Sensor Widgets Documentation

Release 1.0

Oscar Fonts

Dec 11, 2017

Contents

CHAPTER 1

SOS in a nutshell

Sensor Widgets are a visualization tool for Sensor Observation Service (SOS) OGC standard services.

The widgets use a SOS client implementation prepared for version 2.0 of the standard, and, at this stage, they require a JSON encoding endpoint, which is not a standard requirement, but an optional encoding provided by 52 north's SOS server implementation v. 4.0.0 or above.

Note: The embedded SOS client can be extended to implement the mandatory KVP/XML encoding, which would make the Widgets fully compliant with the standard and compatible with any other SOS 2.0 server implementation.

A Sensor Observation Service offers data coming from a collection of sensors. This is how a SOS Service is organized:

1.1 Concepts

Warning: The following is a simplified view of the main SOS concepts, just as a quick reference for newcomers, which will probably be more interested in actually viewing some data than in understanding the full SWE, SOS and O&M models. If you are deploying a SOS service, please don't take this quick note as reference, and take a look at the OGC standard specification documents (see reference at the end of this chapter).

1.1.1 Offering

The data delivered by a SOS service is grouped in different offerings. For example, a "meteo" SOS service could have the following offerings: Satellite, Radar, Station Observations, Prediction Maps, etc. Each offering exposes data from a sensor or sensor network, described in its procedure.

Consider an Offering as a "sections" or "drawer" in a SOS Service, which classifies data based on their different nature or origin.

1.1.2 Procedure

A procedure describes a sensor, a collection of sensors, or a process that outputs some observations. It provides metadata about sensor inputs, outputs, calibration, data processing algorithms, contact information, data availability (space and time extents), etc.

It is generally encoded in SensorML format.

Consider a Procedure as metadata about the sensor(s) or process(es) that generates the data you will see.

One Offering is related to a single Procedure, whereas a Procedure can be used in many Offerings. For instance, a Procedure can be a “Weather Station Network”, and this same Procedure can be used in many Offerings (with differing time spans or periodicities), such as “Weather Station Network Measurements during 2015”.

1.1.3 Feature of Interest

Each measurement in a SOS service is bound to a Feature Of Interest (FoI), which usually describes where the observed phenomenon occurred. For example, for a satellite image, the Feature Of Interest could be the image’s footprint (polygon), and for a temperature measurement, it could be the thermometer location (point).

Consider them as a collection of places where data is referred to.

1.1.4 Observed Property

The thing that is measured, such as: Temperature, Wind Direction, Cloudiness, Number of Vehicles... it can be numerical (a quantity with a unit of measure), logical (true / false), categorical (sunny, cloudy, rainy), or descriptive (a text).

1.1.5 Observation

Finally, an observation is the value of an observed property at a particular time and place (Feature Of Interest). For example: “The temperature at Barcelona on 22/09/2015 at 11:52 AM is 23 degrees celsius”.

1.2 Operations

Any SOS request operation has to indicate the following parameters:

- Service: `SOS`.
- Version: `2.0.0` (the version supported by Sensor Widgets).
- Request: the Operation name, such as `GetCapabilities`.

These are the SOS 2.0 Request operations used in Sensor Widgets:

1.2.1 GetCapabilities

The GetCapabilites response is quite verbose. The GetCapabilities request optionally accepts a `sections` parameter to retrieve only specific parts of the document.

Specifically, the `contents` section describes the service as a collection of Offerings. Each offering containing:

- The Offering name (for instance “10-minute measurements”),
- The Offering identifier,
- The related Procedure identifier,
- The collection of Observable Properties (their identifiers),
- The geographical extent of the measurements (the bbox containing all the Features of Interest),
- The time span of the measurements (the time range containing all the Observations).

Full GetCapabilities JSON request example:

```
POST http://sensors.fonts.cat/sos/json
Content-Type: application/json
Payload:
    {
        "service":"SOS",
        "version":"2.0.0",
        "request":"GetCapabilities",
        "sections":["Contents"]
    }
```

This Capabilities-contents document is used as an entry point to discover the SOS service structure and available data. It provides a lot of identifiers, but little details, which have to be retrieved with subsequent requests to DescribeSensor or GetFeatureOfInterest operations.

1.2.2 DescribeSensor

The DescribeSensor request accepts a `procedure` identifier parameter, and returns a SensorML document, containing metadata about the sensor(s) or process(es) producing the offering's measurements.

The relevant contents are:

- The Procedure Identifier, Short Name and Long Name,
- A collection of keywords (useful for metadata catalog text search engines),
- Some contact information,
- The valid Time Period (redundant with Capabilities response),
- The observed BBOX (redundant with Capabilities response),
- The collection of Feature of Interest identifiers (new information not found in GetCapabilities contents),
- The collection of Offering Identifiers using this procedure (a back reference),
- An Output list: A collection of ObservableProperties with their corresponding IDs, names, types and Units of Measure.

This request is normally used to get the details that GetCapabilities doesn't provide, especially the description of Observable Properties (names and units of measure).

Full DescribeSensor JSON request example:

```
POST http://sensors.fonts.cat/sos/json
Content-Type: application/json
Payload:
    {
        "service":"SOS",
        "version":"2.0.0",
        "request":"DescribeSensor",
        "procedure":"http://sensors.portdebarcelona.cat/def/weather/procedure",
        "procedureDescriptionFormat":"http://www.opengis.net/sensorML/1.0.1"
    }
```

1.2.3 GetFeatureOfInterest

The GetFeatureOfInterest accepts a `procedure` as parameter, and returns all the Features of Interest related to that procedure. In fact, Features of Interest are bound to each Observation, but this operation provides a sort of "list" of all possible Feature values.

It is useful to get the location details, such as their names and geometries. So, it's usually used to draw a map or a place chooser.

Full GetFeatureOfInterest JSON request example:

```
POST http://sensors.fonts.cat/sos/json
Content-Type: application/json
Payload:
    {
        "service":"SOS",
        "version":"2.0.0",
        "request":"GetFeatureOfInterest",
        "procedure":"http://sensors.portdebarcelona.cat/def/weather/procedure"
    }
```

1.2.4 GetDataAvailability

The getDataAvailability request accepts a `procedure`, and optionally a collection of `FeatureOfInterest` and/or `ObservedProperty` as parameters.

It returns the time span of the available observations for each combination of Procedure-Feature-Property. So we can query the available data time span for any particular location and sensor.

Full GetDataAvailability JSON request example:

```
POST http://sensors.fonts.cat/sos/json
Content-Type: application/json
Payload:
    {
        "service":"SOS",
        "version":"2.0.0",
        "request":"GetDataAvailability",
        "procedure":"http://sensors.portdebarcelona.cat/def/weather/procedure",
        "featureOfInterest":["http://sensors.portdebarcelona.cat/def/weather/
↪features#02"],
        "observedProperty":["http://sensors.portdebarcelona.cat/def/weather/
↪properties#31"]
    }
```

1.2.5 GetObservation

Finally, the data about measurements.

A GetObservation request accepts as parameters:

- An `offering`,
- A collection of `FeatureOfInterest`,
- A collection of `ObservedProperties`,
- Temporal or Spatial Filters.

Specially interesting is the filtering, so one can constrain the query to a particular time period or geographical area. Sensor Widgets only use the temporal filtering to get either the "lastest" available observation, or a collection of observation in a given time period.

