Client-Side Business Logic Widget Manual

From Snap4City:

- We suggest you read https://www.snap4city.org/download/video/Snap4Tech-Development-Life-Cycle.pdf
- **We suggest you read the TECHNICAL OVERVIEW:**
  - https://www.snap4city.org/download/video/Snap4City-PlatformOverview.pdf
- **To see slides go to** https://www.snap4city.org/944
- https://www.snap4city.org
- https://www.snap4solutions.org
- https://www.snap4industry.org
- https://twitter.com/snap4city
- https://www.facebook.com/snap4city
- https://www.youtube.com/channel/UC3tAO09EbNba8f2-u4vandg

Coordinator: Paolo Nesi, Paolo.nesi@unifi.it
DISIT Lab, https://www.disit.org
DINFO dept of University of Florence,
Via S. Marta 3, 50139, Firenze, Italy
Phone: +39-335-5668674

Access Level: public
Date: 07-10-2024
Version: 4.8

Sommario

1. Introduction

The following manual has the purpose of describing a feature of the Dashboard Builder which allows some widgets to send and/or receive actions, events and data through JavaScript code with parameters from/to other widgets, in order to make them interact/smart with each other. For example, you may be able to click on a section of a pie chart to extract a piece of data and view it in more detail on a table on another widget; make spatial drill-down/up on maps, temporal drill down/up on time series data etc.

Please refer to the Server-Side Business Logic section on the following document **https://www.snap4city.org/download/video/Snap4Tech-Development-Life-Cycle.pdf**
Server-Side Business Logic implies to send commands to the Proc.Logic / IoT App implemented as Node-RED Node.JS processes running on server-side cloud via a visual editor.

Please remind that only AreaManager can access at features of the Client-Side Business Logic on Snap4city.org platform, and they have to perform a request to the platform administrator by sending an email to snap4city@disit.org : If you need to become an AreaManager, please also ask to the administrator by sending an email to snap4city@disit.org .
This feature is also available on MicroX installations, in those cases the access to the CSBL functionality is provided by registering the user on the database table providing the access.

1.1 Snap4City Development Environment

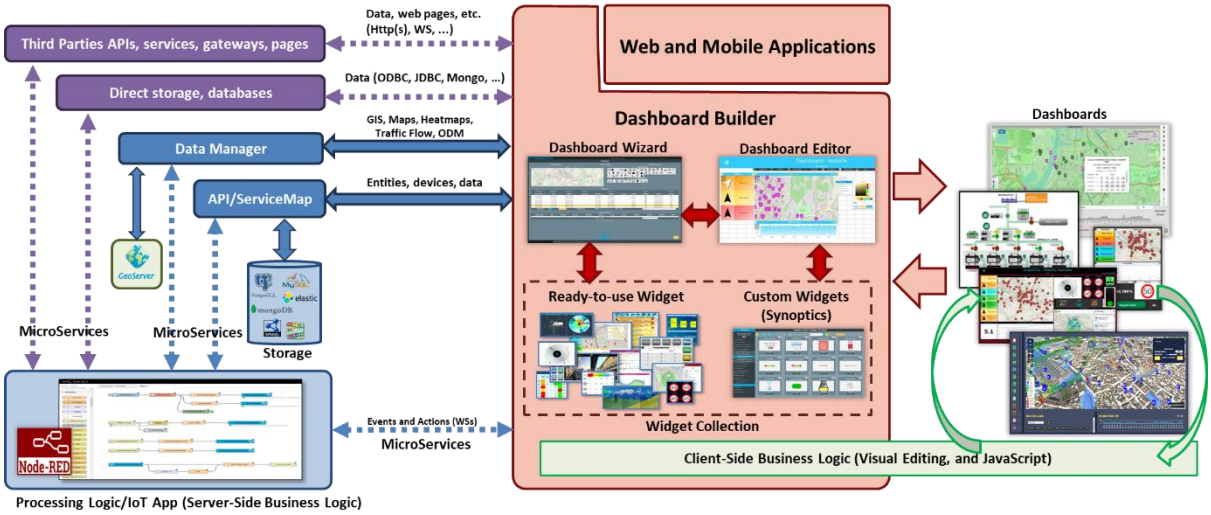
This approach is an exclusive prerogative of Snap4City development environment.

In Snap4City, Client-Side Business Logic is a solution to close the loop from user actions and effects on widgets directly on the client side, on the browser. This approach allows a separated context for each user since any Business Logic computation is performed on client side at every connection, thus reducing the computational workload on server side. Actually, Client-Side Business Logic, CSBL, and Server-Side Business Logic, SSBL, may be present at the same time behind a Dashboard and thus behind a Business Intelligence / Smart Application. This approach may be used to save some context and share it with other users. In CSBL the logic code is formalized in JavaScript only, while in SSBL the logic is formalized in Processing Logic which is Node-RED plus some JavaScript, also called IoT Apps. Both SSBL and CSBL can access/query platform data and external data from any kind of APIs, including third party APIs.

The design and development of the smart applications with Graphic User Interface, GUI, means to develop views which put in connection the back-office data with some dynamic visual

representation and provides to the users' interactive elements to change the observed views or go for another dashboard as well.

In Snap4City, the views are implemented as smart Dashboards which are composed by graphical widgets connecting storage on the basis of the Entity Instances, Processing Logic (IoT App) data/nodes in stream, Synoptics, External Services, and Brokers. Widgets can be maps, time trends, chords, spidernet, bar series, tables, buttons, animated elements, sliders, etc., from libraries such as D3 and Highcharts or custom Synoptics widgets by using an SVG-based integrated tool and templates. Moreover, the Dashboard may dress different styles / themes in their rendering and the designer / developer can select them soon or later in the creation process.



The power of the user interface is grounded on the possibility of easily connecting the graphic widgets with the Entity Instances in a smart interactive manner, keeping humans in the loop. For example: selecting an area by clicking on a Pin/service on map and connecting related data to widgets such as a pie chart and time series, as well as producing some computation, for instance calculating the average values or other metrics. To this end, the User Interface needs to provide some business logic which can be on server-side (formalized in Proc.Logic/IoT App, Node-RED) to serve all the users at the same time, and client-side for evolving user interface behavior on each client device autonomously. Client devices are typically a browser but could also be a mobile app. Please note that user actions can also produce effects on Brokers, can activate AI/XAI processes, etc., these activations cane performed via SSBL or directly acting on API provided by Python / Rstudio processes.

According to SMADELC, ***SMart Applications' DEvelopment Life Cycle***, the design of views starts with the production of Dashboards' mockups (see the development life cycle document). For each Dashboard, a set of information is requested such as: aim, status, Subnature classification target device, GUI style, list of widgets and their description, and the business logic kind.

The Dashboards are called passive when the data represented come only from platform storage without any event driven connection from platform to dashboard and vice versa via SSBL for presenting/producing data in real time. Active views/dashboards are those that

provide those connections which can be SSBL or CSBL. They are initially developed as passive Dashboards in first version and get smarter and smarter in successive sprints of the development life cycle. Moreover, for fast prototyping the SSBL is developed by using IoT App Node-RED, which provides a set of nodes corresponding to graphical widgets on dashboards (via secure WebSocket), thus overcoming the limitation of Node-RED native dashboards. This approach is very effective and viable for fast prototyping and to realize strictly synchronized smart applications in which all the actions on the web interface are shared with all the users (typically limited) as happens in the IoT world. For example, the monitoring panel of an industrial plant should present the same data to all the users connected to it.

On the other hand, when the users expect to play with the data rendered on dashboards for implementing some business intelligence tool, their experience and the evolving data represented in the interface according to their activities is going to change. Those aspects are personal, and context based as one expects to have on a smart application, leading to an evolving user interface that should have a CSBL. In Snap4City, the development of CSBL can be implemented by adding JavaScript functions embedded into the graphic widgets as call back actions, which can perform actions on other widgets and on the platform, and also on third party services, such as a REST Call to API.

To develop this phase one has to follow the Dashboard descriptions performed in the design phase. And at the same time, he/she has to answer questions such as:

- The user interface has to provide some dynamic changes on the basis of the users' actions? Which kind of changes? (CSBL would be need in the affirmative case)
- How many users are using it? (CSBL would be need in the multiple users performing their own private analysis)
- Is it a view for the control room and decision makers of a business intelligence tool for playing with data and solutions? (SSBL would be need if the idea is to create a dashboard in which all the users are going to see the evolution of data even if they are requested by only one of them: sharing the experience or the view)
- Etc.

How to proceed: The developers on Snap4City can visually create dashboards with a drag and drop tool by using Dashboard Builder which is assisted by a number of tools:

- **Wizard** to match data with widgets. It is an expert system for immediate matching HLT (High Level Types: IoT Devices, heatmaps, traffic flows, POI, ODM, etc.) data vs graphics representation for creating Dashboards by rendering and acting on data with a large range of graphics widgets, which may have intelligence in the back by means of:
  - **SSBL**: Processing Logic (IoT App) on data flow combining powerful Microservices/nodes, Data Analytics and API, to implement SSBL.
  - **CSBL**: implemented as visual programming and JavaScript on specific Dashboard Widgets as described in the following of the life cycle document mentioned above.
- **Custom Widget** production tool for creating new widgets and Synoptics by using visual tools and templates: for real time rendering data on graphical scenographic tools, and for graphic interaction on the systems from dashboard to actuators through end-to-end secure connection.

- **Style Theme** modeling for deciding which style to use on the front-end dashboards. It is easy to change the graphical theme and style of the dashboard to have your precise fitting on your applications and portals.
- **External Services** for integrating external tools via SSBL / Proc.Logic and/or via IFRAME into an External Content Widget, or directly calling them as rest API or other means from CSBL.

The list of graphical widgets available in Snap4City to compose the user interface is accessible at the following link: https://www.snap4city.org/download/video/course/p2/
In Snap4City, there is a specific tutorial for the Dashboard development with several examples: https://www.snap4city.org/download/video/course/p2/

Thus, in CSBL we have IN, OUT and IN/OUT widgets (quite similar to SSBL):

- **IN Widgets** are those that are prepared to **receive some actions/commands** (inputs) from the Users or from other widgets as events/messages/commands. For example, a click on a button, a click on the map, etc. They are in some way passive, in the sense that they do not produce any action on other widgets. They do not have Custom CSBL JavaScript inside.
- **OUT Widgets** are those that are prepared to produce **send some messages / triggers to other widgets**. For example, a view of a bar series on some other data, a rendering of a time series, a rendering of a set of Entities on the map, etc. They can be active in the sense that they are capable to produce actions on other widgets, via the Custom CSBL JavaScript they have inside.
- **IN/OUT Widgets** are those that provide capabilities of both IN and OUT Widgets. For example, a map can receive an IN commands to show a PIN, and can send an OUT command to show the data of a selected set of PINs on some barseries.

In the development of the CSBL, two modalities are possible: (i) mixt of visual programming and JavaScript, (ii) fully JavaScript (also called former full CSBL). The JavaScript may be activated by events / actions performed by the User on the GUI of the Widget, and may also send messages/events on other widgets as well perform some API call to Snap4City services or to external services of any kind. While the OUT Widgets are ready to receive commands/messages from CSBL JavaScript (other widgets), similarly to what they do when receiving commands from SSBL via Web Socket to perform actions on the widget or any other JavaScript action/activity.

The Developers that would like to develop CSBL have to be singularly authorized, please ask to snap4city@disit.org, and in MicroX local installation ask to us for the location on your local database for the activation of the functionality to your users. They will be entitled to go in the widget More Options tab and to find and edit a the CK editor to insert the JavaScript code according to this manual, and examples.

When working in **SSBL**, widgets can be created and edited from Node-RED Processing Logic. When working in a CSBL context, widgets have to be created through the Dashboard Wizard as well as from New Dashboard Wizard which provides much more capabilities, as shown in *Figure 1*.

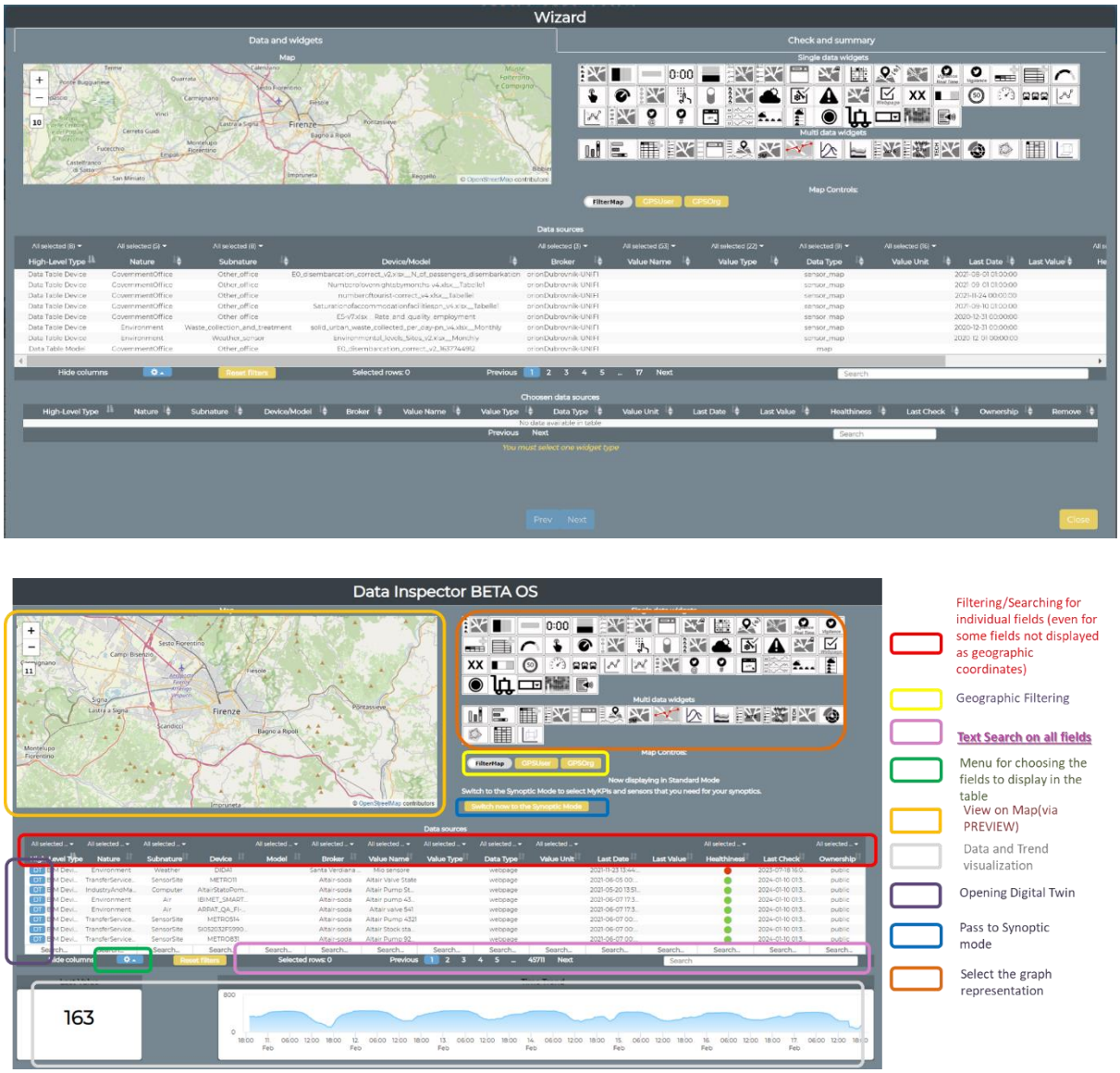


figure 1: Dashboard Wizard of Snap4City, former and newer. The Dashboard wizard is also accessible as: (i) Data Inspector to search and navigate on data of any kind, (ii) tools for selecting variables for Synoptics panels.

The desired widget type can be chosen from the up-right section of the Wizard showing the available widgets. The desired data source can be chosen from the "Data Sources" table, exploiting the filters on column headers, as well as the text search box in the bottom-right corner of the same table. When the desired choices have been performed, by clicking on the "Next" button a final "Check and Summary" tab is shown for final check. At last, clicking on the "Create widget" magic wand the desired widget(s) are created in the current user dashboard. In this way, the created widget(s) are in the form of IN widgets, and they are ready to receive and show data and actions by JavaScript, as described in the table "Commands which are ready to execute from JavaScript" in the next pages.

In order to create OUT widgets (those widgets which have this capability are listed in the following table "*Users' Action Description and effects*"), an authorized developer can add the

desired JavaScript code in the CK Editor box in the widget “More Options” box, as shown in *Figure 2*. The JavaScript code should be provided as a single JavaScript function named “execute”. Please refer to the specific user manual for more detailed instructions.

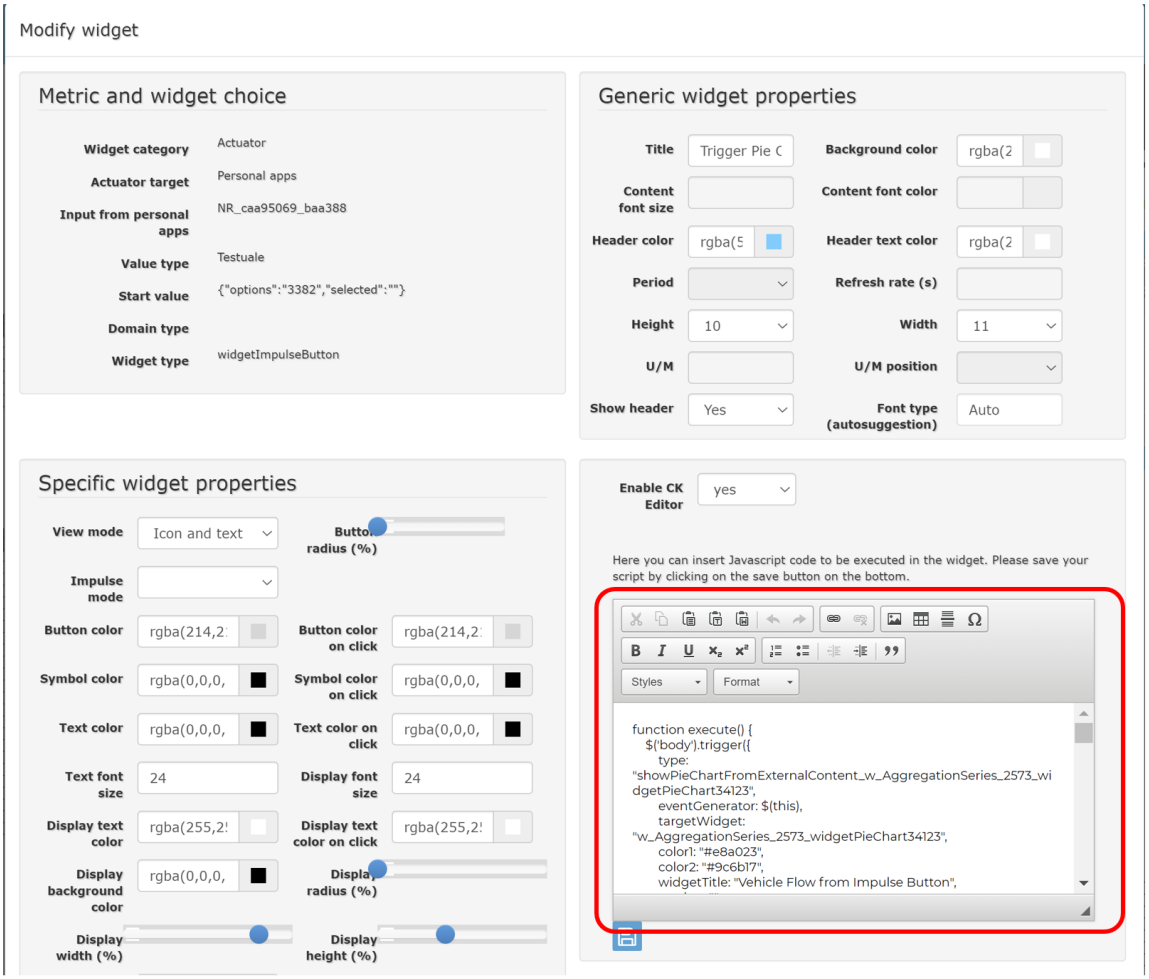


Figure 2: More Options of Widget with activated CK Editor for CSBL.

OUT and IN/OUT Widgets which present the possibility of scripting in JavaScript when an action is performed on their graphic user interface are reported in the following table. The performed action by the user provokes the activation of a call back that can be filled in the JavaScript editor of the Widget to formalize the action to be performed. At the moment in which an action is triggered, a number of parameters can be provided. For example, geographic coordinates can be passed at a click on map, etc. Into the JavaScript, the developer can code how this information can be used to command IN Widgets, and also REST Calls to Smart City API and other activities.

In general, a widget may receive commands/events (IN) (from other widgets (triggers) and from the user) and may send commands/events to the GUI part of the widget automatically and to the CSBL JavaScript part for custom actions/effects. Moreover, the received inputs commands and events my provoke changes on its own representation as well as to other widgets by sending events, triggers commands outside. The following figure represents a schema.

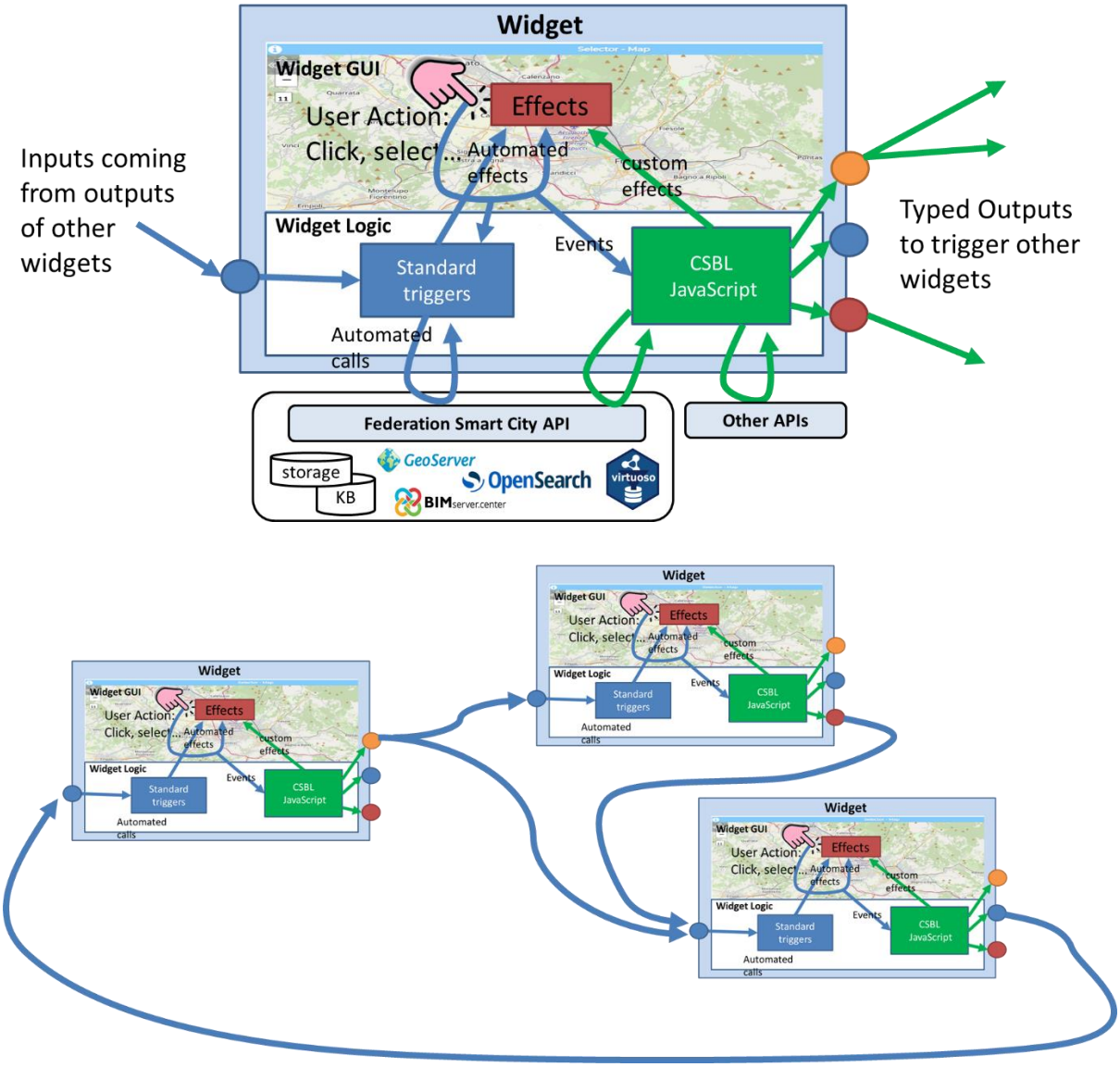


Figure 2bis – The CSBL concepts on Widgets: single and composition.

The inputs received by a widget (coming from others) provoke standard triggers and thus automated effects on the GUI counterpart of the widget. This means that standard triggers cannot be used to execute CSBL JavaScript and custom effects in the same widget in which they are received, but the Effect on GUI are constrained to be compliant with the standard automated effects. In summary: IN widgets provide only blue and red parts of the above figure; pure OUT widgets provide red and green; IN/OUT provide all parts.
This approach avoids the possibility to perform indefinite loops among widgets. On the other hand, it is not a limitation on logic since each widget can control any number of other widgets with its outputs. This mechanism allows the developers to create business intelligence tools in which the action on some GUI Widget (e.g., click, drill down, zoom on map, etc.) can produce a result directly shown on other widgets.

Outputs are usually typed, in the sense that the JSON in out should be compliant with some standard message format, plus other custom messages can be defined as well.

The OUT widgets provide space for Custom JavaScript editable via CKEditor with the following structure, in which is evident that the widget is going to execute function **execute (),** in which the Events may be processed with specific JavaScript segments. The CSBL JavaScript is activated by Events coming from the Users' Actions. The CSBL JavaScript segment may include:

- calls to ASCAPI (Advanced Smart City API of Snap4City), usage of AJAX, authentications, etc.
- call to external API, usage of AJAX, authentications, etc.
- production of Typed Outputs to trigger other widgets (one or more) of the dashboard provided that the widget IDentifier is known.
- commands to open other dashboards/views passing also parameters to them.

In the Snap4City tools there are two modalities of producing Business Intelligence dashboards. In particular they are called:

- **CSBL Visual Editor:** create the CSBL JavaScript in assisted manner via CSBL Visual editor in which the structure of the code and the connections / messages are automatically predefined and generated. **See Section 3.**
- **Full Custom CSBL:** create the CSBL JavaScript according to the user manual reported in **Sections 4, 5**, etc.

In the CKeditor of each Widget via More Options, you can create or you can find a different template as reported in the following table. The left version allows you to be compliant with the Full Custom CSBL in JavaScript, while the right version allows you to be compliant with CSBL Visual Editor as described in the following. There is a certain level of compatibility among the two approaches. The Cull Custom CSBL allows to exploit in deep all the functionalities of the Dashboard Builder for smart applications.

/* template compliant with the **Full Custom CSBL** JavaScript */	/* template (comments included) if you want your CK Editor code to be compliant with the **CSBL Visual Editor.** */

function execute ()
{
 	
Var e = JSON.parse(param)
if (e.event == ‘’…….’’) {
……
}
else if (e.event == ‘’…….’’) {
……
}
else {
	function execute(){
	// Connections JSON Template (CSBL Editor)
	var connections =
	[{"port_name":"<PORT_NAME>",
	"output_type":"<OUT_PORT_TYPE>",
	"linked_target_widgets":[
	{"widget_name":"<TARGET_WIDGET_NAME>",
	"widget_type":"<TARGET_WIDGET_TYPE>"}]}];
	
	var e=readInput(param, connections);
	
	//events_code_start
	if(e.event == "<EVENT_NAME>"){

```
        ……..……..
        $('body').trigger ({……..});
    }
}
```

```
        //events_code_part_start
        //events_code_part_end
    }
//events_code_end

}
```

As a results, a business intelligence application and any smart applications can be built controlling the widgets and defining the CSBL with the simple insertion of CSBL JavaScript.

Please note that not all widgets provide IN/OUT capabilities. Some of them are limited to IN other to OUT and most of them are IN/OUT. The following two Tables provide you a short summary. If you need to have more IN/OUT CSBL functionalities on widgets, please send an email to snap4city@disit.org

1.2 - Table of Input Commands to OUT and IN/OUT Widgets

Please note that OUT Widgets may receive commands from the User Actions on widget GUI, and they can be automatically capable to make queries to the Smart City API (ASCAPI/SCAPI) (automated calls in the figure above). They are typically configured to work on the basis of query (for example a selector) or a SURI (Service URI, as defined in the development manual and on training course of Snap4City) from which they perform the query to the ASCAPI from which they receive the Entities data to be shown in the widget itself. This means that the Widgets are filled/initialized with some data on which the user may perform some Action producing and internal Event which can provoke effect outside via CSBL on to trigger other connected widgets as well.

Those marked in Green are also validated in the CSBL Visual Editor. The column "**Full Custom CSBL"** refers to the full custom CSBL JavaScript coding approach described in this document, while the CSBL Visual editor compatibility is reported I the last column. The **CSBL Visual Editor** approach is also described in this document.

OUT and IN/OUT Widgets	Users' Action Description and effects	Full Custom CSBL	CSBL Visual editor
widgetTimeTrend	Drill-Down on time interval selection (zoom), providing, SURI, value name, start and end time stamp	Time, SURI, ValueName, starttime, endtime	
	Send Reset Drill-Down	DateTime	DateTime
	Click on a single time instant, providing time stamp, SURI and value name	DateTime	DateTime
widgetMap (multidatamap)	Click on a generic point on the map, providing coordinates	Space Selection	
	Click on a PIN, providing coordinates and ServiceURI of the clicked PIN	SURI Selection	SURI
	Select the bounding box area shown on the map, and the zoom level in order to perform	Space Selection	ListSURI

	geographical Drill-Down on the entities click on dedicated button on map (devices identified by SURIs, Points of Interest etc.) which are currently shown on map		
widgetPieChart	Click on a sector that identifies the name of a metric, providing: value, timestamp, entity name (from which the SURI can be reconstructed) value name, value type and value unit	SURI Selection	SURI
	click on a sector that identifies a device ID or MyKPI ID, providing: value, timestamp, entity name (from which the SURI can be reconstructed) value name, value type and value unit	SURI Selection	SURI
	Get date and time from the calendar	DateTime	DateTime
	Click on legend, providing the status (e.g.: "checked" or "unchecked") of the metric/SURI which has been clicked (under development)	SURI Selection	SURI
widgetBarSeries	Click on a bar, providing: value, timestamp, entity name (from which the SURI can be reconstructed) value name, value type and value unit	SURI Selection	SURI
	Get date and time from the calendar	DateTime	DateTime
	Click on legend, providing the visibility status of each metric/SURI	SURI Selection	SURI
widgetRadarSeries	Click on a radar axis related to a specific metric of a specific device, providing: value, timestamp, entity name (from which the SURI can be reconstructed) value name, value type and value unit	SURI Selection	SURI
	Get date and time from the calendar	DateTime	DateTime
	Click on legend, providing the visibility status of each metric/SURI	SURI Selection	SURI
widgetCurvedLineSeries (multi series)	Drill-Down on time interval selection (zoom), providing: start and end time stamp, and list of SURI. It is also possible to program the synchronization of multiple widgetCurvedLineSeries widgets.	Time, SURI, ValueName, starttime, endtime	DateTime, SURI
	Click on a single time instant, providing: time stamp and list of objects including SURIs and related entity names and value names	Time Selection, list of SURI, value names	
	Click on legend, providing the visibility status of each metric/SURI	SURI Selection	SURI
	Get date and time from the calendar	DateTime	DateTime
	Send Reset Drill-Down	reset zoom	
widgetDeviceTable	Click on the action buttons, providing the action type, the corresponding SURI and a list of attributes with their corresponding values	SURI Action	

widgetImpulseButton	Click on button as a trigger (no parameters are provided)	Action	
widgetOnOffButton	Click on button, providing the new status	Action (param)	
widgetButton	Click on button, providing some action with parameters, a SURI plus type, for example.	Action (param)	
widgetKnob	Drag on knob, providing the value selected on the knob	Action (param)	
	Click on minus and plus action (under development)	Action (param)	
widgetNumericKeyboard	Click on the confirm button, providing the numeric value typed on the keyboard	Action (param)	
widgetEventTable	Click on the action buttons, providing the action type, the corresponding event SURI and the ordering criteria	SURI Action	
widgetExternalContent	It can support HTML pages and SVG Synoptics, in addition to JavaScript, so that it can perform a wide range of actions that can be defined in the HTML/SVG/JS code by the users.	Depending on program	

IN and IN/OUT Widgets present the possibility of receiving Input commands from other widgets and are reported in the following table. In Full Custom CSBL, each IN Widget to be controlled/targeted by other widgets have to be identified/referred by their ID/name. The ID/identifier of each targeted Widget is accessible in the More Options of the widget to be targeted, the so-called Widget name/ID (which is automatically generated and not editable). In the CSBL Visual Editor the connections are automatically addressed by the CSBL visual editor, **and maintained also when you clone a dashboard.**

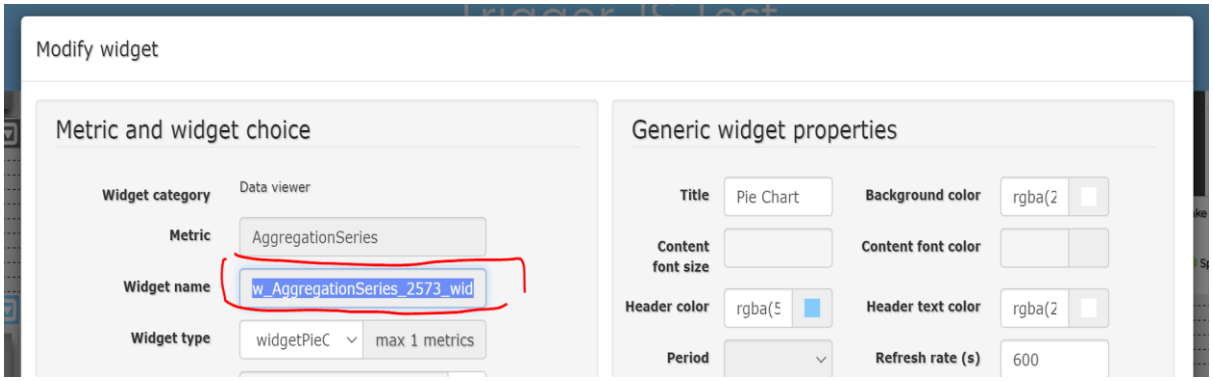


Figure 3 – Widget name/ID from the More Options config.

In most cases, the commands to be sent to an IN Widget includes a JSON very similar or identical to the JSON to be sent to the widget from the Node-RED in the context of writing a SSBL. So, please refer to the same help manual and web pages for the same widgets, and to this document.

1.3 - Table of Input Commands to IN and IN/OUT Widgets

Those market in Green are also tested in the current version of CSBL Visual Editor. The column "**Full Custom CSBL"** refers to the full custom CSBL JavaScript coding approach described in this document, while the CSBL Visual editor compatibility is reported I the last column. The **CSBL Visual Editor** approach is also described in this document.

