ESWC 2011 Tutorial

# Building Semantic Sensor Webs and Applications: Sensor Data and Semantic Mashups

*Hands-on material*

The code for the mashup can be found on the Virtual Machine (VM) in the directory:

*/var/www/hlapi/apps/surfstatus/boscombe/*

There are three versions of the main mashup application file in this directory:

- *boscombe.php* (developed by our tamed mashup developer)
- *boscombe-annotated.php* (with comments referenced below added)
- *boscombe-cutouts.php* (with sections of code removed for you to add)

To switch between them, change the index.php symlink. That is:

```
rm index.php
ln -s <boscombe version of choice PHP>  index.php
```

GEdit (installed on the VM) can be used to edit the PHP file with syntax highlighting (as can any other editor of your choice).

## *1) Try the mashup application*

This is the complete, working, mashup (we will dissect it in later sections!)

- Symlink the boscombe.php version (this should be the default) and point the web browser in the VM at:
  *http://apps.semsorgrid.ecs.soton.ac.uk/surfstatus/boscombe/*

## *2) (Re-)Writing and editing the mashup source*

Both *boscombe-annotated.php* and *boscombe-cutouts.php* contain the comments referenced below. If you only wish to look, open *boscombe-annotated.php* in your editor; if you would like to edit too, open *boscombe-cutouts.php* !

## *3) The latest wave height observation (entry point to the API)*

Search for *NOTE#1* in source code. This is where the mashup uses the URI for the latest wave height observation of Boscombe to retrieve the observation as RDF.

Below *CODE_INSERT#1* add the following line of code:

```
$observationsURI = "http://id.semsorgrid.ecs.soton.ac.uk/observations/cco/boscombe/Hs/latest";
```

(If you are familiar with RDF you can also try dereferencing the URI. You might also try other accept type, e.g. application/xml)

## *4) Manipulating the Observations using ontology terms*

Find *NOTE#5*. Before plotting the current wave height values, the mashup must retrieve wave height values from the observations.

Below *CODE_INSERT#2* add the following lines of code:

```
if ($observationNode->get("ssn:observedProperty") != PROP_WINDWAVEHEIGHT)
        continue;
```

You can search for the definition of PROP_WINDWAVEHEIGHT – it comes from a domain ontology. It is associated with the observation using the observedProperty from the SSN ontology.

## *5) Using links to move between collection*

The mashup moves ("pages") wave height data by following "next"/"previous" links between collections. Find *NOTE#6*, where this functionality is implemented.

Below *CODE_INSERT#3* add the following line of code:

```
$prevobservation = $observations[0]->get("DUL:directlyFollows");
```

You can see that there is a corresponding *directlyPrecedes* a few lines later.

## *6) Finding linked data via a named URI (from the sensor)*

For some queries to external data sources (e.g. in this version, when the mashup finds nearby utilities) using a named point asserted to be "based_near" the sensor.

(For others, we can use the actual latitude and longitude of the sensor. You can see where the mashup starts the method by finding the sensor ("platform") that made the observation at *NOTE#8*).

Go to *NOTE#9*. Below *CODE_INSERT#4* add the following line of code:

```
$based_near = $graph->resource($sensorURI)->get("foaf:based_near");
```

(You can also search later in the code to find where the *based_near->uri* is passed to the *nearbyamenities()* function).

## *7) Querying the Observations API*

The RDF representation of the observations API is also stored in a triplestore so that it can be queried using a SPARQL endpoint. The mashup uses this to list other sensors that can observe waveheights (and in the non-VM version, can automatically generate mashups for these sensor URIs too).

Find *CODE_INSERT#6*. Below this add the following lines of code which make the SPARQL query:

```
$otherwavesensors = sparqlquery(ENDPOINT_CCO, "
        SELECT DISTINCT ?sensor ?sensorname
        WHERE {
                ?obs
                        a ssn:Observation ;
                        ssn:observedProperty <" . PROP_WINDWAVEHEIGHT . "> ;
                        ssn:observedBy ?sensor ;
                .
                OPTIONAL {
                        ?sensor rdfs:label ?sensorname .
                }
        }
");
```

## *8) Querying for amenities via based_near*

The code that generates a list of nearby amenities is split amongst several sections in the code. We have already seen where the based_near location is retrieved (6).

This is passed to the nearbyamenities() function along with the type of amenities to search for using a LinkedGeoData SPARQL query in the nearbyamenities() function (see *NOTE#15*).

Arrays of amenity types to search for are defined as arrays at *NOTE#13* using URIs from LinkedGeoData (search for the namespace definitions). Below *CODE_INSTERT#7* add the following array:

```
$types_pub = array(
        "lgdo:Pub",
        "lgdo:Bar",
);
```

## *9) Querying for road statistics (and please link your data!)*

Go to *CODE_INSERT#5* and add the following lines to query for National Road statistics:

```
$rows = sparqlquery(ENDPOINT_EUROSTAT, "
        SELECT DISTINCT ?region ?injured ?killed ?population WHERE {
                ?ourregion
                        a eurostat:regions ;
                        eurostat:name \"$euroRegion\" ;
                        eurostat:parentcountry ?country .
                ?region
                        a eurostat:regions ;
                        eurostat:parentcountry ?country ;
                        eurostat:population_total ?population ;
                        eurostat:injured_in_road_accidents ?injured ;
                        eurostat:killed_in_road_accidents ?killed .
        }
");
```

This works, but only because the string matches between two datasets. Ideally, the URI would be passed instead...

Kevin Page, University of Southampton
29/05/2011