Full GetObservation JSON request example:

```
POST http://sensors.fonts.cat/sos/json
Content-Type: application/json
Payload:
    {
        "service":"SOS",
        "version":"2.0.0",
        "request":"GetObservation",
        "offering":"http://sensors.portdebarcelona.cat/def/weather/offerings#10m",
        "featureOfInterest":["http://sensors.portdebarcelona.cat/def/weather/
↪features#P3"],
        "observedProperty":["http://sensors.portdebarcelona.cat/def/weather/
↪properties#31"],
        "temporalFilter":[{
            "equals":{
                "ref":"om:resultTime",
                "value":"latest"
            }
        }]
    }
```

The response is a collection of observations, each one containing:

- Its related Offering Identifier,
- Its related Procedure Identifier,
- Its related Feature of Interest (with its corresponding Name, Identifier and full Geometry),
- Its related Observable Property Identifier,
- Phenomenon time (when something happened) and result time (when the resulting measurement was obtained),
- Finally, the result, which is composed of a **value** and a unit of measure.

The whole response is tediously verbose and redundant, with some element descriptions being repeated again and again hundreds or thousands of times in the same response. Imagine a series of 5000 observations from the same sensor. All the fields except times and values are repeated 5000 times without need. This seriously impacts on SOS service response speed and lightness.

Some service implementors (namely 52n SOS 4.0.0+) provide some strategies that extend the core standard to alleviate the situation, such as the aforementioned JSON format service encoding, and an extension called `MergeObservationsIntoDataArray`, that “collapse” all the observations sharing the same procedure, feature of interest and observed property into a single `SweArrayObservation`.

Note: The Sensor Widgets don’t take advantage of the `MergeObservationsIntoDataArray` extension. This is a potential future improvement.

1.3 Reference

Standards documents from the Open Geospatial Consortium:

- OGC® Sensor Web Enablement: Overview And High Level Architecture v. 3 (White Paper). Ref. OGC 07-165.
- OpenGIS® SWE Service Model Implementation Standard v. 2.0. Ref. OGC 09-001.
- OGC® SWE Common Data Model Encoding Standard v. 2.0.0. Ref. OGC 08-094r1.
- Sensor Observation Service v. 1.0. Ref. OGC 06-009r6.
- OGC® Sensor Observation Service Interface Standard v. 2.0. Ref. OGC 12-006.
- OpenGIS® Sensor Model Language (SensorML) Implementation Specification v. 1.0.0. Ref. OGC 07-000.
- OGC Abstract Specification - Geographic information — Observations and measurements v.2.0. Ref. OGC 10-004r3.
- Observations and Measurements - XML Implementation v.2.0. Ref. OGC 10-025r1.

CHAPTER 2

Using the Sensor Widgets

Each widget has a collection of mandatory inputs, and some optional inputs. Setting up a widget is essentially choosing the correct input values to get the desired result.

2.1 The Wizard

The easiest way to configure a Widget is using the Wizard, which will help us choose the input values based in a range of valid values. For instance, most of the widgets take an "offering", one or more "features" and one or more "properties" as mandatory inputs. The wizard will inspect a SOS service for you and let you pick from a list of existing offerings, features and properties.

Other typical mandatory inputs are the refresh interval, for widgets showing live data that has to be updated periodically, or the time range, for widgets that show different values during a period of time. In the later case, the time picker will limit its possible values to the available data time range, and will format the initial and final dates for us.

Typical optional parameters are the "footnote", a free text to be displayed along with the widget, and a Custom CSS URL, a mechanism to override the default widget style.

For details on each widget inputs and their format, please refer to the next chapter.

So, once we fill the widget configuration form, we can click on "Create Widget", and will get a widget preview, and three ways to "take away" the resulting widget: As a standalone web page, as an embeddable HTML component, or as a piece of code to be integrated in a larger javascript application.

2.2 Take away: Link and Embed

If we click on the generated "link" from the Wizard, we will get a rather long URL. This URL opens a web page with the configured widget. We can just use the widget as is and stop bothering about the link's internal structure.

But for those that want to understand how the links work (for instance, to generate or manipulate widgets manually, without having to go through the wizard), let's see how they are built, decomposing an example into its parameters:

```
http://sensors.fonts.cat/widget/
    name=compass
    service=http://demo.geomati.co/sos/json
    offering=http://sensors.portdebarcelona.cat/def/weather/offerings#10m
    feature=http://sensors.portdebarcelona.cat/def/weather/features#P3
    property=http://sensors.portdebarcelona.cat/def/weather/properties#31
    refresh_interval=5
    lang=en
```

Note: To have a valid URL, the parameter values have to be encoded using javascript's standard `encodeURIComponent` function (or equivalent in your language of choice). For clarity, we have presented them decoded in this example.

This is mostly the widget form input values. The wizard form let us choose an offering, feature and property names, but the widget configuration works with identifiers instead. The wizard inspected the SOS service for us to grab all the available name-identifier pairs. You can get the valid identifiers manually via a `GetCapabilities` operation.

There are a couple of extra parameters which are not widget inputs:

- The first one "name": It is the widget name, to know which widget has to be created.
- The last one, "lang": It is used to translate the possible text labels. It is optional and defaults to English ("en"). Other supported languages are Spanish ("es") and Catalan ("ca").

The "embed" option just wraps the link in an iframe tag, so it can be embedded in any other web site:

```
<iframe src="..." width="570" height="380" frameBorder="0"></iframe>
```

Width and height are taken from the widget form (recommended dimensions) but can be customized by just resizing the widget view (mind the handle in the bottom left corner).

2.3 Usage in Javascript

Finally, the most flexible way of using the widgets is programmatically. You just need to load the Sensor Widgets javascript library, which is available at http://sensors.fonts.cat/js/SensorWidgets.js , and instantiate the widget using the SensorWidget factory, which takes three parameters:

```
SensorWidget(widget_name, widget_configuration, dom_element);
```

The widget name is a string, the widget configuration is an object whose properties are the input name&values, and the DOM element indicates where in the HTML page to render the widget.

The most practical way to generate a widget is to use the wizard and copy&paste the code snippet. Then you can add dynamism by changing some of its configuration values.

See a live example here: http://bl.ocks.org/oscarfonts/5ad801cf830d421e55eb

Note: The `SensorWidget` function has no return value, but some of the parameters accept a callback function. Widgets are created asynchronously. In case of error, an error message will be displayed to the user in place of the widget.

2.3.1 Making low level SOS calls with Javascript

Warning: Direct access to low-level SOS operations is experimental. The API described here can change at any time.

The SOS client instance is obtained asynchronously:

```
getSOS(function(SOS) {
    // You must indicate a 52n SOS 4.x UrL with JSON encoding
    SOS.setUrl("http://sensorweb.demo.52north.org/sensorwebtestbed/service");
    // Then call any other SOS method
});
```

This is the API:

```
SOS.getCapabilities(callback, error); // Get the GetCapabilties "contents" section.
SOS.describeSensor(procedure, callback, error); // Get the SensorML document␣
↪converted to a JSON structure.
SOS.getFeatureOfInterest(procedure, callback, error); // Get all the␣
↪FeatureOfInterest for the given procedure.
SOS.getDataAvailability(procedure, offering, features, properties, callback,␣
↪error); // Get the time range of availability for each combination of procedure␣
↪+ feature + property.
SOS.getObservation(offering, features, properties, time, callback, error); // Get␣
↪the observations for the given combination of parameters.
```

Where the parameters are:

- *callback* (function) gathers the response as a Javascript object (parsed JSON).
- *error* (function) callback invoked in case the SOS service returns an error.
- *procedure* (string) procedure identifier.
- *offering* (string) offering identifier.
- *features* (array de strings) list of Features Of Interest for which we want to get a response.
- *properties* (array de strings) list of Observable Properties for which we want to get a response.
- *time* the instant (if it's a string) or time range (if it's an array of 2 strings) for which we eant to get a response. Times are indicated in UTC, format "yyyy-mm-ddThh:mm:ssZ". The special "latest" value is used to get the most recent available observation.

And their optionality:

- The *callback* function is always mandatory, and the *error* function is always optional.
- It is mandatory to indicate the *procedure* for *describeSensor* and *getFeatureOfInterest* methods.
- For *getDataAvailability* and *getObservation*, the filters (procedure, offering, features, properties, time) are optiona. Set them to *undefined* in case you don't want to filter by a particular concept.

