Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Department of Measurement and Information Systems

Monitoring Cyberphysical Systems

MSc Thesis

Készítette
Velinszky László

Konzulens
dr. Strausz György

March 29, 2015

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Velinszky László*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, March 29, 2015

Velinszky László
hallgató

Kivonat

Minden ember nap mint nap találkozik olyan szenzorokkal, melyek hálózatba kötve kommunikálnak egymással vagy egy központi egységgel. Ezek lehetnek akár időjárás érzékelők, utcai kamerák, mobiltelefonok. Az eszközök "Internet of things" (dolgok internetje) részévé válnak, hogy aztán a méréseket kiértékelve következtetéseket vonjunk le és automatikusan beavatkozhassunk a környezetünkbe. Az ilyen kiberfizikai rendszerek felett azonban nincsenek szemantikai kapcsolatok, általában a szenzorok információit csak adott feladatokra hazsnálják ki. Gazdaságosabb lehet az egyes szenzorokat több feladatra is felhasználni, ehhez azonban több dologra is szükség van. Egy részről szükséges egy szabványos elérés az adatok eléréséhez, ez a SensorML. Másodsorban szükséges egy értelmezési, szemantikai megközelítés mely alapján a szenzorok kereshetőek lehetnek. Nem utolsó sorban a két dolgot össze kell kapcsolni. A diplomamunkám témája az utóbbi részek megvalósításának jelen helyzete és egy mintaprogram kidolgozása, mellyel ezek a kiberfizikai rendszerek monitorozhatóak.

Abstract

Most of us encounters different sensors that act together while connected into a network every day. These can be either weather sensors or CCTV cameras or smart-phones. These devices become part of the Internet of Things by evaluating their observations and interacting with its environment. Such cyberphisical systems does not contain semantic connections between its sensors, they are only used for one purpose. It would be more efficient to be able to use each sensor for several tasks. Many things are needed for that. Firstly there should be a standard language for accessing information, this is called SensorML. Secondly, there shall exist a semantic knowledge about the sensors to make them easily search-able. In this thesis the latter parts are described and a sample implementation is introduced.

Introduction

Nowadays, most of our devices are connected through the Internet. Our computers, mobile phones, surveillance cameras share their information via the world wide web. Each device has many sensors, such as GPS, gravity, acceleration sensor, imaging devices or just processing powers. These sensors or resource's information are stored individually on each device.

The world is emerging into a state that information needs to be shared between peers and should be stored in an easy to reach independent location called Cloud. The Cloud is a distributed information store which can be reached through the internet. This enables sensor information to be stored and processed for new uses, because the same sensor can be used for different purposes. An outdoor surveillance camera can be used to protect from intruders or to provide weather information. This data can be used for different purposes in various regions. A local measurement can control the local heating system, but a grid of weather stations can be used to provide forecast data.

It is not enough to measure all the data using such sensors. It has to be stored for analytics as it can be a source for predictions. Storing has become easier in the BigData world we live in. There are expectations of the method of storing the data. It should be transparent letting different systems share information with each other. To create such a standardized way of storing sensor measurement data Open Geospatial Consorptium started and maintains the SensorMl format. One of SensorML's couple layers is the SOS (Sensor Observation Service) which provides different ways to reach the observed data. This storage engine makes it available for outer services and inner services (like virtual sensors) to reach the desired data and run analytics and trigger monitoring events on them.

Storing sensor data does not give any semantic knowledge about the system. Such knowledge can be that a wind sensor is also a weather sensor or a camera is a visual sensor. To describe such semantic connections an ontology has to be created. There are many ways to store ontologies. Usually they are described as triples like in the most common RDF format but there are newer formats which has extended capabilities and can describe natively much more things (such as temporal logic) like OWL format. Storing such ontologies are done using special RDF databases. Such databases can analyze connections faster it is even possible gain new knowledge from predefined conditions using reasoning. This way we can describe such things that if an Anemometer is a kind of wind sensors and wind sensors

are weather sensors than without specifying explicitly that the Anemometer is a weather sensor the system already knows the answer.

If we have such an enormous system it can be hard to maintain it. There can be thousands or even more sensors in a network with different capabilities and efficiency. The operation of the sensors should be monitored in an easy to access method, where each sensor's state should be easy to reach via a modern user interface. The purpose of this document is to describe such a large system's architecture and provide a tool with such a monitoring task can be done.

Such a system is described in this paper. The chosen format is a dynamic, interactive web page. The page consists of an engine that is capable of showing the state of the sensors using the RDF database. The sensor's state is displayed on a user friendly page. The sensors can be filtered to show only a group of sensors. A summary page for the error is also included to have a big picture of the state of the system.

In the next two section the basics of the used technologies are described. First there is a detailed introduction to the SOS server and the SensorML standard. After that the semantic description language and its storage engine is introduced. In the third chapter the actual system is shown which the monitoring supports. There will be a detailed introduction to the components and the used sensors and some use case for the system. The final chapter describes the monitoring system in details. It shows its structure, the used third party frameworks and a usage manual. In the end the whole work will be summarized.

Chapter 1

Introduction of SensorML and SensorWeb

1.1 Goals of observation gathering

Most phyisical quantities are measured with great accuracy on very basic devices. From the most basic sound measurements with microphone, to the nowadays popular inertial measurement units every data can be measured. There are also weather sensors and solar sensors available for everyone.

With the evolution of sensor fusion algorithms these data can be merged together to give an even better accuracy or a general knowledge of our environment. This data can be used to predict weather conditions by merging neighborhood sensors into one. It can be used to make predictions based on the weather conditions in an area and the direction of the wind on the location.

Figure 1.1. *SWE architecture*

The other challenge is to use a sensor's data for multiple purposes. For example a CCTV camera picture can be used to count the number of cars on the road, to get an estimation

of the speedings in a crossing or to get visual weather data. Computer Vision algorithms, regressions and special machine learning algorithms need different computing resources. Those can be outsourced to different computers which must have access to the data. The Sensor Observation Server is used to make this all available.

