SOS.js Documentation

Release stable

September 09, 2015

Contents

SOS.js is a Javascript library to browse, visualise, and access, data from an Open Geospatial Consortium (OGC) Sensor Observation Service (SOS).

CHAPTER 1

Overview of the SOS Library

The library consists of a number of modules, which along with their dependencies build a layered abstraction for communicating with a SOS.

The core module - SOS.js, contains a number of objects that encapsulate core concepts of SOS, such as managing the service connection parameters, the service's capabilities document, methods to access the service's Features of Interest (FOIs), offerings, observed properties etc. It also contains various utility functions, available as methods of the SOS.Utils object. The objects of this module are:

- SOS
- SOS.Offering
- SOS.Utils

This module is built on top of OpenLayers, for low-level SOS request/response handling.

The user interface module - SOS.Ui.js, contains the UI components of the library. These components can be used standalone, but are also brought together in the default SOS.App object as a (somewhat) generic web application. The objects of this module are:

- SOS.Plot
- SOS.Table
- SOS.Map
- SOS.Menu
- SOS.App

This module is built on top of OpenLayers which provides simple mapping for discovery; jQuery for the UI and plumbing; and flot, which is a jQuery plugin, for the plotting.

In addition, there are a number of separate modules that contain UI extension components that are built on top of the above standard components.

- SOS.MapSet.js
- SOS.Plot.Rose.js

All the styling for the UI components is contained in the library style sheet - SOS.Styles.css.

CHAPTER 2

Example Usage

Here we discuss examples of using the various objects of the library. For fully working examples, see the examples directory in the library distribution.

2.1 SOS

The core SOS object can be used for low-level communication with a SOS. After instantiating a SOS object, the user then interacts with the object via a series of event handling callbacks.

To instantiate a SOS object, we pass it a number of options. Only the URL to the SOS is required, so at its simplest, this will suffice:

```
var options = {
  url: "http://sosmet.nerc-bas.ac.uk:8080/sosmet/sos"
};

var sos = new SOS(options);
```

Typically the first thing that is required after instantiation, is to fetch the capabilities document of the SOS. As this call is asynchronous, we need to setup a callback to handle the `sosCapsAvailable` event, which signifies that the SOS object has received and parsed the capabilities document from the given SOS. We can accomplish this via the following:

```
sos.registerUserCallback({
  event: "sosCapsAvailable",
  scope: this,
  callback: capsHandler
});

sos.getCapabilities();

function capsHandler(evt) {
...
```

whereupon our `capsHandler` function can then inspect the capabilities of the SOS via the available method calls of the SOS object [1]. As a convenience, we can pass the name of our callback function as an argument to the `getCapabilities` call, which will then register this callback function to handle the `sosCapsAvailable` event with a scope of `this`; so identical to the above explicit `registerUserCallback` call:

[1] The parsed capabilities document is stored as a JSON object in the SOS object as `this.SOSCapabilities`. The structure of this document may change in future versions of the library, so direct access is discouraged.

```
sos.getCapabilities(capsHandler);
```

To unregister a callback, we can issue the following:

```
sos.unregisterUserCallback({
  event: "sosCapsAvailable",
  scope: this,
  callback: capsHandler
});
```

Once we have our capabilities document, we can inspect the available offerings and FOIs of the given SOS:

```
var offIds = sos.getOfferingIds();
var offNames = sos.getOfferingNames();
var foiIds = sos.getFeatureOfInterestIds();
```

2.2 SOS.Offering

Once we've identified an offering we're interested in, we can fetch a SOS.Offering object that encapsulates that offering:

```
var offering = sos.getOffering(offId);
```

or we can fetch an array of SOS.Offering objects pertaining to a given FOI:

```
var offerings = sos.getOfferingsForFeatureOfInterestId(foiId);
```

We can inspect the details of a particular offering, via its method calls:

```
var foiIds = offering.getFeatureOfInterestIds();
var procIds = offering.getProcedureIds();
var propIds = offering.getObservedPropertyIds();
var propNames = offering.getObservedPropertyNames();
```

and furthermore we can fetch observations of the offering's observed properties. By default, observations for all the offering's observed properties will be retrieved, however, often we may only want observations for a particular observed property, or subset of observed properties. This can be achieved by filtering the offering's observed properties, thus:

```
// Fetch the air temperature only
offering.filterObservedProperties("air_temperature");

// Fetch the wind data only
offering.filterObservedProperties(["wind_speed", "wind_direction"]);
```

To reset an offering's observed properties list, we unfilter:

```
offering.unfilterObservedProperties();
```

Once we have specified the desired observed property(s), we can fetch observation records, given a date range [2]. This is an asynchronous call, so just like the capabilities call above, we can explicitly setup a callback event handler:

```
offering.registerUserCallback({
  event: "sosObsAvailable",
  scope: this,
```

[2] All dates and times passed to the library must be in an ISO 8601 compliant format. For example, for the 31st of August 2013, that would be `2013-08-31` or `2013-08-31T00:00:00.000Z` etc.

```
  callback: obsHandler
});

offering.getObservations(startDatetime, endDatetime);

function obsHandler(evt) {
...
```

or alternatively, we can use the convenience of passing our callback function as an argument to the `getObservations` call:

```
offering.getObservations(startDatetime, endDatetime, obsHandler);
```

In our observation handler, we can then iterate over the observation records that were returned by the SOS, using the `getCountOfObservations` and `getObservationRecord` method calls. For example, to display the data in an HTML table, we could do something like:

```
for(var i = 0, len = offering.getCountOfObservations(); i < len; i++) {
  var ob = offering.getObservationRecord(i);
  tbody += '<tr>';
  tbody += '<td>' + ob.observedPropertyTitle + '</td>';
  tbody += '<td>' + ob.time + '</td>';
  tbody += '<td>' + ob.result.value + ' ' + ob.uomTitle + '</td>';
  tbody += '</tr>';
}
```

The observation record that is returned by a call to `getObservationRecord` is an Observations and Measurements om:Measurement resultModel representation, as returned by SOS, with additional convenience members of `time`, `observedPropertyTitle` and `uomTitle`. It has the following structure:

```
{
  samplingTime: {
    timeInstant: {
      timePosition: "2013-08-25T00:00:00.000Z"
    }
  },
  procedure: "urn:ogc:object:feature:Sensor:BAS:bas-met-halley-met",
  observedProperty: "urn:ogc:def:phenomenon:OGC:1.0.30:air_temperature",
  fois: [{
    features: [{
      layer: null,
      lonlat: null,
      data: {
        id: "foi_34579",
        name: "Halley"
      },
      id: "OpenLayers.Feature.Vector_1570",
      geometry: {
        id: "OpenLayers.Geometry.Point_1569",
        x: -26.7,
        y: -75.58
      },
      state: null,
      attributes: {
        id: "foi_34579",
        name: "Halley"
      },
      style: null
    }]
  }],
  result: {
    value: "-40.3",
    uom: "Cel"
  },
  time: "2013-08-25T00:00:00.000Z",
  observedPropertyTitle: "Air Temperature",
  uomTitle: "&deg;C"
}
```