IN and IN/OUT Widgets	Commands which are ready to be executed from Widget	Full Custom CSBL	CSBL Visual editor
widgetPieChart	Receive a JSON containing a list of SURI, metric names and/or values, and show their corresponding values on a Pie Chart graph.	List of SURI, or data	ListSURI
	set date and time from the calendar	DateTime	DateTime
widgetRadarSeries	Receive a JSON containing a list of SURI, metric names and/or values, and show their corresponding values on a Radar/Kiviat graph.	List of SURI, or data	ListSURI
	set date and time from the calendar	DateTime	DateTime
widgetBarSeries	Receive a JSON object containing a list of SURI, metric names and/or values, and show their corresponding values on a Bar graph.	List of SURI, or data	ListSURI
	set date and time from the calendar	DateTime	DateTime
widgetSingleContent	Receive a SURI and a metric name, or a value, or a text string, and show the corresponding value.	SURI or data, etc.	SURI
	Receive and show a HTML/JS page	HTML	
widgetSpeedometer	Receive a SURI and a metric name, or a value, and show the corresponding value on a speedometer graph.	SURI, or data	SURI
widgetGaugeChart	Receive a SURI and a metric name, or a value, and show the corresponding value on a gauge graph.	SURI, or data	SURI
widgetTimeTrend	Receive a SURI and a metric name, or a value, and show the corresponding time-series on a line, spline, area or stacked area graph.	SURI, or data	SURI
	Receive reset zoom	Reset	
widgetTable	Receive a JSON containing a list SURI, metric names and/or values, and show the corresponding time-series on a HTML static table.	List of SURI, or data	ListSURI
	Receive start datetime, end datetime without change sources IDs (under development)		
widgetCurvedLineSeries	Receive a JSON containing a list of SURI, metric names and/or values, and show the corresponding time-	List of SURI, or data	ListSURI

	series on a line, spline, area or stacked area graph.		
	Receive start datetime, end datetime without change sources IDs	time interval	DateTime Interval
	set date and time from the calendar	Time	DateTime
	Receive reset zoom	Reset	
widgetDeviceTable	Receive a JSON containing a list of SURI representing IoT devices, and show their related attributes and values on an interactive table which provides action buttons.	List of SURI, or data	ListSURI
widgetEvent	Receive a JSON containing a list of SURI representing events as virtual devices, and show their related attributes (e.g., start and end date) and values on an interactive table which provides action buttons.	List of SURI, or data	ListSURI
widgetMap	Receive a JSON containing a list of SURI or entities (such as heatmaps, categories of Points of Interest etc.) and show them on an interactive map as clickable markers, dynamic SVG pins, traffic flows, heatmaps etc.	List of SURI, or data	ListSURI
widgetOnOffButton	Receive and show a value representing the status	Action	Action
widgetKnob	Receive and show a value	Action	Action
widgetNumericKeyboard	Receive and show a value	Action	Action
WidgetExternalContent	A SURI and a type, eventual parameters	Action	Action

2 - Generic CSBL JS Pre-defined Functions and Variables

Valid for any JavaScript in: Full Custom CSBL and CSBL Visual Editor.

In order to make easier for the user to implement basic/fundamental operations which can be useful to send, receive and manage data in the Snap4City CSBL approach, some pre-defined variables and functions have been implemented, which can be used as a sort of function library inside the JavaScript code. First of all, the following CSBL standard JavaScript variables have been defined for every widget (thus, with a scope limited to the code related to each single widget):

- ***e*** represents a JSON of parameters and data coming from each widget (for instance, the SURI or array of SURI passed when clicking on a widget graphic element, map, etc., see table in Section 5.1 for reference about data and parameter format sent by each widget) and that may be passed to a pre-defined function after the occurrence of an event, as well as used in your own custom JavaScript;
- ***connections*** represent a JSON mapping the connections for each widget.

The pre-defined functions, with details about their parameters, a short description, and a list of supported widget events that can exploit the specific function, are listed in the following table.

Pre-defined function	***Parameters and supported OutputTypes***	***Description***	***Supported Events Sect. 3.1***
sendSURI (<PORT_NAME>, <JSON_DATA>)	<PORT_NAME>: name of the output port to which data are sent <JSON_DATA>:_data related to a single Snap4City Service URI which are sent to the output port specified by <PORT_NAME>. *It is typically assigned the value of the pre-defined **e** variable, as above described*. Supported OutputTypes: **SURI**	Send data, as passed in the <JSON_DATA> parameter (e.g.: high-level type, metric etc.) related to a single Snap4City Service URI to be sent to the output port specified by the parameter <PORT_NAME>	• **click**
sendListSURIAndMetrics (<PORT_NAME>, <JSON_DATA>)	<PORT_NAME>: name of the output port to which data are sent <JSON_DATA>:_data related to an array of multiple Snap4City Service URI which are sent to the output port specified by <PORT_NAME>.	Send data, as passed in the <JSON_DATA> parameter (e.g.: high-level type, metric etc.) related to an array of multiple Snap4City Service URI to be sent to the output port specified by the parameter <PORT_NAME>	• **legend_item_click**

	*It is typically assigned the value of the pre-defined **e** variable, as above described*. Supported OutputTypes: **ListSURI**		
sendMapSURI (<PORT_NAME>, <JSON_DATA>)	<PORT_NAME>: name of the output port to which data are sent <JSON_DATA>:_data related to a single Snap4City Service URI which are sent to the output port specified by <PORT_NAME>. *It is typically assigned the value of the pre-defined **e** variable, as above described*. Supported OutputTypes: **ListSURI**	Send data, as passed in the <JSON_DATA> parameter (e.g.: high-level type, metric etc.) related to an array of multiple Snap4City Service URI (formatted in a specific way for the **widgetMap**), to be sent to the output port specified by the parameter	• **click** • **geo_drill_down**
sendTimeRange (<PORT_NAME>, <JSON_DATA>)	<PORT_NAME>: name of the output port to which data are sent <JSON_DATA>: time interval (specified by lower and upper timestamp, t1 and t2 respectively, in Unix milliseconds) sent to the output port specified by <PORT_NAME>. *It is typically assigned the value of the pre-defined **e** variable, as above described*. Supported OutputTypes: ***DateTime*, *DateTimeInterval***	Send time interval, specified in the <JSON_DATA> parameter (in Unix milliseconds) which is typically used to zoom temporal view in **widgetCurvedLineSeries**	• **time_zoom** • **reset_zoom**

We can observe from the above table that the user has only to specify the output port name (<PORT_NAME>), since the correct <JSON_DATA> is automatically passed to the pre-defined JavaScript variable. Therefore, handling the correct data format, which may be the most difficult part for non-technically skilled users, is completely transparent to the user.

As an alternative, you can implement your custom CSBL by writing directly JavaScript code from scratch in the CKEditor of the widget (see next Section 4), which can be accessed in the "More Options" tab of the widget in the dashboard editor mode.

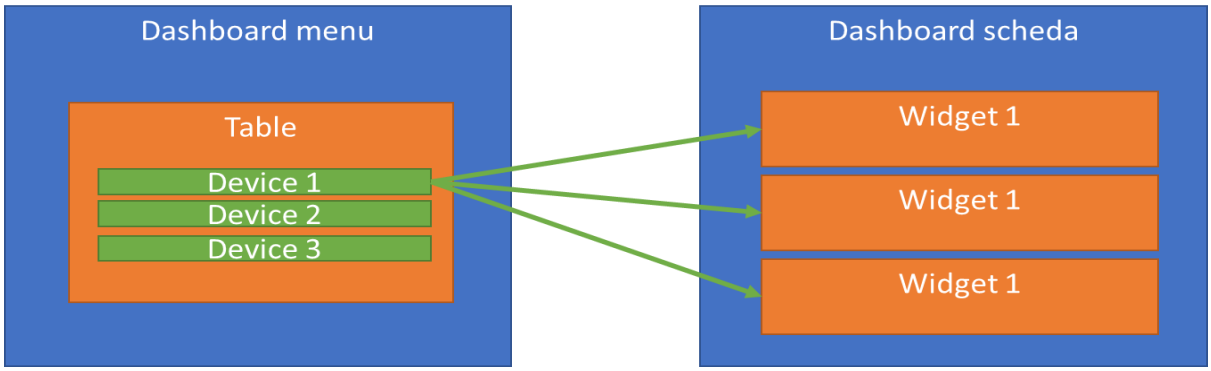
2.1 – Macro functions on CSBL JavaScript for Widgets and dashboards's connections

Other functions which can be exploited on Actions JavaScript segments:

- **Open a New Dashboard: openNewDashboard()**
- **Get parameters: getParams()**
- **Etc. other**

In this manner, it is possible to activate in a new dashboard some actions on specific elements. ***This means that you can from a Dashboard with CSBL JavaScript embedded into a Widget to Open a different dashboard passing to it a parameter, which is in turn passed to the JavaScript into the Widgets. So that the new dashboard can show any kind of data and services according to the parameter received. This approach can be performed to create dashboards focused on showing data for devices of a given type, and receive the specific device ID to focus only by the parameter….***

It is assumed that you need to create a system of interfaces (dashboard) that allow the following interaction: **Following an event launched by a widget in dashboard A, dashboard B opens and is given the parameters necessary for a widget in dashboard B to retrieve data and display it.**



See Section 9 for an example with call and called dashboard.

2.2 -- Time Machine, history of commands as do/undo on Business Intelligence Dashboards

In each dashboard where at least one of its widgets provides a CSBL JavaScript code saved in the CK Editor, the history of all the different actions performed and triggered in the CSBL is locally saved in the browser, until the dashboard is closed. This is done in order to allow the user to come back to different views performed by earlier actions. The history of CSBL actions can be viewed by clicking on the history clock icon which appears in these cases on the top-right corner of the dashboard header.

See **section 8** on the time machine history of commands as do/undo.

3 - CSBL Visual Editor modality

The Snap4City **CSBL Visual Editor** is a graphical interface which is integrated in the Snap4City Dashboard Builder to make easier to implement business intelligence dashboards and thus smart applications. CSBL Visual Editor has been designed to reduce the coding for producing high valuable results and it is accessible for non-programming skilled users. On t the other hand it also enables fully skilled JavaScript programmers to exploit more functionalities, by scripting, and may be passing at the Full Custom CSBL to exploit full potentialities of snap4City Dashboard Builder.

The **CSBL Visual Editor** can be opened in the dashboard page (from the edit mode), by clicking on the "CSBL Editor" button in the upper menu bar. A detail of the graphic interface of the CSBL Editor is displayed in **Figure 4**. In the CSBL Visual Editor, each node of the flow chart represents a widget of the dashboard (identified by its unique ID displayed in the "WidgetName" box). In these nodes/widgets the user can write the JavaScript code (in the "Code" box) that implements the desired CSBL for each event related to the specific widget (shown in the "Event" box), so that each event and action of the current widget can trigger the execution of a specific JavaScript code (check **Section 2.1** for a list of the available events). Therefore, changing the event selection in the "Event" changes also the visualization of the specific code associated to that event, which is displayed in the "Code" box.

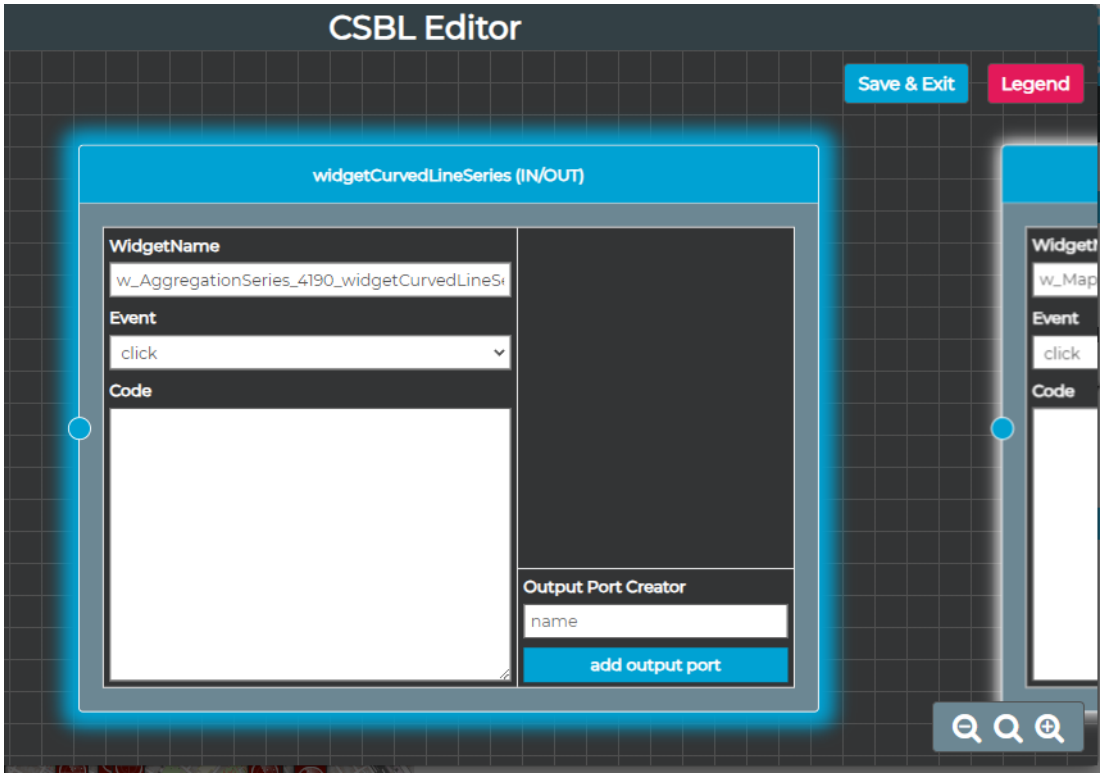


Figure 4 – The CSBL Visual Editor Graphical User Interface: single widget view

At the present version of this document, the CSBL Editor is working for the following widgets (implementing their IN, OUT and IN/OUT functionalities as described in the tables in Section 1.1): **widgetBarSeries**, **widgetCurvedLineSeries**, **widgetMap**, **widgetPieChart**,

widgetRadarSeries, widgetGaugeChart, widgetSpeedometer, widgetSingleContent. Regarding the other widgets not included in the previous list, the CSBL Editor may allow to view and edit the JavaScript code (as explained in the following), but it may be not still fully compliant to handle IN and OUT connections. Further adaptations and developments of the CSBL Editor will be made to improve compliance with all the other widgets will be evaluated and implemented in future deployments.

3.1. Widget Events

In the following table, all the available events are listed and all the widgets supporting each event at the current state of the CSBL Visual Editor. In future developments, all the widgets (according to tables in **Sections 1.2/1.3**) will be leveraged. User events are those coming from the GUI of the widget acted by the users, while CMD are those that may arrive from other widgets from the Input port of the widget. See Table in **Section 1.2.**

Events	***Supported Widget Types***	***Description***	***User/ Cmd***	***Output Type***
click	widgetBarSeries widgetCurvedLineSeries widgetMap widgetPiechart widgetRadarSeries	Click on a single bar / kiviat branch / pie-chart sector / time-series sample	user	SURI
legend_item_click	widgetBarSeries, widgetCurvedLineSeries, widgetPiechart widgetRadarSeries	Click on legend item	user	ListSURI
geo_drill_down	widgetMap	Geographic drill down considering all the devices and entities displayed on the current bounding box of the widgetMap	user	ListSURI
time_zoom	widgetCurvedLineSeries	Temporal zoom on multi-series widgets	User	DateTime
reset_zoom	widgetCurvedLineSeries	Reset temporal zoom on multi-series widgets	User	DateTime
external_commands (compatibility event with Full Custom CSBL, read in the following)	widgetBarSeries widgetCurvedLineSeries widgetMap widgetPiechart widgetRadarSeries	Event for implementing custom JavaScript code (in this case, it is recommended to check the code compliance with the CSBL Editor template as	Cmd	custom

		described in **Section 3)**		

Currently, the “external_commands” event is a special event aimed at handling Full Custom CSBL segments (included directly into the CKEditor from More Options) which are not compliant with the CSBL Visual Editor template

3.2. IN/OUT Connections

Each Widget in the CSBL Visual Editor presents a single input port, to which one or multiple output ports from other widgets can be connected. In those widgets that involves also output functionalities (i.e., sending messages to other widgets), an additional section is shown in the right part of the node, which is dedicated to specifying the output ports, as depicted in **Figure 5.**

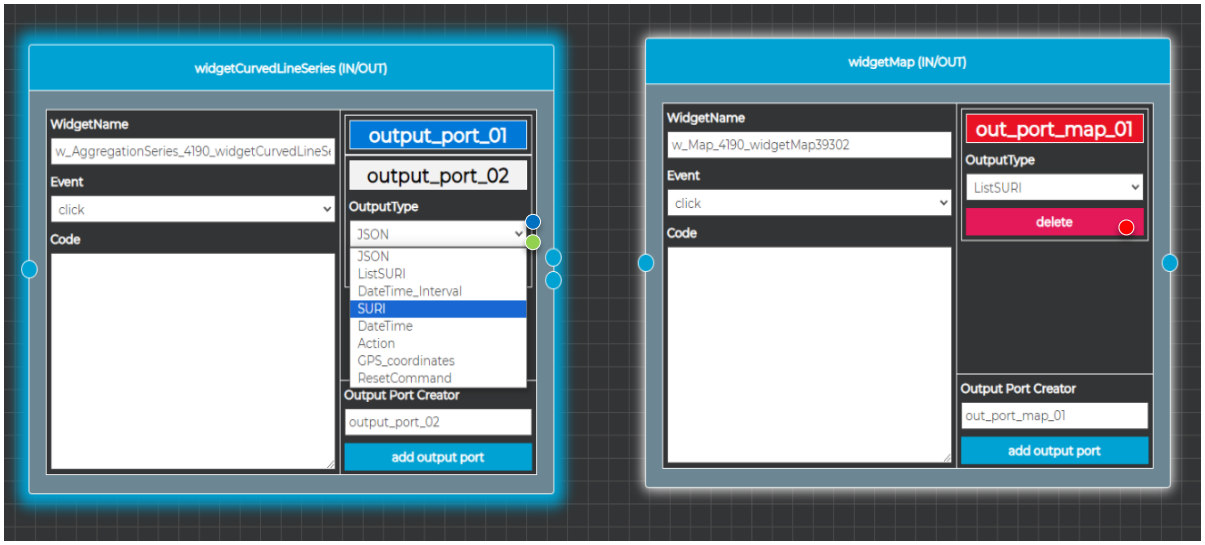


Figure 5 – Adding output ports to a Widget/node into the visual editor.

At the bottom of the widget representation into the CSBL Visual Editor, the user can add a new output port by writing a desired output port name and by clicking the “add output port” button. The names of the output ports must be unique in the Widget. After adding an output port, it is possible to access further information by clicking the button with its name. Actually, each output port has to be classified to be of a specific data type (OutputType) which is produced from that port. Moreover, a delete button is provided to remove the selected port from the design (see the right node/widget in **Figure 5**).

The OutputTypes are coherent with the possible Inputs that can be received by the IN and IN/OUT Widgets as described in Section 1.3.

The Full Custom CSBL has less/no restrictions on this kind of message types.

The different OutputTypes for the messages exchanged among widgets available are listed in the following (see the left node in Figure 5):

- ***SURI***: Snap4City **Service URI**. This OutputType is suitable in those cases in which it is needed to send data related to a single Service URI. For instance, when clicking on a single graphical element of a widget related to a single IoT Device (a single bar, a single pie chart sector etc.).
- ***ListSURI***: array of multiple Snap4City Service URI. This OutputType may be suitable in those cases in which it is needed to send data related to multiple Service URI at once (for instance, clicking on the legend of a widget when the user wants to hide from view data related to some devices, while sending data related to all the other devices which are not excluded from current visualization.
- ***DateTime***: a timestamp in Unix epoch milliseconds format. This may be suitable to send single timestamp, for instance when clicking on a single chart point in a time-trend or multi-series widget.
- ***DateTimeInterval***: a temporal interval identified by lower (t1) and upper (t2) limits, represented in Unix epoch milliseconds format. This may be suitable when selecting a temporal interval to zoom in a time-trend or multi-series widget.
- ***GPSCoordinates:*** geographic coordinates (Latitude, Longitude) which are typically sent when clicking on a free area on a map widget.
- ***JSON:*** generic JSON format which can be useful when the user needs to send generic data which can be handled by Full Custom CSBL.
- ***Action:*** They are actions performed by the user (such as the click of a button, the drag of the knob, etc. See table in **Section 1.2**) or from Input port of the Widget (typically avoided to prevent loops). They can be complex JSON.
- ***ResetCommand:*** rest is typically the return at the initial status, for example in the time default time range or condition of the widget (to be implemented in future developments). For example, the reset to conditions before drill down in time, see table in **Section 1.2**.

Each output port can be connected to the desired target widget(s), as depicted in **Figure 6**, by click/drag and drop a connections with the mouse, starting by clicking on the desired output circular pin related to the corresponding output port of the specific node (which have the same color and positional order from top to bottom, as shown in **Figure 6**) and dragging the connection line to the desired target widget input port (which has the same light blue color for all the nodes). Changing the type of output port will also reset the connections related to that port. Moreover, the position of each widget box can be arranged as the user wants in the CSBL Visual Editor panel. When ready, the user can click on the “Save & Exit” button to save the current CSBL Visual Editor configuration and return to the dashboard edit page visualization. The results would be a general configuration to all widgets of the dashboard, business intelligence view. Please remind that a set of views/dashboards can be connected / activated each other as well.

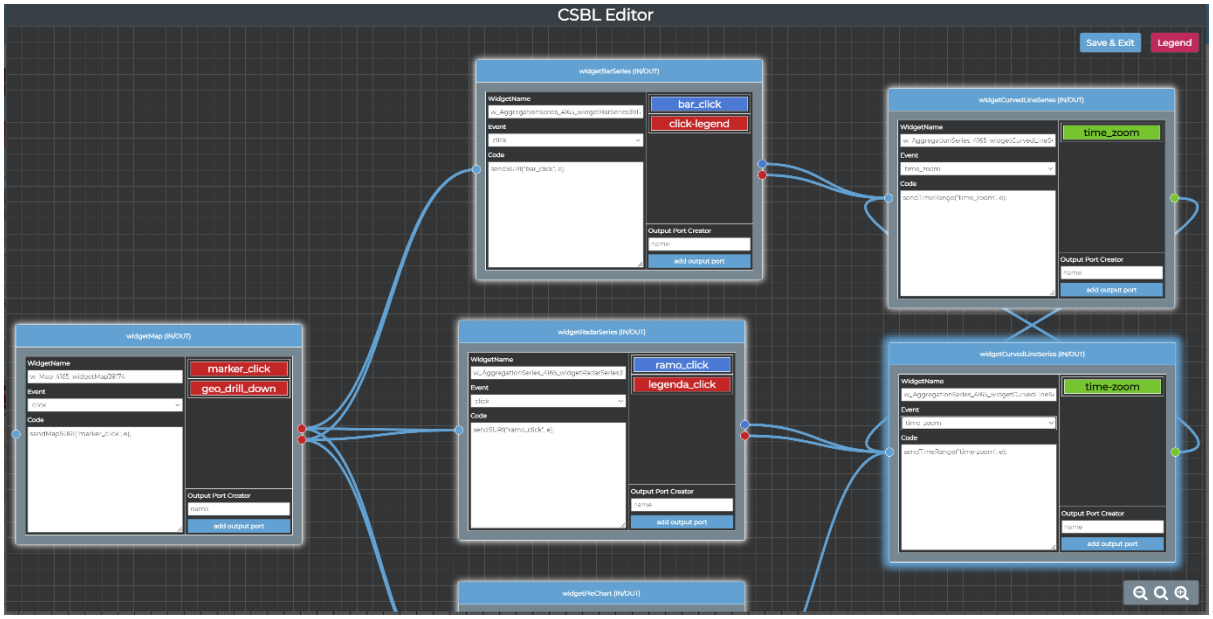


Figure 6 – Widgets connections in the CSBL Visual Editor of a Dashboard.

The "Legend" button shows a short info box with different colors of Output ports and the main shortcuts and actions that can be performed in the CSBL Visual Editor (see **Figure 7**).

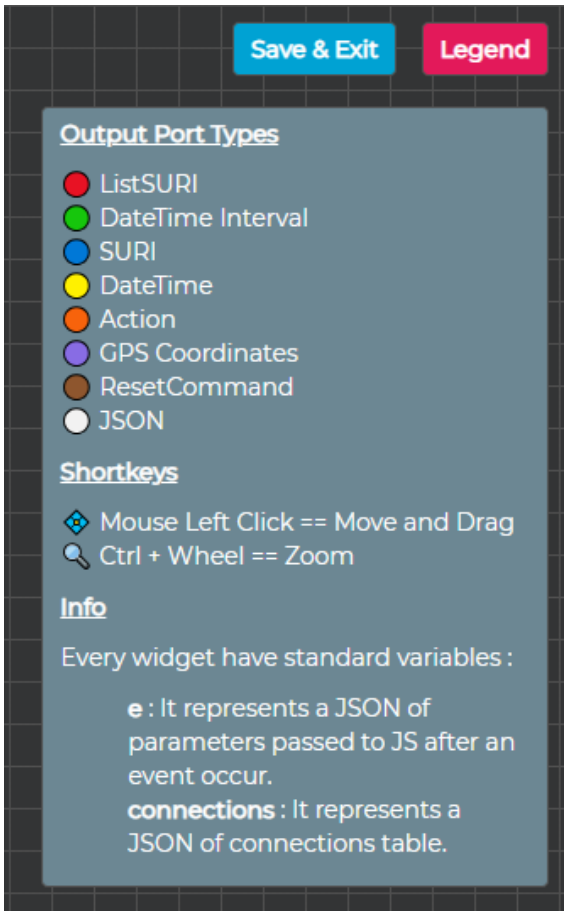


Figure 7 – CSBL Editor Legend

3.3 Compatibility from Full Custom and CSBL Visual editor

However, in order to make the CKEditor code compliant with the CSBL Visual Editor (so that you can use both modalities, i.e. CKEditor and CSBL Visual Editor GUI to view and edit your CSBL), a default template has been added to the CKEditor Code which helps the user to understand how to write CSBL code in a way which is compliant with the CSBL Editor (see Section 4).

In case the CKEditor of a certain widget code is not compliant with the CSBL Visual Editor template of JavaScript, a yellow header is displayed for that widget, as well as warning message advising the user that continuing to edit and save the current code may lead to unexpected working or saving problems (see **Figure 8**):

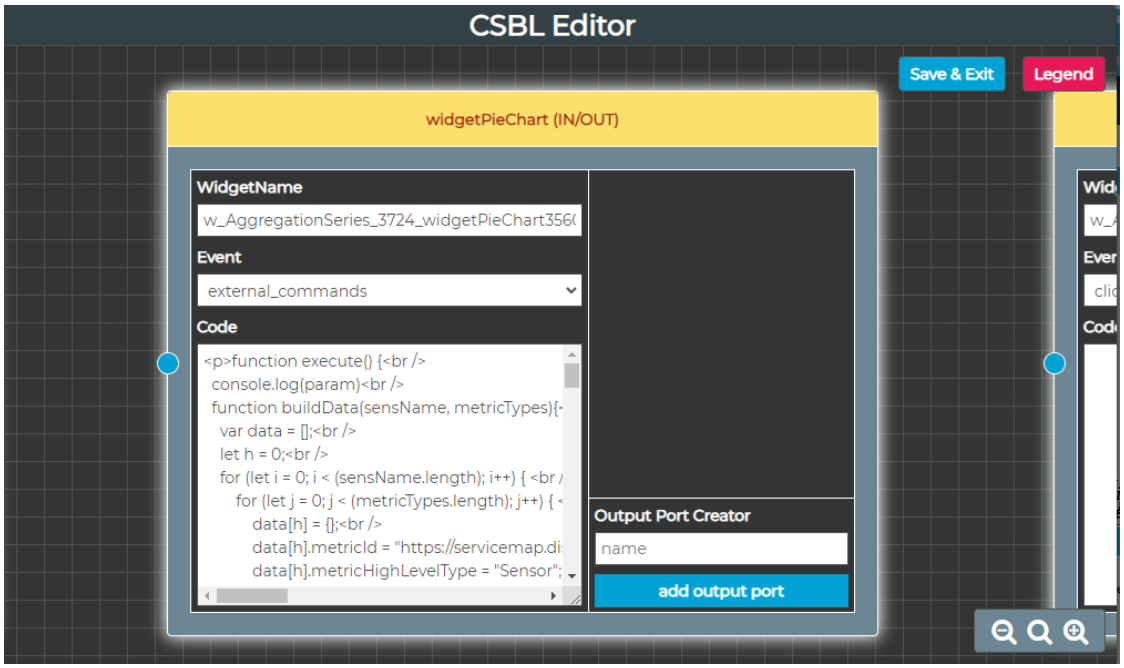
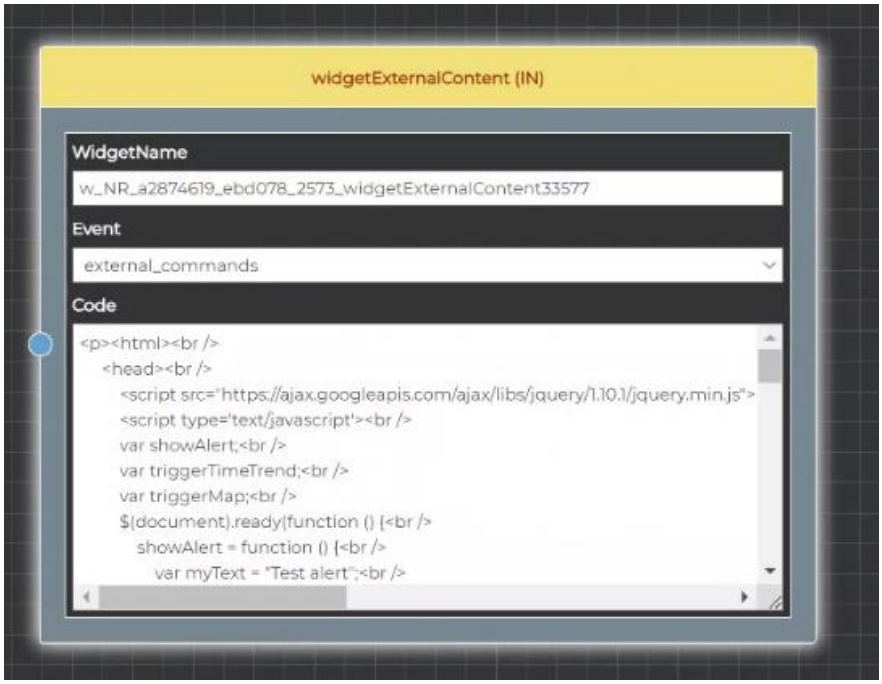
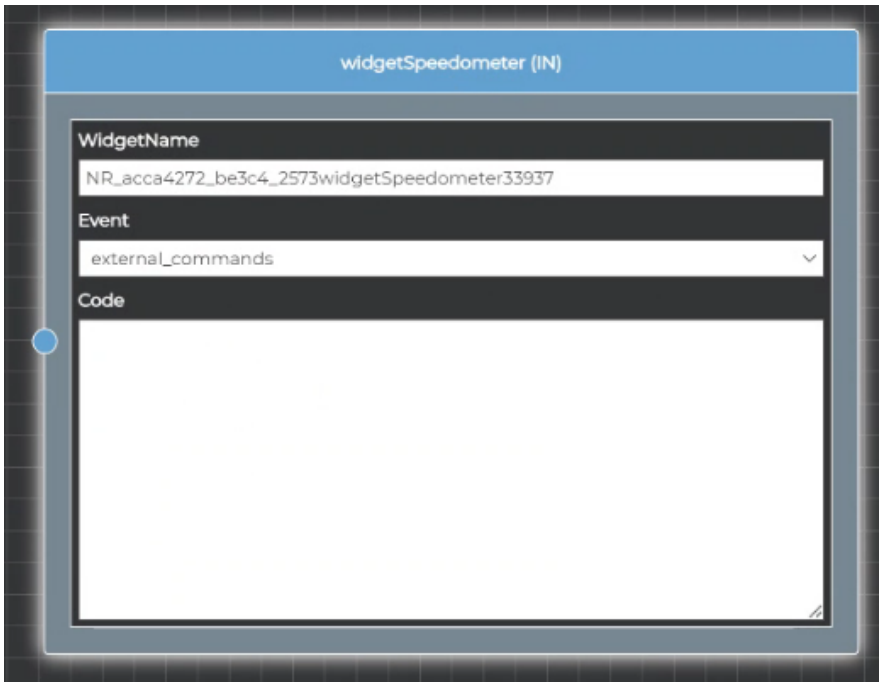


Figure 8 – Yellow header of a widget node warning that CKEditor code is currently not compliant with CSBL Editor template reported in the following.

The widget external content in the following figure has to be addressed as a full custom CSBL editor widget and thus the JavaScript is not structured according to CSBL Visual Editor template. This is the reason since it is typically market in Yellow. Please note that it is in effect and IN/OUT Widget without the output ports, providing maximum flexibility.



In the following figure an IN Widget, no output is present. In effect it is not present on Table of Section 1.2.



3.4. Example of Usage

In the public Snap4City dashboard "CSBL Editor Test" depicted in **Figure 9,** the CSBL Editor has been exploited to easily create a number of connections and interactions among the different widgets. The dashboard is available at the following URL: https://www.snap4city.org/dashboardSmartCity/view/index.php?iddasboard=NDE2NQ== in view mode; the edit mode is accessible to the dashboard owner only.

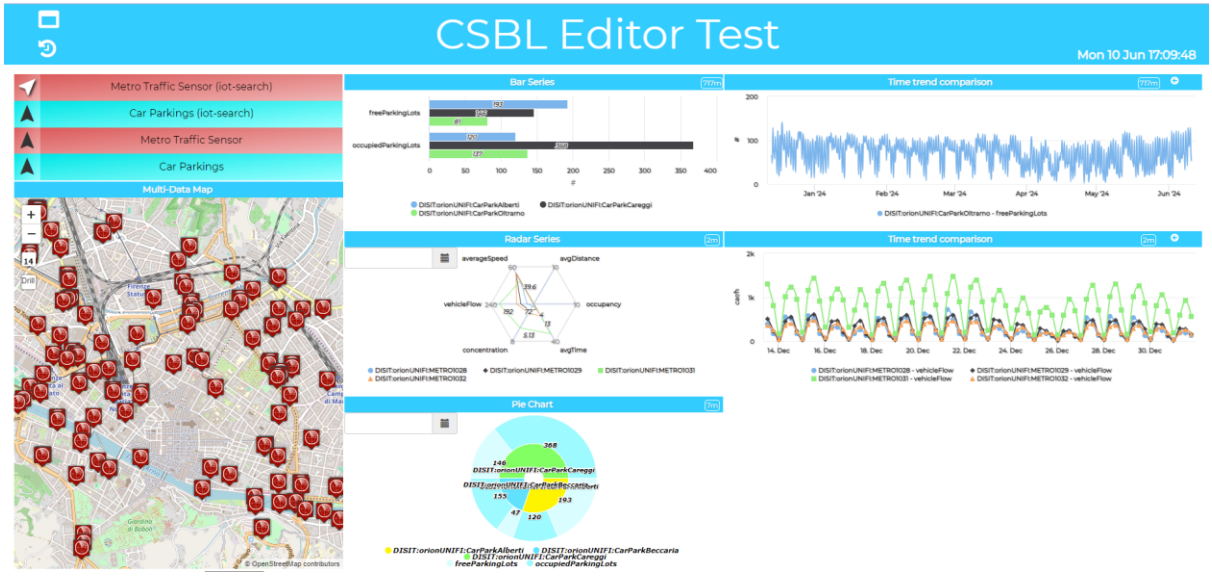


Figure 9 – CSBL Editor Test dashboard.

In the widgetMap on the left, it is possible to click on a single device marker, as well on the "Drill" button on the left for geographic drill-down, in order to send single SURI or an array of SURI, respectively. These data are sent to the three widgets on the centre, i.e.: a bar-series widget, a pie-chart widget and a radar widget, which can show the last values of passed data according to the different modalities. In the CSBL Editor, this is obtained in a very easy way as shown in **Figure 10**.