1.2 Types of sensors

Sensors are all around us. We have sensors in our cell phones or any handheld devices. Most of them are connected to the internet or a network. The measurements of all of these devices can be stored and queried from a database. The usable data in some devices are:

- Smartphone
  - Camera: Visual image of the phone itself
  - Accelerometer: measurement of the acceleration of a system. Derivate of speed.
  - Weather sensors: Often smartphones has built in temperature sensors, rarely barometer is also installed.
  - Computing resource: ability to run additional softwares by using up its resources
  - Gyroscope: Orientation sensor.
- Weather sensor
  - Tempareture sensor: Inner and outer temperature.
  - Humidity sensor: percentage of humidity.
  - Barometer: Air pressure.
- Beagleboard
  - Computing resource.

The examples show that a standard way to retrieve the measurements should include not only the measurement, but the type of the sensor, the measured unit, the location of the sensor and many additional information like this.

Storing and retrieving this information is done using the SensorML format.

1.3 The SensorML

The Open Geospatial Consortium approved the SensorML language to be able to describe all the necessary information about measured data[2]. This is an XML based language that is used to describe the sensors, add, update, delete or retrieve information. The SensorML

is able to solve the above mentioned problems. It is an abstract definition of the sensor information.

To have a general structure for the data, there is an abstract hierarchy that represents sensors and measurements. They have their own concepts which is described in as follows [?]:

- FeatureOfInterest: additional constraints provided for the measurement, e.g. time or location of measurement.
- ObservedProperty: what type of physical property is measured, e.g. speed, temperature.
- Procedure: the virtual or real sensor itself that is providing the data.
- Offering: the property itself that has been measured, e.g. wind speed, air humidity, traffic size.
- Result: the output of the procedure.

This way gives as enough abstraction to separate the measured properties, the physical properties, the actual sensors and the results from each other.

It is able to define the location of the sensor, the timestamp of the measurement and the sensor data itself, with many additional information. The data can be stored in SOS servers that can retrieve the data on different interfaces.

1.4 Server Observation Service

Server Observation Servers store SensorML data and let others query, manipulate and add sensors to the database. These sensors can be derived sensors. Such derived sensors are called procedures. A procedure can be a traffic information based on the CCTV camera. An SOS server should be able to handle dependencies based on which procedure requires other procedures to provide data. Sometimes derived sensors are also called virtual sensors, because they don't do physical measurement but provide a derived value.

There are many closed and open source implementations of SOS servers. Some open source implementations are introduced here.

The MapServer is written in C, C++, however it can be extended in many other languages[3]. It has a built in GUI to view data, however it is not yet available for the SOS implementation. The server can only be reached by one interface. It was originally developed by University of Minnesota. In the time of writing the paper the code is maintained and supported by the Open Source Geospatial Foundation. It is mostly used for mapping not for serving data for derived sensors. It was used in some projects between NASA and the University of Minnesota.

istSOS is another implementation in Python[7]. It runs its scripts from Apache web server just like MapServer. It uses PostgreSQL database backend to store values. It has a nice GUI for administration. Only supports standard SensorML interface to retrieve data. It is a project of the SUPSI University in Switzerland.

OOSThetys is a basic toolkit for enabling SensorML communication. It is written in Perl and Java. It is fairly documentated and seems to be abandoned because there has been no new release since the February of 2012.

52north SOS is a sample implementation written in Java[1]. It runs as a web service from Apache tomcat to serve requests. It also uses PostgreSQL with PostGIS extension. The application is used in the sample implementation and it is covered in details later.

All SOS servers and the whole SensorML standard is missing the semantic information about the sensors meaning that all sensor data are stored in such a database and all properties can be described for the sensors but no connection between the concepts can be described for the sensor.

1.5 52north SOS server

This implementation is done by the non profit organization with the identical name. The software supports many interfaces to query the information needed. The standard SOAP can be used to work with Java Web services. There are KVP and POX to retrieve or add data using standard GET queries. A big advantage is that 52north SOS supports JSON interface. It enables JSON queries to retrieve information efficiently in JavaScript, PHP, Python or other modern scripting languages.

The new 4.2 version is easy to install, it has a graphical user interface to set up the database connection and initialize the database. PostgreSQL with PostGIS is required. The basic usage is described on 52north webpage, but sample queries are shown in the built in test client.

The project is built with Maven and uses Spring framework too. The included unit tests ensure that the software's architecture is still consistent after compilation. The project was chosen to be part of Google's Summer of Code 2014 and the project gained a lot improvements from that.

A request can be sent using the connector of each interface. This used to be done by adding the interface name to the application url but in recent versions the service automatically detects the interface from the HTTP metadata. For example http://152.66.253.152:8080/52n-sos-webapp/service is the url where all interface can be reached. The first part is the host of the server. The tomcat application server is listening on port 8080. 52n-sos-webapp is the default name of the web application, service it the default connector for the webservice. The sent HTTP content-type header marks the interface. A detailed description of the interfaces can be seen on table 1.1.

Interface	Content type	Description
SOAP	application/soap+xml	Standard SOAP request mostly used in JAVA, it is sending the request in XML format using POST request.
POX	application/xml	Very similar to SOAP but has a different XML format.
KVP	application/xml	This is a GET request, meaning that all request parameters are encoded in the URL. The answer is in XML format.
JSON	applicaton/json	This is a POST request which sends and receives information using JSON objects. Most programming languages can easily parse this type of request.

Table 1.1. *Interfaces for 52n SOS server*

1.6 SOS commands

There are many commands to query or manipulate the SOS server. In this section some of them will be shown using the JSON interface.

The GetCapabilities command (shown on listing 1.1) allows the clients to retrieve a list of the reachable sensors of the server and their configuration. This command lists all the available measurements on the server too. This call does not have any required parameters, only if the details of the response should be controlled. Such detail is the section to be displayed. There are 4 different sections:

- ServiceIdentification: the list of profiles that the server has been using. These profiles describe the format that the data is represented in.
- ServiceProvider: in this section is the contact information for the operator of the SOS service.
- OperationMetadata: this section provides the available options and their possible values.
- FilterCapabilities: what fields are available for filtering. Can be spatial filtering for location or temporal filtering for time range.
- Contents: summary of the type and FOI of each procedure.