2.4 Custom styling

All the widgets accept a `custom_css_url` input parameter. You can point to a css stylesheet published elsewhere that overrides the default widget styles.

All widgets are contained in a div element with two classes: the `widget` class, and the widget's name class. For instance, the following rule will apply to all widgets:

```
.widget {
    border: 2px solid black;
}
```

And the following one will apply only to the `compass` widget:

```
.widget.compass {
    background-color: grey;
}
```

Another common element is the `footnote` class:

```
.widget .footnote {
    font-color: red;
}
```

One could even hide some components if not needed. For example, the title:

```
.widget.thermometer h1 {
    display: none;
}
```

For more specific styling, the best practice is to inspect the widget DOM, and apply css rules to the observed elements.

CHAPTER 3

Available Widgets

3.1 Compass

The Compass widget belongs to the “single instant measure” category. It displays the latest value for a particular Property expressing a direction or angle. The widget will periodically interrogate the server to get the latest value.

Mandatory inputs are:

- “service”, “offering”, “feature” and “property”: selects a particular property to be shown. Property should take values between 0 and 360 (unit of measure is supposed to be degrees).
- “refresh_interval” (in seconds): is the time between updates (launches successive getObservations every Nth second).

Other optional inputs:

- “title”: If not specified, defaults to the name of the Feature.
- “display_utc_times”: The SOS service dates are in UTC, and they are converted to local time fuse by default. To display them in UTC, set this parameter to *true*.
- “footnote”: Optional small text caption at the bottom.
- “custom_css_url”: css stylesheet to be applied to the widget.

3.2 Gauge

Another “single instant measure”, but for percentage values ranging 0 to 100.

Mandatory inputs are:

- “service”, “offering”, “feature” and “property”: selects a particular property to be shown. Property should take values between 0 and 100 (unit of measure is supposed to be %).
- “refresh_interval” (in seconds): is the time between updates (launches successive getObservations every Nth second).

Other optional inputs:

- “footnote”: Optional small text caption at the bottom.

- "display_utc_times": The SOS service dates are in UTC, and they are converted to local time fuse by default. To display them in UTC, set this parameter to *true*.
- "custom_css_url": css stylesheet to be applied to the widget.

3.3 Jqgrid

It displays a jqGrid table, with a collection of observations for a given period of time, each row being an observation. Results are paginated and can be sorted by any column value (Result time, Feature name, Property Name, Value and Unit of Measure).

Mandatory inputs:

- "service", "offering", a collection of "features" and a collection of "properties": selects a collection of feature-property combinations to be shown.
- "time_start" and "time_end": The result time range of the observations to be displayed.
- "title": the widget title.

Other optional inputs:

- "footnote": Optional small text caption at the bottom.
- "display_utc_times": The SOS service dates are in UTC, and they are converted to local time fuse by default. To display them in UTC, set this parameter to *true*.
- "custom_css_url": css stylesheet to be applied to the widget. Please note that jqGrid look & feel is taken from the underlying jQuery-ui theme.

Note: this widget depends on jQuery, jQuery UI and the jgGrid plugin itself. It's a rather heavy and not much customizable (it was made as an integration exercise with a legacy application). We recommend the use of other widgets such as the "table" one, which is more in the spirit of Sensor Widgets: lightweight, compact and easily customizable.

3.4 Map

This widget is special in many senses. First off, it represents a GetFeatureOfInterest response, instead of the most usual GetObservation response.

Sencondly, it is a highly customizable widget, with a lot of configuration options. Fortunately most of the inputs are optional, so its basic usage is in fact very simple.

It is built on the Leaflet mapping library.

The only strictly mandatory parameters are:

- "service" and "offering": Determines the offering whose Features of Interest are to be displayed on a map.

This will display a map with the Features on it. Placing the mouse pointer over the map features will display a little tooltip with the feature name.

There are another couple of mandatory (but not so mandatory) inputs:

- "features": One can select which features to display. If none is selected, *all* of the possible features are displayed (no filtering aplied). But you *have* to explicitly indicate an empty array of features as input.
- "properties": If one or more properties are selected, each feature's tooltip will be an embedded panel widget, displaying the list of properties. Again, you can indicate an empty array of properties. In this case, no property values are shown.

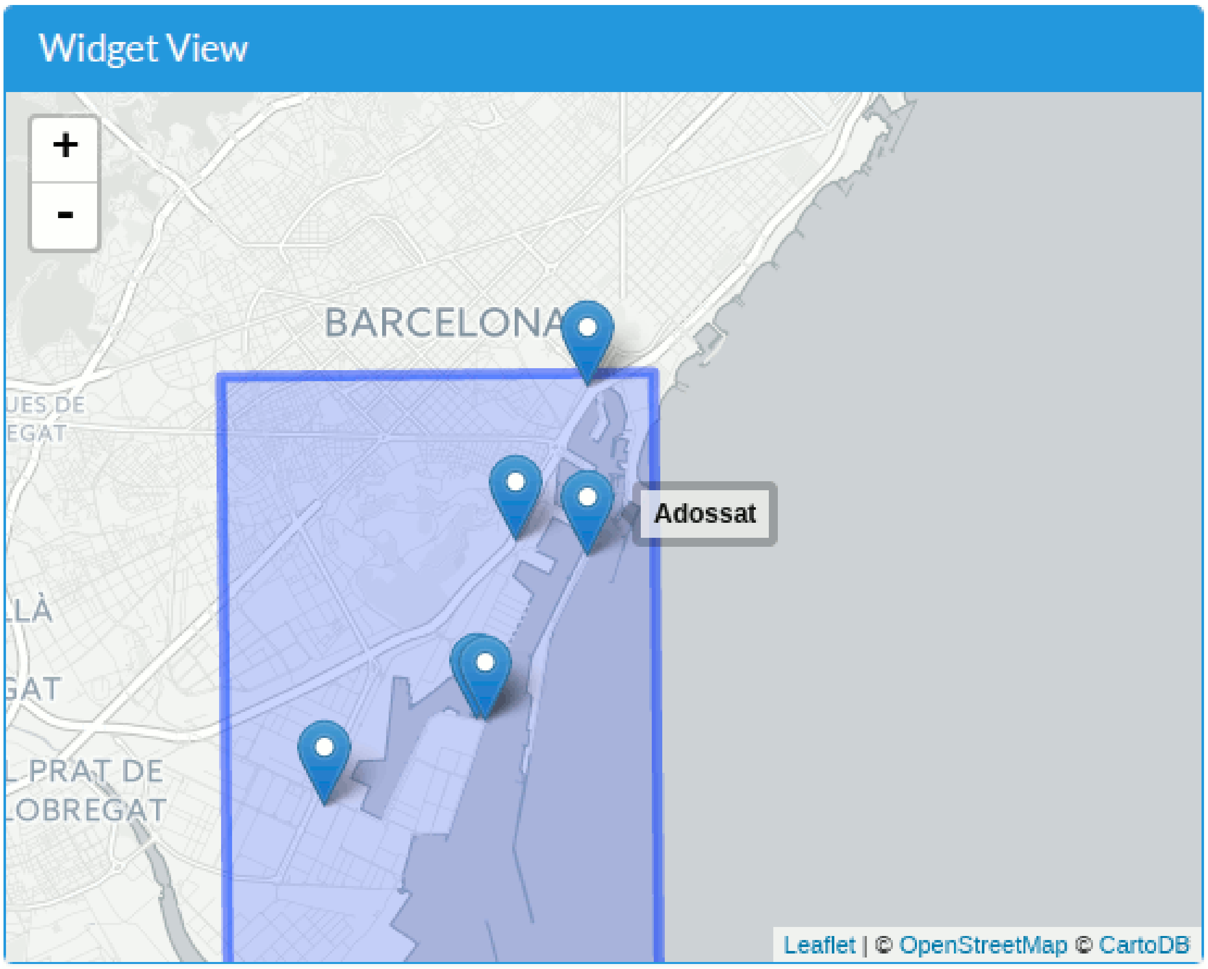


Fig. 3.1: Simple map with no features and no properties indicated.