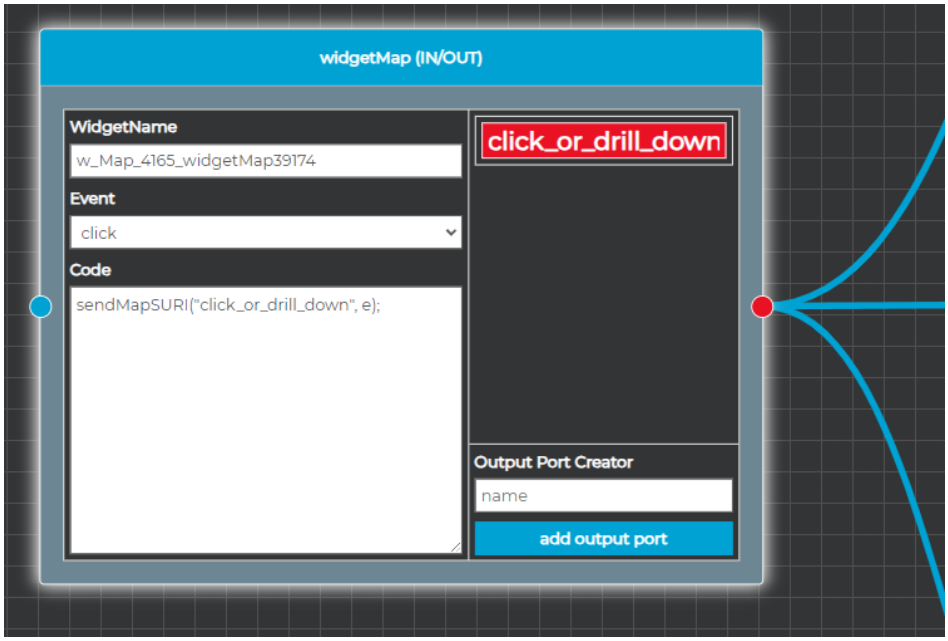
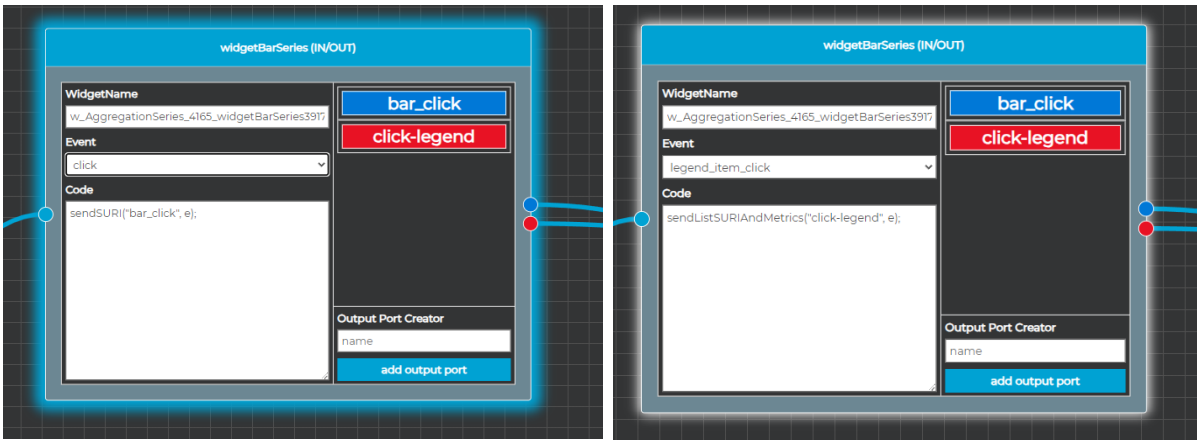


Figure 10 – Detail of the CSBL Editor related to the widgetMap node

In the widgetMap, the pre-defined **sendMapSURI ("click_or_drill_down", e)** function (described in the table of Section 3.3, which can handle both SURI and List of SURI data sending modalities for the widgetMap) is called (that is, is written in the Code box) for both

"click" and "geo_drill_down" events. In both cases, the widget data are automatically passed in the correct format in the pre-defined JavaScript **e** variable to the unique "click_or_drill_down" output port (actually, a single output port can handle all data of the same OutputType (as described in Section 3.2), even if related to multiple events. The three connections coming out from this node go to the bar-series, pie-chart and radar widgets, respectively (see **Figure 13** for a general overview of the CSBL Editor flowchart).

Then, let's consider the widgetBarSeries. In this case, the corresponding node in the CSBL Editor has been configured to: (i) send data to the multi-series on its right when clicking on a single bar (thus, related to a single Service URI) and to (ii) send data to the multi-series on its right when clicking on a legend item (thus, related to multiple SURI, i.e. the ones that are not excluded from visualization after clicking on the legend item). This can be done very easily in the CSBL Visual Editor by: (i) calling the pre-defined **sendSURI("bar_click", e);** function associated to the "click" event, which sends data passed in the pre-defined **e** variable to the "bar_click" output port (which is set to SURI as OutputType) and (ii) calling the pre-defined **sendListSURIAndMetrcis**("click-legend", e) function associated to the "legend_item_click" event, which sends data passed in the pre-defined **e** variable to the "click-legend" output port (which is set to SURI as OutputType), as depicted in **Figure 11**.



(i) (ii)

Figure 11 – Detail of the CSBL Editor of the widgetBarSeries node related to the click event (i) and the legend_item_click event (ii)

Then, the multi-series widget which is target of the previous mentioned bar-series can send to itself time interval related to the time_zoom and reset_zoom events. This can be easily done in the CSBL Editor by calling the pre-defined **sendTimeRange**("time_zoom", e) function to the "time_zoom" output port for both time_zoom and reset_zoom events (as shown in **Figure 12**).

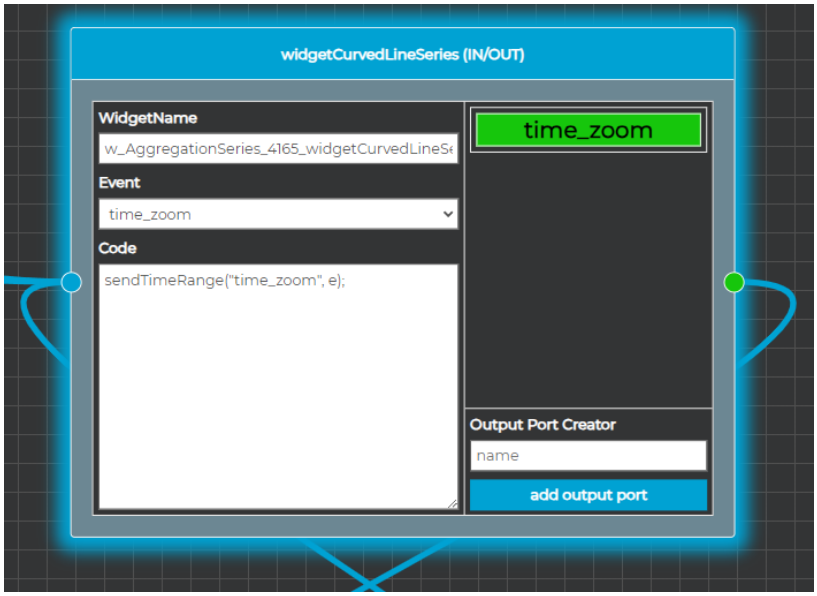


Figure 12 – Detail of the CSBL Editor related to the widgetMap node

Finally, a general overview of the whole CSBL Editor flowchart is provided in **Figure 13**:

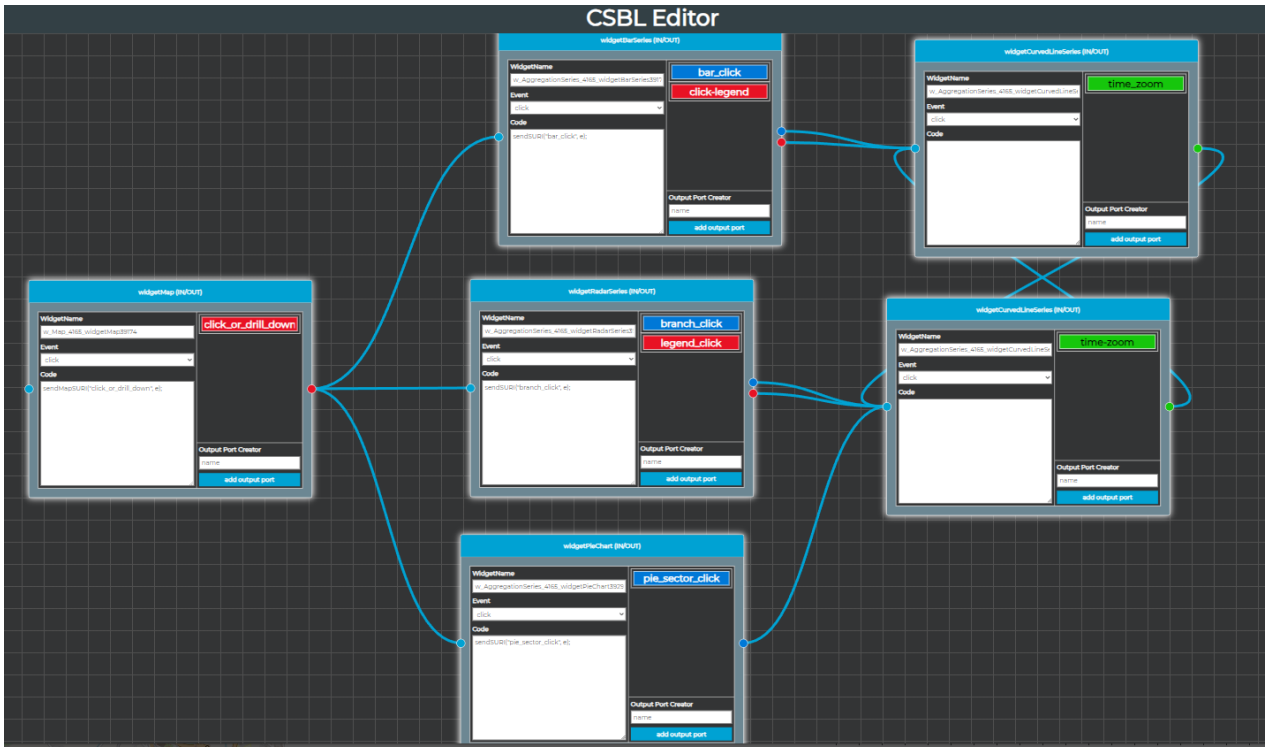


Figure 13 – General overview of the CSBL Editor flowchart for the dashboard described in this Section

4 - Full Custom CSBL: CKEditor JavaScript code

This section describes how to implement your CSBL from scratch, writing the full JavaScript code without the necessity to use the CSBL Editor. This option may be suitable for programming skilled user, since it offers a higher freedom in CSBL code development with fewer constraints, however in this case the user has to manually implement the whole logic for preparing data to be sent, for handling received data, for handling event triggers etc. Within the dashboard builder application access a dashboard in editor mode and create a widget from the wizard.
After creating the widget, access the more options menu, activate script insertion in the Enable CK Editor menu.
At first instance, the following default template is shown in the CKEditor, which shows the user the format which is compliant with the CSBL Editor (see previous Section 3):

```
/*
    Use the following template (comments included)
    if you want your CK Editor code to be compliant
    with the CSBL Editor.
*/

function execute(){
        // Connections JSON Template (CSBL Editor)
        var connections =
[{"port_name":"<PORT_NAME>","output_type":"<OUT_PORT_TYPE>","linked_target_widgets
":[{"widget_name":"<TARGET_WIDGET_NAME>","widget_type":"<TARGET_WIDGET_TYPE>"}]}];

        var e=readInput(param, connections);

        //events_code_start
        if(e.event == "<EVENT_NAME>"){
            //events_code_part_start
            //events_code_part_end
        }
        //events_code_end

}
```

The CSBL Editor actually exploits the JavaScript array of objects `connections` to define and map all the input and output connections for each widget. This array of objects is automatically created by the CSBL Editor when the user graphically sets all the desired connections among the widgets. Therefore, it is not shown in the CSBL Editor, so that it cannot be edited directly in the CSBL Editor. However, it is shown in the CKEditor in order to give user the full flexibility to see and edit it. In particular, “<PORT_NAME>” is the name of the specific output port; “<OUT_PORT_TYPE>” represents the specific OutputType (as described in Section 3.2) of the port; "<TARGET_WIDGET_NAME>” and "<TARGET_WIDGET_TYPE>" represent the name and type of the current widget, respectively; "<EVENT_NAME>" is the name of the specific event associated to the current widget (as described in Section 3.1).

In addition, the CSBL calls a pre-defined function `readInput` (in order to correctly prepare the data sent by the specific event which is represented by the JavaScript variable "e"), and it also uses the comment tags to properly split the code related to each single event.

In the following part of this Section, it is described how to implement your CSBL from scratch in the CKEditor, without the necessity to follow the above-mentioned template. However, be aware that in this case the CSBL Editor may not work properly with your custom CKEditor logic.
In the CKEditor code it is necessary to specify the identification code of the widget that will receive the content of the script (Most details on the function in Section 5)
Once the code has been entered, click on the save icon to store it in the database.

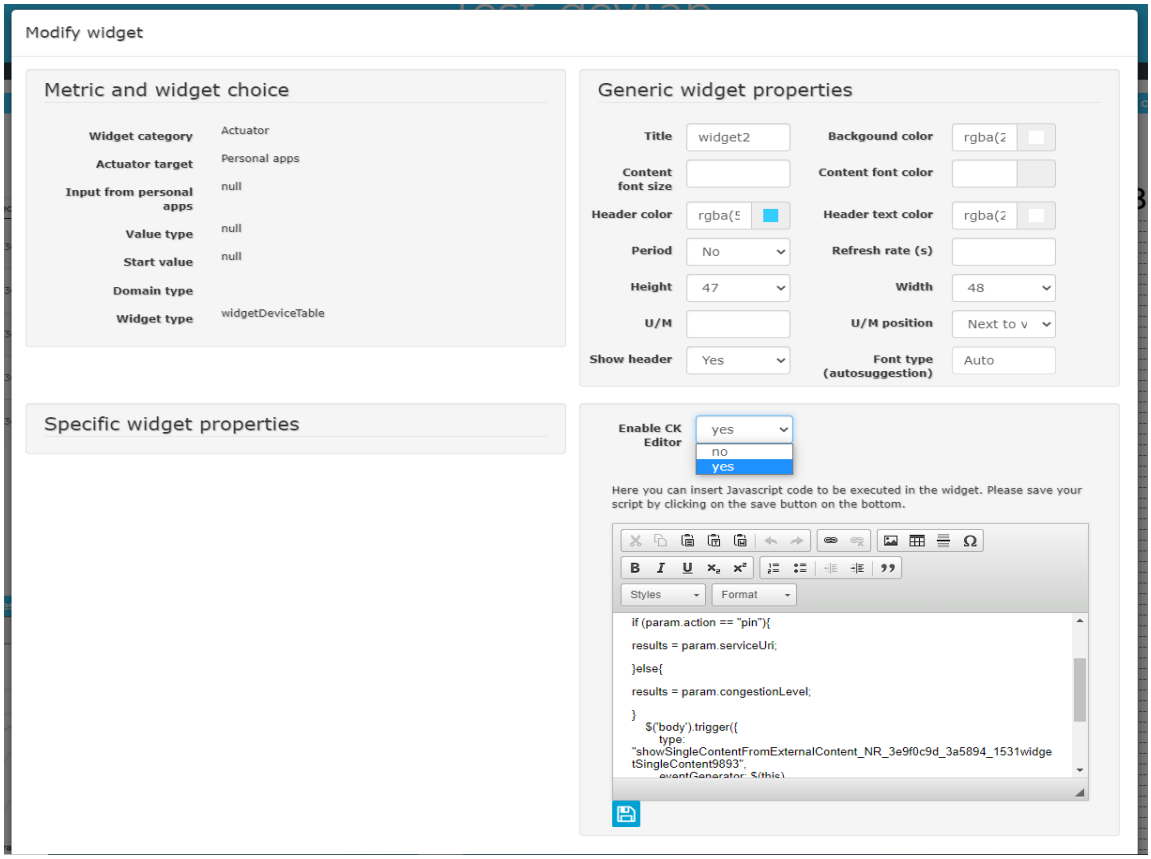


Figure 14

At this point, the widget indicated as a target in the function can read and interpret the contents of the script if an event occurs that triggers the execution of the function. This event is typically generated by a click on a specific data representation.

4.1 Template JavaScript

Let's consider a first simple example to understand how to implement the JavaScript execute function in a widget CK Editor, which is needed to perform the desired logic on some target widget(s). First of all, in this example let's consider a dashboard in which an existing widget is present, the id of which, <TARGET_WIDGET_NAME>, must be noted. Let the target widget have id <TARGET_WIDGET_NAME>.

The JavaScript function to be inserted in the appropriate CK Editor box (in more options) of the current widget is of the following type:

```
function execute() {
        $('body').trigger({
        type: "<TARGET_WIDGET_EVENT_HANDLER_METHOD_NAME>_<TARGET_WIDGET_NAME>",
        eventGenerator: $(this),
        targetWidget: "<TARGET_WIDGET_NAME>",
        passedData: { "dataOperation": param }
    });
}
```

The “type” key to be valorized in the JSON inside the $('body').trigger is a string obtained by concatenation of the name of the event handler method, <TARGET_WIDGET_EVENT_HANDLER_METHOD_NAME>, which is defined in the target widget source code (see Section 6 for detailed specifications for each widget), the underscore character and the target widget id <TARGET_WIDGET_NAME>. Data can be passed (see the “param” variable in the above example, which is passed as value of the “passedData” key in the JSON, as detailed in Section 4.2), for instance a JSON containing a list of SURI, corresponding metric names and/or values etc. which are currently viewed on the source widget.

From a widget it is also possible to send parameters to more than one widget simultaneously by activating more than one trigger event, each one associated with the reading widget id. In this case the same parameters are sent from a widgetmap to multiple widgets of various types:

```
function execute(){
    var e = JSON.parse(param)
    var metrics =
["anomalyLevel","averageSpeed","avgTime","concentration","congestionLevel","vehicleFlow"];
    var data = [];
    let h = 0;
    for (var l in e.layers) {
        for(var m in metrics){
            data[h] = {};
            data[h].metricId = "https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri="+
e.layers[l].serviceUri;
            data[h].metricHighLevelType = "Sensor";
            data[h].metricName = "DISIT:orionUNIFI:" + e.layers[l].name;
            data[h].metricType = metrics[m];
            h++;
        }
    }
    $('body').trigger({
        type: "showPieChartFromExternalContent_w_AggregationSeries_1_widgetPieChart65",
        eventGenerator: $(this),
        targetWidget: "w_AggregationSeries_1_widgetPieChart65",
        passedData: data
    });
    $('body').trigger({
        type: "showBarSeriesFromExternalContent_w_AggregationSeries_1_widgetBarSeries48",
        eventGenerator: $(this),
        targetWidget: "w_AggregationSeries_1_widgetBarSeries48",
        passedData: data
    });
    $('body').trigger({
        type: "showRadarSeriesFromExternalContent_w_AggregationSeries_1_widgetRadarSeries49",
```

```
        eventGenerator: $(this),
        targetWidget: "w_AggregationSeries_1_widgetRadarSeries49",
        passedData: data
    });
}
```

In the example above, it is to be noticed that some data preparation is performed before passing data to the target widgets.

4.2 Parameters

The *param* variable consists of the input value generated by the widget state change, as well as by the entities and related attributes currently displayed on the widget at the moment of triggering the action. You can send it in passedData, or use it to perform operations in JavaScript:

The passedData field can be:

```
passedData: {
        "dataOperation": param
}
```

5 - Full Custom CSBL: List of Widgets' actions and functionalities

5.1 Table of non actuator widgets (partial table)

The following table shows the available actions/functionalities (with the corresponding data/parameters sent) of the various non-actuating widgets. In particular, the Sended Data / Parameters column of the following table show the data and parameters format sent by each widget to the CSBL execute function, which is assigned to the **e** variable of CSBL Visual Editor template, as described in Section 3.3.

Widget name	Action/ Functionality	Sended Data/Parameters

widgetTime Trend	Zoom on a desired time interval to perform temporal Drill-Down on selected time range	<pre>{
		  "t1" : <FIRST_TIMESTAMP_MILLIS>,
		  "t2" : <SECOND_TIMESTAMP_MILLIS>,
		  "sUri": <SERVICE_URI>,
		  "metricName": <VALUE_NAME>
		}</pre>
		Example:
		<pre>{
		  "t1": 1655186608240.9412,
		  "t2": 1659528987182.6824,
		  "sUri":
		"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/ARPAT_QA_FI
		-GRAMSCI_SV",
		  "metricName": "PM10"
		}</pre>

	Click on a single point of the chart to perform temporal Drill down on a single time instant	in case of Drill-Down on a single time instant t1 = t2.

widgetMap	Click on an Entity/Device marker on map and send information (as retrieved by querying the Snap4City SmartCity API) of the clicked Entity/Device	<pre>{
		  "event": "click",
		  "layers": [{
		  "distance": "0.9243",
		  "hasGeometry": false,
		  "multimedia": "",
		  "name": "METRO759",
		  "photoThumbs": [],
		  "serviceType": "TransferServiceAndRenting_Traffic_sensor",
		  "serviceUri":
		"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO759",
		  "tipo": "Traffic_sensor",
		  "typeLabel": "Traffic sensor",
		  "targetWidgets": ["[]"],
		  "color1": "#ffdb4d",
		  "color2": "#fff5cc",
		  "pinattr": "square",
		  "pincolor": "#959595",
		  "symbolcolor": "undefined"
		  }]
		}</pre>

	Click on geographical Drill-Down button to send information of the Entities/Devices (as retrieved by	<pre>{
		  "event": "zoom",
		  "layers": {
		  "0": {
		  "distance": "0.7987",
		  "hasGeometry": false,
		  "multimedia": "",
		  "name": "METRO24",
		  "photoThumbs": [],</pre>

widgetMap	querying the Snap4City SmartCity API) currently displayed in the actual bounding-box	"serviceType": "TransferServiceAndRenting_Traffic_sensor",
		"serviceUri":
		"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO24",
		"tipo": "Traffic_sensor",
		"typeLabel": "Traffic sensor",
		"targetWidgets": ["[]"],
		"color1": "#ffdb4d",
		"color2": "#fff5cc",
		"pinattr": "square",
		"pincolor": "#959595",
		"symbolcolor": "undefined"
		},
		"1": {
		"distance": "0.9243",
		"hasGeometry": false,
		"multimedia": "",
		"name": "METRO759",
		"photoThumbs": [],
		"serviceType": "TransferServiceAndRenting_Traffic_sensor",
		"serviceUri":
		"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO759",
		"tipo": "Traffic_sensor",
		"typeLabel": "Traffic sensor",
		"targetWidgets": ["[]"],
		"color1": "#ffdb4d",
		"color2": "#fff5cc",
		"pinattr": "square",
		"pincolor": "#959595",
		"symbolcolor": "undefined"
		},
		"2": {
		"distance": "0.9603",
		"hasGeometry": false,
		"multimedia": "",
		"name": "METRO758",
		"photoThumbs": [],
		"serviceType": "TransferServiceAndRenting_Traffic_sensor",
		"serviceUri":
		"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO758",
		"tipo": "Traffic_sensor",
		"typeLabel": "Traffic sensor",
		"targetWidgets": ["[]"],
		"color1": "#ffdb4d",
		"color2": "#fff5cc",
		"pinattr": "square",
		"pincolor": "#959595",
		"symbolcolor": "undefined"
		},
		"3": {
		"distance": "1.0086",
		"hasGeometry": false,
		"multimedia": "",
		"name": "METRO950",
		"photoThumbs": [],
		"serviceType": "TransferServiceAndRenting_Traffic_sensor",
		"serviceUri":
		"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO950",
		"tipo": "Traffic_sensor",
		"typeLabel": "Traffic sensor",
		"targetWidgets": ["[]"],
		"color1": "#ffdb4d",
		"color2": "#fff5cc",
		"pinattr": "square",
		"pincolor": "#959595",
		"symbolcolor": "undefined"
		}
		},
		"bounds": {
		"_southWest": {
		"lat": 43.780126595782,
		"lng": 11.252961158752441
		},

widgetMap		

		<pre>"_northEast": {
		"lat": 43.78555645110887,
		"lng": 11.259355545043947
		}
		}
		}</pre>

widgetPieChart	Click on a circular sector that identifies the name of a certain metric to perform Drill-Down (sending information related to the selected metric for all the displayed Entities/Devices)	<pre>[
		{
		"value": "57",
		"metricType": "NO2",
		"metricName": " DISIT:orionUNIFI:ARPAT_QA_GR-SONNINO_SV",
		"measuredTime": "2023-03-04T23:59:00.000+01:00",
		“metricValueUnit”: "µg/m³"
		},
		. . .
		{
		"value": "59",
		"metricType": "NO2",
		"metricName": " DISIT:orionUNIFI:ARPAT_QA_FI-GRAMSCI_SV",
		"measuredTime": "2023-03-04T23:59:00.000+01:00",
		“metricValueUnit”: "µg/m³"
		}
		]</pre>
		N.B.: through the "metricName" value, which is represented in the following type: "<ORG>:<BROKER>:<DEVICE_ID>", it is possible to obtain the Service URI: http://www.disit.org/km4city/resource/iot/<BROKER>/<ORG>/<DEVICE_ID>

	Click on a circular sector that identifies an Entity/Device to perform Drill-Down (sending information related to all the metrics of the selected Entity/Device)	<pre>[
		{
		"value": "54.6",
		"metricType": "PM10",
		"metricName": "Helsinki:orionFinland:fmi-100742",
		"measuredTime": "2022-11-18T10:00:00.000+01:00"
		},
		. . .
		{
		"value": "14.3",
		"metricType": "PM2.5",
		"metricName": "Helsinki:orionFinland:fmi-100742",
		"measuredTime": "2022-11-18T10:00:00.000+01:00"
		}
		]</pre>
		N.B.: through the "metricName" value, represented as "<ORG>:<BROKER>:<DEVICE_ID>", it is possible to obtain the Service URI: http://www.disit.org/km4city/resource/iot/<BROKER>/<ORG>/<DEVICE_ID>

widgetBarSeries	Click on a single bar to perform Drill-Down (sending information of the selected metric and related Device/Entity)	<pre>{
		"event": "click",
		"value": {
		"value": 42,
		"metricType": "Temperature",
		"metricName": "BatteryGalaxyNote",
		"measuredTime": "2019-11-21T14:51:00Z",
		"metricValueUnit": "°C"
		}
		}</pre>

	Click on chart legend to perform Drill-Down (sending information of the Entities/Devices or metrics currently displayed in the chart legend, including their visibility status)	<pre>{
		"event": "legendItemClick",
		"layers": [{
		"name": "DISIT:orionUNIFI:ARPAT_QA_GR-SONNINO_SV",
		"visible": false
		}, {
		"name": "DISIT:orionUNIFI:ARPAT_QA_FI-GRAMSCI_SV",
		"visible": true
		}],
		"metrics": ["NO2"]
		}</pre>

widgetRadarSeries	Click on a single radar metric to perform Drill-Down (sending information of the selected metric and related Device/Entity)	<pre>{
		"event": "click",
		"value": {
		"metricType": "avgTime",
		"metricName": "DISIT:orionUNIFI:METRO24"
		}
		}</pre>

	Click on chart legend to perform Drill-Down (sending information of the Entities/Devices or metrics currently displayed in the chart legend, including their visibility status)	<pre>{
		"event": "legendItemClick",
		"layers": [{
		"name": "DISIT:orionUNIFI:METRO24",
		"visible": true
		}, {
		"name": "DISIT:orionUNIFI:METRO759",
		"visible": false
		}, {
		"name": "DISIT:orionUNIFI:METRO758",
		"visible": true
		}],
		"metrics": ["anomalyLevel", "averageSpeed", "avgTime", "concentration", "congestionLevel", "vehicleFlow"]
		}</pre>

widgetCurvedLineSeries	Zoom on a desired time interval to perform temporal Drill-Down on selected time range.
	It is also possible to synchronize the time alignment of multiple different widgetCurvedLineSeries
		<pre>{
		"series": <SERIES_OBJECT>
		"t1" : <PRIMO_TIMESTAMP_MILLIS>,
		"t2" : <SECONDO_TIMESTAMP_MILLIS>,
		"series": <SERVICE_URI>,
		"metricName": <VALUE_NAME>
		}</pre>
		Example:
		<pre>{
		"t1": 1673212579086.2698,
		"t2": 1673366990478.7363,
		"series": [{
		"metricId": "https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1&format=json",
		"metricHighLevelType": "IoT Device Variable",
		"metricName": "DISIT:orionUNIFI:METRO1",
		"smField": "vehicleFlow",
		"serviceUri": "http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1"</pre>

	Click on a single point of the chart to perform temporal Drill down on a single time instant	}, {
		"metricId":
		"https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1028&format=json",
		"metricHighLevelType": "IoT Device Variable",
		"metricName": "DISIT:orionUNIFI:METRO1028",
		"smField": "vehicleFlow",
		"serviceUri":
		"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1028"
		}]
		}
		
		in case of Drill-Down on a single time instant t1 = t2.

widgetDeviceTable	Click on Action buttons to send information related to selected Entity/Device	{
		<LIST_OF_OBJECT>
		}
		
		Example:
		{
		"anomalyLevel":110.16865,
		"averageSpeed":86.4935,
		"avgDistance":"",
		"avgTime":13.1625,
		"concentration":1.671046,
		"congestionLevel":101.25,
		"dateObserved":"2023-01-16T09:36:00.000Z",
		"occupancy":"",
		"speedPercentile":"",
		"thresholdPerc":"",
		"vehicleFlow":289.06924,
		"device":"METRO54",
		"serviceUri":"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO54",
		"action":"pin"
		}

widgetEventTable	Click on Action buttons to send information related to selected Event	Example:
		{
		"device":"Alarm001",
		
		"prefix":"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/",
		"ordering":"startDate",
		"action":"pin"
		}

5.2. Table of widget actuator

The following table is a summary of the actions/features for trigger widgets.

Widget name	Action/ Functionality	Sended Data/Parameters
widgetImpulseButton	Action on click	
widgetOnOffButton	Action on click	(string) status;
widgetKnob	Action on click	(float) value;
widgetNumericKeyboard	Action on click	(float) value;

6. Full Custom CSBL: Details for widget types

In this section, specifications on how to implement the JavaScript execute() function for all the available widgets are provided. The JavaScript execute() function allows to pilot target widgets, sending them actions, data etc.

6.1 widgetRadarSeries (IN/OUT)

6.1.1 widgetRadarSeries as Reading (IN) widget

First of all, an existing widgetRadar must be identified in the dashboard, of which the id <TARGET_WIDGET_NAME> must be noted.
The JavaScript function to be inserted in the appropriate CK Editor box (in more options) of another widget of the same dashboard (for example, a button widgetImpulseButton) in order to pilot the <TARGET_WIDGET_NAME> widgetRadar is of the following type:

```
function execute() {
	$('body').trigger({
		type: " showRadarSeriesFromExternalContent_<TARGET_WIDGET_NAME>",
		eventGenerator: $(this),
		targetWidget: <TARGET_WIDGET_NAME>,
		passedData:
[{"metricId":"https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/IT0952A1&format=json","metricHighLevelType":"Sensor","metricName":"DISIT:orionUNIFI:IT0952A1","metricType":"O3_"},{"metricId":"https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/IT0952A1&format=json","metricHighLevelType":"Sensor","metricName":"DISIT:orionUNIFI:IT0952A1","metricType":"NO2"},{"metricId":"https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/IT0957A1&format=json","metricHighLevelType":"Sensor","metricName":"DISIT:orionUNIFI:IT0957A1","metricType":"O3_"}
]
	});
}
```

6.1.2 widgetRadarSeries as Writing widget: Click on widgetRadarSeries element

When clicking on an element of the widgetRadarSeries, a JSON object called param is passed to the execute() function set in the CK Editor, in which there are the type of event (in this case "click") and all the elements present inside the widget in the layers field. The following example shows how to send the information to a multi-series widget (widgetCurvedLineSeries) related to a selected Entity/Device metric which has been clicked on the Radar Series chart:

```
function execute() {
    var e = JSON.parse(param);
    console.log(e);
```

```
    if (e.event == "click") {
       let serviceUri =
"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/" +
e.value.metricName.slice(17, e.value.metricName.length);
       var data = [];
       data[0] = {};
       data[0].metricId=
"http://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://www.disit.org/k
m4city/resource/iot/orionUNIFI/DISIT/" + e.value.metricName.slice(17,
e.value.metricName.length) + "&format=json";
       data[0].metricHighLevelType = "Sensor";
       data[0].metricName = "DISIT:orionUNIFI:" + e.value.metricName.slice(17,
e.value.metricName.length);
       data[0].smField = e.value.metricType;
       data[0].serviceUri = serviceUri;
       $('body').trigger({
          type:
"showCurvedLinesFromExternalContent_w_AggregationSeries_3721_widgetCurvedLineSerie
s35547",
          eventGenerator: $(this),
          targetWidget: "w_AggregationSeries_3721_widgetCurvedLineSeries35547",
          range: "7/DAY",
          field: data.smField,
          passedData: data
       });
    }
}
```

6.1.3 widgetRadarSeries as Writing widget: Click on Legend Items

When clicking on an element of the widgetRadarSeries legend, a JSON object called param is passed to the execute() function set in the CK Editor, in which there are the type of event (in this case "legendItemClick"), all the elements present inside the widget in the layers field with inside even if they are visible or not (corresponding to whether or not they are selected in the legend) and finally all the metrics present.
Below is an example of how to send the sensors visible in the legend of the command widget to a Radar Series when clicking on a legend item of it:

```
function execute(){
   var e = JSON.parse(param);
   if(e.event == "legendItemClick"){
      var date = [];
      let name, h = 0;
      for (var l in e.layers) {
         if(e.layers[l].visible == true){
            name = e.layers[l].name.slice(17,e.layers[l].name.length);
            for(var m in e.metrics){
               date[h] = {};
```

```
                data[h].metricId =
"https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://www.disit.org/
km4city/resource/iot/orionUNIFI/DISIT/"+name ;
                data[h].metricHighLevelType = "Sensor";
                data[h].metricName = e.layers[l].name;
                data[h].metricType = e.metrics[m];
                h++;
             }
          }
       }
       $('body').trigger({
          type:
"showRadarSeriesFromExternalContent_w_AggregationSeries_1_widgetRadarSeries49",
          eventGenerator: $(this),
          targetWidget: "w_AggregationSeries_1_widgetRadarSeries49",
          passedData: data
       });
    }
}
```

6.1.4 widgetRadarSeries Time Selection

The Radar Series widget also allows you to send a date and time parameter as input to another widget to insert in the calendar. In this way it is possible to synchronize more than one widget at the same moment in time.

For example, if in a dashboard there are two RadarSeries widgets with different data and I want them to always show data relating to the same moment in time, I can activate this feature so that when I use the calendar in one of the two, the other is also changed.