Listing 1.1. *JSON getCapabilities POST request*

```
{
  "request": "GetCapabilities",
  "service": "SOS",
  "sections": [
    "ServiceIdentification",
    "ServiceProvider",
```

```
    "OperationsMetadata",
    "FilterCapabilities",
    "Contents"
  ]
}
```

If the procedure is known the DescribeSensor command will describe the measurements of the selected sensor. The response will include a description field which contains the details about the sensor in the standard XML based SensorML format. The sample request code can be seen on listing 1.2.

Listing 1.2. *JSON DescribeSensor POST request*

```
{
  "request": "DescribeSensor",
  "service": "SOS",
  "version": "2.0.0",
  "procedure": "http://www.52north.org/test/procedure/1",
  "procedureDescriptionFormat": "http://www.opengis.net/sensorML/1.0.1"
}
```

After knowing the necessary parameters the getObservation property can be called. This command shows the measured values for some given observations. The input can be complex, it can filter based on procedures, offerings, observed properties, on any feature of interest or temporal or spatial filters. A sample can be read from listing 1.3.

Listing 1.3. *JSON GetObservation POST request with all filters*

```
{
  "request": "GetObservation",
  "service": "SOS",
  "version": "2.0.0",
  "procedure": [
    "http://www.52north.org/test/procedure/6",
    "http://www.52north.org/test/procedure/1"
  ],
  "offering": [
    "http://www.52north.org/test/offering/6",
    "http://www.52north.org/test/offering/1"
  ],
  "observedProperty": [
    "http://www.52north.org/test/observableProperty/1",
    "http://www.52north.org/test/observableProperty/6"
  ],
  "featureOfInterest": [
    "http://www.52north.org/test/featureOfInterest/6",
    "http://www.52north.org/test/featureOfInterest/1"
  ],
  "spatialFilter": {
    "bbox": {
      "ref": "om:featureOfInterest/sams:SF_SpatialSamplingFeature/sams:shape",
      "value": {
        "type": "Polygon",
        "coordinates": [
          [ [ 50, 7 ],
            [ 53, 7 ],
            [ 53, 10],
            [ 50, 10],
```

```
            [ 50, 7 ]
          ]
        ]
      }
    }
  },
  "temporalFilter": [
    {
      "during": {
        "ref": "om:phenomenonTime",
        "value": [
          "2012-11-19T14:00:00+01:00",
          "2012-11-19T14:05:00+01:00"
        ]
      }
    },
    {
      "equals": {
        "ref": "om:phenomenonTime",
        "value": "2012-11-19T14:08:00+01:00"
      }
    }
  ]
}
```

The response (on listing 1.4) will contain the usual header which is contained by all the responses. It tells the requester the version of the response. After that comes the list of the observations that fit the conditions given. In the observation objects there are the procedures that measured the result, the feature of interests and the observable property that has been measured. The result object contains the uom field which is an abbreviation for the unit of measurement and the specific value. It can be an integer value, floating point or some encoded binary data.

Listing 1.4. *JSON GetObservation response*

```
{
    "request": "GetObservation",
    "version": "2.0.0",
    "service": "SOS",
    "observations": [
        {
            "type":
                "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_Measurement",
            "procedure": "http://www.52north.org/test/procedure/6",
            "offering": "http://www.52north.org/test/offering/6",
            "observableProperty":
                "http://www.52north.org/test/observableProperty/6",
            "featureOfInterest": {
                "identifier": {
                    "codespace": "http://www.opengis.net/def/nil/OGC/0/unknown",
                    "value": "http://www.52north.org/test/featureOfInterest/6"
                },
                "sampledFeature":
                    "http://www.52north.org/test/featureOfInterest/world",
                "geometry": {
                    "type": "Point",
                    "coordinates": [
                        51.447722,
```

```
                        7.270806
                    ]
                }
            },
            "phenomenonTime": "2012-11-19T13:09:00.000Z",
            "resultTime": "2012-11-19T13:09:00.000Z",
            "result": {
                "uom": "test_unit_6",
                "value": 2.9
            }
        },
        {
            "type":
                "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_Measurement",
            "procedure": "http://www.52north.org/test/procedure/6",
            "offering": "http://www.52north.org/test/offering/6",
            "observableProperty":
                "http://www.52north.org/test/observableProperty/6",
            "featureOfInterest": {
                "identifier": {
                    "codespace": "http://www.opengis.net/def/nil/OGC/0/unknown",
                    "value": "http://www.52north.org/test/featureOfInterest/6"
                },
                "sampledFeature":
                    "http://www.52north.org/test/featureOfInterest/world",
                "geometry": {
                    "type": "Point",
                    "coordinates": [
                        51.447722,
                        7.270806
                    ]
                }
            },
            "phenomenonTime": "2012-11-19T13:03:00.000Z",
            "resultTime": "2012-11-19T13:03:00.000Z",
            "result": {
                "uom": "test_unit_6",
                "value": 2.3
            }
        }
    ]
}
```

If only the result fields are important to the user the shorter GetResult command can be used. It is shown on figure 1.5.

Listing 1.5. *JSON minimal GetResult POST request*

```
{
  "request": "GetResult",
  "service": "SOS",
  "version": "2.0.0",
  "offering": "http://www.52north.org/test/offering/6",
  "observedProperty": "http://www.52north.org/test/observableProperty/6"
}
```

The response will be a list of values of the last measurement for the queried properties. The results will be given as a concatenated string where each measurement is separated

by a comma and in each measurement the value and its timestamp is separated by a hash mark. It can be seen on figure 1.6.

Listing 1.6. *JSON GetResult response*

```
{
    "request": "GetResult",
    "version": "2.0.0",
    "service": "SOS",
    "resultValues":
        "10#2012-11-19T13:00:00.000Z,2012-11-19T13:00:00.000Z,2.0#2012-11-19T13:01:00.000Z,2012-11
}
```

There are other commands that manipulate the database, like InsertSensor, DeleteSensor to add or remove sensors. There are commands to add observable properties, and add results to the database too.

1.7 SOS client applications

52north has a client application for the SOS server called 52n Sensorweb Client. This is a Java web application that can be configured to connect to multiple hosts and display their sensor data on maps and charts. It is developed using the same tools as the SOS server. This client is a web application which is can be opened in a web browser, but some parts run on the web server. This is a relatively complex application that does not yet understand semantic connections between the procedures. A screenshot can be seen on figure 1.2.