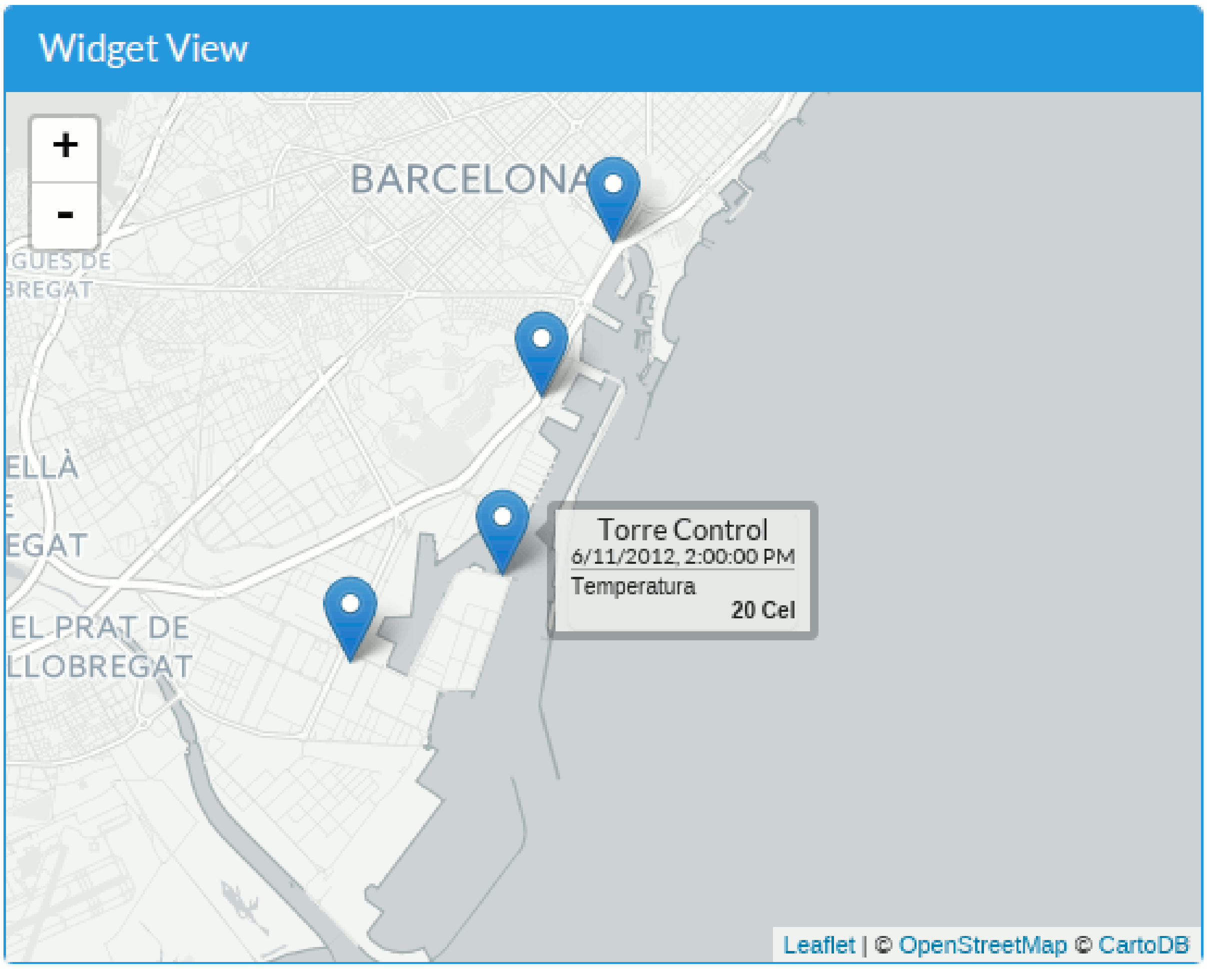


Fig. 3.2: Simple map with four features and one property selected.

The "permanent_tooltips" optional parameter, if set to "true", will force the tooltips to be always shown, not only on mouse hover.

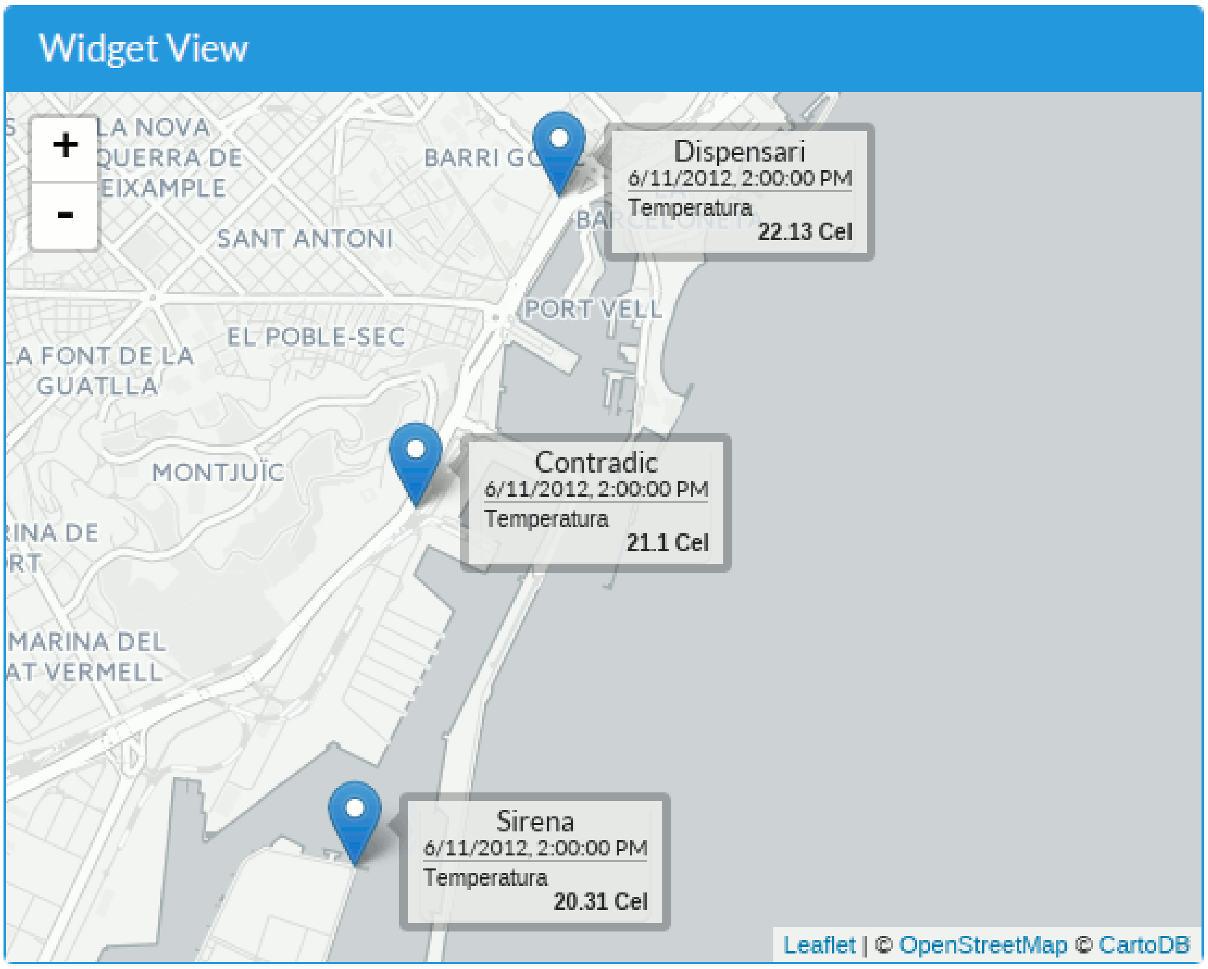


Fig. 3.3: Map with permanent tooltips.

