

Available libraries and frameworks

In your widget you can use the following set of libraries and frameworks which are included in the project:

- JQuery (http://jquery.com)
- plugin JQuery noUiSlider (http://plugins.jquery.com/nouislider/)
- plugin JQuery knob (http://plugins.jquery.com/knob/)
- Boostrap v3 (CSS+Javascript) (http://getbootstrap.com)
- Font Awesome v4 (http://fontawesome.io)
- UnderScore.js (http://underscorejs.org/)
- Backbone.js (http://backbonejs.org/)

File script.js

Before proceeding with further part of documentation we recommend to read the documentation of Backbone.js library http://backbonejs.org/ .

In your JavaSctipt code in collection of widget classes you need to add your own extensions for base classes – Widget (model) and WidgetView (view), by realizing necessary set of fields and methods as listed in the example below:

```
Widget["your widget ID"] = Widget.extend({
    /* description of widget settings*/
    config : {
        ...
    },

    /*is called when objects values are changed on Logic Machine*/
    changeValue : function(name, value) {
        ...
    }
}

WidgetView["your widget ID"] = WidgetView.extend({
    /*width of widget in cells*/
    width : ...,
    /*height of widget in cells*/
    height : ....,

    /* is called when the object is initialized */
    init : function() {
        ....
    }

    /*main template of widget*/
    template : ...,

    /*template for viewing in editor mode*/
    preview : ....,

    /*is called after rendering of widget during initialization*/
    afterRender : function() { ...}

    ...
})
```

Instead of "your widget ID" you need to use your widget ID and specify it also when installing the widget from interface. Besides described service attributes you can use any of your helper methods and properties during implementation of these classes, which are responsible for data conversion, validation, calculations, working with DOM of your widget etc.

Base classes Widget and WidgetView are extensions of base class model (Backbone.Model) and view (Backbone.View), Backbone respectively includes set of helper methods to work with LogicMachine objects which can be used in your implementation.

For each type of widget created by Mosaic user there are following classes created.

```
model = new Widget["your widget ID"]();
view = new WidgetView["your widget ID"]({model:model});
```

Thus, model method calls within model itself should be carried out using *this*:

```
this.func()
```

Call of properties and model methods within view should be done in the following way:

```
this.model.func()
```

Widget configuration

All widget settings required to bind it with LogicMachine objects as well as other settings and data required for the formation of the forms and editing in Mosaic constructor have to be described in the property *config* of your model.

Example:

```
Widget["your widget ID"] = Widget.extend({
    config : {
        title : "Name of widget",
        fields : {
            title : {
                datatype : "string",
                title : "Title"
            }
        },
        objects : {
            key : {
                datatype : dt.bit,
                title : "Object"
            }
            ...
        },
        settings : {
            "min-value" : {
                datatype : "number",
                title : "Minimum value"
            }
            ...
        },
        systems:["blinds", "ac"]
    },
    ...
}
```

The field «title» is used to specify the name of the widget. It will be displayed in a preview mode and editing mode.

The field «fields» is used to specify fields of the widget which should be configured by user in the constructor. Object keys will be used to access the values of these properties. The access to fields inside the model itself can be done with the following command *this.field*. We recommend to use this field only to edit the property *title* and all other widget settings to bring in *settings* section.

The field «settings» is used to specify list of widget settings which are available for user editing in the constructor mode. Object keys will be used to access settings thanks to helpers described below. For every property you need to specify *title*, which will be seen in widget editing mode, and *datatype*. Currently two data types are supported - *string* and *number*.

The field «objects» is used to show list of objects from LogicMachine (group addresses) which can be linked to a particular widget. Specific objects should be specified by user with a help of widget creation form in Mosaic constructor. Object keys will be used to access object values and properties thanks to helpers

described below. *Title* also should be specified for every object which will be shown in the widget editing form as well as KNX datatype number. A list of all supported datatypes can be found here http://openrb.com/docs/lua.htm#grp-info. To provide specific numeric constants we recommend to use the object *dt* with a list of fields defined in the link above. If a specific KNX subtype is specified as datatype, the user will be offered to choose objects with this type only. If a specific type is specified as datatype, the user will be offered to choose objects with this data type or its subtypes.

The field «systems» is used to specify a list of engineering systems which will be used in Mosaic navigation. Widgets of a particular type will be shown on a screen of all engineering systems listed in an array. Here is a list of identifier of engineering systems which can be used:

- lighting
- blinds
- power
- windows
- ac
- ventilation
- floor

Auxiliary methods of class Widget (base model)

Here is a list of predefined methods Widget which you can use in your models and views.

Working with LogicMachine objects (group addresses):

`getValue(key)`

Returns current object value.

`existsValue(key)`

Returns *true* if user has specified a specific object in Mosaic configurator, or *false* if the object is now specified.

`getObject(key)`

Returns information on the object – group address, data type and object name. Example of reply:

```
{
    updatetime:1461825143,
    address : "1/1/1",
    units "",
    value : true,
    name : "alarm",
    datatype: 1005
}
```

`write(key, value)`

Write object value to the controller/fieldbus.

Other methods and properties

`conf(key)`

Returns value of setting (described in the section *settings* of widget configuration), specified by user

`title()`

Returns value of the field «title» which is specified by user during setting up the widget in Mosaic constructor.

`ctitle()`

Returns widget name which is specified in «title» section of widget configuration.

`isPreview`

The field has value *true* if the widget is shown in preview mode. Note! The widget is shown in a preview mode prior to user settings i.e. it has no group addresses or other settings set up.

Call of methods of class Widget.