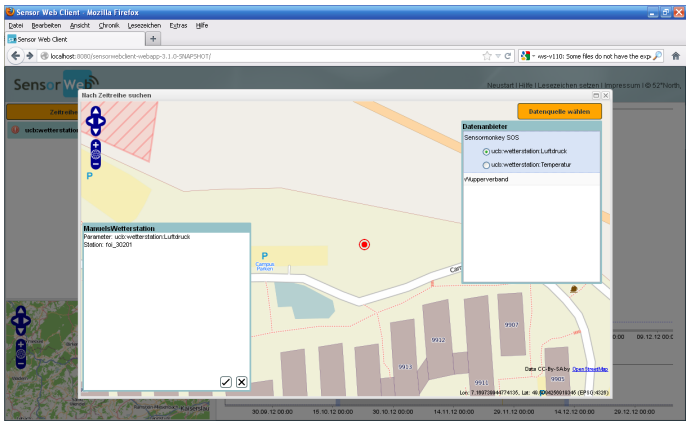

Figure 1.2. *Screenshot of SWE Client 3*

There is a JavaScript framework that enables users to use only client side tools to connect to SOS servers and display data called SOS.js. Unfortunately, because of security restrictions on JavaScript cross-site request the data cannot be retrieved from the SOS server directly, the client has to be served from the same domain as the server. This is often not doable. That is why this application needs a proxy that forwards the data to the same domain to bypass this restriction. Screenshot can be seen on figure 1.3.

There are external softwares that can display their own measurements but not the SensorML standard. To make them usable with 52north SOS the developers created tools to

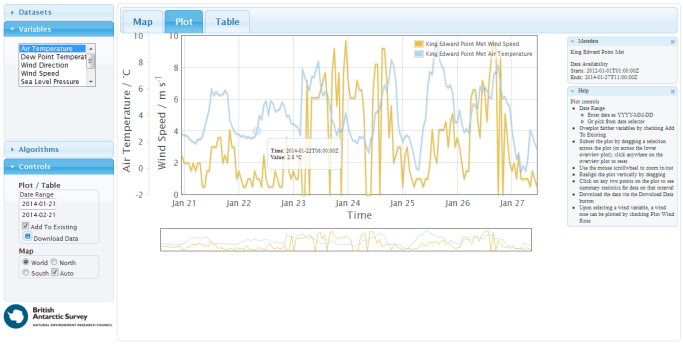

Figure 1.3. *Screenshot of sos.js*

extend such existing softwares to be able to import data from SOS. Such extension is the ArcGIS extension which makes SOS data available to ArcGIS server. This is also available to other programs such as μDig.

There are other tools to export data to R language or to use with other Geographic Information Systems (GIS) softwares. However, the problem is that no client software has the ability to make search available by semantic connections. A client has to be extended with such information to enable convenient filtering when monitoring a cyberphisical system.

Chapter 2

Semantic Connection

2.1 Advantages of semantic information

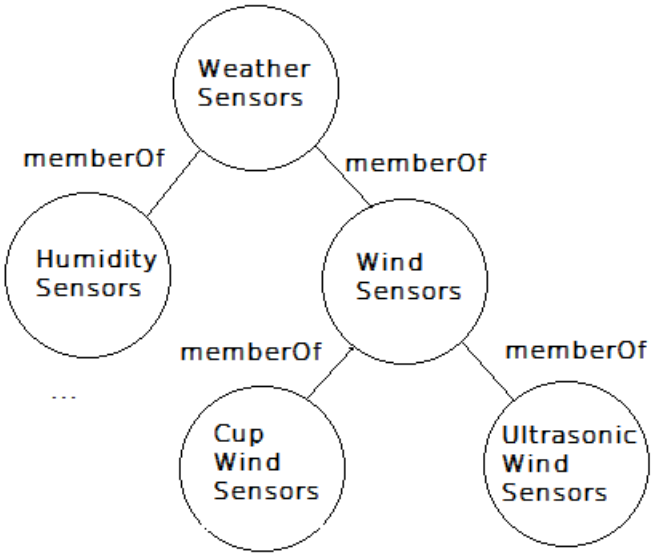

Figure 2.1. *Using triples in the example*

In a cyberphisical system stored in an SOS every sensor has a name and a type. These are identifiers that correspond to a sensor and there may be other descriptors that are not informative for human reader. Sensors can be called on different names like Anemometer, Wind instrument, windmeter that means the same, measures the speed of the wind. Filtering for name is not a good aproach for filtering the information. To make monitoring simpler. A semantic information is needed that organizes the data in a way that is convenient for the human reader to read. The sensors should be separated into different groups and subgroups and these groups should have an easily distinguishable name. Such groups can be Phyisical sensors -> Weather Sensors -> Wind sensors -> Wind speed sensors. This information yet can not be stored on the SOS server a separate ontology database shall be created. The usual way to store such information is in an RDF database.

2.2 Resource Definition Framework

The RDF standard[6] is created to be used with the Semantic Web approach to expand web pages with additional meanings that makes machines capable of understanding and reasoning about a web page. For example a web store can be easily understand by a customer: it has items, prices, shipping information, etc. However, for machines without saying explicitly that the value in one field is the price in USD it can be easily confused by the dimensions or the performance. Although nowadays these problems can be solved by machine learning, it is still a resource intensive process. To solve this problem an XML based standard has been introduced. This is done by using triples.

The triple describes which page or entity is connected on what property to which other page or entity. These triples are separate three unique values. Each value could have another triple describing it. Each unique identifier is a fully qualified domain name and a hash tag and a unique name, like a URL with an anchor on a web page. The first part is called the subject, the second one is the predicate, the third one is the object.

Such recursive data can be represented in graph databases, where reasoning is only walking in the database. There are graph databases to store these triples and also dedicated RDF databases. The standard way to query RDF databases is using SPARQL queries. Although RDF can represent data and connections it can not describe the rules how the reasoning, the walks in the graph should be done. These are represented by OWL or SWRL which are an extension to the RDF. Most RDF databases also support such rules.