This feature is activated when you edit the calendar. The following parameters must be set in the execute() function of the widget writer:

```
event: "set_time"
```

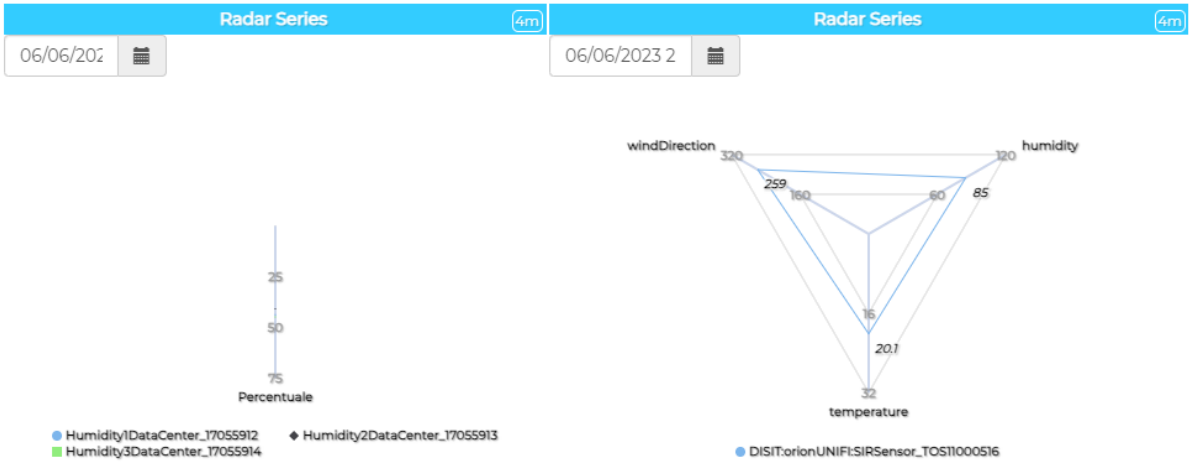
It is an important parameter because it is used to activate the synchronization function. It doesn't work without it.

```
datetime: param
```

This parameter must receive the date of the datetime automatically from the selection of the Calendar.

```
passedData: []
```

Usually when writing the Radar Series widget, it is possible to send a list of elements to the target widget, in the case of the selectionTime this parameter can be represented as an empty array. However, it must be present for the feature to work properly.




```
function execute() {
    $('body').trigger({
        type:
"showRadarSeriesFromExternalContent_w_AggregationSeries_3865_widgetRadarSeries3714
1",
        targetWidget: "w_AggregationSeries_3865_widgetRadarSeries37141",
        widgetTitle: "RadarWidget",
        event: "set_time",
        datetime: param,
        passedData: []
    });
}
```

6.2 widgetTable (IN)

First of all, an existing widgetTable must be identified in the dashboard, of which the id <TARGET_WIDGET_NAME> must be noted.
The JavaScript function to be inserted in the appropriate CK Editor box (in more options) of another widget of the same dashboard, in order to pilot the <TARGET_WIDGET_NAME> widgetTable is of the following type:

```
function execute() {
      $('body').trigger({
            type: "showTableFromExternalContent_<TARGET_WIDGET_NAME>",
            eventGenerator: $(this),
            targetWidget: <TARGET_WIDGET_NAME>,
            passedData: [  {
                "metricId":
            "https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://w
            ww.disit.org/km4city/resource/iot/orionUNIFI/DISIT/IT0952A1&format=js
            on",
                "metricHighLevelType": "Sensor",
                "metricName": "DISIT:orionUNIFI:IT0952A1",
                "metricType": "O3_"
              },  {
```

```
                "metricId":
            "https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://w
            ww.disit.org/km4city/resource/iot/orionUNIFI/DISIT/IT0952A1&format=js
            on",
                "metricHighLevelType": "Sensor",
                "metricName": "DISIT:orionUNIFI:IT0952A1",
                "metricType": "NO2"
              },  {
                "metricId":
            "https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://w
            ww.disit.org/km4city/resource/iot/orionUNIFI/DISIT/IT0828A1&format=js
            on",
                "metricHighLevelType": "Sensor",
                "metricName": "DISIT:orionUNIFI:IT0828A1",
                "metricType": "NO2"
        }
    ]}
);}
```

6.3 widgetSingleContent, widgetSpeedometer and widgetGaugeChart (IN)

First of all, an existing widgetSingleContent, widgetSpeedometer or widgetGaugeChart must be identified in the dashboard, of which the id <TARGET_WIDGET_NAME> must be noted. The JavaScript function to be inserted in the appropriate CK Editor box (in more options) of another widget of the same dashboard, in order to pilot the <TARGET_WIDGET_NAME> is of the following type:

```
function execute() {
      $('body').trigger({
            type: "showLastDataFromExternalContentGis_<TARGET_WIDGET_NAME>"
            eventGenerator: $(this),
            targetWidget: <TARGET_WIDGET_NAME>,
            color1: "#acb2fa",
            color2: "#231d5c",
            widgetTitle: "Occupied Parking Lots (Alberti Car Park)",
            field: "occupiedParkingLots",
            serviceUri:
"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/CarParkPal.Giustizia"
      });
}
```

In the above example, the parameters color1, color2 and widgetTitle are required in order to set the style of the target widget.

6.3.1 (Alternative) widgetSingleContent with structured data

First of all, an existing widgetSingleContent must be identified in the dashboard, of which the id <TARGET_WIDGET_NAME> must be noted.

The JavaScript function to be inserted in the appropriate CK Editor box (in more options) of another widget of the same dashboard, in order to pilot the <TARGET_WIDGET_NAME> widgetSingleContent is of the following type:

```
function execute() {
	$('body').trigger({
		type: "showSingleContentFromExternalContent_<TARGET_WIDGET_NAME>"
		eventGenerator: $(this),
		targetWidget: <TARGET_WIDGET_NAME>,
		color1: "#acb2fa",
		color2: "#231d5c",
		widgetTitle: "Occupied Parking Lots (Alberti Car Park)",
		passedData: { 'dataOperation': param}
	});
}
```

The param variable is made up of the input value generated by the current widget's state change.

6.4 widgetTimeTrend (IN/OUT)

6.4.1 widgetTimeTrend as Reading widget

First of all, an existing TimeTrend widget must be identified in the dashboard, of which the id <TARGET_WIDGET_NAME> must be noted.
The JavaScript function to be inserted in the appropriate CK Editor box (in more options) of another widget of the same dashboard, in order to pilot the <TARGET_WIDGET_NAME> widgetTimeTrend is of the following type:

```
function execute() {
	$('body').trigger({
		type: "showTimeTrendFromExternalContentGis_<TARGET_WIDGET_NAME>",
		eventGenerator: $(this),
		targetWidget: <TARGET_WIDGET_NAME>,
		range: "7/DAY",
		color1: "#9b93ed",
		color2: "#231d5c",
		widgetTitle: "Occupied Parking Lots (Alberti Car Park) from Impulse
Button",
		field: "occupiedParkingLots",
		serviceUri:
"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/CarParkPal.Giustizia",
	});
}
```

6.4.2 widgetTimeTrend as Writing widget: Click on widgetTimeTrend chart point or zoom on desired time interval

When clicking on chart point of the widgetTimeTrend, as well as selecting a desired time interval, a JSON object called param is passed to the execute() function set in the CK Editor, in which there are the SURI, the metric name and the starting and ending timestamp of the selected time interval (when clicking on a single chart point, the two timestamps are the same). In the following, a more advanced example is provided (the same illustrated in the Section 7.1), showing how to retrieve data in the selected time interval of a specific Device (by performing an ajax call to Snap4City SmartCity API), handling the results to calculate the mean value for each device's metric, and finally send the results to be shown in a widgetBarSeries:

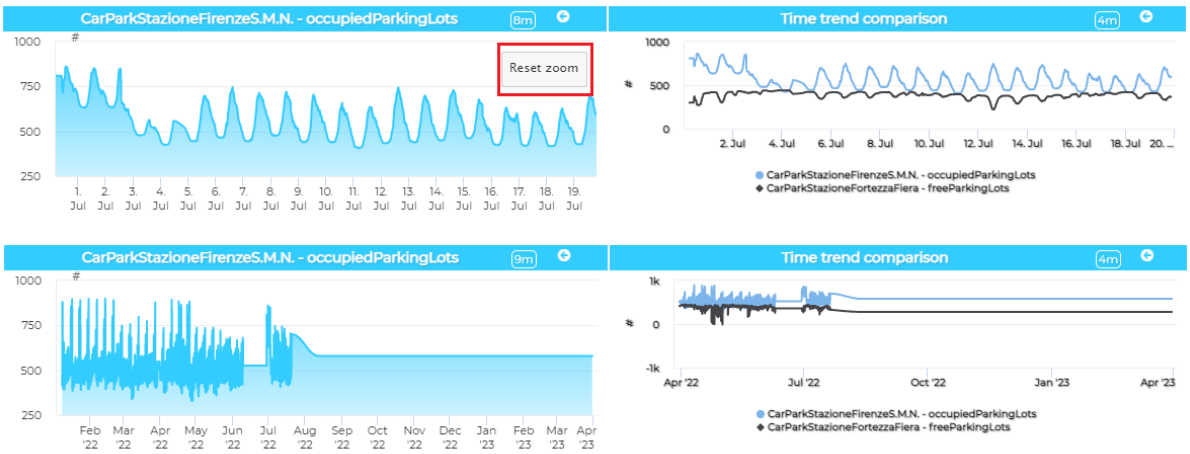
```
function execute() {
    let minT, maxT = null;
    if (param['t1'] != param['t2']) {
        minT = param['t1'];
        maxT = param['t2'];
    } else {
        minT = param['t1'] - 10000000;
        maxT = param['t2'] + 10000000;
    }
    let dt1 = new Date(minT);
    let dt1_iso = dt1.toISOString().split(".")[0];
    let dt2 = new Date(maxT);
    let dt2_iso = dt2.toISOString().split(".")[0];

    function getMean(originalData) {
        var singleOriginalData, singleData, convertedDate = null;
        var convertedData = {
            data: []
        };
        var originalDataWithNoTime = 0;
        var originalDataNotNumeric = 0;
        var meanDataObj = {};
        if (originalData.hasOwnProperty("realtime")) {
            if (originalData.realtime.hasOwnProperty("results")) {
                if (originalData.realtime.results.hasOwnProperty("bindings")) {
                    if (originalData.realtime.results.bindings.length > 0) {
                        let propertyJson = "";
                        if (originalData.hasOwnProperty("BusStop")) {
                            propertyJson = originalData.BusStop;
                        } else {
                            if (originalData.hasOwnProperty("Sensor")) {
                                propertyJson = originalData.Sensor;
                            } else {
                                if (originalData.hasOwnProperty("Service")) {
                                    propertyJson = originalData.Service;
                                } else {
                                    propertyJson = originalData.Services;
```

```
                                }
                            }
                        }
                        for (var j = 0; j <
originalData.realtime.head.vars.length; j++) {
                            var singleObj = {}
                            var field = originalData.realtime.head.vars[j];
                            var numericCount = 0;
                            var sum = 0;
                            var mean = 0;
                            if (field == "updating" || field == "measuredTime" ||
field == "instantTime" || field == "dateObserved") {
                                // convertedDate =
singleOriginalData.updating.value;
                                continue;
                            }
                            for (var i = 0; i <
originalData.realtime.results.bindings.length; i++) {
                                singleOriginalData =
originalData.realtime.results.bindings[i];
                                if (singleOriginalData[field] !== undefined) {
                                    if
(!isNaN(parseFloat(singleOriginalData[field].value))) {
                                        numericCount++;
                                        sum = sum +
parseFloat(singleOriginalData[field].value);
                                    }
                                }
                            }
                            mean = sum / numericCount;
                            meanDataObj[field] = mean;
                        }
                        return meanDataObj;
                    } else {
                        return false;
                    }
                } else {
                    return false;
                }
            } else {
                return false;
            }
        } else {
            return false;
        }
    }

    function buildDynamicData(data, name) {
        var passedJson = [];
        for (const item in data) {
```

```
            var singleJson = {};
            singleJson["metricId"] = "";
            singleJson["metricHighLevelType"] = "Dynamic";
            singleJson["metricName"] = name;
            singleJson["metricType"] = item;
            singleJson["metricValueUnit"] = "";
            singleJson["value"] = data[item];
            passedJson.push(singleJson)
        }
        return passedJson;
    }
    $.ajax({
        url: "../controllers/superservicemapProxy.php/api/v1/?serviceUri=" +
encodeServiceUri(param['sUri']) + "&fromTime=" + dt1_iso + "&toTime=" + dt2_iso,
        // url: "../controllers/superservicemapProxy.php/api/v1/?serviceUri=" +
encodeServiceUri(sUri) + "&fromTime=" + dt1_iso + "&toTime=" + dt2_iso +
"&valueName=" + param['metricName'],
        type: "GET",
        data: {},
        async: true,
        dataType: 'json',
        success: function(data) {
            if (data.realtime.results) {
                var meanData = getMean(data);
                var passedJson = buildDynamicData(meanData,
data.Service.features[0].properties.name);

                $('body').trigger({
                    type:
"showBarSeriesFromExternalContent_w_AggregationSeries_3664_widgetBarSeries35029",
                    eventGenerator: $(this),
                    targetWidget: "w_AggregationSeries_3664_widgetBarSeries35029",
                    color1: "#f22011",
                    color2: "#9c6b17",
                    widgetTitle: "Air Quality IT from Impulse Button",
                    passedData: passedJson
                });
            } else {
$('#w_AggregationSeries_3664_widgetBarSeries35029_chartContainer').hide();
$('#w_AggregationSeries_3664_widgetBarSeries35029_noDataAlert').show();
            }
        },
        error: function(data) {
            console.log("Errore in scaricamento dati da Service Map");
            console.log(JSON.stringify(data));
        }
    });
}
```

6.4.3 widgetTimeTrend reset zoom

In addition, to command a zoom to one or more widgets, widgetTimeTrend can also reset the zoom of these widgets, that are time series and curved lines.



By clicking on the "Reset zoom" button, you can send a command to the other widgets to reset the magnification of the other widgets.
This operation is performed using the execute() function present in the CkEditor by sending it parameters present in the following code:

```
{
    "event": 'reset zoom',
    "t1": 1654434511034.4827,
    "t2": 1658936003172.4138,
    "sUri": "http://www.disit.org/km4city/resource/CarParkStazioneFirenzeS.M.N.",
    "metricName": "occupiedParkingLots"
}
```

The user can then use these parameters to decide how to handle the zoom reset in the various widgets. By default this operation acts on widgets in which a zoom event has been defined in the ckeditor code.

```
function execute(){
    let dt1 = new Date(param['t1']);
    let dt1_iso = dt1.toISOString().split(".")[0];
    let dt2 = new Date(param['t2']);
    let dt2_iso = dt2.toISOString().split(".")[0];
    $('body').trigger({
        type:
"showCurvedLinesFromExternalContent_w_AggregationSeries_3545_widgetCurvedLineSerie
s35689",
        eventGenerator: $(this),
        targetWidget: "w_AggregationSeries_3545_widgetCurvedLineSeries35689",
        range: "7/DAY",
        color1: "#9b93ed",
        color2: "#231d5c",
        widgetTitle: "Example",
```

```
        t1: dt1_iso,
        t2: dt2_iso,
        event:"zoom"
    });
}
```

For example, in this case in the ckeditor JavaScript code the target widget is "*w_AggregationSeries_3545_widgetCurvedLineSeries35689*", consequently zooming and resetting the zoom will only affect this widget.

6.5 widgetCurvedLineSeries (IN/OUT)

6.5.1 widgetCurvedLineSeries as Reading widget

First of all, an existing widgetCurvedLineSeries must be identified in the dashboard, of which the id <TARGET_WIDGET_NAME> must be noted.
The JavaScript function to be inserted in the appropriate CK Editor box (in more options) of another widget of the same dashboard, in order to pilot the <TARGET_WIDGET_NAME> widgetCurvedLineSeries is of the following type:

```
function execute() {
      $('body').trigger({
                  type:
"showCurvedLinesFromExternalContent_<TARGET_WIDGET_NAME>",
                  eventGenerator: $(this),
                  targetWidget: <TARGET_WIDGET_NAME>,
                  range: "7/DAY",
                  field: "vehicleFlow",
                  passedData:
[{"metricId":"https:\/\/servicemap.disit.org\/WebAppGrafo\/api\/v1\/?serviceUri=ht
tp:\/\/www.disit.org\/km4city\/resource\/iot\/orionUNIFI\/DISIT\/METRO658&format=j
son","metricHighLevelType":"Sensor","metricName":"METRO658","smField":"vehicleFlow
","serviceUri":"http:\/\/www.disit.org\/km4city\/resource\/iot\/orionUNIFI\/DISIT\
/METRO658"}]
            });
}
```

The passedData field can be:

- An array of JSON for the CurvedLineSeries widget, as described in https://www.snap4city.org/drupal/node/575

E.g.:

```
[{
      "metricId": <METRIC_IDENTIFIER_OR_API_REF>,
      "metricHighLevelType": <HIGH_LELVE_CLASS_CATEGORY>,
      "metricName": <METRIC_NAME>,
      "metricType": <METRIC_TYPE>,
      "serviceUri": <IOT_DEVICE_SERVICE_URI>,
```

```
},
. . .
{
       "metricId": <METRIC_IDENTIFIER>,
       "metricHighLevelType": <HIGH_LELVE_CLASS_CATEGORY>,
       "metricName": <METRIC_NAME>,
       "metricType": <METRIC_TYPE>,
       "metricValueUnit": <UNIT_OF_MEASURE>,
       "meausuredTime": <DATE_TIME_OF_MEASUREMENT>,
       "value": [<ARRAY_OF_VALUES>]
}]
```

An Example:

```
[{
       "metricId":
"https:\/\/servicemap.disit.org\/WebAppGrafo\/api\/v1\/?serviceUri=http:\/\/www.di
sit.org\/km4city\/resource\/iot\/orionUNIFI\/DISIT\/METRO18&format=json",
       "metricHighLevelType": "Sensor",
       "metricName": "METRO18",
       "smField": "vehicleFlow",
       "serviceUri":
"http:\/\/www.disit.org\/km4city\/resource\/iot\/orionUNIFI\/DISIT\/METRO18"
},
{
        "metricId": "",
        "metricHighLevelType": "Dynamic",
        "metricName": "BatteryTemperatureGalaxyNote",
        "smField": "Gradi Centigradi",
        "metricValueUnit": "°C",
        "values": [
                    [now-64*base, 19.5],
                    [now-60*base, 20.0],
                    [now-56*base, 20.5],
                    [now-52*base, 18.5],
                    [now-48*base, 19],
                    [now-44*base, 18.5],
                    [now-40*base, 21.5],
                    [now-36*base, 22.0],
                    [now-32*base, 19],
                    [now-28*base, 17.5],
                    [now-24*base, 16.5],
                    [now-20*base, 17.0],
                    [now-16*base, 18.5],
                    [now-12*base, 20.0],
                    [now-8*base, 19.5],
                    [now-4*base, 21.5],
                    [now-1*base, 21]
        ]
    }]
```

6.5.2 widgetCurvedLineSeries as Writing widget: Zoom alignment on WidgetCurvedLineSeries

Through a zoom event on a Curved Lines it is possible to send the command to align other Curved Lines, as in the following example:

```
$('body').trigger({
        type: "showCurvedLinesFromExternalContent_w_AggregationSeries_1_widgetCurvedLineSeries70",
        eventGenerator: $(this),
        targetWidget: "w_AggregationSeries_1_widgetCurvedLineSeries70",
        range: "7/DAY",
        field: data.smField,
        passedData: date,
        t1: dt1_iso,
        t2: dt2_iso,
        event:"zoom"
});
```

Note that it is necessary to specify the start and end time of the zoom (found in param) and the characterizing event "zoom", in this case the passedData, event and fields parameters are not necessary since the receiving Curved Lines will maintain the same metric present before the zoom.

6.5.3 widgetCurvedLineSeries as Writing widget: Temporal Drill-Down by Zoom on WidgetCurvedLineSeries

At the time zoom event on a curve lines, the execute function is executed inside the relative ckeditor in which a JSON object called param is passed in which there are start and end times and an array that presents all the metrics that the widget is showing, within each element there are the characteristics of the device and the identified metric. Param has the following form:

```
{
     "t1":1677060516253.012,
     "t2":1677063881726.334,
     "series":[{
       "metricId":"https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1&format=json",
       "metricHighLevelType":"IoT Device Variable",
       "metricName":"DISIT:orionUNIFI:METRO1",
       "smField":"concentration",
       "serviceUri":"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1"
      }]
}
```

The following example shows how to use this input parameter to calculate the average of the metrics present in the device and send this data to a Bar Series to be displayed:

```
function execute() {
```

```
var newParam = {};
newParam.t1 = param['t1'];
newParam.t2 = param['t2'];
newParam.metricName = param.series[0].smField;
newParam.sUri = param.series[0].serviceUri;

param = newParam;

let minT, maxT = null;
if (param['t1'] != param['t2']) {
    minT = param['t1'];
    maxT = param['t2'];
}
else {
    minT = param['t1']-10000000;
    maxT = param['t2']+10000000;
}
let dt1 = new Date(minT);
let dt1_iso = dt1.toISOString().split(".")[0];
let dt2 = new Date(maxT);
let dt2_iso = dt2.toISOString().split(".")[0];

function getMean(originalData){
    var singleOriginalData, singleData, convertedDate = null;
    var convertedData = {
        data: []
    };
    var originalDataWithNoTime = 0;
    var originalDataNotNumeric = 0;
    var meanDataObj = {};
    if(originalData.hasOwnProperty("realtime")){
        if(originalData.realtime.hasOwnProperty("results")){
            if(originalData.realtime.results.hasOwnProperty("bindings")){
                if(originalData.realtime.results.bindings.length > 0){
                    let propertyJson = "";
                    if(originalData.hasOwnProperty("BusStop")){
                        propertyJson = originalData.BusStop;
                    }
                    else{
                        if(originalData.hasOwnProperty("Sensor")){
                            propertyJson = originalData.Sensor;
                        }
            else{
              if(originalData.hasOwnProperty("Service")){
                propertyJson = originalData.Service;
              }
              else{
                propertyJson = originalData.Services;
              }
            }
```

```
              }
           for(var j = 0; j < originalData.realtime.head.vars.length; j++){
             var singleObj = {}
             var field = originalData.realtime.head.vars[j];
             var numericCount = 0;
             var sum = 0;
             var mean = 0;
             if (field == "updating" || field == "measuredTime" || field ==
"instantTime" || field == "dateObserved") {
               continue;
             }
          for (var i = 0; i < originalData.realtime.results.bindings.length; i++) {
               singleOriginalData = originalData.realtime.results.bindings[i];
               if (singleOriginalData[field] !== undefined) {
                 if (!isNaN(parseFloat(singleOriginalData[field].value))) {
                   numericCount++;
                   sum = sum + parseFloat(singleOriginalData[field].value);
                 }
               }

             }
             mean = sum / numericCount;
             meanDataObj[field] = mean;

           }
           return meanDataObj;
         } else {
           return false;
         }
       } else {
         return false;
       }
     } else {
       return false;
     }
   } else {
     return false;
   }
}

function buildDynamicData(data, measuredTime) {
  var passedJson = [];
  for (const item in data) {
    var singleJson = {};
    singleJson["metricId"] = "";
    singleJson["metricHighLevelType"] = "Dynamic";
    singleJson["metricName"] = name;
    singleJson["metricType"] = item;
    singleJson["metricValueUnit"] = "";
    singleJson["measuredTime"] = measuredTime;
```

```
    singleJson["value"] = data[item];
    passedJson.push(singleJson)
  }
  return passedJson;
}

$.ajax({
  url: "../controllers/superservicemapProxy.php/api/v1/?serviceUri=" +
encodeServiceUri(param['sUri']) + "&fromTime=" + dt1_iso + "&toTime=" + dt2_iso,
  // url: "../controllers/superservicemapProxy.php/api/v1/?serviceUri=" +
encodeServiceUri(sUri) + "&fromTime=" + dt1_iso + "&toTime=" + dt2_iso +
"&valueName=" + param['metricName'],
  type: "GET",
  data: {},
  async: true,
  dataType: 'json',
  success: function(data){
    if (data.realtime.results) {
      var meanData = getMean(data);
      var passedJson = buildDynamicData(meanData,
data.Service.features[0].properties.name,
data.realtime.results.bindings[0].measuredTime );


      $('body').trigger({
            type:
"showBarSeriesFromExternalContent_w_AggregationSeries_1_widgetBarSeries48",
            eventGenerator: $(this),
            targetWidget: "w_AggregationSeries_1_widgetBarSeries48",
            passedData: passedJson
      });
    } else {
      $('#w_AggregationSeries_1_widgetBarSeries48_chartContainer').hide();
      $('#w_AggregationSeries_1_widgetBarSeries48_noDataAlert').show();
    }

  },
  error: function (data){
    console.log("Errore in scaricamento dati da Service Map");
    console.log(JSON.stringify(data));
  }
});
}
```

6.5.4 widgetCurvedLines providing status from legend

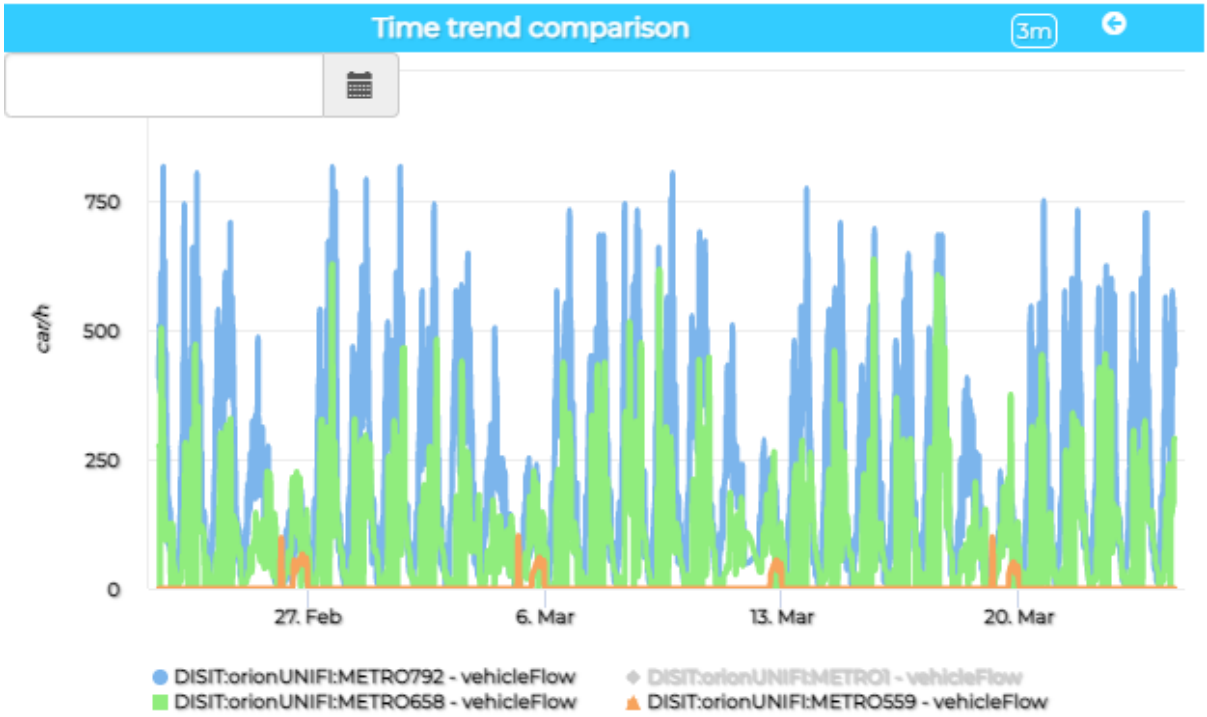
widgetCurvedLines also allows you to output data relating to the active or deactivated status of the metrics to another widget by clicking on the legend of the graph.

This is done by placing a JavaScript execute function in the CKEditor code. As in the example below.

```
function execute(){
    $('body').trigger({
        type:
"showSingleContentFromExternalContent_w_DISIT_orionUNIFI_METRO1_1536_widgetSingleC
ontent9999",
        targetWidget: "w_DISIT_orionUNIFI_METRO1_1536_widgetSingleContent9999",
        widgetTitle: "Show legend status",

passedData: { 'dataOperation': param}

    });
}
```

"param" in this case is a fixed variable which is used to send the data from the graph, it is not customizable by the user.

To activate this function, click on a label in the graph legend.



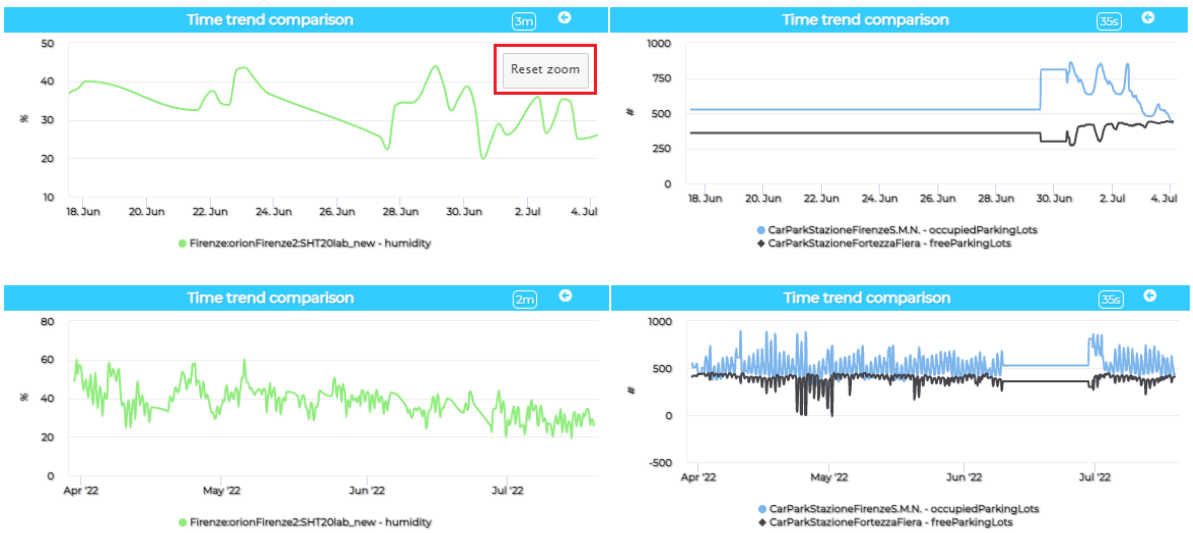
The Curved Lines widget will send as output a json with a list of metrics associated with the visible status true/false and a list of temporal data visible in the graph.

```
{ "event":"legendItemClick",
   "layers":[ {
         "name":"DISIT:orionUNIFI:METRO792 - vehicleFlow",
         "visible":true
      }, {
```

```
            "name":"DISIT:orionUNIFI:METRO1 - vehicleFlow",
            "visible":false
        }, {
            "name":"DISIT:orionUNIFI:METRO658 - vehicleFlow",
            "visible":true
        },{
            "name":"DISIT:orionUNIFI:METRO559 - vehicleFlow",
            "visible":true
        }],
"metrics":[1677069000000,1677069900000,1677070800000,1677078900000,1677079800000,
1677080700000,1677081600000, 1677082500000, 1677083400000, 1677084300000,
1677085200000,1677086100000,1677087000000,1677090600000, 1677091500000,
1677092400000,1677093300000,1677094200000,1677095100000]}
```


6.5.5 widgetCurvedLines reset zoom

In addition to commanding a zoom to one or more widgets (5.5.2), widgetCurvedLines can also reset the zoom of these widgets.



By clicking on the "Reset zoom" button, you can send a command to the other widgets to reset the magnification of the other widgets.

This operation is performed using the *execute()* function present in the CkEditor by sending parameters present in the following code.


```
{
   "event":"reset zoom",
   "t1":null,
   "t2":null,
   "series":[
      {
"metricId":"https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://www
.disit.org/km4city/resource/CarParkStazioneFirenzeS.M.N.&format=json",
         "metricHighLevelType":"Sensor",
```

```
        "metricName":"CarParkStazioneFirenzeS.M.N.",
        "smField":"occupiedParkingLots",
"serviceUri":"http://www.disit.org/km4city/resource/CarParkStazioneFirenzeS.M.N."
      }
   ]
}
```

The structure of the "reset zoom" message is similar to that of zoom, however the *t1* and *t2* keys are set to null to cancel filtering on widget target dates.

The user can then use these parameters to decide how to handle the zoom reset in the various widgets. By default this operation acts on target widgets in which a zoom event has been defined in the ckeditor code.

```
function execute(){
    let dt1 = new Date(param['t1']);
    let dt1_iso = dt1.toISOString().split(".")[0];
    let dt2 = new Date(param['t2']);
    let dt2_iso = dt2.toISOString().split(".")[0];
    $('body').trigger({
        type:
"showCurvedLinesFromExternalContent_w_AggregationSeries_3545_widgetCurvedLineSerie
s35689",
        eventGenerator: $(this),
        targetWidget: "w_AggregationSeries_3545_widgetCurvedLineSeries35689",
        range: "7/DAY",
        color1: "#9b93ed",
        color2: "#231d5c",
        widgetTitle: "Example",
        t1: dt1_iso,
        t2: dt2_iso,
        event:param['event'];
    });
}
```

For example, in this case in the ckeditor JavaScript code the target widget is "*w_AggregationSeries_3545_widgetCurvedLineSeries35689*", consequently zooming and resetting the zoom will only affect this widget.

6.5.6 WidgetCurvedLine Time Selection

Widget CurvedLine also allows you to send a date and time parameter as input to another widget to insert in the calendar. In this way it is possible to synchronize more than one widget at the same moment in time.

For example, if in a dashboard there are two CurvedLine widgets with different data and I want them to always show data relating to the same moment in time, I can activate this feature so that when I use the calendar in one of the two, the other is also changed.