If map elements appear located in the opposite side of the world, you probably have to change the coordinate axis order. Setting the optional parameter "swap_axis"=true, latitude and longitude will be switched, and this effect will be fixed.

Besides the tooltip, we can also attach a sub-widget to each feature, which will be displayed when clicking the feature. The "popup_widget" input is a JSON structure which contains a Widget definition. The "service", "offering" and "feature(s)" inputs for the widget are taken from the *parent* map widget, so are not needed. The "name" property indicates which widget to be instantiated.

For instance, if we want to open a popup containing a "timechart" on each feature click, we have to indicate:

- "name": "timechart",
- ...all the timechart widget inputs, except for "service" and "offering".

That is:

```
{
    "name": "timechart",
    "title": "Temperatures",
    "properties": [
        "http://sensors.portdebarcelona.cat/def/weather/properties#32M",
        "http://sensors.portdebarcelona.cat/def/weather/properties#32",
        "http://sensors.portdebarcelona.cat/def/weather/properties#32N"
    ],
    "time_start": "2015-09-03T05:05:40Z",
    "time_end": "2015-09-03T08:05:40Z"
}
```

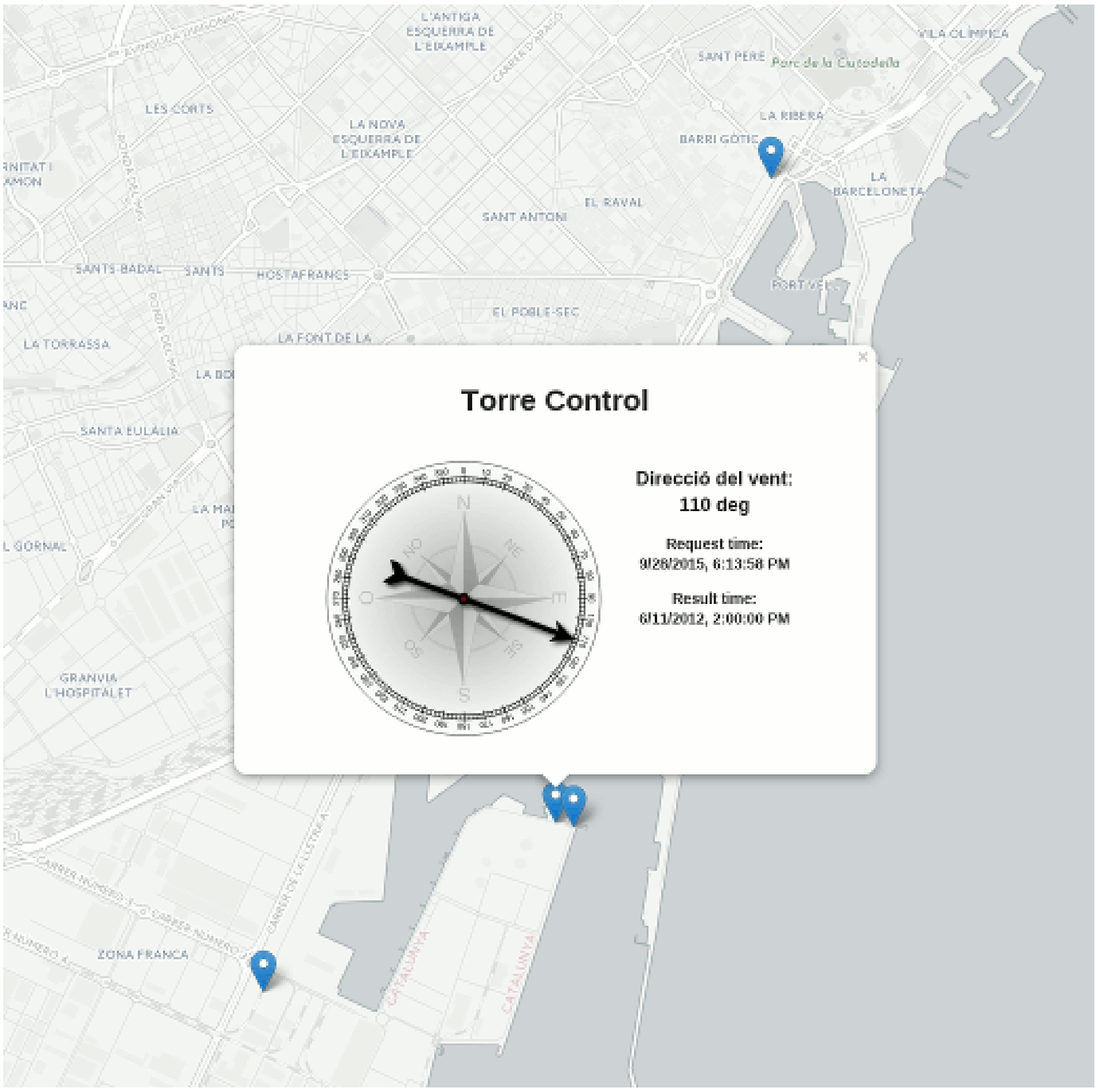


Fig. 3.4: Map with a "compass" popup.

Apart from customizing both tooltips and popups with details about each feature, we can indicate a custom base layer for the map, via the "base_layer" input. Two layer types can be specified:

- A Tile layer: Specify an "url", and a collection of "options". For example:

```
{
    "url": "http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png",
    "options": {
            "maxZoom": 19,
            "attribution": "&copy; <a href='http://www.openstreetmap.org/
↪copyright'>OpenStreetMap contributors</a>"
        }
}
```

The "url" and "options" parameters correspond to Leaflet's TileLayer "urlTemplate" and "TileLayer_options" respectively.

There's a good collection of free tile layers here: http://leaflet-extras.github.io/leaflet-providers/preview/

- A WMS layer: Specify "type": "wms", an "url" and a collection of "options". For example:

```
{
   "type": "wms",
   "url": "http://geoserveis.icc.cat/icc_mapesbase/wms/service",
   "options": {
      "layers": "orto5m",
      "format": "image/jpeg",
      "attribution": "Ortofoto 1:5.000: CC-by <a href='http://www.icc.cat'␣
↪target='_blank'>Institut Cartogràfic de Catalunya</a>"
   }
}
```