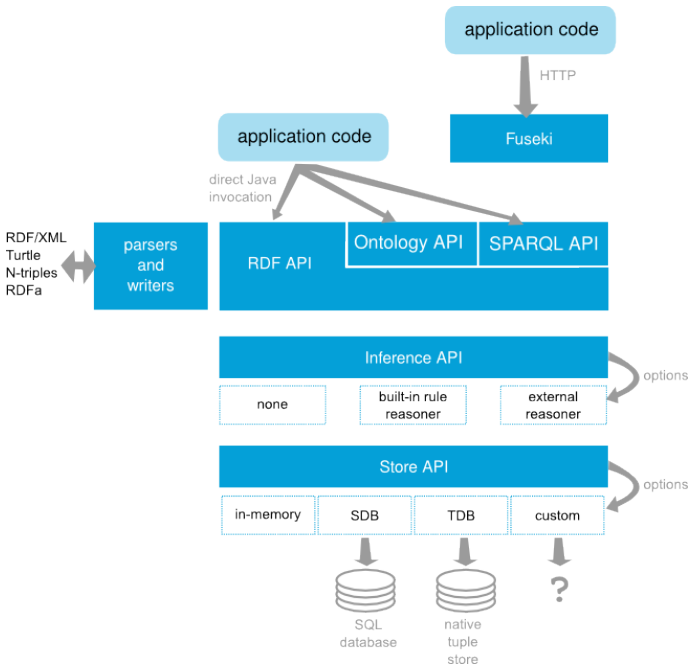

Figure 2.2. *Architecture of Apache Jena*

2.3 Some open souce RDF databases

Apache Jena is an open source RDF datastore supported by the Apache foundation. It is written in Java[4]. It has many interfaces and supports many database backends. It can be used with in memory databases, SQL RDBs, triplestores. There is a built-in reasoner in the datastore, however it can be changed to other external reasoners too. The architecture of the software is shown on figure 2.2.

OpenRDF Sesame is another tool for storing RDF triples. It is also written in Java. It has three different interfaces for communication: the SAIL API, the RIO interface and an HTTP client. The whole application runs from a Java container like Tomcat.

There is an out of the box tool that contains better reasoners, has a basic GUI and accepts different data types. This is Stardog database. It is a commercial application. However, it has a community version with a few restrictions. The software is written in Java and run from a web container, like Tomcat. It supports many interfaces such as HTTP and SNARL, it has a built in reasoner with integrated constraint validation. Connectors for different programming languages are ready to use. It can be easily queried using SPARQL. It has support for OWL 2 rules. Because it is an easy to use, out of the box tool this database is used in the sample application.

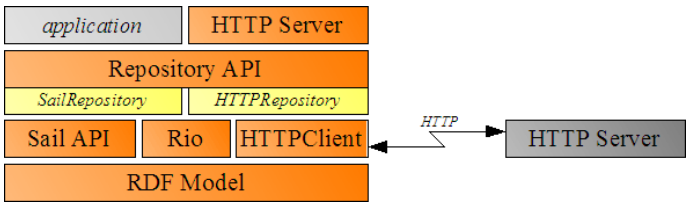

Figure 2.3. *Components of Sesame*

2.4 SPARQL for the queries

To retrieve the neccessary metadata the SPARQL query language is used in the RDF databases. These queries are less human readable than standard SQL queries, although they look similar. The language builds on the subject-predicate-object triples. In a result all matching data is responded. A SPARQL query usually has a selection part and a filtering part. The selection part shows which parameters should be returned. The filtering part shows which attributes are defined and it can also apply built-in functions for example language selection. Multiple conditions are separated with dots or commas. Commas are only used when the condition is about the same object as the previous query. A sample SPARQL query is shown in listing 2.1.

Listing 2.1. *Sample SPARQL that queries all filterable objects*

```
select ?s { <uri#filterable> <uri#subPropertyOf> ?s }
```

Chapter 3

The actual use case of SensorML and sensor ontology

3.1 Usage of the designed software

The designed monitoring software is made to be part of the system created for the Future Internet Research, Services and Technology project started by ETIK organization. The system is a prototype of a sensor network where the output of sensors can be used to create so called virtual sensors and store the data in a central data store. The system can reason using the ontology built on the sensors. A detailed introduction can be found in this chapter.

3.2 Goal of the project

A new virtual machine has been created for the test environment. It is using free and open source tools that run the application. Ubuntu 13.10 is used as the operating system. The virtual machine has a limited 512 MB of RAM, and a max. 10 GB storage. It's network card is hidden behind a NAT provided by the host computer. This makes easier to work on a laptop on different locations. Java Runtime Environment is installed on the virtual machine for Stardog and SOS. For the web application NodeJS has been set up.

The chosen RDF database is Stardog Community edition. At first the software preallocates too much memory. The startup script had to be changed to make the software start. Changing this parameter have not decreases the performance significantly, compared to another computer with more memory.

The chosen SOS server is 52north SOS 4.0. This is a recent release of the software. The used development version enables JSON communication and other experimental tools. The server requires PostgreSQL database backend and Tomcat application server.

PostgreSQL 9.1.12 is used as the database backend for SOS. PostGIS environment had to be added. The databes can be administrated using pgAdmin III from the host computer using port forward. For the test environment no new users has been added, the SOS server uses the admin user to connect to the database. The database had to be created manually but tables and configuration is added during the installation automatically.

Tomcat 7 is installed to support SOS. To keep the system separated a new user is created to run the container and the application. The Stardog database is also running in Tomcat like environment that is why it is started by the same user as the SOS server.

3.3 NodeJS: The base of the connector application

The connector application is written in JavaScript. The frontend and the backend of the application is the same language. Since the V8 engine exists JavaScript can be compiled and run significantly faster[5]. NodeJS builds upon the V8 engine and lets JavaScript run on backend. Another advantage is that NodeJS is single threaded, however event driven. That enables running applications faster, without worrying about thread safety. The architecture of the NodeJS environment can be seen on figure 4.1. To keep the integrator separated a different user is added to run server side code. No web application container is needed to run a NodeJS application. The software itself handles TCP connections and other modules help do it similar to Java Servlets. Because it has smaller overhead and dependencies, the created application should run faster than a Java Web application. NodeJS has an easy to use packaging system that makes dependency handling simple. All necessary modules names are added to the related part of the package.json file and required packages are downloaded using the npm install command from a central repository.