This feature is activated when you edit the calendar. The following parameters must be set in the execute() function of the widget writer:

```
event: "set_time"
```

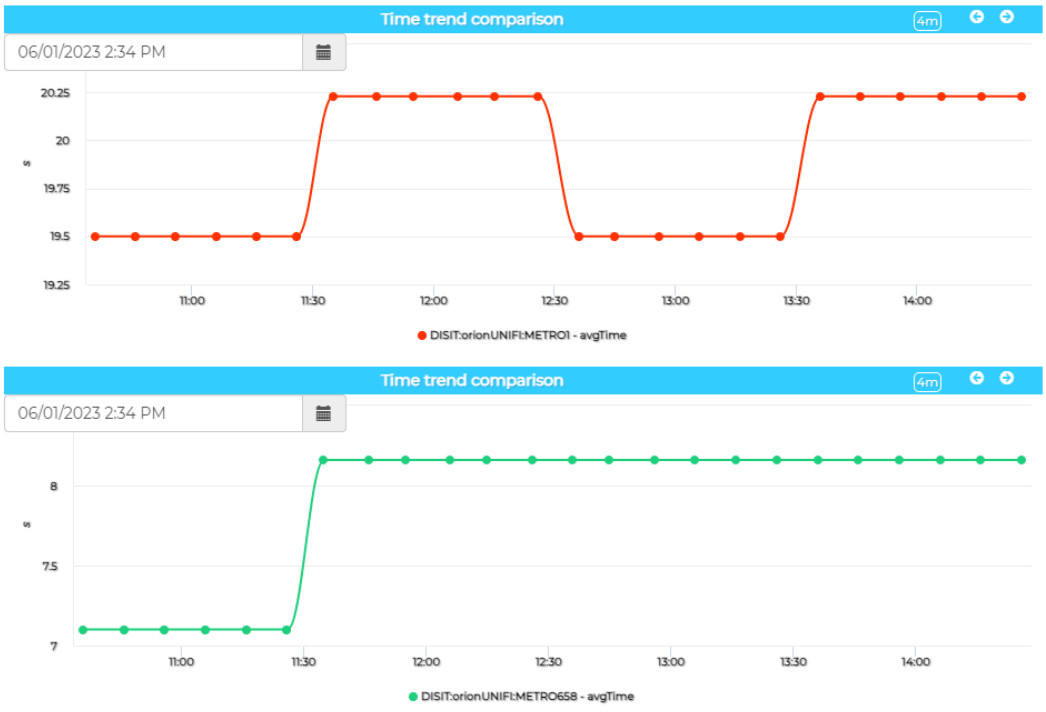
It is an important parameter because it is used to activate the synchronization function. It doesn't work without it.

```
datetime: param
```

This parameter must receive the date of the datetime automatically from the selection of the Calendar.

```
passedData: []
```

Usually when writing the CurvedLine widget it is possible to send a list of elements to the target widget, in the case of the selectionTime this parameter can be represented as an empty array. However, it must be present for the feature to work properly.




```
function execute() {
    $('body').trigger({
type:
"showCurvedLinesFromExternalContent_w_AggregationSeries_1542_widgetCurvedLineSerie
s10065",
        targetWidget: "w_AggregationSeries_1542_widgetCurvedLineSeries10065",
        widgetTitle: "Curved Lined Report",
```

```
        event: "set_time",
        datetime: param,
        passedData: []
    });
}
```

6.6 widgetDeviceTable

First of all, an existing widgetDeviceTable must be identified in the dashboard, of which the id <TARGET_WIDGET_NAME> must be noted.
The JavaScript function to be inserted in the appropriate CK Editor box (in more options) of another widget of the same dashboard, in order to pilot the <TARGET_WIDGET_NAME> widgetDeviceTable is of the following type:

```
function execute() {
      $('body').trigger({
            type: "showDeviceTableFromExternalContent_<TARGET_WIDGET_NAME>",
            eventGenerator: $(this),
            targetWidget: "<TARGET_WIDGET_NAME>",
            passedData:
{ordering:"vehicleFlow",query:"https://www.snap4city.org/superservicemap/api/v1/io
t-
search/?selection=42.014990;10.217347;43.7768;11.2515&model=metrotrafficsensor&val
ueFilters=vehicleFlow>0.5;vehicleFlow<300",actions:["pin"],
columnsToShow:["dateObserved","vehicleFlow"]}
      });
}
```

The passedData field must be of the type described in https://www.snap4city.org/drupal/node/809.

```
{
      ordering: <ORDER_BY_COLUMN_NAME>,
      query: <QUERY_SUPERSERVICEMAP>,
      actions:[<ARRAY_OF_ICONS_FOR_ACTION_COLUMN],
      columnsToShow: [<ARRAY_OF_TABLE_COLUMN_TO_SHOW>]
}
```

Example

```
{
    ordering: "dateObserved",
    query: "https://www.snap4city.org/superservicemap/api/v1/iot-
search/?selection=42.014990;10.217347;43.7768;11.2515&model=IBE Air Quality",
    actions:
["pin","https://upload.wikimedia.org/wikipedia/commons/thumb/6/6d/Windows_Settings
_app_icon.png/1024px-Windows_Settings_app_icon.png"]*,
```

```
    columnsToShow: ["dateObserved", "airHumidity", "PM10"]
}
```

*"Pin" is a keyword for displaying a pin icon, to insert icons of other types it is necessary to insert the url of an icon, in this case the icon of a gear present at the url https://upload.wikimedia.org/wikipedia/commons/thumb/6/6d/Windows_Settings_app_icon.png/1024px-Windows_Settings_app_icon.png

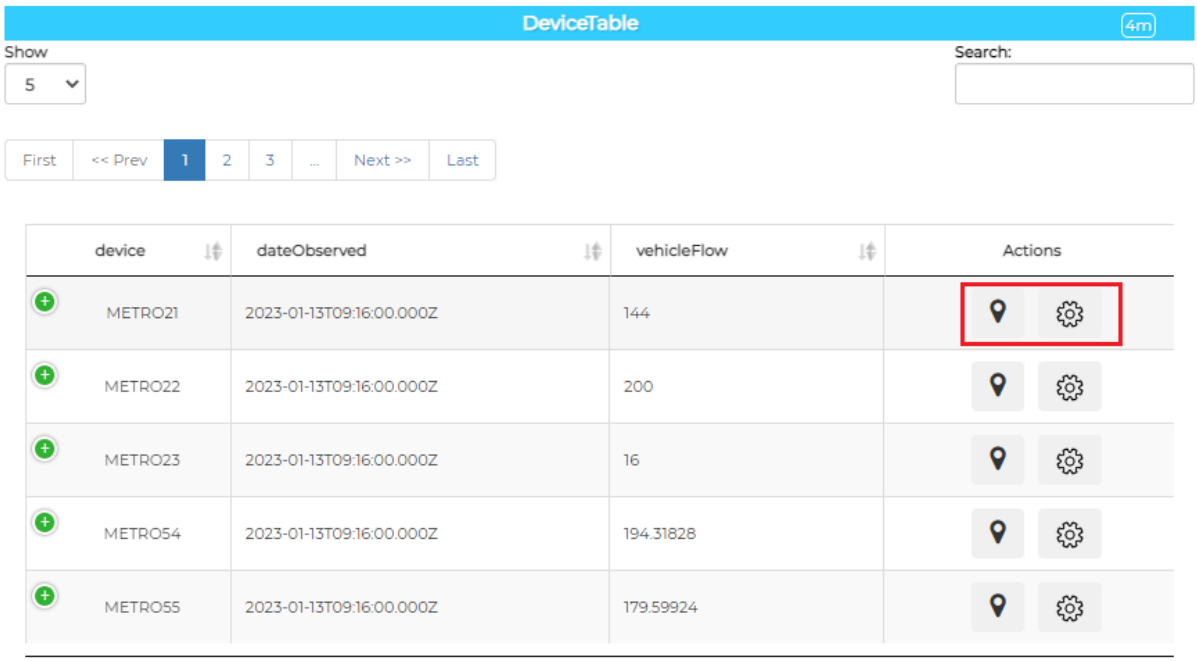


Figure 15: widgetDeviceTable example

6.7 widgetMap (IN/OUT)

6.7.1 widgetMap as Reading widget

First of all, an existing Map widget must be identified in the dashboard, of which the id <TARGET_WIDGET_NAME> must be noted.
The JavaScript function to be inserted in the appropriate CK Editor box (in more options) of another widget of the same dashboard, in order to pilot the <TARGET_WIDGET_NAME> widgetMap is of the following type:

```
function execute() {
	var coordsAndType = {};
	coordsAndType.eventGenerator = $(this);
	coordsAndType.desc = "CarPark";
	coordsAndType.query =
"https://servicemap.disit.org/WebAppGrafo/api/v1/?selection=43.64471;11.005751;43.89471;11.505751&categories=Car_park&maxResults=200&format=json&model=CarPark";
	coordsAndType.color1 = "#ebb113";
```

```
        coordsAndType.color2 = "#eb8a13";
        coordsAndType.targets = <TARGET_TIME_TREND_WIDGET_NAME_(OPTIONAL)>;
        coordsAndType.display = "pins";
        coordsAndType.queryType = "Default";
        coordsAndType.iconTextMode = "text";
        coordsAndType.pinattr = "square";
        coordsAndType.pincolor = "#959595";
        coordsAndType.symbolcolor = "undefined";
        coordsAndType.bubbleSelectedMetric = "";

        $('body').trigger({
               type: "addSelectorPin",
               target: <TARGET_WIDGET_NAME>,
               passedData: coordsAndType
        });
}
```

coordsAndType.query can be of the following types:

- **Device category:** To find all entities (e.g. IoT Devices, sensors, etc.) of a certain category (specified in the query's categories parameter, which corresponds to the Subnature column in the wizard table) and produced by a certain IoT Model (model parameter in the query, corresponding to the Device/Model column of the wizard). A display bounding box must also be specified (parameter selection)

  Subnature and Model examples:
  - categories=Car_park&model=CarPark
  - categories=Traffic_sensor&model=metrotrafficsensor
  - categories=Weather_sensor&model=SirSensors

- **Individual devices:** To find individual entities (e.g. IoT Devices, sensors, etc.) identified by their unique serviceURI.

Example:

https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=<SERVICE_URI>

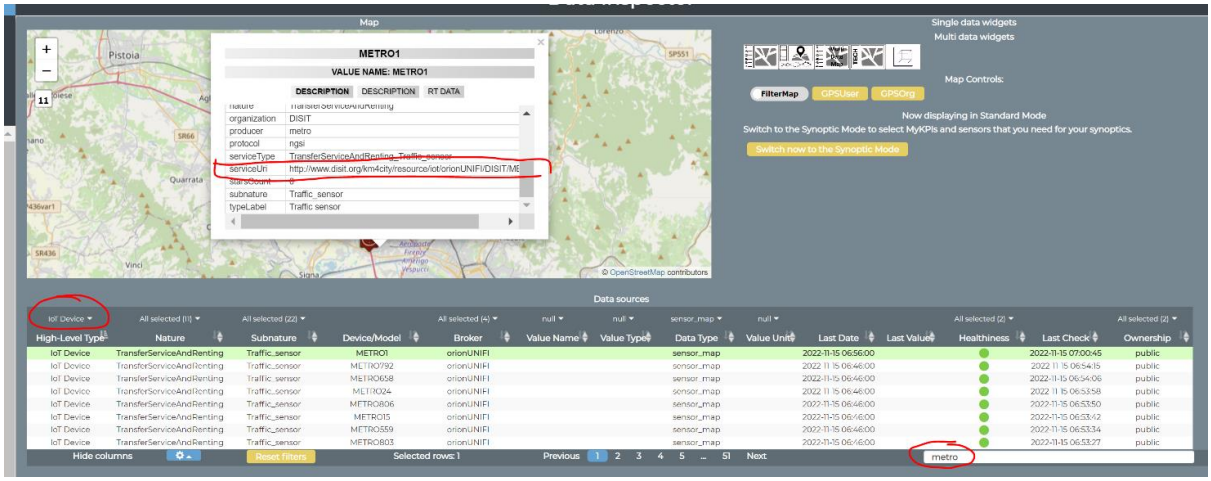
Examples of serviceURIs:

- http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/SIRSensor_TOS01001205 (Environmental sensor)

- http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1 (Traffic sensor)

- http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/CarParkPal.Giustizia (Parking sensor)

Where to find the ServiceURI in the Wizard or Inspector:
(Traffic sensors):



(Weather sensors):

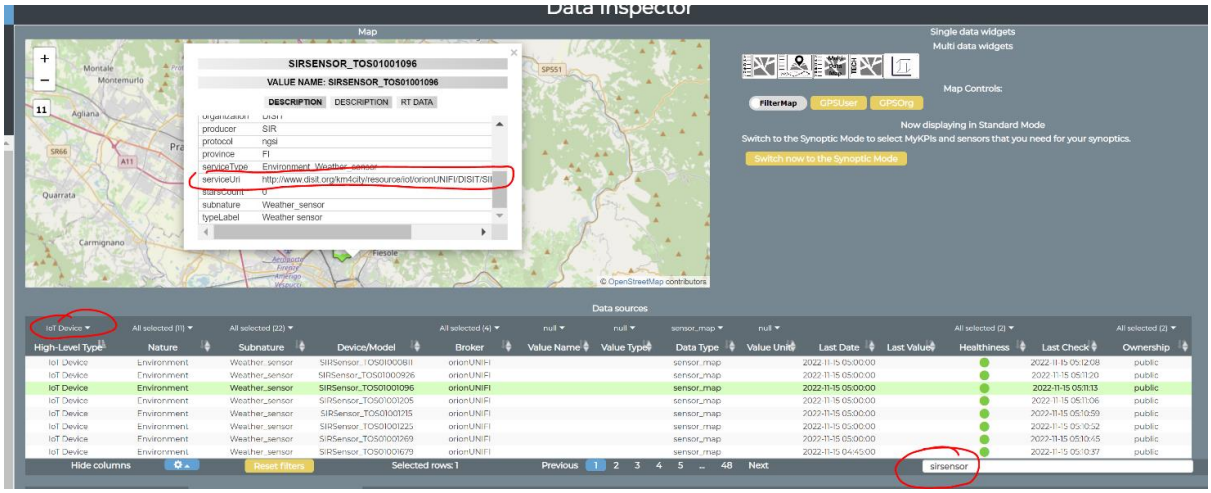


Figure 16- 17: Device Selection by Wizard

6.7.2 widgetMap as Writing widget: Geographic Drill-Down by Zoom on Widget Map

At the click on the geographic drill-down button on the widget map, the JavaScript function script in the CKeditor is executed passing as a parameter a JSON named "param" containing the zoom bounds, the type of event (in this case "zoom") and a list containing all the selected layers (those inside the bounds) and their properties.

The properties of the sensors passed have to be adapted to prepare data in the suitable format to be read by the target widget, in order to retrieve and display them. Therefore, it is necessary to build a JSON with proper data format. Below is an example of how to construct a JSON to be passed starting from the input parameter and an array of metrics of interest:

```
var e = JSON.parse(param)
var metrics =
["anomalyLevel","averageSpeed","avgTime","concentration","congestionLevel","vehicl
eFlow"];
var data = [];
```

```
let h = 0;
for (var l in e.layers) {
       for(var m in metrics){
              data[h] = {};
              data[h].metricId =
"https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri="+
e.layers[l].serviceUri;
              data[h].metricHighLevelType = "Sensor";
              data[h].metricName = "DISIT:orionUNIFI:" + e.layers[l].name;
              data[h].metricType = metrics[m];
              h++;
       }
}
```

The javascript “data” object can be sent for display to other widgets, here is an example with a PieChart:

```
$('body').trigger({
   type: "showPieChartFromExternalContent_<TARGET_WIDGET_NAME>",
   eventGenerator: $(this),
   targetWidget: "<TARGET_WIDGET_NAME>",
   passedData: date
});
```

6.7.3 widgetMap as Writing widget: Marker click on WidgetMap

At the click on a specific marker on widget map, the JavaScript function script in the CKeditor is executed passing as a parameter a JSON named "param" with a list containing all the selected layers (those inside the bounds) and their properties.

The properties of the sensors passed have to be adapted to prepare data in the suitable format to be read by the target widget, in order to retrieve and display them. Therefore, it is necessary to build a JSON with proper data format.

The same example shown in section 6.7.2 is suitable also for this action: in this case, a single SURI will be passed in the JavaScript function, instead of a list of SURI as in the previous case.

6.7.4 widgetMap as Writing widget: Click on a generic point on WidgetMap

At the click on a generic point on widget map, the JavaScript function script in the CKeditor is executed passing as a parameter a JSON named "param" with the geographical coordinates of the clicked point. The JSON has the following structure and attributes:

```
{
	"event": "mapClick",
	"coordinates": {
		"latitude": 43.780043758868835,
		"longitude": 11.26093190389292
	}
}
```

6.8 widgetOnOffButton (IN/OUT)

First of all, an existing widget must be identified in the dashboard to be the target of the triggered action, of which the id <TARGET_WIDGET_NAME> must be noted. In this example the target is a WidgetSingleContent.
The JavaScript function to be inserted in the appropriate CK Editor box (in more options) of the current OnOffButton widget is of the following type:

```
function execute() {
	$('body').trigger({
		type: "showSingleContentFromExternalContent_<TARGET_WIDGET_NAME>",
		eventGenerator: $(this),
		targetWidget: "<TARGET_WIDGET_NAME>",
		color1: "#e8a023",
		color2: "#9c6b17",
		widgetTitle: "ShowDouble",
		passedData: { "dataOperation": param}
	    });
}
```

The passedData field can be:

```
	passedData: {

		"dataOperation": <VALUE>

	}
```

The param variable consists of the input value generated by the state change of the widgetOnOffButton widget. you can send it in passedData, or use it to perform operations in JavaScript:

```
function execute() {
var check_status='NOT ACTIVED';
	if (param !== null){
		check_status = 'ACTIVED';
	}
	$('body').trigger({
		type: "showSingleContentFromExternalContent_<TARGET_WIDGET_NAME>",
		eventGenerator: $(this),
		targetWidget: "<TARGET_WIDGET_NAME>",
		color1: "#e8a023",
		color2: "#9c6b17",
		widgetTitle: "ShowOnOffStatus",
```

```
            passedData: { "dataOperation": check_status}
          });
}
```

In this example in the <TARGET_WIDGET_NAME> a string "ACTIVED" or "NOT ACTIVED" will appear if the status of the widget widgetOnOffButton is null or not, for this reason before the function $('body').trigger a variable check_status is created and a check on the content of the param, which corresponds to the current state of the widget, and then send it in the PassedData.

6.8.2 widgetOnOffButton as Reading widget

It is possible to send a status value from another widget as a read parameter in the widget on/off button using a JavaScript function inserted in the CKEditor of the writing widget.

In this example, you can send a parameter from an impulse button widget to an on/off button widget. In the CKEditor of the writing widget this execute function must be written whose type must be written as follows.
showOnOffButtonFromExternalContent_<TARGET_WIDGET_NAME>

```
function execute() {
$('body').trigger({
    type: "showOnOffButtonFromExternalContent_<TARGET_WIDGET_NAME>",
    eventGenerator: $(this),
    targetWidget: "<TARGET_WIDGET_NAME>",
    widgetTitle: "ShowData",
    passedData: { "dataOperation": <VALUE>}
    });
}
```

The <VALUE> must be set as “Off” or “On”.

6.9 widgetKnob (IN/OUT)

6.9.1 widgetKnob as Writing widget

First of all, an existing widget must be identified in the dashboard to be the target of the triggered action, of which the id <TARGET_WIDGET_NAME> must be noted. In this example the target is a WidgetSingleContent.
The JavaScript function to be inserted in the appropriate box (in more options) of the current widgetKnob is of the following type:

```
function execute() {
     $('body').trigger({
        type: "showSingleContentFromExternalContent_<TARGET_WIDGET_NAME>",
        eventGenerator: $(this),
        targetWidget: "<TARGET_WIDGET_NAME>",
```

```
        color1: "#e8a023",
        color2: "#9c6b17",
        widgetTitle: "ShowKnobStatus",
       passedData: { "dataOperation": param }
    });
}
```

The param variable consists of the input value generated by the widgetKnob widget state change. you can send it in passedData, or use it to perform operations in JavaScript:

The passedData field can be:

```
passedData: {

        "dataOperation": param

}
```

6.9.2 widgetKnob as Reading widget

It is possible to send a numeric value from another widget as a read parameter in the widget knob using a JavaScript function inserted in the CKEditor of the writing widget.

In this example, you can send a parameter from an impulse button widget to a knob widget. In the CKEditor of the writing widget this excute function must be written whose type must be written as follows. *showKnobFromExternalContent_<TARGET_WIDGET_NAME>*

```
function execute() {
$('body').trigger({
    type: "showKnobFromExternalContent_w__1540_widgetKnob10040",
    eventGenerator: $(this),
    targetWidget: "w__1540_widgetKnob10040",
    color1: "#e8a023",
    color2: "#9c6b17",
    widgetTitle: "ShowData",
    passedData: { "dataOperation": <VALUE>}
    });
}
```

The passedData field can be:

```
passedData: {

        "dataOperation": <VALUE>

}
```

6.10 widgetNumericKeyboard (IN/OUT)

6.10.1 widgetNumericKeyboard as a Writing widget

First of all, an existing widget must be identified in the dashboard to be the target of the triggered action, of which the id <TARGET_WIDGET_NAME> must be noted. In this example the target is a WidgetSingleContent.
The JavaScript function to be inserted in the appropriate box (in more options) of the current widgetNumericKeyboard is of the following type:

```
function execute() {

      $('body').trigger({
    type: "showSingleContentFromExternalContent_<TARGET_WIDGET_NAME>",
    eventGenerator: $(this),
    targetWidget: "<TARGET_WIDGET_NAME>",
    color1: "#e8a023",
    color2: "#9c6b17",
    widgetTitle: "ShowDouble",
                passedData: { "dataOperation": param}
     });
}
```

The param variable consists of the input value generated by the change of state of the widgetNumericKeyboard widget, when you click on the "confirm" button.
you can send it in passedData, or use it to perform operations in javascript:

```
   function execute() {
        $('body').trigger({
                   type: "showSingleContentFromExternalContent_<TARGET_WIDGET_NAME>",
                   eventGenerator: $(this),
                   targetWidget: "<TARGET_WIDGET_NAME>",
                   widgetTitle: "ShowDouble",
                   passedData: { "dataOperation": param*2}
                 });
        }
}
```

In this example we want the target widget to show the double value compared to the one inserted in the current widgetNumericKeyboard.

The passedData field can be:

```
        passedData: {
                "dataOperation": <VALUE>
        }
```

6.10.2 widgetNumericKeyboard as a Reading widget

It is possible to send a numeric value from another widget as a read parameter in the widgetNumericKeyboard using a JavaScript function inserted in the CKEditor of the writing widget.

In this example, you can send a parameter from an impulse button widget to a numeric keyboard widget. In the CKEditor of the writing widget this execute function must be written whose type must be written as follows.
showNumericKeyboardFromExternalContent_<TARGET_WIDGET_NAME>

```
function execute() {
  type:
"showNumericKeyboardFromExternalContent_w__1540_widgetNumericKeyboard10039",
    eventGenerator: $(this),
    targetWidget: "w__1540_widgetNumericKeyboard10039",
    color1: "#e8a023",
    color2: "#9c6b17",
    widgetTitle: "ShowData",
passedData: { "dataOperation": param}

}
```

The passedData field can be:

```
passedData: {
        "dataOperation": <VALUE>
        }
```

6.11 widgetPieChart

6.11.1 widgetPieChart as Reading widget

First of all, an existing widgetPieChart must be identified in the dashboard, of which the id <TARGET_WIDGET_NAME> must be noted.
The JavaScript function to be inserted in the appropriate CK Editor box (in more options) of another widget of the same dashboard, in order to pilot the <TARGET_WIDGET_NAME> widgetPieChart is of the following type:

```
function execute() {

    $('body').trigger({
            type: "showPieChartFromExternalContent_<TARGET_WIDGET_NAME>",
            eventGenerator: $(this),
            targetWidget: "<TARGET_WIDGET_NAME>",
```

```
                passedData:
        [{"metricId":"https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=h
        ttp://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO759","metric
        HighLevelType":"Sensor","metricName":"DISIT:orionUNIFI:METRO759","metricTyp
        e":"averageSpeed"},{"metricId":"https://servicemap.disit.org/WebAppGrafo/ap
        i/v1/?serviceUri=http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT
        /METRO759","metricHighLevelType":"Sensor","metricName":"DISIT:orionUNIFI:ME
        TRO759","metricType":"avgTime"},{"metricId":"https://servicemap.disit.org/W
        ebAppGrafo/api/v1/?serviceUri=http://www.disit.org/km4city/resource/iot/ori
        onUNIFI/DISIT/METRO759","metricHighLevelType":"Sensor","metricName":"DISIT:
        orionUNIFI:METRO759","metricType":"vehicleFlow"}]
            });
}
```

6.11.2 widgetPieChart as Writing widget

When clicking on an element of the widgetPieChart, a JSON object called param is passed to the execute() function set in the CK Editor, in which there are the type of event (in this case "click") and all the elements present inside the widget in the layers field.
The properties of the sensors passed have to be adapted to prepare data in the suitable format to be read by the target widget, in order to retrieve and display them. Therefore, it is necessary to build a JSON with proper data format.
The following example shows how to send the information to a Radar widget (widgetRadarSeries) related to a selected Entity/Device or metric which has been clicked on the Pie chart:

```
var e = param;
    let sensName = [];
    let metricTypes = [];
    var s;
    for (let i = 0; i < param.length; i++) {
        s = param[i].metricName.replace("DISIT:orionUNIFI:", '');
        if (!sensName.includes(s)) {
            sensName.push(s);
        }
        if (!metricTypes.includes(s)) {
            metricTypes.push(param[i].metricType);
        }
    }
    let data = buildData(sensName, metricTypes);
    if (data.length > 1) {
        $('body').trigger({
            type:
"showRadarSeriesFromExternalContent_w_AggregationSeries_3721_widgetRadarSeries35546",
            eventGenerator: $(this),
            targetWidget: "w_AggregationSeries_3721_widgetRadarSeries35546",
            passedData: data
        });
    }
}
```

```
function buildData(sensName, metricTypes) {
    var data = [];
    let h = 0;
    for (let i = 0; i < (sensName.length); i++) {
        for (let j = 0; j < (metricTypes.length); j++) {
            data[h] = {};
            data[h].metricId =
"https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://www.disit.org/km4city/r
esource/iot/orionUNIFI/DISIT/" + sensName[i];
            data[h].metricHighLevelType = "Sensor";
            data[h].metricName = "DISIT:orionUNIFI:" + sensName[i];
            data[h].metricType = metricTypes[j];
            h++;
        }
    }
    return data;
}
```

6.11.3 WidgetPieChart Time Selection

Widget PieChart also allows you to send a date and time parameter as input to another widget to insert in the calendar. In this way it is possible to synchronize more than one widget at the same moment in time.

For example, if in a dashboard there are two PieChart widgets with different data and I want them to always show data relating to the same moment in time, I can activate this feature so that when I use the calendar in one of the two, the other is also changed.

This feature is activated when you edit the calendar. The following parameters must be set in the execute() function of the widget writer:

```
event: "set_time"
```

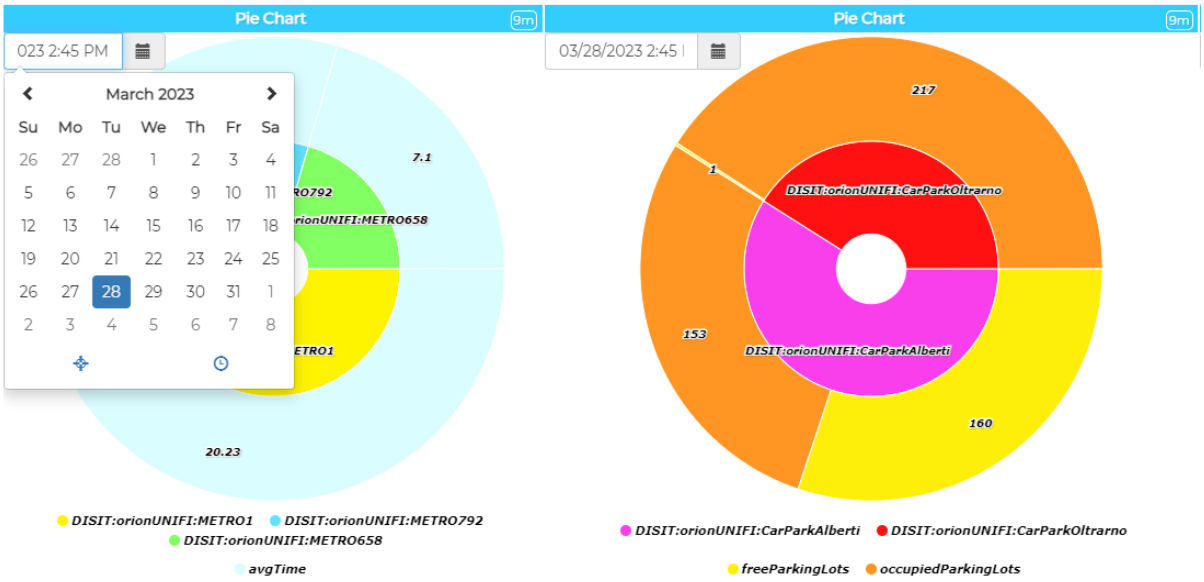
It is an important parameter because it is used to activate the synchronization function. It doesn't work without it.

```
datetime: param
```

This parameter must receive the date of the datetime automatically from the selection of the Calendar.

```
passedData: []
```

Usually when writing the PieChart widget it is possible to send a list of elements to the target widget, in the case of the selectionTime this parameter can be represented as an empty array. However, it must be present for the feature to work properly.




```
function execute() {
    $('body').trigger({
        type:
"showPieChartFromExternalContent_w_AggregationSeries_3854_widgetPieChart37089",
        targetWidget: "w_AggregationSeries_3854_widgetPieChart37089",
        widgetTitle: "Pie Chart time selection",
        event: "set_time",
        datetime: param,
        passedData: []
    });
}
```

6.12 widgetBarSeries (IN/OUT)

6.12.1 widgetBarSeries as Reading widget

First of all, an existing widget must be identified in the dashboard, of which the id <TARGET_WIDGET_NAME> must be noted. In this example the target is a WidgetSingleContent.
The JavaScript function to be inserted in the appropriate CK Editor box (in more options) of another widget of the same dashboard, in order to pilot the <TARGET_WIDGET_NAME> widgetBarSeries is of the following type:

```
function execute() {

    $('body').trigger({
            type: " showBarSeriesFromExternalContent_<TARGET_WIDGET_NAME>",
            eventGenerator: $(this),
            targetWidget: "<TARGET_WIDGET_NAME>",
```

```
            passedData:
        [{"metricId":"https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=h
        ttp://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO759","metric
        HighLevelType":"Sensor","metricName":"DISIT:orionUNIFI:METRO759","metricTyp
        e":"averageSpeed"},{"metricId":"https://servicemap.disit.org/WebAppGrafo/ap
        i/v1/?serviceUri=http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT
        /METRO759","metricHighLevelType":"Sensor","metricName":"DISIT:orionUNIFI:ME
        TRO759","metricType":"avgTime"},{"metricId":"https://servicemap.disit.org/W
        ebAppGrafo/api/v1/?serviceUri=http://www.disit.org/km4city/resource/iot/ori
        onUNIFI/DISIT/METRO759","metricHighLevelType":"Sensor","metricName":"DISIT:
        orionUNIFI:METRO759","metricType":"vehicleFlow"}]
            });
}
```

6.12.2 widgetBarSeries as Writing widget: Click on Legend item on WidgetBarSeries

When clicking on an element of the widgetBarSeries legend, the execute function is executed inside the CKEditor which is passed a JSON object called param in which there are the type of event (in this case "legendItemClick"), all the elements present inside the widget in the layers field with inside even if they are visible or not (corresponding to whether or not they are selected in the legend) and finally all the metrics present.
Below is an example of how to send the sensors visible in the legend of the command widget to a Radar Series when clicking on an element of it:

```
function execute(){
        var e = JSON.parse(param);
        if(e.event == "legendItemClick"){
                var date = [];
                let name, h = 0;
                for (var l in e.layers) {
                        if(e.layers[l].visible == true){
                        name = e.layers[l].name.slice(17,e.layers[l].name.length);
                        for(var m in e.metrics){
                        date[h] = {};
                        data[h].metricId =
                        "https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://www.disit.
                        org/km4city/resource/iot/orionUNIFI/DISIT/"+name ;
                        data[h].metricHighLevelType = "Sensor";
                        data[h].metricName = e.layers[l].name;
                        data[h].metricType = e.metrics[m];
                        h++;
                        }
                        }
                }
                $('body').trigger({
                type: "showRadarSeriesFromExternalContent_w_AggregationSeries_1_widgetRadarSeries49",
                eventGenerator: $(this),
                targetWidget: "w_AggregationSeries_1_widgetRadarSeries49",
                passedData: date
```

```
        });
        }
}
```

6.12.3 widgetBarSeries as Writing widget: Click on Bar

When clicking on a bar of the widgetBarSeries, a JSON object called param is passed to the execute() function set in the CK Editor, in which there are the type of event (in this case "click") and all the elements present inside the widget in the layers field.
The properties of the sensors passed have to be adapted to prepare data in the suitable format to be read by the target widget, in order to retrieve and display them. Therefore, it is necessary to build a JSON with proper data format.
The following example shows how to send the information to a multi-series widget (widgetCurvedLineSeries) related to a selected Entity/Device metric which has been clicked on the Bar chart:

```
function execute() {
    var e = JSON.parse(param);
    if (e.event == "click") {
        let serviceUri =
"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/" +
e.value.metricName.slice(17, e.value.metricName.length);
        let smField = e.value.metricType;
        var data = [];
        data[0] = {};
        data[0].metricId =
"http://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://www.disit.org/k
m4city/resource/iot/orionUNIFI/DISIT/" + e.value.metricName.slice(17,
e.value.metricName.length) + "&format=json";
        data[0].metricHighLevelType = "Sensor";
        data[0].metricName = "DISIT:orionUNIFI:" + e.value.metricName.slice(17,
e.value.metricName.length);
        data[0].smField = e.value.metricType;
        data[0].serviceUri = serviceUri;
        console.log(data);
        $('body').trigger({
            type:
"showCurvedLinesFromExternalContent_w_AggregationSeries_3721_widgetCurvedLineSerie
s35563",
            eventGenerator: $(this),
            targetWidget: "w_AggregationSeries_3721_widgetCurvedLineSeries35563",
            range: "7/DAY",
            color1: "#9b93ed",
            color2: "#231d5c",
            widgetTitle: "Occupied Parking Lots (Alberti Car Park) from Impulse
Button",
            field: data.smField,
            passedData: data
        });
    }
}
```

6.12.4 widgetBarSeries Time Selection

Widget BarSeries also allows you to send a date and time parameter as input to another widget to insert in the calendar. In this way it is possible to synchronize more than one widget at the same moment in time.

For example, if in a dashboard there are two BarSeries widgets with different data and I want them to always show data relating to the same moment in time, I can activate this feature so that when I use the calendar in one of the two, the other is also changed.

This feature is activated when you edit the calendar. The following parameters must be set in the execute() function of the widget writer:

```
event: "set_time"
```

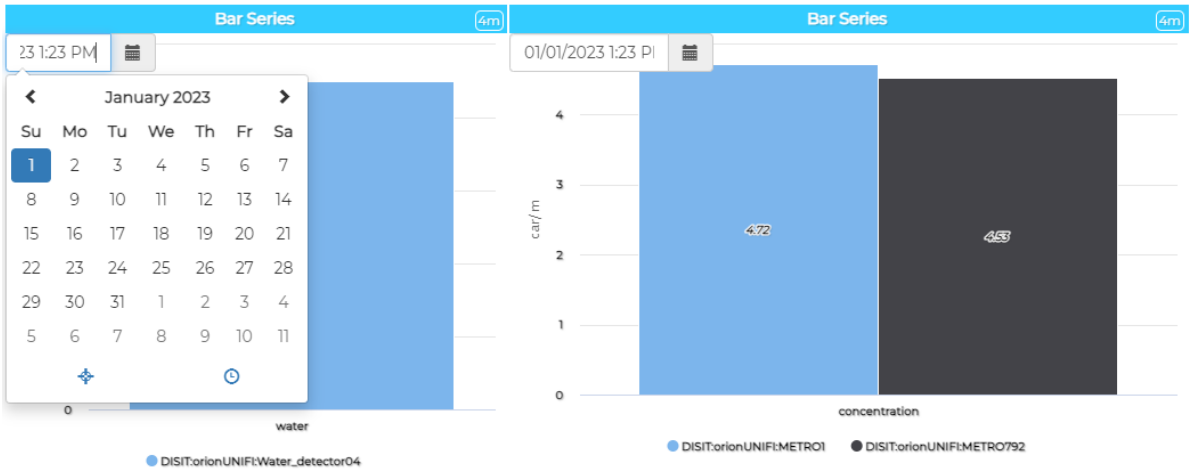
It is an important parameter because it is used to activate the synchronization function. It doesn't work without it.

```
datetime: param
```

This parameter must receive the date of the datetime automatically from the selection of the Calendar.

```
passedData: []
```

Usually when writing the BarSeries widget it is possible to send a list of elements to the target widget, in the case of the selectionTime this parameter can be represented as an empty array. However, it must be present for the feature to work properly.



Example:

```
function execute() {
    $('body').trigger({
        type:
"showBarSeriesFromExternalContent_w_AggregationSeries_3865_widgetBarSeries37138",
        targetWidget: "w_AggregationSeries_3865_widgetBarSeries37138",
        widgetTitle: "Example Bar Series",
```

```
        event: "set_time",
        datetime: param,
        passedData: []
    });
}
```

6.13 widgetEventTable

First of all, an existing widget must be identified in the dashboard, of which the id <TARGET_WIDGET_NAME> must be noted. In this example the target is a WidgetSingleContent.
The JavaScript function to be inserted in the appropriate box (in more options) of the current widgetEventTable is of the following type:

```
function execute() {
     $('body').trigger({
              type: "showSingleContentFromExternalContent_<TARGET_WIDGET_NAME>",
              eventGenerator: $(this),
              targetWidget: "<TARGET_WIDGET_NAME>",
              color1: "#e8a023",
              color2: "#9c6b17",
              widgetTitle: "ShowDouble",
             passedData: { "dataOperation": param}
          });
}
```

The param variable consists of the input value generated by the change of state of the widgetEventTable widget, when one of the icons in the action column of the EventTable graph is clicked.

The passedData field can be:

```
      passedData: {
            "dataOperation": <VALUE>
      }
```

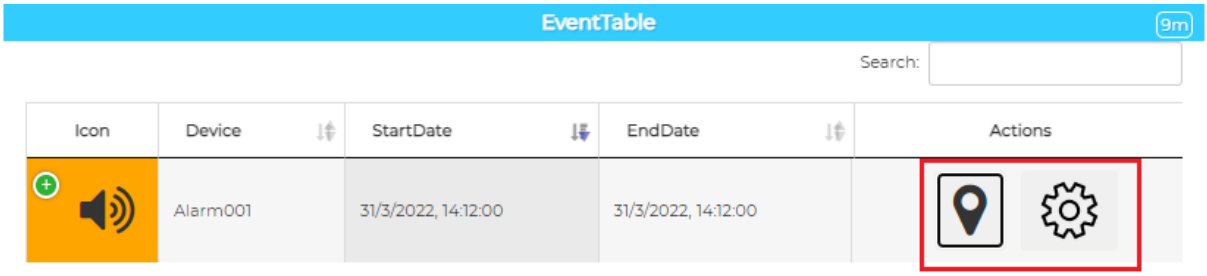


Figure 8: widgetEventTable example

6.14 widgetExternalContent

widgetExternalContent is a widget that allows you to insert both JavaScript scripts and HTML code using the CKEditor.
This feature differentiates it from most other types of widgets.

For example, the ExternalContent widget could be used to create small HTML graphical interfaces to which JavaScript functions are associated.

Figure 19: widgetExternalContent example

In the following example it is in fact possible to define an iframe with two buttons that send data to two different widgets, a time trend and a widget map.