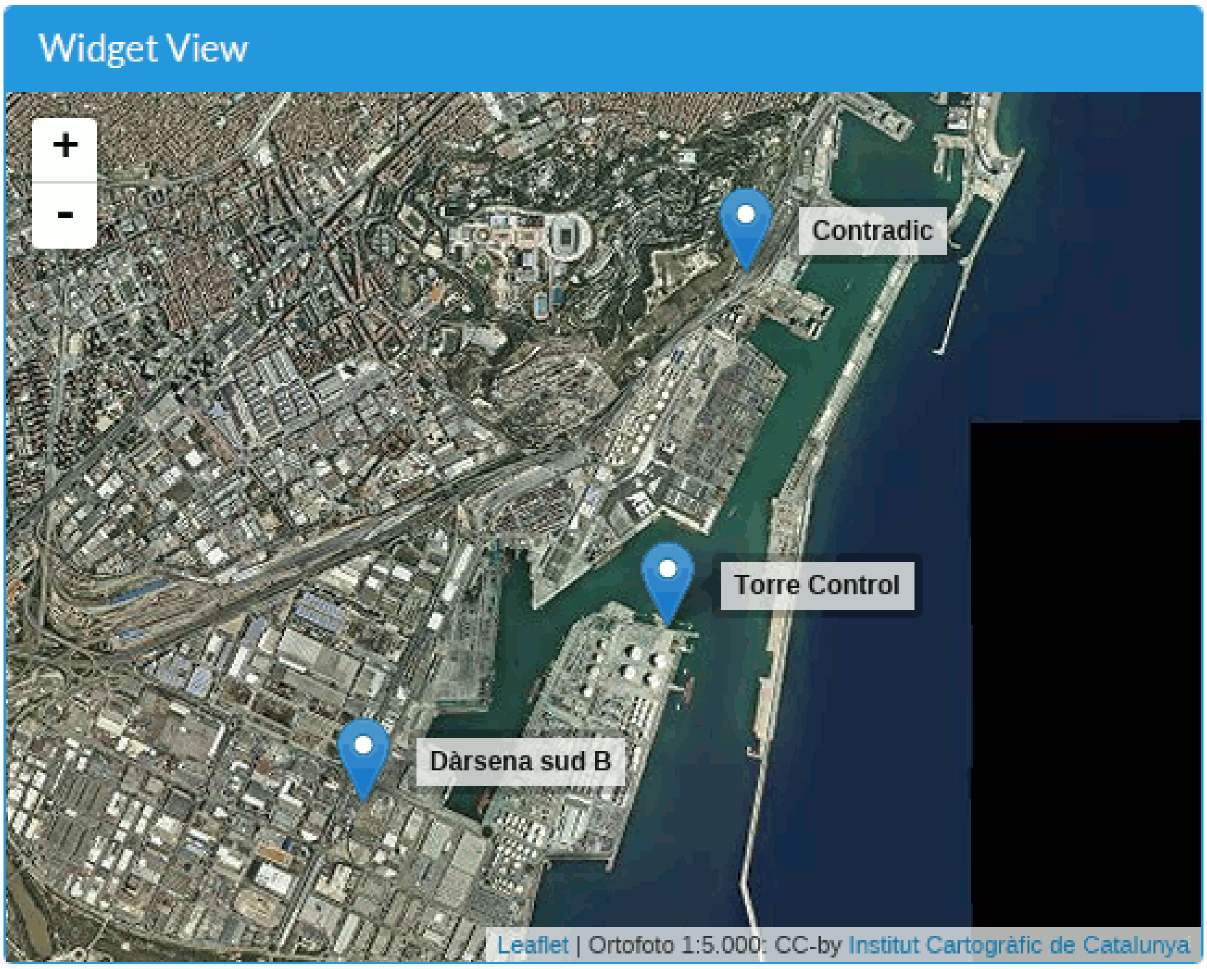

Widget View
+
-
Contradic
Torre Control
Dàrsena sud B
Leaflet | Ortofoto 1:5.000: CC-by Institut Cartogràfic de Catalunya

Fig. 3.5: Map with a custom base WMS layer.

The "url" and "options" parameters correspond to Leaflet's TileLayer.WMS "baseUrl" and "TileLayer.WMS_options" respectively.

Another optional input is "max_initial_zoom": It indicates the maximum zoom level to use when the map is first rendered. This avoids to zoom in too much, so we loose context, especially when a single point feature is drawn.

When there is a risk of marker overlapping on the map, a clustering mechanism is applied automatically. This automatic clustering can be disabled setting to *true* the optional "no_clustering" parameter.

If creating the widget with Javascript, it is possible to capture the "click" event on a map marker and get its details:

```
"on_click": function(marker) {
   console.log(marker.feature);
}
```

Finally, the common “display_utc_times”, “footnote” and “custom_css_url” inputs are also available.

See a **complete live example** here: http://bl.ocks.org/oscarfonts/265d734349396cf4372c

3.5 Panel

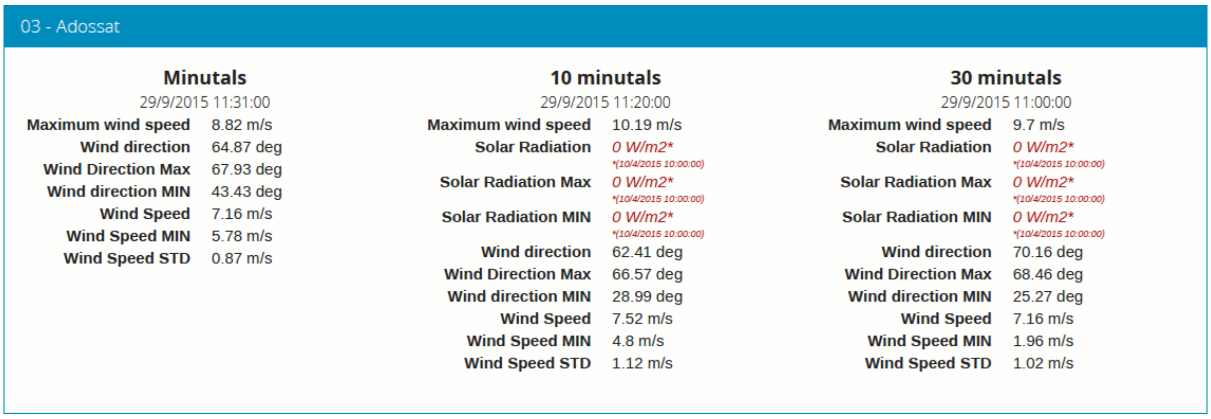
The “panel” widget is used to display all (or some of) the last property values for a particular Feature. It is built as an HTML Definition List, compatible with Bootstrap CSS classes. The widget will auto-refresh periodically.

Its mandatory inputs are:

- The usual “service”, “offering” and “feature”.
- A list of “properties” to be displayed.
- The “refresh_interval”, in seconds.

And the optional inputs: “title”, “display_utc_times”, “footnote” and “custom_css_url”.

The panel will show the result time as a subtitle. In case some of the propertie’s result time is previous to the common one, the value will be displayed in red and the particular result time for that observation displayed explicitly.


03 - Adossat

Minutals
29/9/2015 11:31:00

Maximum wind speed	8.82 m/s
Wind direction	64.87 deg
Wind Direction Max	67.93 deg
Wind direction MIN	43.43 deg
Wind Speed	7.16 m/s
Wind Speed MIN	5.78 m/s
Wind Speed STD	0.87 m/s

10 minutals
29/9/2015 11:20:00

Maximum wind speed	10.19 m/s
Solar Radiation	*0 W/m2** *(10/4/2015 10:00:00)*
Solar Radiation Max	*0 W/m2** *(10/4/2015 10:00:00)*
Solar Radiation MIN	*0 W/m2** *(10/4/2015 10:00:00)*
Wind direction	62.41 deg
Wind Direction Max	66.57 deg
Wind direction MIN	28.99 deg
Wind Speed	7.52 m/s
Wind Speed MIN	4.8 m/s
Wind Speed STD	1.12 m/s

30 minutals
29/9/2015 11:00:00

Maximum wind speed	9.7 m/s
Solar Radiation	*0 W/m2** *(10/4/2015 10:00:00)*
Solar Radiation Max	*0 W/m2** *(10/4/2015 10:00:00)*
Solar Radiation MIN	*0 W/m2** *(10/4/2015 10:00:00)*
Wind direction	70.16 deg
Wind Direction Max	68.46 deg
Wind direction MIN	25.27 deg
Wind Speed	7.16 m/s
Wind Speed MIN	1.96 m/s
Wind Speed STD	1.02 m/s

Fig. 3.6: Three Panel widgets, some of them showing outdated values.

3.6 Progressbar

Another instant measure widget, this time displayed as a proportion bar between two values. It is useful to show how a value relates to its boundary values. It can be used to display a percentage if min/max values ranging from 0 to 100, but it could also be used to display a liquid level, or a pressure. Sort of a “gauge” but displayed linearly and with custom value range.