To access the above helper functions use the correct context. Example of calling a function title():

- in model methods *this.title()*
- in view methods *this.model.title()*
- in templates *title()*

Helper methods of class WidgetView (basic views)

List of predefined methods of WidgetView which can be used in your views.

`size(width, height)`

Allows to resize the widget after initialization (on the fly). Widget size should be specified in the number of occupied cells. The size of one cell is 110px*110px, but taking into account fields the real size of widget can be determined by the following formula (110*width-10) px * (110*height-10) px

`el`

DOM-model of an HTML container in which your widget is enclosed. More detailed usage is described in the documentation of library Backbone.js http://backbonejs.org/#View-el . We recommend to use it wherever you are using the selectors to access the DOM-model of your widget, in order to avoid collisions with other widgets. Example of search on your widget title page:

```
$("div.widget-title", this.el)
```

WidgetView class methods can only be called from your view. *this* is used to access them. (For example, *this.size(1,2)*)

Trigger methods

Parts of methods described in structure of JS file are events. They are called by program kernel at specific cases.

`changeValue(key, value) for the model`

If you have implemented this method in your model, it will be called each time when there was an object value changed in LogicMachine (group addresses which are related to your widget). The object key (as provided in the configuration) and its new value will be given as parameters for the method.

`init()  for the view`

If you have implemented this method in your view, it will be called once during initialization of view (prior its rendering). Here you can initialize start parameters of the view which can be used in the template during its rendering.

`afterRender() for the view`

If you have implemented this method in your view, it will be called once – right after rendering the widget. It can be used to initialize individual controls which cannot be made in the template (e.g. dimmer knob)

Widget templates

In the view class of widget you need to implement two methods:

`template(model)`

It will be called by the system during widget initialization (for rendering the widget in common list of widgets). As an input parameter the initialized model will be sent.

`preview(model)`

It will be called by the system during widget initialization for its preview rendering in Mosaic constructor. Note that in this case the model will not contain objects and widget settings which are set by user during widget creation process.

For templating we strongly recommend to use method _.template from Underscore.js library http://underscorejs.org/#template . But you can use also any other templating or create your widget DOM-objects in JavaScript. In this case do not forget that the whole widget is placed in a container which you can access in your view by *this.el*.

All further example in this documentation use Underscore.js library which is already included in the project.

Recommended widget structure

We recommend to use the following HTML structure to apply all base classes of system widgets

```
WidgetView["your widget ID"]  = WidgetView.extend({
    ...
    template: _.template(" \
        <div class=\"widget\"> \
            <div class=\"widget-title\"> \
                <div class=\"txt\"><%=title() || ctitle()%></div> \
            </div> \
            widget controls
        </div> \
    "),
    ...
})
```

Also you can add your own classes and define styles for them in your style.css or use another widget structure. Remember that all created CSS classes for your widget can affect the appearance of other elements of the application. We recommend to use unique names for new CSS classes to avoid such collisions.

Usage of methods and properties of model in templates

If you use _.template Underscore.js for templating of widget then since as a widget model is passed as an input parameter in your template, you can call any model methods and its properties directly in template without specifying the context.

Example:

```
WidgetView["your widget ID"]  = WidgetView.extend({
    ...
    template: _.template(" \
        <div class=\"widget\"> \
            <div class=\"widget-title\"> \
                <div class=\"txt\"><%=title() || ctitle()%></div> \
            </div> \
             <% if (existsValue('object')) { %> \
             Object value <%= getValue('object')%> \
             <% } %> \
        </div> \
    "),
    ...
})
```

Preview template

You can implement a separate method to display the widget in preview mode in Mosaic constructor:

```
 WidgetView["your widget ID"]  = WidgetView.extend({
    ...
    template: _.template(" \
        <div class=\"widget\"> \
            <% if (existsValue(\"object\") { %> \
                <%=getValue(\"object\")%> \
            <%  } %> \
        </div> \
    "),
    preview: _.template(" \
        <div class=\"widget\"> \
            Object value \
        </div> \
    ")
    ...
})
```

If the *preview* method will not be implemented then during the initialization phase of widget the method *template* will be called to do preview. Thus one single template can be implemented for widget, but keep in mind that in preview mode it doesn't contain objects and settings, thus all necessary checks should be made in the template.

The previous example but by using a single template for both widget display modes:

```
WidgetView["your widget ID"]  = WidgetView.extend({
    ...
    template: _.template(" \
        <div class=\"widget\"> \
            <% if (isPreview) { %> \
                Object value \
            <% } else if (existsValue(\"object\")) { %> \
                <%=getValue(\"object\")%> \
            <%  } %> \
        </div> \
    "),
    ...
})
```

Data exchange among the model and view

Any property of the model is available in the view as the object instance is its attribute, but not vice versa. Besides often a situation arises when the attribute change of the model should affect the view of the widget. In this case we recommend to use Backbone events http://backbonejs.org/#Events .

The following examples shows how to display the changed object value on the page:

```
Widget["your widget ID"] = Widget.extend({
    ...
    changeValue : function(name, value) {
        //initiate event
        if (name=="status") return this.trigger("changedStatus");
        ...
    }
    ...
}

WidgetView["your widget ID"]  = WidgetView.extend({
    ...
    init : function() {
        //subscribe to a specific event model
        this.listenTo(this.model, "changedStatus", this.changedStatus);
    },
    template: _.template(" \
        <div class=\"widget\"> \
            <%=getValue(\"status\")%> \
        </div> \
    "),
    changedStatus : function() {
        //change the value in HTML code
        $(".widget", this.el).text(this.model.getValue("status"))
    },
    ...
})
```