The HTML code and the JavaScript function to be inserted in the appropriate box (in more options) of the current widget is of the following type:

```
<html>
<head>
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.10.1/jquery.min.js"></script>
         <script type='text/javascript'>
         var showAlert;
         var triggerTimeTrend;
         var triggerMap;
         $(document).ready(function () {
             showAlert = function () {
                 var myText = "Test alert";
                 alert (myText);
             }
             $('#triggerTTrend').click(function (event) {
                 parent.$('body').trigger({
    type: "showTimeTrendFromExternalContentGis_<TARGET_WIDGET_NAME>",
                     eventGenerator: $(this),
                     targetWidget: "<TARGET_WIDGET_NAME>",
                     range: "7/DAY",
                     color1: "#34eb6e",
                     color2: "#114a23",
```

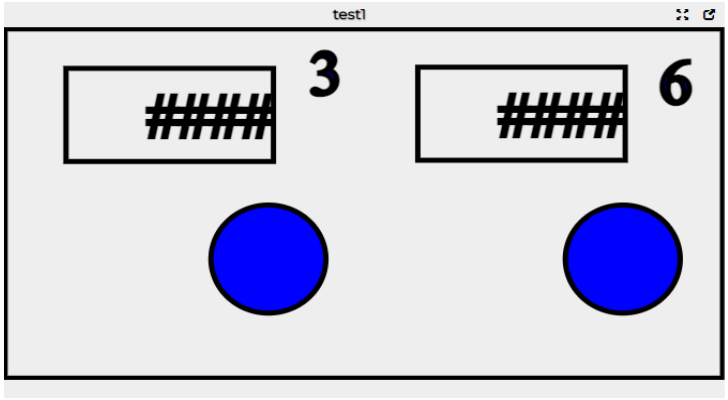
```
                    widgetTitle: "Free Parking Lots data from External Content",
                    field: "freeParkingLots",
                    serviceUri: "<SERVICE_URI>",
                    marker: "",
                    mapRef: "",
                    fake: false
                });
            });
            $('#triggerMap').click(function (event) {
                let coordsAndType = {};
                coordsAndType.eventGenerator = $(this);
                coordsAndType.desc = "CarPark";
            coordsAndType.query = "<SERVICE_URI>";
                coordsAndType.color1 = "#ebb113";
                coordsAndType.color2 = "#eb8a13";
            coordsAndType.targets = "w_DISIT_orionUNIFI_<TARGET_WIDGET_NAME>";
                coordsAndType.display = "pins";
                coordsAndType.queryType = "Default";
                coordsAndType.iconTextMode = "text";
                coordsAndType.pinattr = "square";
                coordsAndType.pincolor = "#959595";
                coordsAndType.symbolcolor = "undefined";
                coordsAndType.bubbleSelectedMetric = "";

                parent.$('body').trigger({
                    type: "addSelectorPin",
                    target: "<TARGET_WIDGET_NAME>",
                    passedData: coordsAndType
                });
            });
        });
        </script>
    </head>
    <body>
        <h2>Trigger dashboard widgets from External Content iframe</h2>
        <div>
            <button id="triggerTTrend">Trigger data on Time-Trend</button>
            <button id="triggerMap">Trigger data on Map</button>
        </div>
    </body>
</html>
```

6.14.1 Use of Synoptic SVG in ExternalContent Widget

The External Content widget allows the loading of custom widgets in SVG format. They can be used with SSBL and CSBL. For general design, develop and test SVG synoptics see:

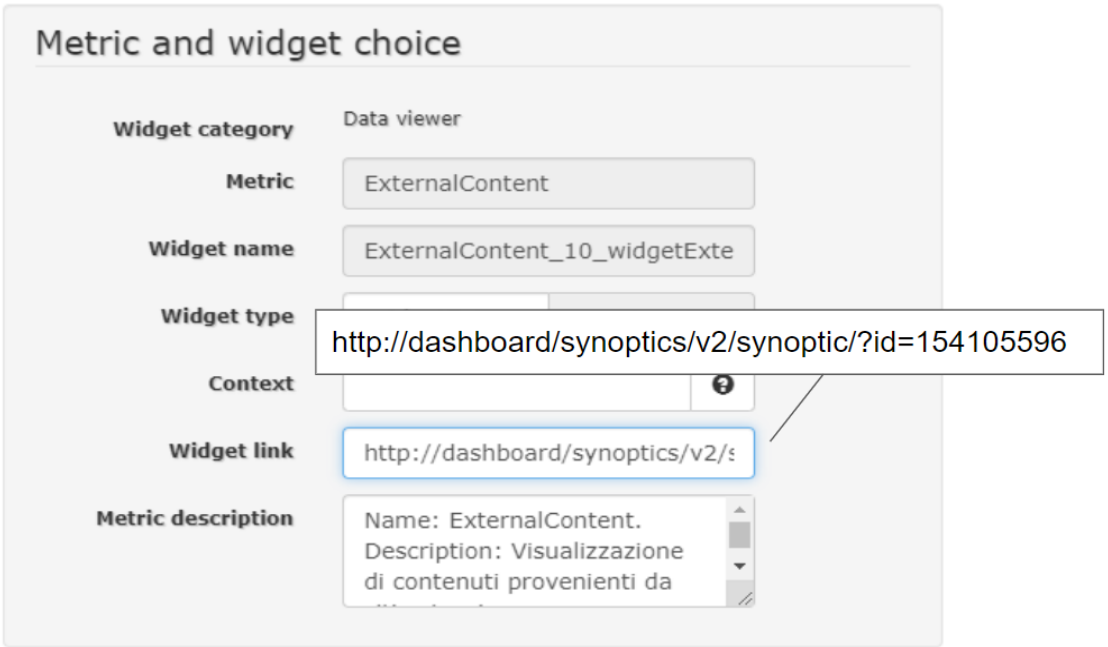
- TC1.22a: Create and configure a Snap4City SVG Custom Widget for real-time interaction

- TC1.22b: Create and configure a Snap4City SVG Custom Widget for real-time interaction
- Custom Widgets: Table explanation, as SVG
- TC1.26: Use customised SVG pins in a map
- Custom Synoptics and Widgets for Dashboards



This section describe how to use SVG on web pages directly without using them from server side business logic.

During the creation of an External content widget it is possible to link the url of a previously created CustomWidget or Synoptic in the Widget link menu.



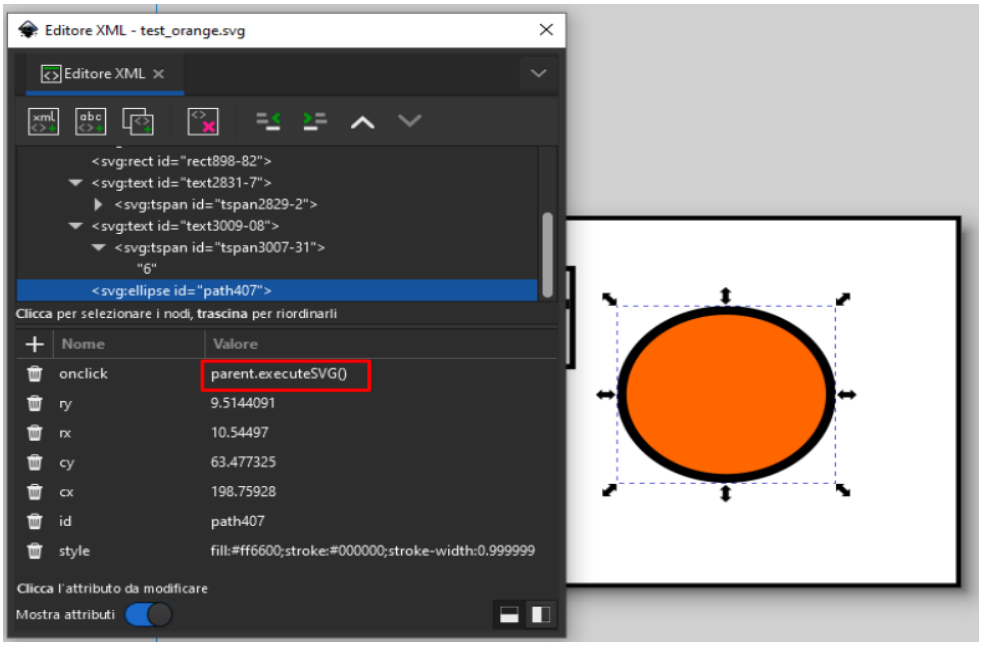
To allow the embedding of the screen svg code in the html code of the widget it is necessary to activate as true the menu "*SVG Mode*" and "*Enable CKEditor*" in the lower right section of the widget modification form.

6.14.1.1 widgetExternalContent as Writing widget: From Svg to Dashboard Widget.

A javascript function can be defined in the CKeditor area of the widgetExternalContent and executed by the SVG synoptic.
In this example, clicking on an element of the synoptic svg it is possible to send parameters to a radar widget.

```
function executeSVG() {
    $('body').trigger({
        type: "showRadarSeriesFromExternalContent_<TARGET WIDGET NAME>",
        eventGenerator: $(this),
        targetWidget: <TARGET WIDGET NAME>,
        range: "7/DAY",
        color1: "#9b93ed",
        color2: "#231d5c",
        events: "sendData",
        passedData:
[{"metricId":"https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://www.disit.
org/km4city/resource/iot/orionUNIFI/DISIT/IT0957A1&format=json","metricHighLevelType":"Sens
or","metricName":"DISIT:orionUNIFI:IT0957A1","metricType":"O3_"}]
    });
};
```

The function, which in this case is called *executeSVG()*, must be present in the svg synoptic and be called as *"parent.executeSVG()*".



6.14.1.2 widgetExternalContent as Reading widget: From Dashboard Widget to Svg

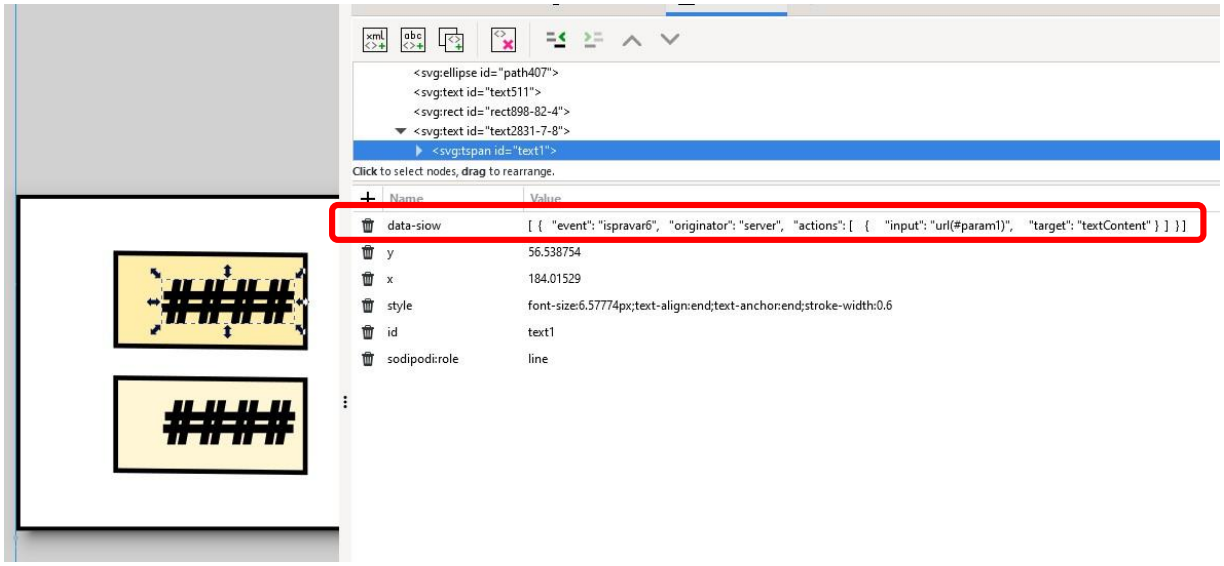
The Svg synoptic embedded in the widgetExternal content can receive some parameters from another widget and use them.
These parameters have to be written in a specific array of objects with the following keys:

```
{
```

```
“target”: <id of element in svg file>
“attributes”: <element with data-siow parameters of the object>
}
```

The following is an example of a selected text into a svg XML with a defined data-siow attribute defining the variable “ispravar6” that can be governed in the Client Side as explained below. Please refer to the following webpages for more information about hot to define a data-siow attribute in an SVG:

- TC1.22a: Create and configure a Snap4City SVG Custom Widget for real-time interaction
- TC1.22b: Create and configure a Snap4City SVG Custom Widget for real-time interaction



In the following example we want to send input values to two text elements present in the synoptic svg which have "text1" and "text2" as identifiers by using a widgetImpulseButton that will contains the client logic to send these parameters.

The structure of the function in the widgetImpulseButton is no different from other trigger functions:

```
function execute() {
    let svg_data = [{
        "target": "text1",
        "attributes": [{
            "event": "ispravar6",
            "originator": "server",
            "actions": [{
                "input": "5",
                "target": "textContent"
            }]
        }]}, {
        "target": "text2",
        "attributes": [{
```

```
            "event": "ispravar6",
            "originator": "server",
            "actions": [{
                "input": "3",
                "target": "textContent"
            }]
        }]
    }];

    $('body').trigger({
        type: "showExternalContentFromExternalContent_<TARGET WIDGET NAME>",
        eventGenerator: $(this),
        targetWidget: <TARGET WIDGET NAME>,
        range: "7/DAY",
        color1: "#9b93ed",
        color2: "#231d5c",
        events: "sendData",
        passedData: {
            "dataOperation": svg_data
        }
    });
}
```

A public example of dashboard with this interaction is available at the following link: https://www.snap4city.org/dashboardSmartCity/view/index.php?iddasboard=MzgyOQ==

6.14.2 widgetExternalContent to define data template

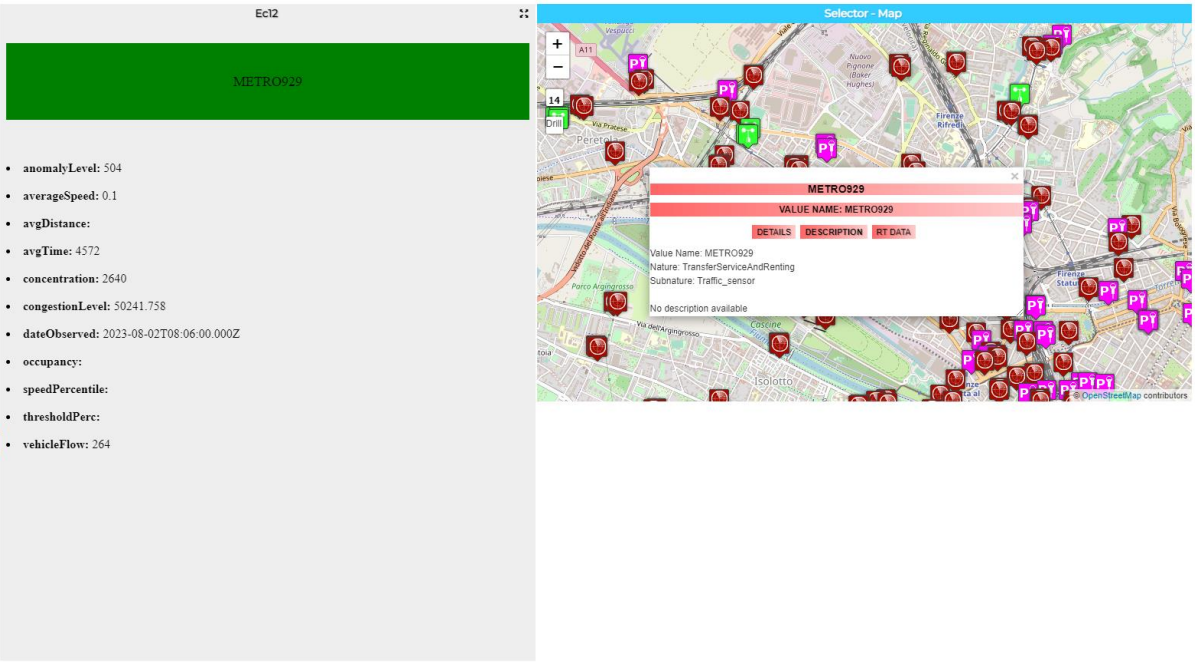
The External Content widget allows you to receive input values, for example a serviceUri, and use them to display data according to a precise template defined in the external content CKEditor.

To send data from an input widget, such as a widgetButton or a widgetMap, the "**events**" parameter must be setted as "**sendcontent**". To define an example of input request to send to widgetExaternalContent

```
    function execute() {
        const myJSONObject = {
             serviceuri: '<SERVICE_URI>',
            type: 'floor'
        };
        const jsonString = JSON.stringify(myJSONObject);
        $('body').trigger({
    type: 'showExternalContentFromExternalContent_<TARGET_WIDGET_NAME>',
            eventGenerator: $(this),
            targetWidget: '<TARGET_WIDGET_NAME>',
            range: '7/DAY',
            events: 'sendContent',
            passedData: jsonString
        })
    };
```

This example shows an HTML and Javascript script to be inserted in the CKEditor menu of the ExternalContent widget which is associated with an 'action' function which processes the data received as input and uses them.
This script receives a serviceuri as input by a widgetmap clicking on the marker on the map, and in the action function processes it through an ajax request to extract a value and subsequently display it in the output of the external content widget as an HTML content filled by values.

```
<!DOCTYPE html>
<html>
<head>
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.10.1/jquery.min.js"></script>
</head>
<body>
<div id="messageBox" style="width: 100%; height: 100px; background-color: #FFD700;
display: flex; justify-content: center; align-items: center; font-size: 18px;"> no
message </div>
<script>


function action(paramString) {
	var param = JSON.parse(paramString);
	$.ajax({
	url:
"https://www.snap4city.org/dashboardSmartCity/controllers/superservicemapProxy.php
/api/v1/?serviceUri="+ param.serviceuri + "&format=json",
type: "GET",
data: {},
async: true,
dataType: 'json',
success: function(data) {
	if (data) {
		<SCRIPT TO DEFINE HOW TO VIEW DATA>
	} else {
		console.log("Error in ASCAPI Data Retrieval.");
	}
},
error: function(data) {
	console.log(JSON.stringify(data));
	}
   });
 }
</script>
</body>
</html>
```



To allow the reading of the trigger received as input, a function called **action** has to be defined in this code.
The HTML content to view the values is fully customizable by the client, the client for example could define some templates to show the value in different ways on the base of some typology or amounts of values.

6.14.3 From other Widgets to External Content Triggering (IN)

Typically a widget is controlled by an Exernal Content widget, for example, the External Content may have CBLS to control other widget. It can be frequently used to create HTML/CSS Javascript CSBL to search data (from external API and SCAPI), control data, send these data and commands to other widgets. This feature is presented in other sections and not in this one.

In this section, the other way around is presented. That is a solution when a Widget could send a message and data/parameters to an External Content widget to execute some CSBL taking into account of these parameters.

For example, the CSBL JavaScript from a Widget Map can have:

```
function execute() {
   var e = JSON.parse(param)
   var coords = []
   if (e.event == "mapClick") {
      coords[0] = e.coordinates.latitude
      coords[1] = e.coordinates.longitude
   }
   const customEvent = new CustomEvent('receivecoordsfrommapclick', {
      detail: {
         targetWidget: "ExternalContent_4272_widgetExternalContent39953",
```

```
        passedData: coords // a set of parameters in this case only coordinates
      }
    })
    document.dispatchEvent(customEvent);
    console.log('Event Triggered: ', coords);
}
```

The counterpart on CSBL JavaScrip of a Widget External Content can be:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>
    <h1>Map Click Event Listener</h1>
    <script>
        function handleMapClickEvent() {
            parent.document.addEventListener('receivecoordsfrommapclick', function(event) {
            console.log('Recieved', event.detail.passedData);
          })
        }
        handleMapClickEvent();
    </script>
    <!-- jQuery CDN (required for the jQuery part of the code) -->
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
</body>
</html>
```

6.15 widgetImpulseButton (OUT)

First of all, an existing widget must be identified in the dashboard to be the target of the triggered action, of which the id <TARGET_WIDGET_NAME> must be noted. In this example the target is a WidgetSingleContent.
The JavaScript function to be inserted in the appropriate CK Editor box (in more options) of the current widgetImpulseButton widget is of the following type:

```
function execute() {
       $('body').trigger({
              type: "showLastDataFromExternalContentGis_<TARGET_WIDGET_NAME>"
              eventGenerator: $(this),
              targetWidget: <TARGET_WIDGET_NAME>,
              color1: "#acb2fa",
              color2: "#231d5c",
```

```
            widgetTitle: "Occupied Parking Lots (Alberti Car Park)",
            field: "occupiedParkingLots",
            serviceUri:
"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/CarParkPal.Giustizia"
      });
}
```

6.16 widgetButton (OUT)

An existing widget must be identified in the dashboard to be the target of the triggered action, of which the id <TARGET_WIDGET_NAME> must be noted. In this example the target is a WidgetSingleContent.
The JS function to be inserted in the appropriate CK Editor box (in more options) of the current widgetButton widget is of the following type:

```
function execute() {
      $('body').trigger({
            type: "showLastDataFromExternalContentGis_<TARGET_WIDGET_NAME>"
            eventGenerator: $(this),
            targetWidget: <TARGET_WIDGET_NAME>,
            color1: "#acb2fa",
            color2: "#231d5c",
            widgetTitle: "Occupied Parking Lots (Alberti Car Park)",
            field: "occupiedParkingLots",
            serviceUri:
"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/CarParkPal.Giustizia"
      });
}
```

7. Advanced examples, usage of Smart City APIs

7.1 JavaScript Example for Business Intelligence: Time Drill-Down from widgetTimeTrend to widgetBarSeries

More advanced example of time drill down from widgetTimeTrend to widgetBarSeries. By zooming in the TimeTrend widget, the following JavaScript function is executed, which makes a call to the Smart City API to retrieve the real-time data of the IoT Device corresponding to the Service URI sent in the parameters to the function itself.
The data received in response to the call to the API is averaged over the time range selected in the zoom (and sent as parameters to the function).

Finally, a data structure conforming to the required JSON is prepared to display the average values for each Device metric in a widgetBarSeries (specified by <TARGET_WIDGET_BAR_SERIES>).

```
function execute() {

    let minT, maxT = null;
    if (param['t1'] != param['t2']) {
        minT = param['t1'];
        maxT = param['t2'];
    } else {
        minT = param['t1']-10000000;
        maxT = param['t2']+10000000;
    }
    let dt1 = new Date(minT);
    let dt1_iso = dt1.toISOString().split(".")[0];
    let dt2 = new Date(maxT);
    let dt2_iso = dt2.toISOString().split(".")[0];

    function getMean(originalData)
    {
        var singleOriginalData, singleData, convertedDate = null;
        var convertedData = {
            data: []
        };
        var originalDataWithNoTime = 0;
        var originalDataNotNumeric = 0;
        var meanDataObj = {};
        if(originalData.hasOwnProperty("realtime"))
        {
            if(originalData.realtime.hasOwnProperty("results"))
            {
                if(originalData.realtime.results.hasOwnProperty("bindings"))
                {
                    if(originalData.realtime.results.bindings.length > 0)
                    {
                        let propertyJson = "";
                        if(originalData.hasOwnProperty("BusStop"))
                        {
                            propertyJson = originalData.BusStop;
```

```
                        }
                        else
                        {
                            if(originalData.hasOwnProperty("Sensor"))
                            {
                                propertyJson = originalData.Sensor;
                            }
                            else
                            {
                                if(originalData.hasOwnProperty("Service"))
                                {
                                    propertyJson = originalData.Service;
                                }
                                else
                                {
                                    propertyJson = originalData.Services;
                                }
                            }
                        }
                        for(var j = 0; j < originalData.realtime.head.vars.length;
j++) {
                            var singleObj = {}
                            var field = originalData.realtime.head.vars[j];
                            var numericCount = 0;
                            var sum = 0;
                            var mean = 0;
                            if (field == "updating" || field == "measuredTime" ||
field == "instantTime" || field == "dateObserved") {
                                continue;
                            }
                            for (var i = 0; i <
originalData.realtime.results.bindings.length; i++) {
                                singleOriginalData =
originalData.realtime.results.bindings[i];
                                if (singleOriginalData[field] !== undefined) {
                                    if
(!isNaN(parseFloat(singleOriginalData[field].value))) {
                                        numericCount++;
                                        sum = sum +
parseFloat(singleOriginalData[field].value);
                                    }
                                }
                            }
                            mean = sum / numericCount;
                            meanDataObj[field] = mean;
                        }
                        return meanDataObj;
                    } else {
                        return false;
                    }
                } else {
                    return false;
                }
            } else {
                return false;
            }
        } else {
            return false;
```

```
        }
    }

    function buildDynamicData(data, name) {
        var passedJson = [];
        for (const item in data) {
            var singleJson = {};
            singleJson["metricId"] = "";
            singleJson["metricHighLevelType"] = "Dynamic";
            singleJson["metricName"] = name;
            singleJson["metricType"] = item;
            singleJson["metricValueUnit"] = "";
            singleJson["value"] = data[item];
            passedJson.push(singleJson)
        }
        return passedJson;
    }

    $.ajax({
        url: "../controllers/superservicemapProxy.php/api/v1/?serviceUri=" +
encodeServiceUri(param['sUri']) + "&fromTime=" + dt1_iso + "&toTime=" + dt2_iso,
        // url: "../controllers/superservicemapProxy.php/api/v1/?serviceUri=" +
encodeServiceUri(sUri) + "&fromTime=" + dt1_iso + "&toTime=" + dt2_iso +
"&valueName=" + param['metricName'],
        type: "GET",
        data: {},
        async: true,
        dataType: 'json',
        success: function(data)
        {
            if (data.realtime.results) {
                var meanData = getMean(data);
                var passedJson = buildDynamicData(meanData,
data.Service.features[0].properties.name);

                $('body').trigger({
                    type:
"showBarSeriesFromExternalContent_<TARGET_WIDGET_BAR_SERIES>",
                    eventGenerator: $(this),
                    targetWidget: "<TARGET_WIDGET_BAR_SERIES>",
                    color1: "#f22011",
                    color2: "#9c6b17",
                    widgetTitle: "Air Quality IT from Impulse Button",
                    passedData: passedJson
                });
            } else {
                $('#<TARGET_WIDGET_BAR_SERIES>_chartContainer').hide();
                $('#<TARGET_WIDGET_BAR_SERIES>_noDataAlert').show();
            }
        },
        error: function (data)
        {
            console.log("Error downloading data from Service Map");
            console.log(JSON.stringify(data));
        }
    });
}
```

7.2 Example X: using a Business Intelligence tool

The dashboard built for demonstrating encapsulates all the functionalities in all the widgets described so far. As can be seen from Figure there is a map and its associated selector, a pie chart, a bar series, a radar series, and each of the latter three widgets is associated with a curved lines.

https://www.snap4city.org/dashboardSmartCity/view/Gea.php?iddasboard=MzcyNA==

In this dashboard, it is possible to view a number of categories of sensors scattered throughout the city of Florence and beyond, with various types of metrics within:

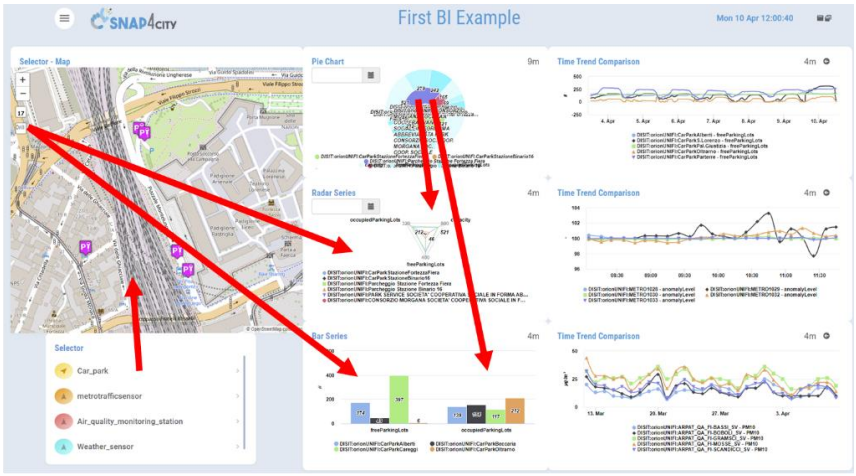
Sensori	Metriche
Metro Traffic Sensors	Anomaly Level Concentration Average Speed Average Time Congestion Level Vehicle Flow
Car Park	Capacity Occupied Parcking Lots Free Parcking Lots
Air Quality Monitoring Stations	PM10 PM2.5 Temperature Humidity
Weather Sensors	Temperature Humidity Wind

The possible interactions with this dashboard are:

- Sensor type selection, using the selector located below the map, several different types of sensors can be selected, including traffic, parking, air quality, and weather sensors.
- Spatial Drill Down, within the map there is a button that says "drill," with this it is possible to drill down with which real time data about the sensors included by the zoom will be shown in the three widgets next to it. This allows a visualization of the usual sensors in different ways and different groupings, the pie chart groups by single sensor, while the other two by metric type.
- Individual sensor selection, to analyze a single sensor of those correctly selected, a selection by marker click is possible on map or click on pie chart inner pie, both events will change the pie chart content, bar series and radar series, which will go to show only the considered sensor with all the metrics it presents.

1) Select the area of interest on map
2) Select the sensors kind of interest
3) Drill down on map

4) The JavaScript CSBL on Map will send data to the programmed Widgets. In this case, arrowed in RED



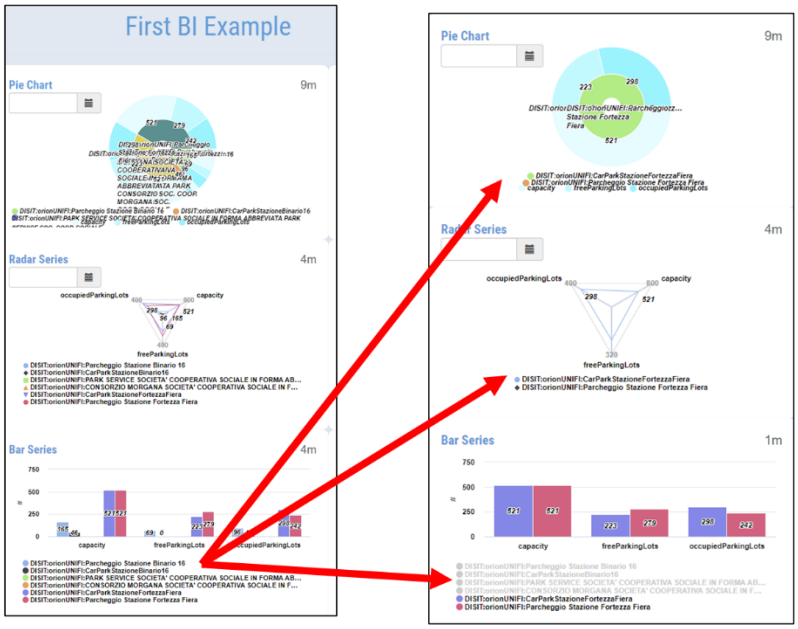
- Drill down time , through the 3 multiple time trends it is possible to specify a time window in which to analyze the trend of the selected metrics, all three trends align at the same start and end time.
- Single metric selection, to analyze a single metric of those currently selected a selection is possible by clicking on pie chart outer pie, bar series bar or single radar series element, these events show the trend of the selected metric in a certain time range of the trend immediately adjacent to the widget with which the selection was made.
- Unselect/Select Single Sensor, in a current display of a number of sensors it is possible to unselect some of them via the bar series and radar series legend, this will "turn off" the uninterested sensors of those present removing them from display in the other widgets. If a sensor has been turned off it will still remain displayed in the legend of the widget where the shutdown occurred, colored gray, it will then be possible to "turn it back on" and allow it to be displayed in the other widgets as well.

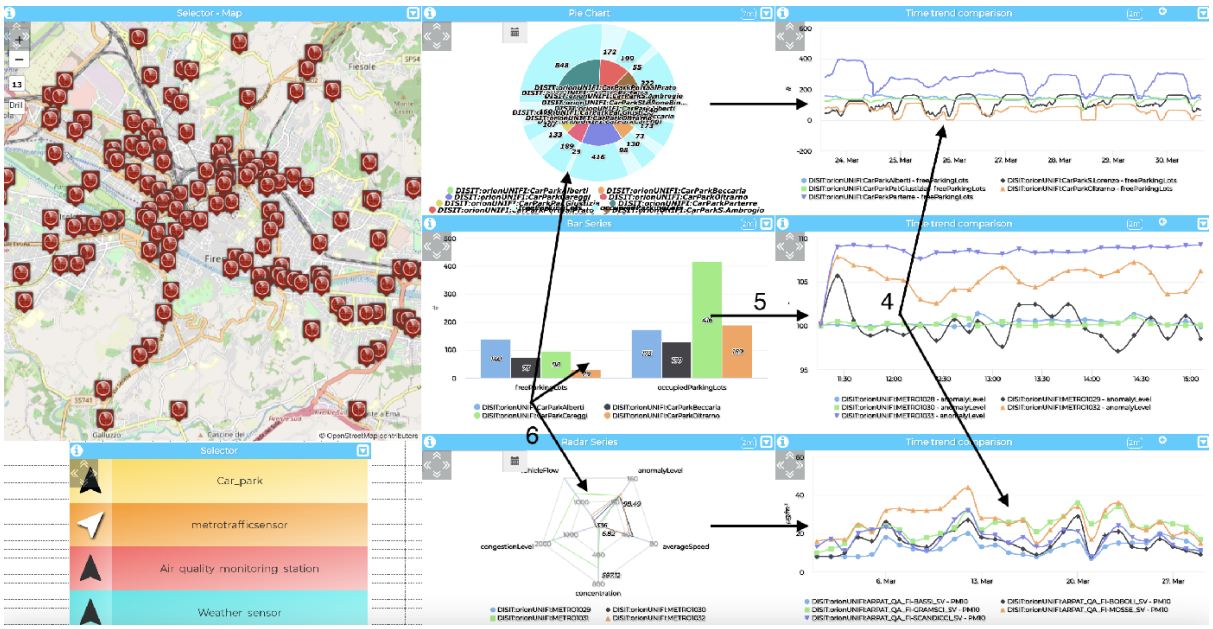
1) Click on the Donut element

2) The JavaScript CSBL on the Donut Widget will send commands to the programmed Widgets to focus on selection, as highlighted by the red arrows

1) Click on the Legenda of Bar Series

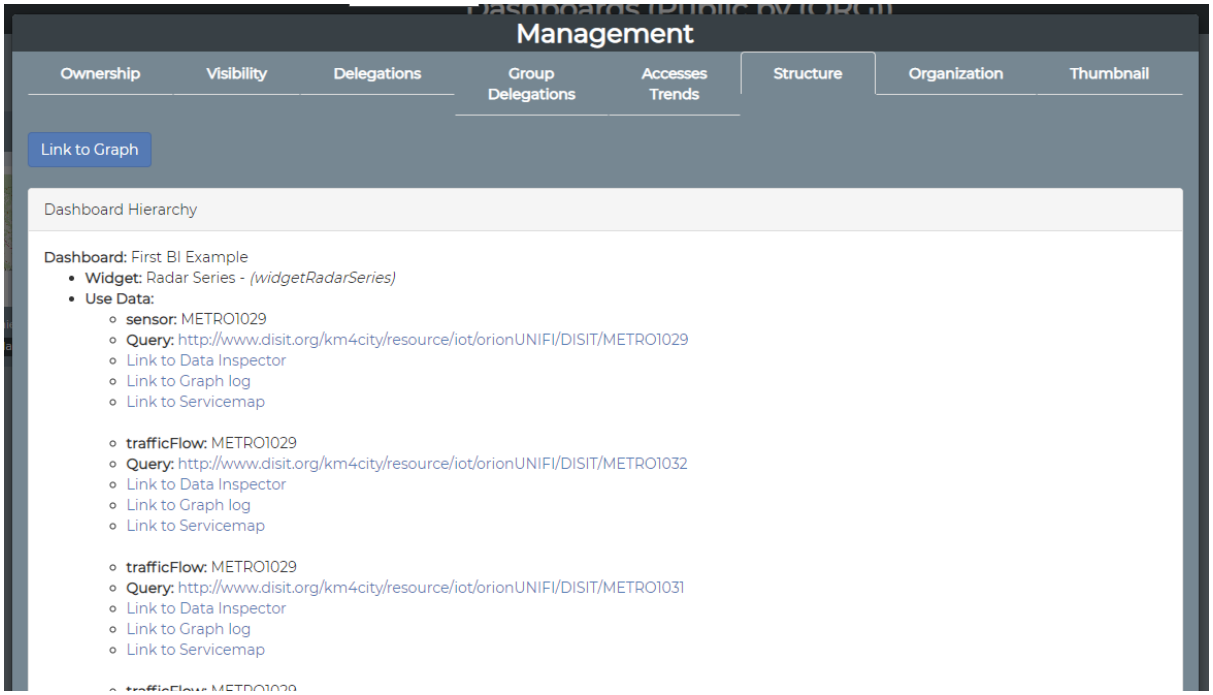
2) The JavaScript CSBL on the Bar Series will send commands to the programmed Widgets to remove the unselected devices, as highlighted by the red arrows





7.3 Dashboard Structure of Example X

To get the Dashboard structure please go on Dashboard Management menu.



Dashboard: First BI Example

- **Widget:** Radar Series - *(widgetRadarSeries)*
- **Use Data:**
  - **sensor:** METRO1029
    - **Query:** http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1029
    - Link to Data Inspector
    - Link to Graph log
    - Link to Servicemap
  - **trafficFlow:** METRO1029
    - **Query:** http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1032
    - Link to Data Inspector
    - Link to Graph log
    - Link to Servicemap
  - **trafficFlow:** METRO1029
    - **Query:** http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1031
    - Link to Data Inspector
    - Link to Graph log
    - Link to Servicemap
  - **trafficFlow:** METRO1029
    - **Query:** http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1030
    - Link to Data Inspector
    - Link to Graph log
    - Link to Servicemap
  - **sensor:** METRO1032
    - **Query:** http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1032

        - Link to Data Inspector
        - Link to Graph log
        - Link to Servicemap
- **Widget:** Selector - *(widgetSelectorNew)*
- **Use Data:**
    - **poi:** ?selection=43.64471;11.005751;43.89471;11.505751&categories=Car_park &maxResults=200&format=json
        - **Query:** https://servicemap.disit.org/WebAppGrafo/api/v1/?selection=43.64471...
        - Link to Data Inspector
        - Link to Graph log
        - Link to Servicemap
    - **poi:** ?selection=43.64471;11.005751;43.89471;11.505751&categories=Car_park &maxResults=200&format=json
        - **Query:** https://servicemap.disit.org/WebAppGrafo/api/v1/?selection=43.64471...
        - Link to Data Inspector
        - Link to Graph log
        - Link to Servicemap
    - **poi:** ?selection=43.64471;11.005751;43.89471;11.505751&categories=Car_park &maxResults=200&format=json
        - **Query:** https://servicemap.disit.org/WebAppGrafo/api/v1/?selection=43.64471...
        - Link to Data Inspector
        - Link to Graph log
        - Link to Servicemap
    - **poi:** ?selection=43.64471;11.005751;43.89471;11.505751&categories=Car_park &maxResults=200&format=json
        - **Query:** https://servicemap.disit.org/WebAppGrafo/api/v1/?selection=43.64471...
        - Link to Data Inspector
        - Link to Graph log
        - Link to Servicemap
- **Widget:** Time trend comparison - *(widgetCurvedLineSeries)*
- **Use Data:**
    - **sensor:** METRO1029
        - **Query:** http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1029
        - Link to Data Inspector
        - Link to Graph log
        - Link to Servicemap
    - **sensor:** METRO1028
        - **Query:** http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1028
        - Link to Data Inspector
        - Link to Graph log
        - Link to Servicemap
    - **sensor:** METRO1033
        - **Query:** http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1033
        - Link to Data Inspector
        - Link to Graph log
        - Link to Servicemap

    - **sensor:** METRO1032
        - **Query:** http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1032
        - Link to Data Inspector
        - Link to Graph log
        - Link to Servicemap
    - **sensor:** METRO1030
        - **Query:** http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1030
        - Link to Data Inspector
        - Link to Graph log
        - Link to Servicemap
- **Widget:** Time trend comparison - *(widgetCurvedLineSeries)*
- **Use Data:**
    - **sensor:** ARPAT_QA_FI-MOSSE_SV
        - **Query:** http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/ARPAT_QA...
        - Link to Data Inspector
        - Link to Graph log
        - Link to Servicemap
    - **sensor:** ARPAT_QA_FI-BASSI_SV
        - **Query:** http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/ARPAT_QA...
        - Link to Data Inspector
        - Link to Graph log
        - Link to Servicemap

ETC.

7.4 JavaScript on MultiDataMap of the Section 7.2 Example X: using a Business Intelligence tool