Its mandatory inputs:

- The usual “service”, “offering”, “feature” and “property”.
- “min_value” and “max_value”, which will determine the extreme values.
- “refresh_interval” in seconds.

And the usual optional inputs: “display_utc_times”, “footnote” and “custom_css_url”.

3.7 Status

The "status" widget displays the whole offering status at a glance. Given an offering, it builds a table with all the possible feature-property combinations, and for each one, the last observed value and its recency. It is a good way to see the offering's health: If new data is being generated and for which sensors.

This widget is meant as a monitoring tool (sort of hypertable), and it's better displayed at full screen.

Its only mandatory inputs are "service" and "offering".

And the common optional inputs: "display_utc_times", "footnote" and "custom_css_url".

3.8 Table

Given a feature and a time range, the table displays property values over a time period. It provides a more compact view than jqGrid widget. The widget is built as a plain HTML table supporting Bootstrap's styling.

- The usual "service", "offering" and "feature".
- A list of "properties" to be displayed.
- "time_start" and "time_end": The result time range of the observations to be displayed.
- The table's "title".

And the common optional inputs: "display_utc_times", "footnote" and "custom_css_url".

3.9 Thermometer

Another "single instant measure" widget, such as Compass or Gauge, but for atmospheric temperature in Celsius degrees. It displays a thermometre drawing, whose values range from -24ºC to 56ºC. Numeric value is also shown. As other widgets in its category, it has built in auto-refresh mechanism.

Mandatory inputs are:

- "service", "offering", "feature" and "property": selects a particular property to be shown. Unit of measure is supposed to be degrees celsius.
- "refresh_interval" (in seconds): is the time between updates.

Other optional inputs:

- "footnote": Optional small text caption at the bottom.
- "display_utc_times", to force times to be expressed in universal time and not in local fuse.
- "custom_css_url": css stylesheet to be applied to the widget.

3.10 Timechart

Given a feature and a time range, it displays property values over a time period. Its interface is the same as the "table" widget, but the results are displayed graphically on a chart.

Charts are built with the Flot charting library, which in turn depends on jQuery.

- The usual "service", "offering" and "feature".
- A list of "properties" to be displayed.
- "time_start" and "time_end": Determines the time period of the observations to be displayed.
- The timechart's "title".

And the common optional inputs: “display_utc_times”, “footnote” and “custom_css_url”.

3.11 Windrose

This is a very specific widget, used to display wind regime statistics, where one can see at a glance the proportions of wind direction and wind speed over a period of time, for a particular location.

Note: The polar chart is built with the Highcharts library. This library is free for non-commercial uses, but **a license must be purchased for commercial uses**.

Mandatory inputs are:

- “service”, “offering”, “feature”: selects a particular location, which should offer both wind direction and wind speed properties.
- “properties”: two and only two properties. One will be wind speed, in `m/s`, and the other wind direction in `deg`. Result times for both properties should be synchronized and obtained in regular time intervals.
- “time_start” and “time_end”: the data time range to be gathered from SOS service.
- “refresh_interval” (in seconds): is the time between updates. As the windrose displays statistcs over a large dataset whose retrieval is expensive, it is recommended that the refresh interval is set to a high value (several minutes).
- “title” the widget’s title.

Parámetros opcionales:

- “colors”: Array of colors in *#rrggbb* format, which will be applied to the chart lines for each of the properties.
- “callback”: Function to be called after widget instantiation. It will get the internal Flot chart instance as parameter.

Optional inputs:

- “subtitle”.
- “display_utc_times”, “footnote” and “custom_css_url”.

This is how data is grouped to build the windrose chart:

1. The wind direction observations are grouped into 16 sectors: N, NNE, NE, ENE, E, ESE, SE, SSE, S, SSW, SW, WSW, W, WNW, NW, NNW and N.
2. For each sector, the corresponding wind speeds are classified in ranges: 0-2 m/s, 2-4 m/s, 4-6 m/s, 6-8 m/s , 8-10 m/s and > 10 m/s.

A polar chart with 16 distinct columns is drawn, each column containing different colored sectors, proportional to the wind speed counting.

Note: Unlike other widgets, which are meant to be lightweight and flexible, this one requires the SOS service to deliver the data in a very specific way. Moreover, it depends on a not completely free charting library. But the results for the specific use case it covers are very good. So, take this one example not as a generic, reusable widget, but as an example of *specialization*. And please, feel free to code your own widgets that better express your own data. See the next chapter, on how to contribute.

CHAPTER 4

Contributing to Sensor Widgets

4.1 Getting the code

Requires git:

```
git clone git@github.com:oscarfonts/sensor-widgets.git
```

If you want to contribute some changes, first fork it on github, work on your own fork, and submit a pull request to the upstream repo.

4.2 Code organization

This is 100% JavaScript. The following tools are used:

- Bower: manages javascript dependencies (libraries).
- Grunt: automates development tasks.
- RequireJS: keeps the code modular and dynamically loads only the needed modules.

From the bottom up, modules are:

4.2.1 main.js

The application entry point. Contains the requirejs configuration, where third party libraries are declared, as long as their transitional dependencies (shims).

4.2.2 XML.js

Utility module to convert from XML to JSON and vice-versa. Based on a script from Stefan Goessner.

It has two methods:

- read(xml): Parses an xml input (as a string, DOM document or DOM element) and returns a JSON object.
- write(object): Parses JSON object and returns an XML string.

The "read" method has a second parameter ("clean") which, if set to true, will generate a much browsable JSON object, ignoring XML namespaces and not prepending attributes with an "@". It is easier to use, but some information is lost in conversion, so the JSON cannot be converted back to an equivalent XML document.

The Sensor Widget library uses this module to convert to a JavaScript Object the SensorML document embedded in a DescribeSensor response (as of 52n SOS server 4.0.0, the SensorML is returned in XML format, even when using the JSON encoding endpoint).

4.2.3 SOS.js

The SOS client. It implements the most usual SOS 2.0 queries, and, at this point, only "talks" 52n's JSON encoding (not the KVP/XML encoding required by the standard). So, it isn't meant as a full standard interface implementation, but as the minimal code needed to retrieve information from a 52n SOS 4.0.0 server.

Thanks to the aforementioned XML.js module, it would be relatively easy to support the KVP/XML encoding as well.

It has a setter method for the sevice's base URL:

- setUrl(url): Sets the SOS service base URL.

And a collection of methods named after the SOS 2.0 operations they implement:

- getCapabilities(callback, errorHandler).
- describeSensor(procedure, callback, errorHandler).
- getFeatureOfInterest(procedure, callback, errorHandler).
- getDataAvailability(procedure, offering, features, properties, callback, errorHandler).
- getObservation(offering, features, properties, time, callback, errorHandler).

Methods accept the parameters required to build the SOS query:

- procedure: the procedure ID.
- offering: an offering ID.
- features: an array of Feature IDs (if single feature, just wrap into a single element array).
- properties: an array of Property IDs (if single property, just wrap into a single element array).
- time: can be the "latest" literal to get the last observation, a single timestamp to apply an "equals" filter, or an array of two timestamp values for a "during" filter. Timestamp values are strings formatted as an UTC ISO 8601 date and time. That is, "YYYY-MM-DD[T]HH:mm:ss[Z]" as *momentjs format definition <http://momentjs.com/docs/#/displaying/format/>*.

Query is sent via AJAX, and one of the two callback functions is called:

- callback(response): All went OK, response contains a JSON object data structure.
- errorHandler(status, url, request, response): Something went wrong.

Simple usage example:

```
SOS.setUrl("http://demo.geomati.co/sos/json");
SOS.getCapabilities(function(contents) {
    console.log("Service has " + contents.length + " offerings.");
});
```

4.2.4 sos-data-access.js

Widgets can use the SOS module directly (for instance, the map widget does), but most widgets have to perform some common tasks before and after calling the SOS getObservation request: validating and formatting the input parameters, getting and caching metadata such as property names, and rearranging the response so it can be consumed easily.