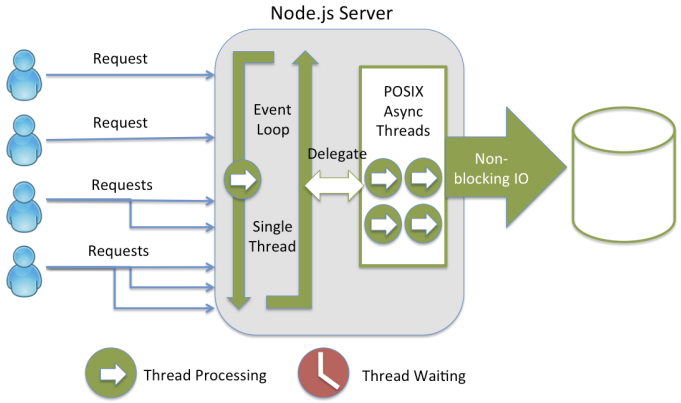

Figure 3.1. *NodeJS's event driven achitecture.*

3.4 Dependent modules

There are external modules that handle HTTP requests, creates responses and read JSON data. There is also a database connector for Stardog.

The ExpressJS module acts as the Servler engine for JavaScript. It handles incomming connections and parsed HTTP request are passed to a callback function which generates the responses. Different urls can have different callback functions to do routing. Express also supports many templating engines. EJS is a popular, easy to understand templating engine that was chosen for the project.

Restler is an easy to use module that can asynchronously read a web page and parse it as JSON data and return it to the callback function. This can be used to communicate with the SOS server.

Stardog.JS is Stardog database servers own connector to get SPARQL queries. It is in early stages, however all the necessary functions are working.

Nodemon is a utility that runs JavaScript codes automatically restarts them on error or code changes. This makes development easier, because every time the source is changed the application automatically restarts. It can be configured to restart on errors to make applications fail safe.

AngularJS is not a server side module, but plays a very important role in the software. This tool is for the browser and it puts a MVC layer on top of the webpage. With the help of Angular, parts of a page can be changed asynchronously based on the value read from the model. It makes the pageload faster by firstly loading the frame of the application and only later inserting the read data from the database. Angular is maintained by Google and it is used in many of their web applications.

3.5 RDF representation of the SOS data

The meta information and the procedure id of an SOS sensor has to be stored in the RDF database to build a semantic representation on top of it. Although a transparent transformation is ready to create RDF XML files from SOS data export, the hierarchy of the sensors is still varying. There are two different approaches.

The first one is to create a tree on top of the sensors manually to describe the connections between each sensor. This often needs to be changed when new sensors arrive in the system. However, this tree can be exported and shared with others just like DBPedia ontologies. This can be also efficiently queried.

The second approach is to use a less redundant way by only creating rules that make sensors part of groups. These rules add a virtual groups based on the conditions defined. This is a resource intensive process that has to be re-run at every query.

For the experiment the first way is used on a small sample ontology. The hierarchy was manually created and some properties have been added. These properties were annotated as filterable or observable. Observable meaning that the property can be seen from the monitoring system and filterable meaning that its value can be given to advance the query.

The structure of the translation is not as straight forward as the created database, however it can be represented in the same way using rules.

3.6 Using the RDF database

To connect to the database the Stardog.js library has been used. The statically retrieved web page asynchronously requested the list API call to get a list of the filterable properties and their range. The ranges can be either other entities, doubles, integers or strings. Depending on what the range of the property is a corresponding input box shall be rendered.

When the AJAX call responds the Angular script reloads the data with the response.

On the backend a separate module handles the API calls, connects to the database and runst the SPARQL request. The result and additional information is returned as JSON response.

3.7 Connecting the SOS server

The JSON API can be reached easily using the Restler module to retrieve information from the SOS server. The server responds to the GetCapabilities queries with its list of the sensors and their parameters. However, yet there has not been progress in implementing the GetObservation command in the JSON interface. The JSON API is still under heavy development, thus further connection using this interface can not be done.

3.8 Solving the challenge

To solve the issue of connecting to the JSON interface there can be three different ways.

The first one is to connect to the SOS server using another interface. There are SOAP interfaces implemented for NodeJS, by adding this layer the GetObservation method can be queried.

The second method is to implement the GetObservation method in the 52north SOS server. The development environment has been already set up, after examining the code the changes shoul be made easily.

Because of the many changes that are needed to display SOS data changing to an existing SOS client should be a great step forward. The existing client can be extended by semantic functions using the Stardog library provided and the some SPARQL queries. The development environment for the SWE Client is also ready to use.

3.9 Using the Software

Thanks to NodeJS package management the software is easy to deploy. After installing NodeJS - in Windows there is a straight-forward installer, for Ubuntu it can be easily installed using apt - the git project has to be cloned to the server. In the project directory (sensormonitor) the npm install command installs the missing modules and the npm start command starts the application with Nodemon. The url of the rdf database can be changed in rdf_parser.js. The installation commands are shown on listing 4.1

Listing 3.1. *Install steps for the software*

```
#0. install NodeJS and npm
sudo apt-get install nodejs
#1. clone the project to your destination
git clone --depth 1 https://djlancelot@bitbucket.org/djlancelot/sensormonitor.git
#2. change into project folder
cd sensormonitor
#3. install missing dependencies
npm install
#4. start the application
npm start
```

Chapter 4

The Implementation

4.1 Goal of the implementation

After introducing the two different data types, in this part a sample application is shown that can handle connection to both data types.

The goal was to show how a connection can be created from third party applications. In further chapters the integration of the different types will be shown. The meaning of this phase is to seek for new ways to implement web application using the connectors provided and measure the needs for such a web application. A whole test system were created to enable this.

4.2 The test environment

A new virtual machine has been created for the test environment. It is using free and open source tools that run the application. Ubuntu 13.10 is used as the operating system. The virtual machine has a limited 512 MB of RAM, and a max. 10 GB storage. It's network card is hidden behind a NAT provided by the host computer. This makes easier to work on a laptop on different locations. Java Runtime Environment is installed on the virtual machine for Stardog and SOS. For the web application NodeJS has been set up.

The chosen RDF database is Stardog Community edition. At first the software preallocates too much memory. The startup script had to be changed to make the software start. Changing this parameter have not decreases the performance significantly, compared to another computer with more memory.