```
function execute(){
    var e = JSON.parse(param)
    sendData(e);
    function buildData(e){
        var carParkSensorsMetrics =
["capacity","freeParkingLots","occupiedParkingLots"];
        var airQualitySensorsMetrics =
["PM10","PM2_5","CO","Benzene","NO2"];
        var weatherSensorsMetrics = ["temperature", "wind",
"humidity"];
        var trafficSensorsMetrics =
["anomalyLevel","averageSpeed","avgTime","concentration","cong
estionLevel","vehicleFlow"];
        var metrics;
        var data = [];
        let h = 0;
        for (var l in e.layers) {
            if(e.layers[l].tipo == "Car_park"){
```

```
                metrics = carParkSensorsMetrics;
            }
            if(e.layers[l].tipo ==
"Air_quality_monitoring_station"){
                metrics = airQualitySensorsMetrics;
            }
            if(e.layers[l].tipo == "Weather_sensor"){
                metrics = weatherSensorsMetrics;
            }
            if(e.layers[l].tipo == "Traffic_sensor"){
                metrics = trafficSensorsMetrics;
            }
            for(var m in metrics){
                data[h] = {};
                data[h].metricId =
"https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri="
+ e.layers[l].serviceUri;
                data[h].metricHighLevelType = "Sensor";
                data[h].metricName = "DISIT:orionUNIFI:" +
e.layers[l].name;
                data[h].metricType = metrics[m];
                if(h<60) h++;
            }
        }
        return data;
    }
    function sendData(e){
        let data = buildData(e);
        $('body').trigger({
            type:
"showRadarSeriesFromExternalContent_w_AggregationSeries_3724_w
idgetRadarSeries35600",
            eventGenerator: $(this),
            targetWidget:
"w_AggregationSeries_3724_widgetRadarSeries35600",
            color1: "#f22011",
            color2: "#9c6b17",
            widgetTitle: "Air Quality IT from Impulse Button",
            passedData: data
        });
        $('body').trigger({
            type:
"showPieChartFromExternalContent_w_AggregationSeries_3724_widg
etPieChart35601",
            eventGenerator: $(this),
            targetWidget:
"w_AggregationSeries_3724_widgetPieChart35601",
            color1: "#f22011",
            color2: "#9c6b17",
            widgetTitle: "Air Quality IT from Impulse Button",
```

```
            passedData: data
        });
        $('body').trigger({
            type:
"showBarSeriesFromExternalContent_w_AggregationSeries_3724_wid
getBarSeries35604",
            eventGenerator: $(this),
            targetWidget:
"w_AggregationSeries_3724_widgetBarSeries35604",
            color1: "#f22011",
            color2: "#9c6b17",
            widgetTitle: "Air Quality IT from Impulse Button",
            passedData: data
        });
    }
}
```

Example of sending a command for heatmap rendering on MDM:

```
const heatMapName = "Rome_NO2_2_2";
            const targetWidgetName =
"w_Map_4252_widgetMap39805";
$('body').trigger({
                "type": "addHeatmap",
                    "target": targetWidgetName,
                    "passedData":
"https://wmsserver.snap4city.org/geoserver/Snap4City/wms?servi
ce=WMS&layers=" + heatMapName,
                    "passedParams": {
                        "desc": heatMapName,
                        "color1":"rgba(0,179,61,0)",
                        "color2":"rgba(114,235,133,1)"
                    }
            });
```

7.5 JavaScript on PieChart of the Section 7.2 Example: using a Business Intelligence tool

```
function execute() {
  console.log(param)
  function buildData(sensName, metricTypes){
    var data = [];
    let h = 0;
    for (let i = 0; i < (sensName.length); i++) {
        for (let j = 0; j < (metricTypes.length); j++) {
            data[h] = {};
            data[h].metricId =
"https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://
www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/" + sensName[i];
            data[h].metricHighLevelType = "Sensor";
```

```
                data[h].metricName = "DISIT:orionUNIFI:" + sensName[i];
                data[h].metricType = metricTypes[j];
                h++;
            }
        }
        return data;
    }
        var e = param;
        let sensName = [];
        let metricTypes = [];
        var s;
        for(let i=0; i < param.length; i++){
            s = param[i].metricName.replace("DISIT:orionUNIFI:", '');
            if(!sensName.includes(s)){
                sensName.push(s);
            }
            if(!metricTypes.includes(s)){
                metricTypes.push(param[i].metricType);
            }
        }
        let data = buildData(sensName, metricTypes);
        console.log(data);
        if(data.length > 1){
            $('body').trigger({
                type:
"showPieChartFromExternalContent_w_AggregationSeries_3724_widgetPieC
hart35601",
                eventGenerator: $(this),
                targetWidget:
"w_AggregationSeries_3724_widgetPieChart35601",
                color1: "#f22011",
                color2: "#9c6b17",
                widgetTitle: "Air Quality IT from Impulse Button",
                passedData: data
            });
            $('body').trigger({
                type:
"showRadarSeriesFromExternalContent_w_AggregationSeries_3724_widgetR
adarSeries35600",
                eventGenerator: $(this),
                targetWidget:
"w_AggregationSeries_3724_widgetRadarSeries35600",
                color1: "#f22011",
                color2: "#9c6b17",
                widgetTitle: "Air Quality IT from Impulse Button",
                passedData:   data
            });
            $('body').trigger({
                type:
"showBarSeriesFromExternalContent_w_AggregationSeries_3724_widgetBar
Series35604",
                eventGenerator: $(this),
                targetWidget:
"w_AggregationSeries_3724_widgetBarSeries35604",
```

```
            color1: "#f22011",
            color2: "#9c6b17",
            widgetTitle: "Air Quality IT from Impulse Button",
            passedData:  data
        });
    }
    if(data.length == 1){
        data = [];
        data[0]={};
        data[0].metricId =
"http://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://w
ww.disit.org/km4city/resource/iot/orionUNIFI/DISIT/"+
e[0].metricName.slice(17, e[0].metricName.length) + "&format=json";
        data[0].metricHighLevelType = "Sensor";
        data[0].metricName = "DISIT:orionUNIFI:" +
e[0].metricName.slice(17,e[0].metricName.length);
        data[0].smField = e[0].metricType;
        data[0].serviceUri =
"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/"+
e[0].metricName.slice(17,e[0].metricName.length);
        console.log(data);
        $('body').trigger({
            type:
"showCurvedLinesFromExternalContent_w_AggregationSeries_3724_widgetC
urvedLineSeries35605",
            eventGenerator: $(this),
            targetWidget:
"w_AggregationSeries_3724_widgetCurvedLineSeries35605",
            range: "7/DAY",
            color1: "#9b93ed",
            color2: "#231d5c",
            widgetTitle: "Occupied Parking Lots (Alberti Car Park)
from Impulse Button",
            field: data.smField,
            passedData: data,
        });
    }
}
```

7.6 JavaScript on RadarSeries the Section 7.2 Example X: using a Business Intelligence tool

```
function execute(){
    var e = JSON.parse(param);
    if(e.event == "click"){
        let serviceUri =
"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/" +
e.value.metricName.slice(17,e.value.metricName.length);
        var data = [];
        data[0]={};
        data[0].metricId =
"http://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://w
ww.disit.org/km4city/resource/iot/orionUNIFI/DISIT/"+
```

```
e.value.metricName.slice(17,e.value.metricName.length) +
"&format=json";
        data[0].metricHighLevelType = "Sensor";
        data[0].metricName = "DISIT:orionUNIFI:" +
e.value.metricName.slice(17,e.value.metricName.length);
        data[0].smField = e.value.metricType;
        data[0].serviceUri = serviceUri;
        $('body').trigger({
            type:
"showCurvedLinesFromExternalContent_w_AggregationSeries_3724_widgetC
urvedLineSeries35609",
            eventGenerator: $(this),
            targetWidget:
"w_AggregationSeries_3724_widgetCurvedLineSeries35609",
            range: "7/DAY",
            color1: "#9b93ed",
            color2: "#231d5c",
            widgetTitle: "Occupied Parking Lots (Alberti Car Park)
from Impulse Button",
            field: data.smField,
            passedData: data
        });
    }
    if(e.event == "legendItemClick"){
        sendData(e);
        function buildData(e){
            var carParkSensorsMetrics =
["capacity","freeParkingLots","occupiedParkingLots"];
            var airQualitySensorsMetrics =
["PM10","PM2_5","CO","Benzene","NO2"];
            var weatherSensorsMetrics = ["temperature", "wind",
"humidity"];
            var trafficSensorsMetrics =
["anomalyLevel","averageSpeed","avgTime","concentration","congestion
Level","vehicleFlow"];
            var metrics;
            var data = [];
            let h = 0;
            for (var l in e.layers) {
                if(e.layers[l].visible == true){
                    e.layers[l].name =
e.layers[l].name.slice(17,e.layers[l].name.length);
                    if(e.layers[l].tipo == "Car_park"){
                        metrics = carParkSensorsMetrics;
                    }
                    if(e.layers[l].tipo ==
"Air_quality_monitoring_station"){
                        metrics = airQualitySensorsMetrics;
                    }
                    if(e.layers[l].tipo == "Weather_sensor"){
                        metrics = weatherSensorsMetrics;
                    }
                    if(e.layers[l].tipo == "Traffic_sensor"){
                        metrics = trafficSensorsMetrics;
```

```
                    }
                    for(var m in e.metrics){
                        data[h] = {};
                        data[h].metricId =
"https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://
www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/"+
e.layers[l].name;
                        data[h].metricHighLevelType = "Sensor";
                        data[h].metricName = "DISIT:orionUNIFI:" +
e.layers[l].name;
                        data[h].metricType = e.metrics[m];
                        h++;
                    }
                }
            }
            console.log(data);
            return data;
        }
        function sendData(e){
            let data = buildData(e);
            $('body').trigger({
                type:
"showBarSeriesFromExternalContent_w_AggregationSeries_3724_widgetBar
Series35604",
                eventGenerator: $(this),
                targetWidget:
"w_AggregationSeries_3724_widgetBarSeries35604",
                color1: "#f22011",
                color2: "#9c6b17",
                widgetTitle: "Air Quality IT from Impulse Button",
                passedData: data
            });
            $('body').trigger({
                type:
"showPieChartFromExternalContent_w_AggregationSeries_3724_widgetPieC
hart35601",
                eventGenerator: $(this),
                targetWidget:
"w_AggregationSeries_3724_widgetPieChart35601",
                color1: "#f22011",
                color2: "#9c6b17",
                widgetTitle: "Air Quality IT from Impulse Button",
                passedData: data
            });
        }
    }
}
```

7.7 JavaScript on BarSeries of the Section 7.2 Example X: using a Business Intelligence tool

```
function execute(){
    var e = JSON.parse(param);
```

```
    console.log(e);
    if(e.event == "click"){
        let serviceUri =
"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/" +
e.value.metricName.slice(17,e.value.metricName.length);
        let smField = e.value.metricType;
        var data = [];
        data[0]={};
        data[0].metricId =
"http://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://w
ww.disit.org/km4city/resource/iot/orionUNIFI/DISIT/"+
e.value.metricName.slice(17,e.value.metricName.length) +
"&format=json";
        data[0].metricHighLevelType = "Sensor";
        data[0].metricName = "DISIT:orionUNIFI:" +
e.value.metricName.slice(17,e.value.metricName.length);
        data[0].smField = e.value.metricType;
        data[0].serviceUri = serviceUri;
        $('body').trigger({
            type:
"showCurvedLinesFromExternalContent_w_AggregationSeries_3724_widgetC
urvedLineSeries35608",
            eventGenerator: $(this),
            targetWidget:
"w_AggregationSeries_3724_widgetCurvedLineSeries35608",
            range: "7/DAY",
            color1: "#9b93ed",
            color2: "#231d5c",
            widgetTitle: "Occupied Parking Lots (Alberti Car Park)
from Impulse Button",
            field: data.smField,
            passedData: data
        });
    }
    if(e.event == "legendItemClick"){
        sendData(e);
        function buildData(e){
            var carParkSensorsMetrics =
["capacity","freeParkingLots","occupiedParkingLots"];
            var airQualitySensorsMetrics =
["PM10","PM2_5","CO","Benzene","NO2"];
            var weatherSensorsMetrics = ["temperature", "wind",
"humidity"];
            var trafficSensorsMetrics =
["anomalyLevel","averageSpeed","avgTime","concentration","congestion
Level","vehicleFlow"];
            var metrics;
            var data = [];
            let h = 0;
            console.log(e);
            for (var l in e.layers) {
                if(e.layers[l].visible == true){
                    e.layers[l].name =
e.layers[l].name.slice(17,e.layers[l].name.length);
```

```
                    if(e.layers[l].tipo == "Car_park"){
                        metrics = carParkSensorsMetrics;
                    }
                    if(e.layers[l].tipo ==
"Air_quality_monitoring_station"){
                        metrics = airQualitySensorsMetrics;
                    }
                    if(e.layers[l].tipo == "Weather_sensor"){
                        metrics = weatherSensorsMetrics;
                    }
                    if(e.layers[l].tipo == "Traffic_sensor"){
                        metrics = trafficSensorsMetrics;
                    }
                    for(var m in e.metrics){
                        data[h] = {};
                        data[h].metricId =
"https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=http://
www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/"+
e.layers[l].name;
                        data[h].metricHighLevelType = "Sensor";
                        data[h].metricName = "DISIT:orionUNIFI:" +
e.layers[l].name;
                        data[h].metricType = e.metrics[m];
                        h++;
                    }
                }
            }
            return data;
        }
        function sendData(e){
            let data = buildData(e);
            $('body').trigger({
                type:
"showRadarSeriesFromExternalContent_w_AggregationSeries_3724_widgetR
adarSeries35600",
                eventGenerator: $(this),
                targetWidget:
"w_AggregationSeries_3724_widgetRadarSeries35600",
                color1: "#f22011",
                color2: "#9c6b17",
                widgetTitle: "Air Quality IT from Impulse Button",
                passedData: data
            });
            $('body').trigger({
                type:
"showPieChartFromExternalContent_w_AggregationSeries_3724_widgetPieC
hart35601",
                eventGenerator: $(this),
                targetWidget:
"w_AggregationSeries_3724_widgetPieChart35601",
                color1: "#f22011",
                color2: "#9c6b17",
                widgetTitle: "Air Quality IT from Impulse Button",
                passedData: data
```

```
            });
        }
    }
}
```

7.8 JavaScript on 1st Time trend comparison of the Section 6.2 Example X: using a Business Intelligence tool

```
function execute() {
    var newParam = {};
    newParam.t1 = param['t1'];
    newParam.t2 = param['t2'];
    newParam.metricName = param.series[0].smField;
    newParam.sUri = param.series[0].serviceUri;
    param = newParam;
    let minT, maxT = null;
    if (param['t1'] != param['t2']) {
        minT = param['t1'];
        maxT = param['t2'];
    }
    else {
        minT = param['t1']-10000000;
        maxT = param['t2']+10000000;
    }
    let dt1 = new Date(minT);
    let dt1_iso = dt1.toISOString().split(".")[0];
    let dt2 = new Date(maxT);
    let dt2_iso = dt2.toISOString().split(".")[0];

     $('body').trigger({
        type:
"showCurvedLinesFromExternalContent_w_AggregationSeries_3724_widgetC
urvedLineSeries35696",
        eventGenerator: $(this),
        targetWidget:
"w_AggregationSeries_3724_widgetCurvedLineSeries35696",
        range: "7/DAY",
        color1: "#9b93ed",
        color2: "#231d5c",
        widgetTitle: "Occupied Parking Lots (Alberti Car Park) from
Impulse Button",
        t1: dt1_iso,
        t2: dt2_iso,
    });
    $('body').trigger({
        type:
"showCurvedLinesFromExternalContent_w_AggregationSeries_3724_widgetC
urvedLineSeries35608",
        eventGenerator: $(this),
        targetWidget:
"w_AggregationSeries_3724_widgetCurvedLineSeries35608",
        range: "7/DAY",
```

```
        color1: "#9b93ed",
        color2: "#231d5c",
        widgetTitle: "Occupied Parking Lots (Alberti Car Park) from
Impulse Button",
        t1: dt1_iso,
        t2: dt2_iso,
    });
    $('body').trigger({
        type:
"showCurvedLinesFromExternalContent_w_AggregationSeries_3724_widgetC
urvedLineSeries35609",
        eventGenerator: $(this),
        targetWidget:
"w_AggregationSeries_3724_widgetCurvedLineSeries35609",
        range: "7/DAY",
        color1: "#9b93ed",
        color2: "#231d5c",
        widgetTitle: "Occupied Parking Lots (Alberti Car Park) from
Impulse Button",
        t1: dt1_iso,
        t2: dt2_iso,
    });
}
```

7.9 JavaScript on 2nd Time trend comparison of the Section 7.2 Example X: using a Business Intelligence tool

```
function execute() {
    var newParam = {};
    newParam.t1 = param['t1'];
    newParam.t2 = param['t2'];
    newParam.metricName = param.series[0].smField;
    newParam.sUri = param.series[0].serviceUri;
    param = newParam;
    let minT, maxT = null;
    if (param['t1'] != param['t2']) {
        minT = param['t1'];
        maxT = param['t2'];
    }
    else {
        minT = param['t1']-10000000;
        maxT = param['t2']+10000000;
    }
    let dt1 = new Date(minT);
    let dt1_iso = dt1.toISOString().split(".")[0];
    let dt2 = new Date(maxT);
    let dt2_iso = dt2.toISOString().split(".")[0];

     $('body').trigger({
        type:
"showCurvedLinesFromExternalContent_w_AggregationSeries_3724_widgetC
urvedLineSeries35696",
        eventGenerator: $(this),
```

```
        targetWidget:
"w_AggregationSeries_3724_widgetCurvedLineSeries35696",
        range: "7/DAY",
        color1: "#9b93ed",
        color2: "#231d5c",
        widgetTitle: "Occupied Parking Lots (Alberti Car Park) from
Impulse Button",
        t1: dt1_iso,
        t2: dt2_iso,
    });
    $('body').trigger({
        type:
"showCurvedLinesFromExternalContent_w_AggregationSeries_3724_widgetC
urvedLineSeries35608",
        eventGenerator: $(this),
        targetWidget:
"w_AggregationSeries_3724_widgetCurvedLineSeries35608",
        range: "7/DAY",
        color1: "#9b93ed",
        color2: "#231d5c",
        widgetTitle: "Occupied Parking Lots (Alberti Car Park) from
Impulse Button",
        t1: dt1_iso,
        t2: dt2_iso,
    });
    $('body').trigger({
        type:
"showCurvedLinesFromExternalContent_w_AggregationSeries_3724_widgetC
urvedLineSeries35609",
        eventGenerator: $(this),
        targetWidget:
"w_AggregationSeries_3724_widgetCurvedLineSeries35609",
        range: "7/DAY",
        color1: "#9b93ed",
        color2: "#231d5c",
        widgetTitle: "Occupied Parking Lots (Alberti Car Park) from
Impulse Button",
        t1: dt1_iso,
        t2: dt2_iso,
    });
}
```

7.10 JavaScript on 3rd Time trend comparison of the Section 7.2 Example X: using a Business Intelligence tool

```
function execute() {
    var newParam = {};
    newParam.t1 = param['t1'];
    newParam.t2 = param['t2'];
    newParam.metricName = param.series[0].smField;
    newParam.sUri = param.series[0].serviceUri;
    param = newParam;
    let minT, maxT = null;
    if (param['t1'] != param['t2']) {
        minT = param['t1'];
        maxT = param['t2'];
    }
    else {
        minT = param['t1']-10000000;
        maxT = param['t2']+10000000;
    }
    let dt1 = new Date(minT);
    let dt1_iso = dt1.toISOString().split(".")[0];
    let dt2 = new Date(maxT);
    let dt2_iso = dt2.toISOString().split(".")[0];

     $('body').trigger({
        type:
"showCurvedLinesFromExternalContent_w_AggregationSeries_3724_widgetC
urvedLineSeries35696",
        eventGenerator: $(this),
        targetWidget:
"w_AggregationSeries_3724_widgetCurvedLineSeries35696",
        range: "7/DAY",
        color1: "#9b93ed",
        color2: "#231d5c",
        widgetTitle: "Occupied Parking Lots (Alberti Car Park) from
Impulse Button",
        t1: dt1_iso,
        t2: dt2_iso,
    });
    $('body').trigger({
        type:
"showCurvedLinesFromExternalContent_w_AggregationSeries_3724_widgetC
urvedLineSeries35608",
        eventGenerator: $(this),
        targetWidget:
"w_AggregationSeries_3724_widgetCurvedLineSeries35608",
        range: "7/DAY",
        color1: "#9b93ed",
        color2: "#231d5c",
        widgetTitle: "Occupied Parking Lots (Alberti Car Park) from
Impulse Button",
        t1: dt1_iso,
        t2: dt2_iso,
```

```
    });
    $('body').trigger({
        type:
"showCurvedLinesFromExternalContent_w_AggregationSeries_3724_widgetC
urvedLineSeries35609",
        eventGenerator: $(this),
        targetWidget:
"w_AggregationSeries_3724_widgetCurvedLineSeries35609",
        range: "7/DAY",
        color1: "#9b93ed",
        color2: "#231d5c",
        widgetTitle: "Occupied Parking Lots (Alberti Car Park) from
Impulse Button",
        t1: dt1_iso,
        t2: dt2_iso,
    });
}
```

7.11 HTML/Javascript to build a Selector-Map scenario showing building shapes on map

In this example, a smart selector is built to show building shapes on a widget map. The shapes color depends on the values of the selected metrics (as described later). Provided that there has been created a specific colormap (through the ColorMap manager in the resource management tool) which should be named by concatenating the string "colormap" and the name of the specific metric (with uppercase first character).
The selector is made by exploiting CSBL inside a widgetExternalContent CK Editor. The selector interface is developed in HTML allowing, for instance, to select: (i) wether to show all the building shapes or a single building shape; (ii) which metrics use to evaluate the shapes color; (iii) wether to show an informative popup on shape click or not.
In the following code, these placeholders have been used:

<BASE_SERVICEURI_URL>: it is the base URL in the service URI representing the building virtual device (e.g.: http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/);

<POPUP_COLOR1>, <POPUP_COLOR2>: colors (in hex or rgb code) for the informative popup on shape click (if enabled);

<TARGET_WIDGET_MAP>: id of the target widget map which show the building shapes;

<TARGET_WIDGET_SINGLE_CONTENT>,<TARGET_WIDGET_TIME_TREND>: id of the target single content and time-trend widgets (if present) to show realtime results for the selected metric from the informative popup;

<SELECTED_METRICS_ARRAY_TO_FILTER>: array of strings representing the name of the desired metrics of the virtual devices to be shown in the informative popup (if enabled);

<DEVICE_MODEL_NAME>: name of the IoT model used to generate the building devices;

<DEVICE_SUBNATURE>: subnature of the building devices, according to the snap4city dictionary;

<MAP_BOUNDS_SW_LAT>: Latitude of South-West corner of desired map Bounding Box;
<MAP_BOUNDS_SW_LNG>: Longitude of South-West corner of desired map Bounding Box;
<MAP_BOUNDS_NE_LAT>: Latitude of North-East corner of desired map Bounding Box;
<MAP_BOUNDS_NE_LNG>: Longitude of North-East corner of desired map Bounding Box;

```
<!DOCTYPE html>
<html lang="en">
<head><script
src="https://www.snap4city.org/dashboardSmartCity/js/jquery-
1.10.1.min.js"></script></head>
<body><div class="form-container" style="display: flex; flex-
direction: column; align-items: flex-start; font-family:arial; font-
size: 16px;">
<div id="midDiv"><label for="modelInstance">All / Single
Device:</label>
    <select id="modelInstance">
      <option value="model">All</option>
      <option value="singleDevice">Single Device</option>
    </select>

<div id="deviceIdDiv" style="display: none;"><br><!-- Insert
deviceName for all deevices -->
      <label for="deviceId">Device ID:</label>
      <select id="deviceId">
        <option value="building2_27B">27B</option>
        <option value="building2_58A">58A</option>
        <option value="building2_100">100</option>
        <option value="building2_101">101</option>
        <option value="building2_102">102</option>
      </select>
    </div>

    </div><br><label id="variable"
for="metric">Variable:</label><select name="metric" id="metric"><!--
Insert device metrics -->
      <option value="capacity">capacity</option>
      <option value="allocation">allocation</option>
```

```
      <option value="occupancy"
selected="selected">occupancy</option>
    </select>

    <label>Popup on Shape Click</label><input type="checkbox"
id="popupCheckbox" checked><!—Check in order to display informative
popups on marker click -->

    <button type="button" id="addToMap"
style="cursor:pointer;background-color: lightblue;border-radius:
6px;">Add To Map</button>
  </div>

  <script>

    $(document).ready(function () {

        $("#modelInstance").change(function(){
            handleModelInstanceChange();
        });

        $("#popupCheckbox").change(function(){
            handlePopupCheckboxChange();
        });

        $("#addToMap").on("click", function() {
            handleAddBimShape();
        });


        function handleModelInstanceChange() {
         var modelInstanceSelect =
document.getElementById("modelInstance");
         var deviceIdDiv = document.getElementById("deviceIdDiv");

         if (modelInstanceSelect.value === "singleDevice") {
           deviceIdDiv.style.display = "block";
         } else {
           deviceIdDiv.style.display = "none";
         }
        }

        function handleAddBimShape() {
         var modelInstanceSelect =
document.getElementById("modelInstance");
         var deviceIdSelect = document.getElementById("deviceId");
         var metricSelect = document.getElementById("metric");
```

```
        var popupCheckbox =
document.getElementById("popupCheckbox");
        var altViewMode="";

        if (popupCheckbox.checked) {
          altViewMode="BimShapePopup";
        } else {
          altViewMode="BimShape";
        }

        params = {
          modelInstance: modelInstanceSelect.value,
          deviceId: deviceIdSelect.value,
          metric: metricSelect.value,
          altViewMode: altViewMode
        };
        triggerEvent("addBimShape", params);
       }

       document.addEventListener("DOMContentLoaded", function() {
        var metricSelect = document.getElementById("metric");
        var options = metricSelect.options;

        for (var i = 0; i < options.length; i++) {
          if (options[i].value === "occupancyPercentage") {
               options[i].selected = true;
               break;
          }
        }
       });

       function triggerEvent(event, params) {
        let baseServiceUri = <BASE_SERVICEURI_URL>;         // e.g.:
"http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/"
        let coordsAndType = {};
        let method = event;

        if (params.modelInstance && params.modelInstance ==
"singleDevice" && params.deviceId != null) {
          coordsAndType.query =
"../controllers/superservicemapProxy.php/api/v1/?serviceUri=" +
baseServiceUri + params.deviceId;
        } else {
          coordsAndType.query =
"../controllers/superservicemapProxy.php/api/v1/iot-
search/?selection=<MAP_BOUNDS_SW_LAT>;<MAP_BOUNDS_SW_LNG>;<MAP_BOUND
```

```
S_NE_LAT>;<MAP_BOUNDS_NE_LNG>&categories=" + categories +
"&format=json&maxResults=300&model=" + deviceModel;
            }
         coordsAndType.desc = params.metric;
         coordsAndType.color1 = <POPUP_COLOR1>;
                coordsAndType.color2 = <POPUP_COLOR2>;
         coordsAndType.targets =
<TARGET_WIDGET_SINGLE_CONTENT>,<TARGET_WIDGET_TIME_TREND>;
         coordsAndType.display = "pins";
         coordsAndType.queryType = "Default";
         coordsAndType.iconTextMode = "text";
         coordsAndType.altViewMode = params.altViewMode;
         coordsAndType.bubbleSelectedMetric = params.metric;
         coordsAndType.modelInstance = params.modelInstance;

         let selectedMetrics = <SELECTED_METRICS_ARRAY_TO_FILTER>
         // e.g.: ["capacity", "allocation", "occupancy"]
         coordsAndType.selectedMetrics = selectedMetrics;

         parent.$('body').trigger({
            type: event,
            target: <TARGET_WIDGET_MAP>,
            passedData: coordsAndType
         });
        }

        console.log("Smart Selector CSBL");
        var deviceModel = <DEVICE_MODEL_NAME>;
        var categories = <DEVICE_SUBNATURE>;

        // AUTO-START
        setTimeout(function() {
            handleAddBimShape();
        }, 800);
    });

  </script>
</body>
</html>
```

Note that in the case of External content one has to use:

- parent.$('body').trigger({...

insteat of

- $('body').trigger({...

7.12 Get Access Token

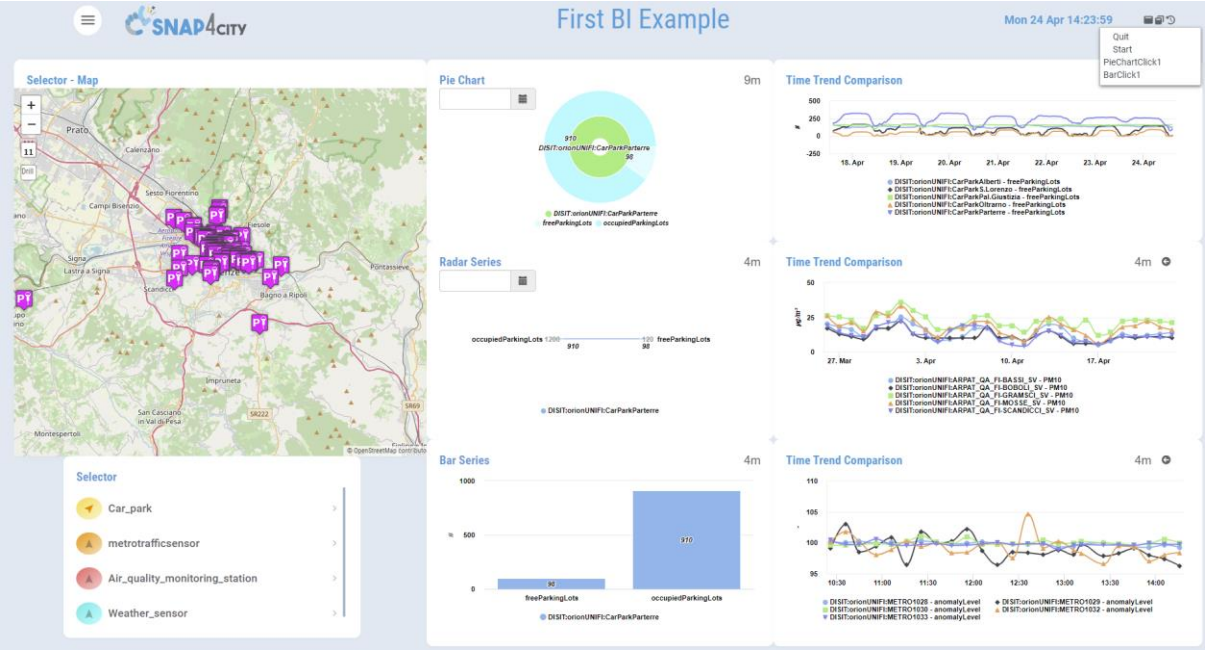
Get Access Token from External Content and from CSBL.

In most Widgets a relative URL can be used “../controllers/getAccessToken.php”