This module returns a "constructor" function that takes as parameters the widget config, a draw callback function, and an error callback function. The constructor returns an object with a single "read" method. Let's see its usage:

```
var widget_inputs = {
    offering: "offeringID",
    features: [...]
    properties: [...],
    ...
}

var data = sos_data_access(widget_inputs, onDraw, onError);

function onDraw(observations) {
    // render the observation values
}

function onError() {
    // display an error message
}

data.read(); // get the data from the SOS service and call the onDraw function on␣
↪success
```

The "read" method will in turn request a SOS.getObservation with the parameters specified in the config object, and call the onDraw function when the response is received. This draw function receives an array of observations, where each observation has the following properties:

```
{
    "time": /* A Date object */,
    "value": 67.17,
    "feature": "Sirena",
    "property": "Wind direction",
    "uom": "deg"
}
```

This is a pruned and flattened version of a full getObservation response, adapted for drawing purposes. That's why it contains feature and property names instead of internal identifiers, for example.

So most widgets won't "see" the SOS protocol directly, not even deal with SWE concepts, but use this "read data" => "draw callback" approach, which is much simpler.

It would be feasible to provide other non-SOS-data-access modules implementing this same interface, so widgets can be used to display data coming from legacy (non-SOS) protocols.

4.2.5 widget-common.js

And again, there are some common features shared by most of the widgets, that have nothing to do with data access. This module provides:

- The common "mandatory" and "optional" input lists, which are:

  ```
  inputs: ["service", "offering"]
  optional_inputs: ["footnote", "custom_css_url"]
  ```

- An initialization method that renders the footnote and loads the custom CSS stylesheet, when provided. So any widget that wants to implement these functions will call `common.init` method within its own init method.

4.2.6 i18n.js and translations.json

The way to translate the application is through the i18n module, which has the following methods:

```
i18n.langs(); // returns a list of supported languages
i18n.setLang('es'); // sets the active lang
i18n.getLang(); // returns the active lang
i18n.t("Original String Text"); // returns a translation of the original string␣
↪text in the active lang
i18n.addTranslations(object); // adds some extra translation strings to the base␣
↪bundle; useful dynamically extend the ``translations.json`` contents
i18n.translateDocTree(dom_element); // translates all the texts contained in this␣
↪dom element; useful to translate static HTML contents
```

The i18n module will load the `translations.json` file, which contains all the translations, like this:

```
"No widget name specified": {
    "es": "No se ha especificado ningún nombre de widget",
    "ca": "Cal especificar un nom de widget"
}
```

The key is the text to be translated in the original language (English), and it contains an object with as many properties as translations provided.

4.2.7 SensorWidget.js

This is *the* entry point to the Sensor Widgets library, and its returned function constitutes all the public interface. This function is a Widget Factory: Given a widget name and some input parameters, it renders the widget into the specified HTML DOM Element:

```
var el = document.getElementById('map-container');

var inputs = {
    service: "http://sensors.portdebarcelona.cat/sos/json",
    offering: "http://sensors.portdebarcelona.cat/def/weather/offerings#10m",
    features: [],
    properties: []
};

var widget = SensorWidget("map", inputs, el);
```

It also provides a default error handling function which will display the error message inside the same DOM Element.

When used as a "global" function it returns nothing, but when used as a require module, it returns an object with some useful stuff:

```
widget.name; // a string with the widget name ("map") in the example
widget.config; // an object with the provided inputs
widget.renderTo; // the provided DOM element where widget is going to be rendered

widget.url(); // returns a link to a web page with a live instance of this widget
widget.iframe(width, height); // returns an <iframe> tag containing the former URL.
widget.javascript(); // returns a javacript snippet to build this widget instance.

widget.inspect(inspect_callback); // provides a method to inspect the widget's␣
↪interface: mandatory and optional inputs, and preferred sizes.

function inspect_callback(mandatory_inputs, optional_inputs, preferred_sizes) {
    // use these values to display information about the widget interface.
    // Used in Wizard to build the Configuration Form by "introspection", and also␣
↪in the project's home page.
}
```

The callback is needed because the SensorWidget factory will load the widget code dynamically on demand, so its interface is only accessible asynchronously. This dynamic (lazy) loading mechanism avoids having to load widget code and the respective library dependencies unless needed. For instance, don't load the Leaflet library until a Map widget has to be created.

4.2.8 widget/<widget_name>.js

As all the common functionality (data access, shared inputs, instantiation) is placed in other modules, the actual widget code is really concise. The "gauge" widget is only 50 lines of code, and the most complex ones ("map", "windrose") take only 150 lines of code.

A widget has to implement the following interface, needed by the `SensorWidget` factory described above:

```
return {
    inputs: ["service", "offering"], // array of mandatory input names
    optional_inputs: ["footnote", "custom_css_url"], // array of optional input␣
↪names
    preferredSizes: [{w: 300, h: 300}], // array of recommended widget dimensions␣
↪in pixels, provide at least one

    init: function(config, element, errorHandler) { // the constructor function
        // Read config, fetch data, draw widget on element
        return {
            destroy: function() {
                // Clear timers and event handlers to prevent leaks
            }
        };
    }
```

4.3 Automated tasks

Grunt is used to automate common javascript development tasks.

Grunt itself is run on nodejs and its dependencies managed with `npm` and the `package.json` file. Make sure to have node and npm installed on your system:

- Mac & Windows: http://nodejs.org/download/
- Debian & Ubuntu: https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager#debian-and-ubuntu-based-linux-distributions

Then:

- Install grunt-cli. For example: *sudo npm install -g grunt-cli*.
- Get the project's npm dependenciess (such as grunt itself and its extensions) running *npm install*.

Now we are prepared to run the different *grunt* tasks:

4.3.1 Bower

Gets the javascript dependencies, such as RequireJS, jQuery, jQuery UI, jqGrid, Flot Charts, Leaflet, Highcharts, etc.

It also picks the needed library files from the `bower_components` directory and places them on the cleaner `js/lib/` directory. This is where requirejs expects to find the external dependencies.

4.3.2 Default

The default task (run as *grunt* without arguments) is to start a local http server that exposes the whole project so it can be tested on the browser. It also uses a *watch* subtask that will reload the page every time a javascript file is changed on disk.

4.3.3 Build

For development purposes, we work on the *src/* directory. But the distribution files are a concatenated and minified version of the source ones. The build task will perform the following subtasks:

- Clean: cleans the *lib* contents (dependencies) and the *dist* contents.
- Bower: fetches the libraries and places the needed files into *lib* again.
- JSHint: warns about coding errors in javascript. The build process will break at this stage until no hint warnings are detected.
- RequireJS: This task concatenates and minifies the source code (using the r.js optimizer and uglify) into various modules: * SensorWidgets.js: The base module, containing requirejs, the main config, and 'XML', 'SOS', 'sos-data-access', 'widget-common', 'i18n', and 'SensorWidget' modules, among others. * widget/<widget_name>.js: Contains the minified version of the widget, and its dependencies inlined (such as svg content). Each widget is kept in a separate module so optimized code can be loaded dinamically as well.
- ProcessHTML: Manipulates the sample page HTML headers so they load the optimized SensorWidget version.

It is recommended to run the 'build' task and test the 'dist' version before pushing changes to the main branch.

4.3.4 Publish

This is not to push source code to git, but to update the http://sensors.fonts.cat contents with an optimized version of your local code status. It runs the build task and uploads the resulting 'dist' directory.

4.4 How to document

This documentation is written in Sphinx and hosted in ReadTheDocs. Documentation is automatically rebuilt on ReadTheDocs when a change is pushed to GitHub.

Please contribute to this documentation via pull request.