The chosen SOS server is 52north SOS 4.0. This is a recent release of the software. The used development version enables JSON communication and other experimental tools. The server requires PostgreSQL database backend and Tomcat application server.

PostgreSQL 9.1.12 is used as the database backend for SOS. PostGIS environment had to be added. The databes can be administrated using pgAdmin III from the host computer

using port forward. For the test environment no new users has been added, the SOS server uses the admin user to connect to the database. The database had to be created manually but tables and configuration is added during the installation automatically.

Tomcat 7 is installed to support SOS. To keep the system separated a new user is created to run the container and the application. The Stardog database is also running in Tomcat like environment that is why it is started by the same user as the SOS server.

4.3 NodeJS: The base of the connector application

The connector application is written in JavaScript. The frontend and the backend of the application is the same language. Since the V8 engine exists JavaScript can be compiled and run significantly faster[5]. NodeJS builds upon the V8 engine and lets JavaScript run on backend. Another advantage is that NodeJS is single threaded, however event driven. That enables running applications faster, without worrying about thread safety. The architecture of the NodeJS environment can be seen on figure 4.1. To keep the integrator separated a different user is added to run server side code. No web application container is needed to run a NodeJS application. The software itself handles TCP connections and other modules help do it similar to Java Servlets. Because it has smaller overhead and dependencies, the created application should run faster than a Java Web application. NodeJS has an easy to use packaging system that makes dependency handling simple. All necessary modules names are added to the related part of the package.json file and required packages are downloaded using the npm install command from a central repository.

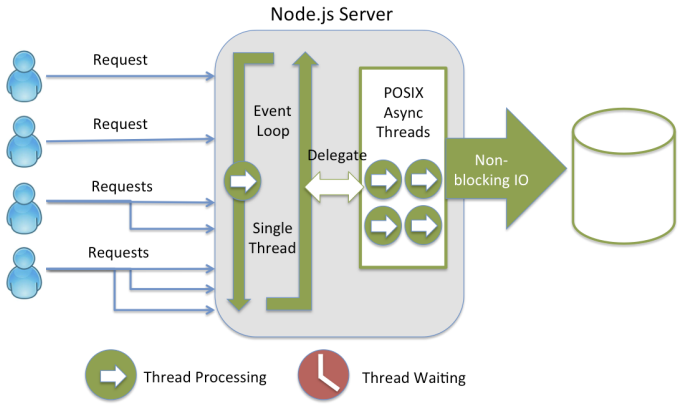

Figure 4.1. *NodeJS's event driven achitecture.*

4.4 Dependent modules

There are external modules that handle HTTP requests, creates responses and read JSON data. There is also a database connector for Stardog.

The ExpressJS module acts as the Servler engine for JavaScript. It handles incomming connections and parsed HTTP request are passed to a callback function which generates

the responses. Different urls can have different callback functions to do routing. Express also supports many templating engines. EJS is a popular, easy to understand templating engine that was chosen for the project.

Restler is an easy to use module that can asynchronously read a web page and parse it as JSON data and return it to the callback function. This can be used to communicate with the SOS server.

Stardog.JS is Stardog database servers own connector to get SPARQL queries. It is in early stages, however all the necessary functions are working.

Nodemon is a utility that runs JavaScript codes automatically restarts them on error or code changes. This makes development easier, because every time the source is changed the application automatically restarts. It can be configured to restart on errors to make applications fail safe.

AngularJS is not a server side module, but plays a very important role in the software. This tool is for the browser and it puts a MVC layer on top of the webpage. With the help of Angular, parts of a page can be changed asynchronously based on the value read from the model. It makes the pageload faster by firstly loading the frame of the application and only later inserting the read data from the database. Angular is maintained by Google and it is used in many of their web applications.

4.5 RDF representation of the SOS data

The meta information and the procedure id of an SOS sensor has to be stored in the RDF database to build a semantic representation on top of it. Although a transparent transformation is ready to create RDF XML files from SOS data export, the hierarchy of the sensors is still varying. There are two different approaches.

The first one is to create a tree on top of the sensors manually to describe the connections between each sensor. This often needs to be changed when new sensors arrive in the system. However, this tree can be exported and shared with others just like DBPedia ontologies. This can be also efficiently queried.

The second approach is to use a less redundant way by only creating rules that make sensors part of groups. These rules add a virtual groups based on the conditions defined. This is a resource intensive process that has to be re-run at every query.

For the experiment the first way is used on a small sample ontology. The hierarchy was manually created and some properties have been added. These properties were annotated as filterable or observable. Observable meaning that the property can be seen from the monitoring system and filterable meaning that its value can be given to advance the query. The structure of the translation is not as straight forward as the created database, however it can be represented in the same way using rules.

4.6 Using the RDF database

To connect to the database the Stardog.js library has been used. The statically retrieved web page asynchronously requested the list API call to get a list of the filterable properties and their range. The ranges can be either other entities, doubles, integers or strings. Depending on what the range of the property is a corresponding input box shall be rendered.

When the AJAX call responds the Angular script reloads the data with the response.

On the backend a separate module handles the API calls, connects to the database and runst the SPARQL request. The result and additional information is returned as JSON response.

4.7 Connecting the SOS server

The JSON API can be reached easily using the Restler module to retrieve information from the SOS server. The server responds to the GetCapabilities queries with its list of the sensors and their parameters. However, yet there has not been progress in implementing the GetObservation command in the JSON interface. The JSON API is still under heavy development, thus further connection using this interface can not be done.

4.8 Solving the challenge

To solve the issue of connecting to the JSON interface there can be three different ways.

The first one is to connect to the SOS server using another interface. There are SOAP interfaces implemented for NodeJS, by adding this layer the GetObservation method can be queried.

The second method is to implement the GetObservation method in the 52north SOS server. The development environment has been already set up, after examining the code the changes shoul be made easily.

Because of the many changes that are needed to display SOS data changing to an existing SOS client should be a great step forward. The existing client can be extended by semantic functions using the Stardog library provided and the some SPARQL queries. The development environment for the SWE Client is also ready to use.