```
async function fetchAccessToken() {
        try {
            const response = await $.ajax({
                url:
"https://www.snap4city.org/dashboardSmartCity/controllers/getAccessToken.php",
                type: "GET",
                dataType: 'json'
            });
            return response.accessToken;
        } catch (error) {
            console.error("Error fetching access token", error);
            throw error;
        }
    }
```

8 - Time Machine, Undo Stack for business intelligence applications

On the dashboard a stack of commands is created at each action. The stack is shown to the user on the top right corner or left corner depending on the stile.



The stack presents the list of commands performed and in particular the actions performed on the widgets.

Top left menu of the dashboard to drill up, each row identifies a past action performed on a certain widget, each of these determine a certain view on the data.

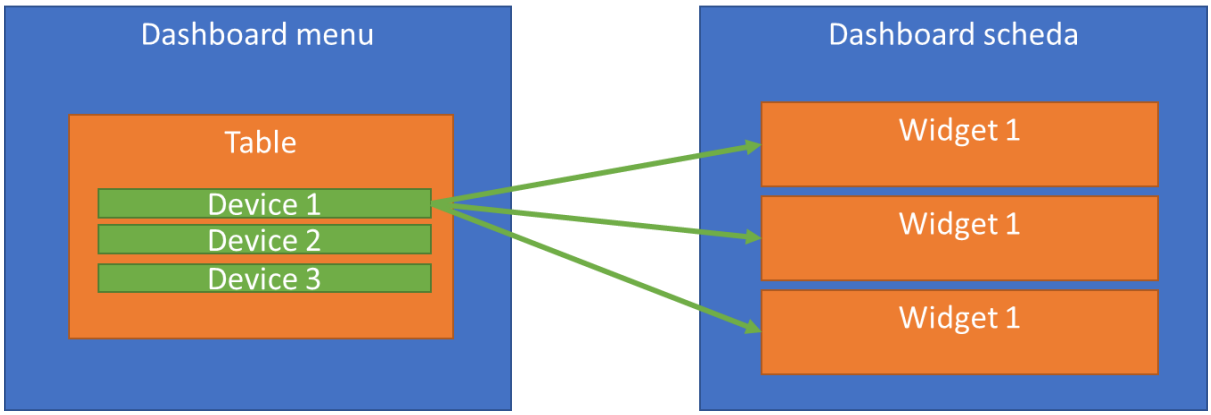
With the menu in the upper left/right corner you can force the dashboard to return at a previously displayed view. Each state of the dashboard is identified by the data that the widgets are displaying, with each action performed these changes in a certain way that depends on the specific action performed and on which widget, so each action identifies a certain state of the dashboard, by clicking on a certain state among those shown on the drop down menu it is possible to reload widgets to show data related to that state, also allowing this sequence to be navigated in either direction.

9 - CSBL JavaScript Library functions for dashboard interaction

It is assumed that you need to create a system of interfaces (dashboard) that allow the following interaction:

Following an event launched by a widget in dashboard A, dashboard B opens and is given the parameters necessary for a widget in dashboard B to retrieve data and display it.

For example, we could have a menu dashboard, with a table that will list certain devices (e.g., vehicles, batteries, users, etc.) and a card dashboard in which a set of widgets (time trends, bar plots, pie charts, etc.)) will show the data collected for a specific device – see figure.
By clicking on a row of the table in the dashboard menu, the dashboard card will open and each widget will independently retrieve the data to be displayed.



To allow this interaction, the following two library functions have been implemented:

- **open New Dashboard**
- **getParams**

Note: Functions are automatically included in widgets, except for externalContent. In that case you need to explicitly specify the source of the JavaScript file to include, e.g.,

```
<script type='text/javascript'
src='https://www.snap4city.org/dashboardSmartCity/js/widgetsCommonFunctions.js'>
</script>
```

taking care to specify the absolute path. The above included Javascript file contains a CSBL function library, including pre-defined functions of the Visual CSBL Editor, as well as the pre-defined functions that are used in the following examples:

9.1 Function: openNewDashboard

The function is included in the CSBL pre-defined function library, and it can be called in the CK Editor of the More Option panel of a widget, has the following definition:

```
function openNewDashboard(url, target)
```

Input:

- **url:** url of the dashboard to open, any parameters to be transmitted can be added in the queue as GET, for example, if we wanted to send the ServiceUri and the model of a device we would use https://www.snap4city.org/dashboardSmartCity/view/index.php?iddasboard=MzY3NA==&suri=' + <serviceUri> + '&model=' + <model>

- **target:** how to open a new dashboard, e.g. _parent, _blank, etc.

Outputs: none
Effect: Opens a new page using the specified url and target

9.2 Function: getParams

The function is included in the CSBL pre-defined function library, and it can be called in the CK Editor of the More Option panel of a widget, has the following definition

```
function getParams(isIFrame = false)
```

Input:

- **isIFrame** (default false): specifies whether the widget implementing the function is loaded in the dashboard as an I-Frame (e.g., externalContent) or not.

Outputs:

- A JSON string containing the GET parameters retrieved from the url of the same dashboard

Example interaction
Supposing to have

- An impulse-button in the dashboard A
- A pie chart in the dashboard B

By clicking on the button in dashboard A, dashboard B will open and the pie-chart will retrieve the data of a specified device and then display them

Code for actuator (e.g. impulse button)

```
function execute() {
   let suri =
'http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/prettoTestBattery03';
   let device = 'prettoBatteryTest03';
```

```
   let url =
'https://www.snap4city.org/dashboardSmartCity/view/index.php?iddasboard=MzY3NA==&s
uri=' + suri + '&device=' + device;
   let target = '_blank';
   openNewDashboard(url, target);
}

function execute() {
    let suri =
'http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1030';
    let model = 'metrotrafficsensor';

    let device = 'METRO1030';
    let url =
'https://www.snap4city.org/dashboardSmartCity/view/index.php?iddasboard=NDIzMA==&s
uri=' +  suri + '&model=' + model + '&device=' +device;
    let target = '_blank';

    openNewDashboard(url, target);
}
```

Note: call to function openNewDashboard must be included in the execute function as it must be invoked when an event is triggered by the actuating widget (e.g., the click on the button, or the selection of a row in the widgetDeviceTable, etc.)

When the button on dashboard A is clicked, dashboard B will open with the following url

https://www.snap4city.org/dashboardSmartCity/view/index.php?iddasboard=NDIzMA==&suri=http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1030&model=metrotrafficsensor&device=METRO1030

Then, on dashboard B, a pie-chart will run the following code to fetch the data of the selected device and then display it.

Receiver code (e.g. pie-chart)

```
let params = JSON.parse(getParams());
if(params.suri && params.device)  {
    let suri = params.suri;
    let device = params.device;

    let passedData= [

        {
            "metricId":
'http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/' + device,
            "metricHighLevelType": 'IoT Device Variable',
            "metricName": device,
            "metricType":'vehicleFlow',
            "serviceUri": suri,
```

```
        },

        {
            "metricId":
'http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/' + device,
            "metricHighLevelType": 'IoT Device Variable',
            "metricName": device,
            "metricType": 'averageSpeed',
            "serviceUri": suri,
        },

        {
            "metricId":
'http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/' + device,
            "metricHighLevelType": 'IoT Device Variable',
            "metricName": device,
            "metricType": 'concentration',
            "serviceUri": suri,
        },

    ];

    setTimeout(function() {

        $('body').trigger({
            type:
"showPieChartFromExternalContent_w_AggregationSeries_4230_widgetPieChart39612",
            eventGenerator: $(this),
            targetWidget: "w_AggregationSeries_4230_widgetPieChart39612",
            color1: "#e8a023",
            color2: "#9c6b17",
            widgetTitle: "Traffic Device Data",
            passedData: passedData
        })

    }, 750);

}
```

Note: to allow the widget to load data, you need to impose a delay on *$('body').trigger()* using for example *setTimeout().* Otherwise the widget will hang and fail to load the data.

9.3 Function: composeSURI(baseUrl, parameterName)

This function is included in the CSBL pre-defined function library, so it can be called directly in the JavaScript code in the CK editor. This function composes the service URI of a virtual

device/entity by the concatenation of a baseURL, <BASE_SERVICEURI_ULR> (e.g.: "http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/"), and a string, <PARAMETER_NAME>, retrieved as a GET parameter from the dashboard URL.

9.4 Function: fetchAjax

Function: var response = fetchAjax(<API_URL>, <DATA_OBJ>, <METHOD>, <DATA_TYPE>, <ASYNC>, <TIMEOUT_VAL>)

This function is included in a global JavaScript library, so it can be called directly in the JavaScript code in the CK editor. This function makes an ajax call passing the input parameters as in the following:

```
$.ajax({
        url: <API_URL>,
        data: <DATA_OBJ>,
        type: <METHOD>,      // e.g. “GET”
        dataType: <DATA_TYPE>,       // 'json'
        async: true,            // asyncFlag
        timeout: timeoutVal
    })
```

It can be useful, for instance, to retrieve IoT device/entities data calling the Sanp4City Smart City API.
For example:

```
var response = fetchAjax("var apiUrl =
"../controllers/superservicemapProxy.php/api/v1/?serviceUri=
http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/METRO1030", null, "GET",
'json', true, 0);
```

The response data (`responseData`) is returned as in the following example:

```
response.done(function(responseData) {
             if (responseData.error == null && responseData.failure == null) {

                    // ...Do your CSBL...

             } else {
                    console.log("Error in retrieving data.");
                    console.log(JSON.stringify(responseData));
             }
      });
```

9.5 Function: triggerMetricsForTrends

Function: triggerMetricsForTrends (<WIDGET_ID>, <LISTENER_NAME>, <JSON_DATA>, <SELECTED_METRICS>, <BASE_KB_URL>, <LEGEND_ENTITY_NAME>, <TIME_RANGE>, <LEGEND_LABELS>)

This function is included in the CSBL pre-defined function library, so it can be called directly in the JavaScript code in the CK editor. This function prepares the required data for triggering the visualization in the target widget (having id **<WIDGET_ID>**), and triggered the prepared data by calling the widget specific listener (represented by **<LISTENER_NAME>**). The data preparation is applied to a json object (**<JSON_DATA>**) which is typically IoT device/entity data retrieved by calling the Snap4City API (for instance, through the above described fetchAjax function). To this aim, the base URL for querying the specific knowledge base has to be passed as a parameter (**<BASE_KB_URL>**, e.g.: https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=). It is possible to select a subset of the IoT device/entity metrics to be sent for visualization, specifying their name in the **<SELECTED_METRICS>** array of strings. It this parameter is null or undefined, all the IoT device/entity metrics values will be displayed.

Other parameters (optional, not required):

<LEGEND_ENTITY_NAME>: array of strings specifying the name of the devices to be represented in widgetCurvedLineSeries legend.

<LEGEND_LABELS>: array of strings specifying custom strings to be represented as metric labels in widgetCurvedLineSeries legend. It must follow the same oreder of the metrics in the **<SELECTED_METRICS>**, if present.

<TIME_RANGE>: specify the visualization time-range, useful for widgetTimeTrendCompare. It can be one of the following: "4/HOUR", "1/DAY", "7/DAY", "30/DAY", "180/DAY", "365/DAY".

9.6 Example

The following example use the above defined functions **composeSURI**, **fetchAjax** and **triggerMetricsForTrends**. The following JavaScript code is not included in the typical execute() function, thus showing another possible use of the CSBL, for instance allowing the execution of JavaScript code directly on dashboard loading. In this case, the following code can be placed in any of the widgets described in section 6 in order to dynamically show data from any virtual device/entity automatically on dashboard load, provided that the device name is passed as a GET parameter in the dashboard (e.g. using the "entityId" GET parameter: https://www.snap4city.org/dashboardSmartCity/view/Gea-Night.php?iddasboard=Mzk4Mg==&entityId=building2_33B). This is useful when you want to use a single dashboard to show data from many different entities by simply passing the device or entity name as a GET parameter in the dashboard URL.

```
var baseUrl = <BASE_SERVICEURI_ULR>;
```

```
var serviceUri = composeSURI(baseUrl, <PARAMETER_NAME>);
if (serviceUri != null) {
	var apiUrl = "../controllers/superservicemapProxy.php/api/v1/?serviceUri="
+ serviceUri;
	var getSmartCityAPIData = fetchAjax(apiUrl, null, "GET", 'json', true, 0);
	getSmartCityAPIData.done(function(jsonData) {
		if (jsonData.error == null && jsonData.failure == null) {
			var selectedMetrics = <SELECTED_METRICS>;
			var legendEntityName = serviceUri.split("_")[1];
			var widgetId = <WIDGET_ID>;
			var baseKbUrl =
"https://servicemap.disit.org/WebAppGrafo/api/v1/?serviceUri=";
			var listenerName = <LISTENER_NAME> + widgetId;
			triggerMetricsForTrends(widgetId, listenerName, jsonData,
selectedMetrics, baseKbUrl, legendEntityName, <TIME_RANGE>, <LEGEND_LABELS>);

		} else {
			console.log("Error in retrieving data from ServiceMap.");
			console.log(JSON.stringify(geoJsonData));
		}
	});
}
```

<BASE_SERVICEURI_ULR>: it is the base URL in the service URI representing the building virtual device (e.g.: http://www.disit.org/km4city/resource/iot/orionUNIFI/DISIT/);

<PARAMETER_SAME>

<SELECTED_METRICS >: array of strings representing the name of the desired metrics of the virtual device to be shown in the widget having id <WIDGET_ID>;

<LISTENER_NAME>: name of the specific widget listener (depending on the <WIDGET_ID>, see seciont 5 for details, e.g.: "showBarSeriesFromExternalContent_" for widgetBarSeries);

NOTE: When building such a scenario, as described in the above example, it is convenient to set "Show CSBL Content on Load": "no" in the widget <WIDGET_ID> more options tab, in order to avoid possible overlapping effects if this widget has default metrics to be shown (for instance, when the dashboard is loaded without entityId parameter in the URL).

10. Selecting DateTime start point in Widgets (so called Calendar Button)

By default, widgets display the values of their metrics based on the most recent data. Some widgets have a feature that allows them to show the values at a specific time, which can be selected from a special input box shown in the widget.

Users can dynamically change the content of the widget selecting a date and time, and it will be recalculated on the bases of the chosen values.

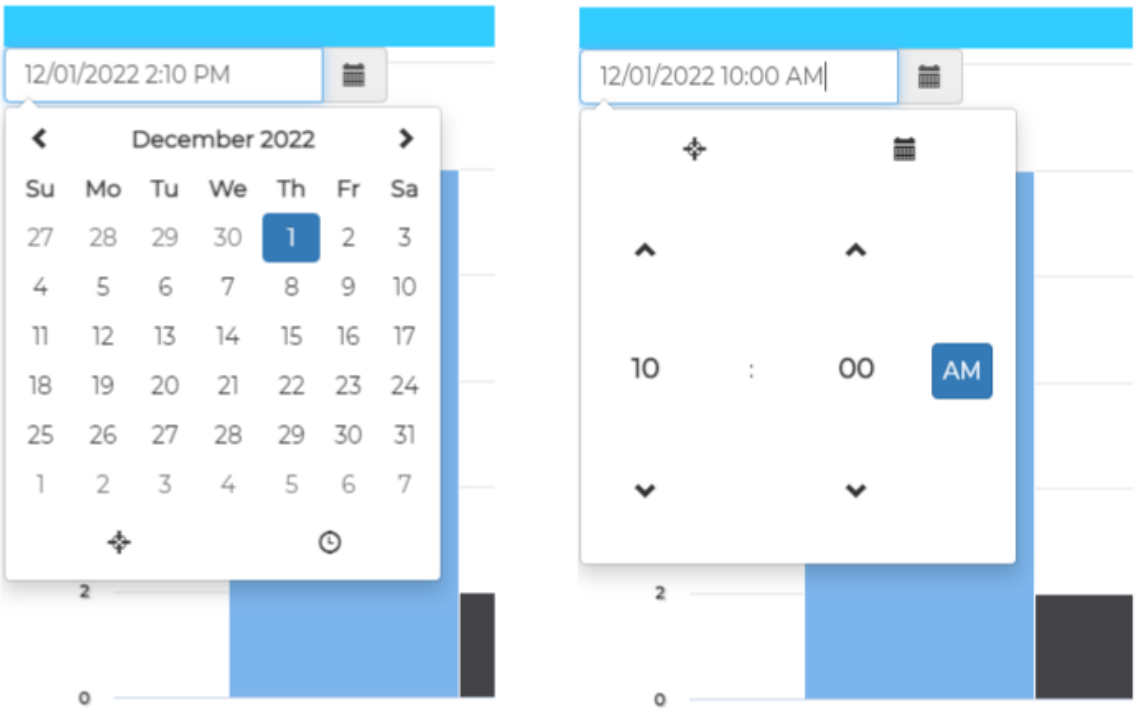


Figure 20: Selection Date and time from the calendar box.

These Widget Types at the moment are the follows:

- *WidgetBarSeries*
- *WidgetPieChart*
- *WidgetRadarSeries*
- *WidgetCurvedLineSeries*

In *widgetRadarSeries* and *widgetPieChart*, the datetime input box is currently by default set as active and uneditable, while in *widgetCurvedLinesSeries* and in *widgetBarSeries* it is set as deactivated and can be activated in the more options menu in the editing mode of the dashboard.

Select "Yes" in the *Show Calendar Button* to activate the input datetime menu in the widget.

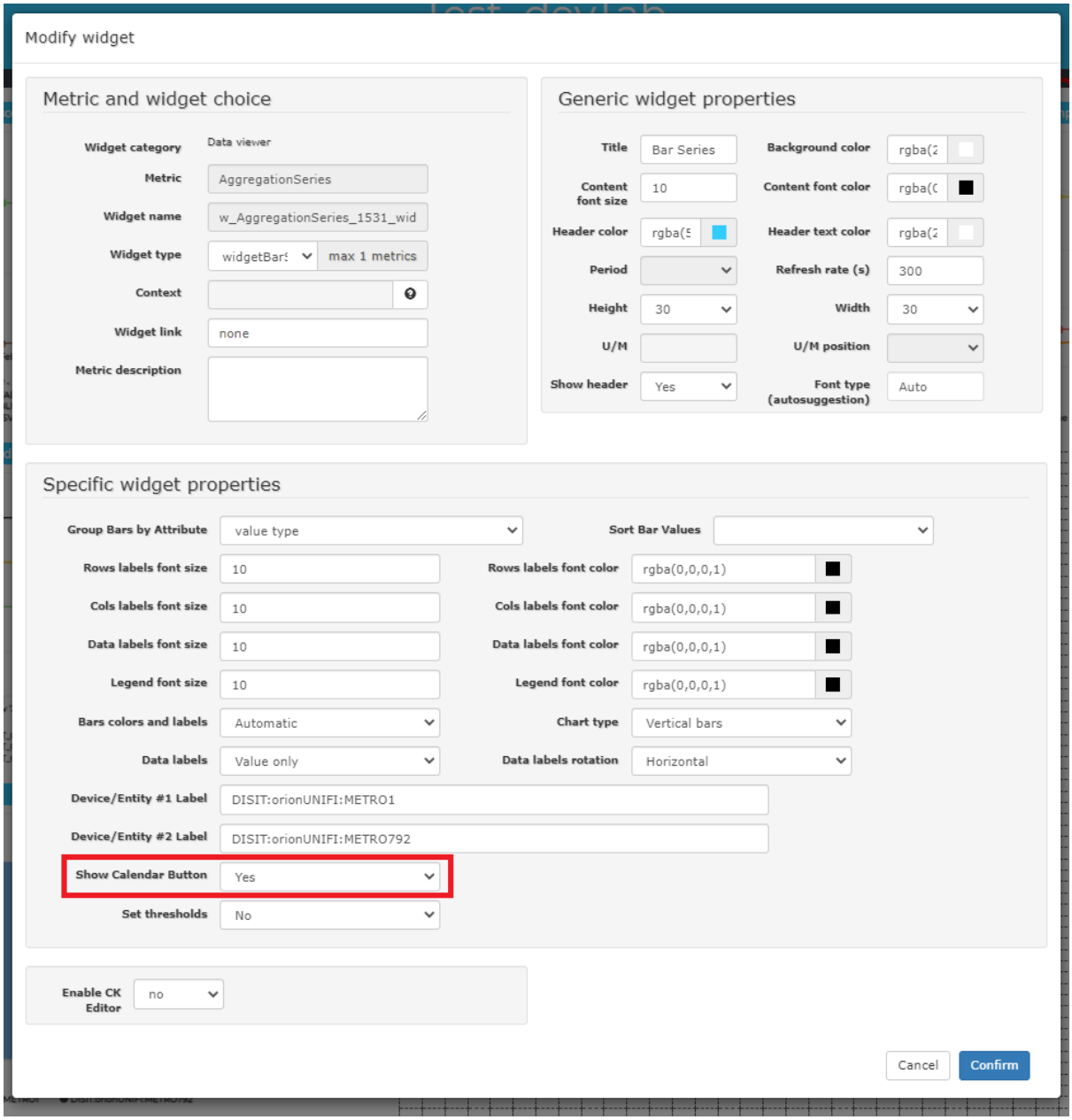
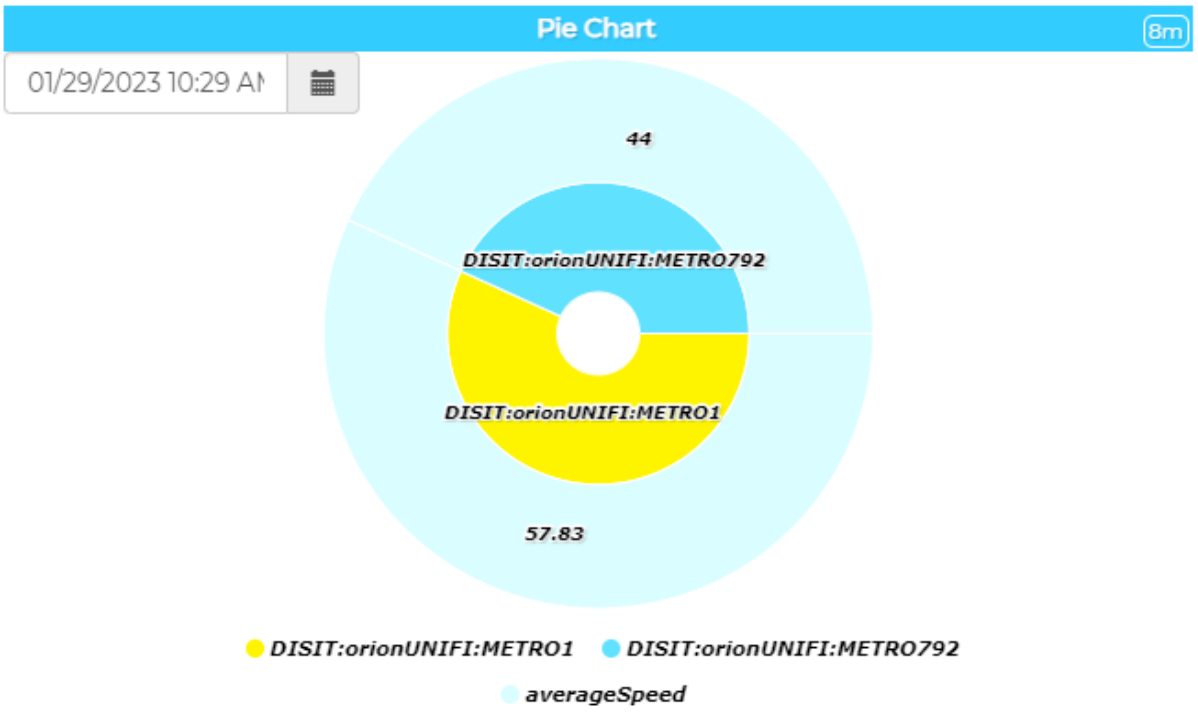
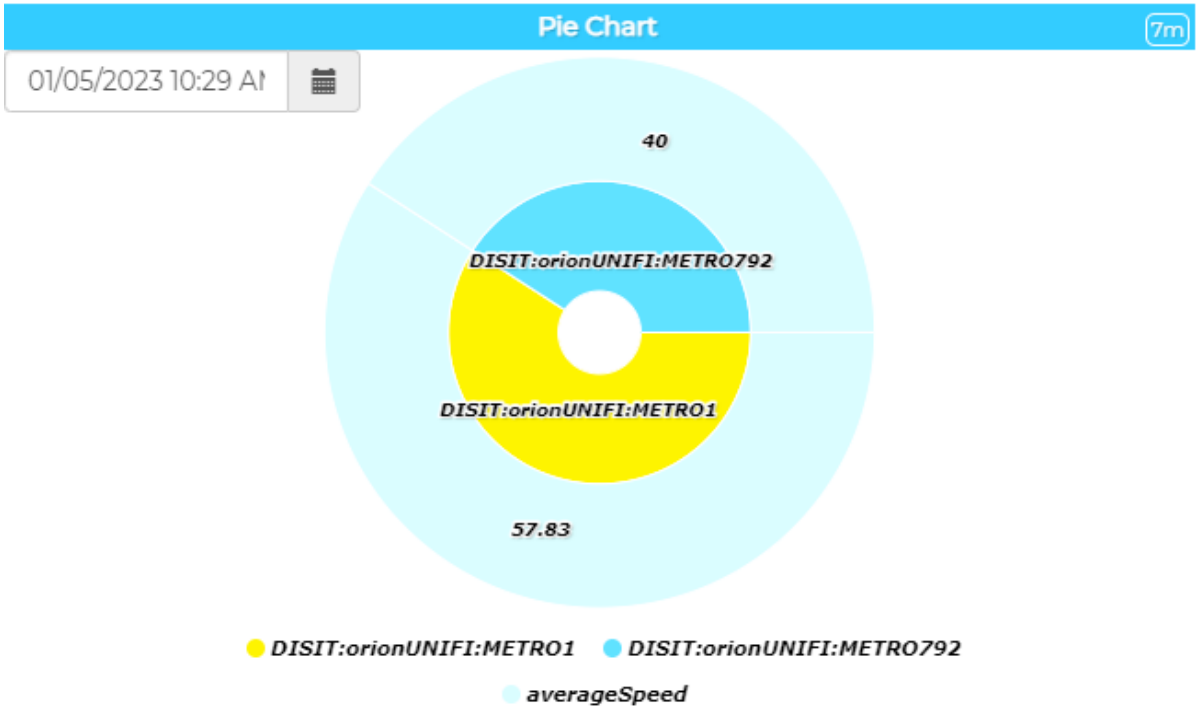


Figure 21: Active or Deactive Calendar Button in modify widget menu.

10.1 widgetPieChart

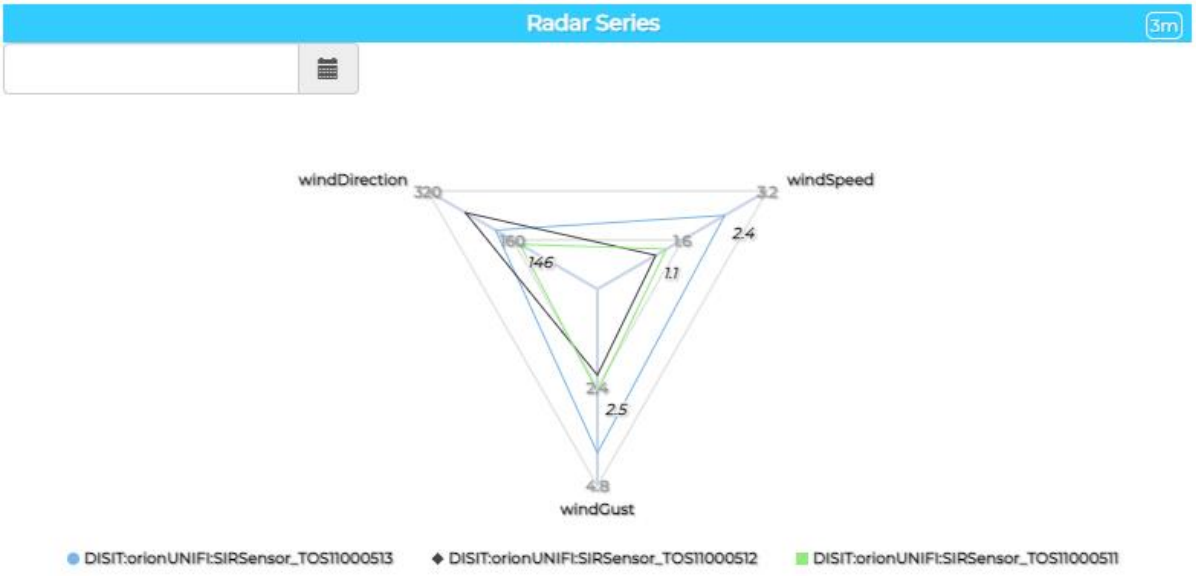
By default, the *widgetPieChart* shows the most recent data, but through the input of the datetime picker it is possible to view the metrics data at a specific date and time.





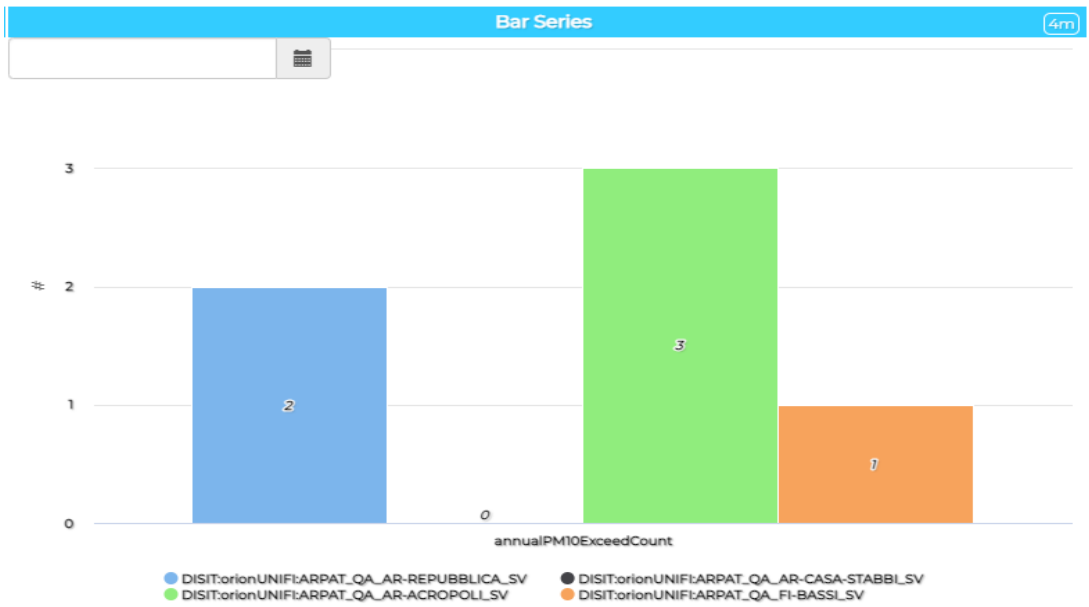
10.2 widgetRadarSeries

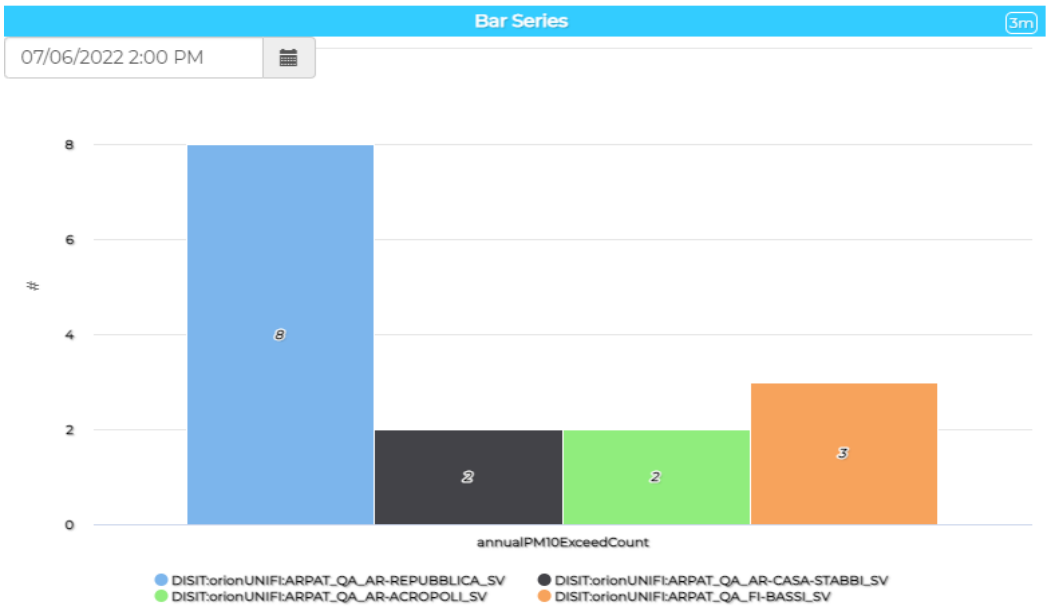
By default, the *widgetRadarSeries* shows the most recent data, but through the input of the datetime picker it is possible to view the metrics data at a specific date and time.



10.3 widgetBarSeries

By default, the *widgetBarSeries* shows the most recent data, but through the input of the datetime picker it is possible to view the metrics data at a specific date and time.





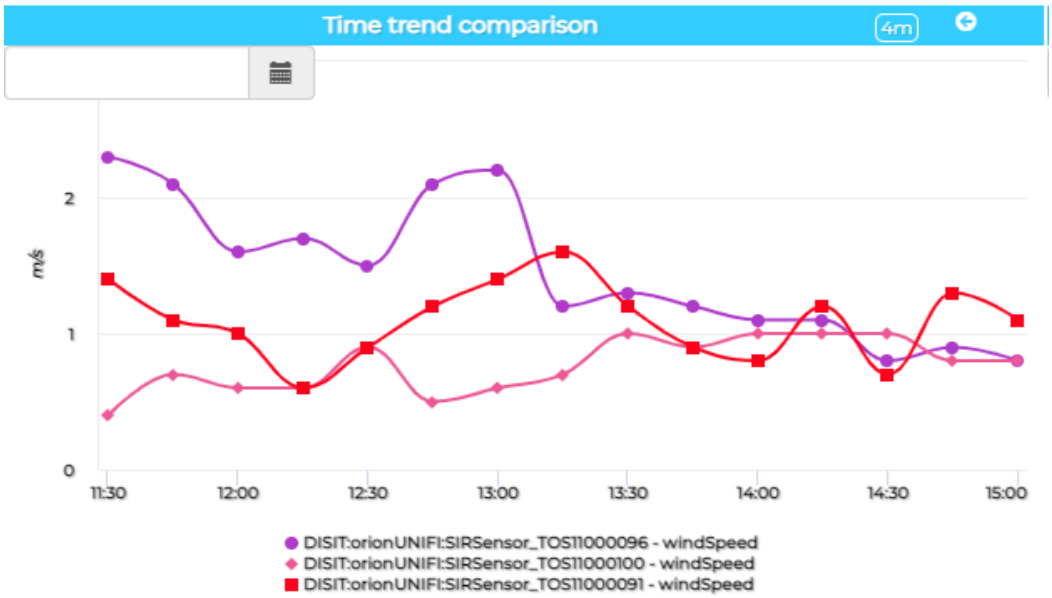
The functionality of activating and deactivating the datetime input can be done through the *more options* menu accessible in the editing mode of the dashboard, at the input selection *Show Calendar Button.*

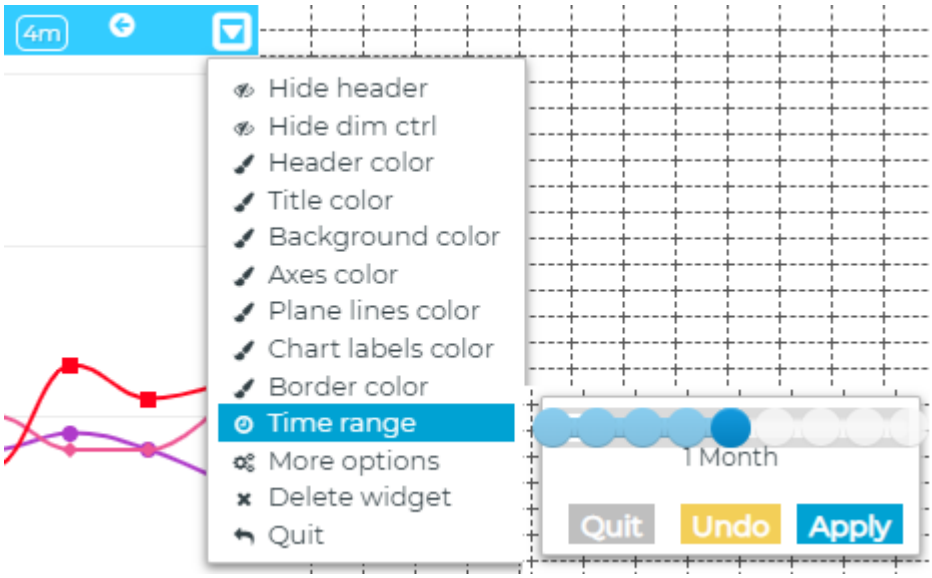
10.4 widgetCurvedLines

The Curved Lines Series widget displays a timeline based on a time range, definable from the edit menu in editing mode, which has by default the date and time of the last metric detection as the graph's last point.

Through the calendar functionality it is possible to define the date and time of the graph's last point. The graph will be recalculated on the basis of the current time range.

For example, if the dashboard is displayed on May 1st 2023 at 10.00 and the time range set is one month, the graph will show the data from 04/01/2023 10:00 to 05/01/2023 10:00.
If we set the datetime picker to 1 February 2023 at 12:00, keeping the same time range, the graph will automatically be recalculated with the data from 01/01/2023 12:00 to 02/01/2023 12:00. It is also possible to change the time range while keeping the new set datetimepicker final point.





In editing, the functionality of activating and deactivating the datetime input can be done through the *more options* menu accessible in the editing mode of the dashboard, at the input selection *Show Calendar Button.*