4.9 Using the Software

Thanks to NodeJS package management the software is easy to deploy. After installing NodeJS - in Windows there is a straight-forward installer, for Ubuntu it can be easily installed using apt - the git project has to be cloned to the server. In the project directory

(sensormonitor) the npm install command installs the missing modules and the npm start command starts the application with Nodemon. The url of the rdf database can be changed in rdf_parser.js. The installation commands are shown on listing 4.1

Listing 4.1. *Install steps for the software*

```
#0. install NodeJS and npm
sudo apt-get install nodejs
#1. clone the project to your destination
git clone --depth 1 https://djlancelot@bitbucket.org/djlancelot/sensormonitor.git
#2. change into project folder
cd sensormonitor
#3. install missing dependencies
npm install
#4. start the application
npm start
```

Acknowledgement

The author is thankful for the help and support of Dr. György Strausz and Dr. András Förhécz.

Bibliography

[1] 52north org. Sensor observation service.

[2] Mike Botts. Ogc sensorml: Model and xml encoding standard.

[3] MapServer Community. Mapserver webpage.

[4] Apache Foundation. Getting started.

[5] Google. Chrome v8.

[6] RDF Working Group. Resource description framework (rdf).

[7] istSOS Community. istsos webpage.

Appendices

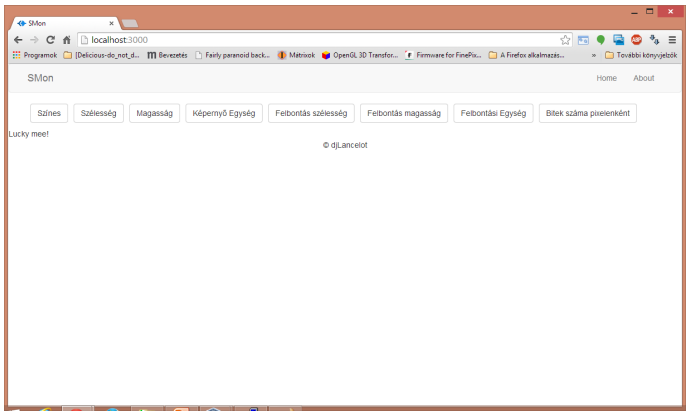

Figure F.0.2. *Screenshot of the test application*

Listing F.0.2. *Client side JavaScript of test application*

```
var smonApp = angular.module('smonApp',['smonSrv']);
  smonApp.controller('TestController',function($scope){
        $scope.message = 'Lucky mee!';
  });
  smonApp.controller('formController',function($scope){
  });
smonApp.controller('listController',['$scope','GetList',
 function($scope, List){
        $scope.addElement=function(id, range){
                switch(range)
                {
                        case "http://www.w3.org/2001/XMLSchema#double":
                        case "http://www.w3.org/2001/XMLSchema#string":
                        case "http://www.w3.org/2001/XMLSchema#integer":
                                break;
                }
                http://www.w3.org/2001/XMLSchema#double
        console.log(range);
        id = '<input type="text"/>';
        }
        $scope.elements = List.query();
  }]);
var smonSrv = angular.module('smonSrv', ['ngResource']);
smonSrv.factory('GetList', ['$resource',
  function($resource){
    return $resource('/api/list', {}, {
      query: {method:'GET', isArray:true}
    });
  }]);
```

Listing F.0.3. *Code for the RDF parser*

```
var stardog = require("stardog");
var conn = new stardog.Connection();
conn.setEndpoint("http://localhost:5820/");
conn.setCredentials("admin", "admin");
exports.index = function(req, res) {
        res.sendfile('./public/index.html'); // load the single view file (angular
            will handle the page changes on the front-end)
};
var getSearchables = function(callback){
        var q = "select ?property ?label ?range{?property
            <http://kli.uni-muenster.de/stations/hbs#filterable> true. ";
                q = q+ "?property <http://www.w3.org/2000/01/rdf-schema#label>
                    ?label. FILTER langMatches( lang(?label), \"hu\" ) ";
                q = q + "?property <http://www.w3.org/2000/01/rdf-schema#range>
                    ?range }";

        conn.query({
                database: "smon",
                query: q,
                limit: 10,
                offset: 0
            }, callback);

};
exports.get_searchables = function(req, res){
         var cb =  function (data) {
                console.log(data.results.bindings);
                res.json(data.results.bindings);
        };
        getSearchables(cb);
};
exports.list_searchables = function(req, res){
         var cb =  function (data) {
                console.log(data.results.bindings);
                res.render('listall', { title: 'List searchable', bindings:
                    data.results.bindings });
        };
        getSearchables(cb);
};
exports.list_all = function(req, res){
        conn.query({
                database: "smon",
                query: "select distinct ?s where { ?s ?p ?o }",
                limit: 10,
                offset: 0
            },
            function (data) {
                console.log(data.results.bindings);
                res.render('listall', { title: 'List all', bindings:
                    data.results.bindings });
        });

};
```

Listing F.0.4. *Angular webpage*

```
<!DOCTYPE html>
<html ng-app="smonApp">
<head>
  <title>SMon</title>
  <link rel="stylesheet"
      href="https://netdna.bootstrapcdn.com/bootstrap/3.0.0/css/bootstrap.min.css"
      />
</head>
<!-- define angular controller -->
<body ng-controller="TestController">
  <nav class="navbar navbar-default">
    <div class="container">
      <div class="navbar-header">
        <a class="navbar-brand" href="/">SMon</a>
      </div>

      <ul class="nav navbar-nav navbar-right">
        <li><a href="#"><i class="fa fa-home"></i> Home</a></li>
        <li><a href="#about"><i class="fa fa-shield"></i> About</a></li>
      </ul>
    </div>
  </nav>
<div class="container" ng-controller="listController">
        <ul class="list-inline">
                <li ng-repeat="el in elements">
                        <a href="" ng-click="addElement(el.input,el.range.value)"
                            class="btn btn-default"> {{el.label.value}}</a>
                        <span ng-bind="el.input"></span>
                </li>
        </ul>
</div>
  <div id="main">
                <!-- this is where content will be injected -->
                {{message}}
  </div>
  <footer class="text-center">
   &copy; djLancelot
  </footer>
    <!-- SPELLS -->
        <script src="/javascripts/angular.js"></script>
        <script src="/javascripts/angular-route.js"></script>
        <script src="/javascripts/angular-resource.js"></script>
        <script src="/javascripts/smon.js"></script>
</body>
</html>
```