



Project No. 249024

NETMAR

Open service network for marine environmental data

Instrument: <i>Please tick</i>	CA	<input checked="" type="checkbox"/> STREP	IP	NOE
--	----	---	----	-----

ICT - Information and Communication Technologies Theme

D7.7 WPS Cookbook

Reference: D7.7_WPS_Cookbook_r1_20111229

Due date of deliverable (as in Annex 1): M0 + 23
Actual submission date: 29 December 2011

Start date of project: 1 February 2010

Duration: 3 years




Plymouth Marine Laboratory

Revision 1

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	



European Commission
Information Society and Media

 	<p>NETMAR Open service network for marine environmental data Project Reference: 249024 Contract Type: Collaborative Project Start/End Date: 01/03/2010 - 31/01/2013 Duration: 36 months</p>
	<p>Coordinator: Prof. Stein Sandven Nansen Environmental and Remote Sensing Center Thormøhlensgate 47, Bergen, Norway Tel.: +47 55 20 58 00 Fax: +47 55 20 58 01 E-mail: stein.sandven@nersc.no</p>

Acknowledgements

The work described in this report has been partially funded by the European Commission under the Seventh Framework Programme, Theme ICT 2009.6.4 ICT for environmental services and climate change adaptation.

Consortium

The NETMAR Consortium is comprised of:

- Nansen Environmental and Remote Sensing Center (NERSC), Norway (coordinator).
Project Coordinator: Prof. Stein Sandven (stein.sandven@nersc.no)
Deputy Coordinator: Dr. Torill Hamre (torill.hamre@nersc.no)
Quality Control Manager: Mr. Lasse H. Pettersson (lasse.pettersson@nersc.no)
- British Oceanographic Data Centre (BODC), National Environment Research Council, United Kingdom
Contact: Dr. Roy Lowry (rkl@bodc.ac.uk)
- Centre de documentation de recherche et d'expérimentations sur les pollutions accidentelles des eaux (Cedre), France.
Contact: Mr. François Parthiot (Francois.Parthiot@cedre.fr)
- Coastal and Marine Resources Centre (CMRC), University College Cork, National University of Ireland, Cork, Ireland.
Contact: Mr. Declan Dunne (d.dunne@ucc.ie)
- Plymouth Marine Laboratory (PML), United Kingdom.
Contact: Mr. Steve Groom (sbg@pml.ac.uk)
- Institut français de recherche pour l'exploitation de la mer (Ifremer), France.
Contact: Mr. Mickael Treguer (mickael.treguer@ifremer.fr)
- Norwegian Meteorological Institute (METNO), Norway.
Contact: Mr. Øystein Torgest (oysteint@met.no)

Author(s)

- Jorge de Jesus, PML, (jmdj@pml.ac.uk)
- Peter Walker, PML, (petwa@pml.ac.uk)

Document approval

- Document status: Revision 1
- WP leader approval: 21 December 2011
- Quality Manager approval: 29 December 2011
- Coordinator approval: 29 December 2011

Revision History

Issue	Date	Change records	Author(s)
Draft	2011-12-13	First draft of the report.	Jorge de Jesus
Draft	2011-12-21	Revisions from review comments	Peter Walker, Jorge de Jesus
1	2011-12-29	Final release approved by coordinator.	Torill Hamre

Executive Summary

The NETMAR project will develop a pilot European Marine Information System (EUMIS) that will enable users to search, download and integrate satellite, in situ and model data from ocean and coastal areas. EUMIS will be a user-configurable system offering flexible service discovery, access and chaining facilities based on open and widely adopted web GIS standards. To support smart search, it will use a semantic framework coupled with ontologies for identifying and accessing distributed data, such as near-real time, forecast and historical data, which are marked up using different, but semantically related, keywords. To support dynamic generation of new composite products and statistics suitable for decision-making, it will use web processing services that can be chained together to form workflows that perform a series of operations on input data chosen by the user.

The EUMIS pilot will target the needs of four user communities:

- Arctic operators, environmental authorities and fishery management
- Oil spill drift forecast and shoreline cleanup assessment services in France
- Ocean colour – Marine Ecosystem, Research and Monitoring users
- The International Coastal Atlas Network (ICAN)

Each user community will have a customised pilot that provides the products and services needed for their line of work. The NETMAR partners have had close contact with selected users in each community to identify and document their needs. These user requirements have been described as a series of use cases and a joint list of requirements, which forms the basis for the EUMIS development. As part of this process, the NETMAR partners have also identified a number of data delivery and processing services that are necessary for each pilot. NETMAR makes heavy use of the Open Geospatial Consortium (OGC), Web Processing Service (WPS) standard when implementing processing services. This document provides a guide to implementers who wish to use WPS (specifically PyWPS) and chain processes provided by these services within their own systems.

WPS defines the protocol by which service providers can expose their geoprocessing services on the web. It specifies how the service should describe itself, how processes should be called and how they return data; allowing simple web clients to make use of potentially complex services.

Service chaining further extends the power of WPS by allowing users to “chain” the output of one process into another, building a custom processing service to fit their needs. These chained processes could themselves be packaged as WPS processes to make it easy for third parties to use them.

Contents

1.	Introduction.....	5
2.	WPS Overview	5
2.1.	Server level metadata (GetCapabilities)	7
2.2.	Process level metadata (DescribeProcess)	8
2.2.1.	Description of Data Input and Process Outputs	10
	<i>Literal Data</i>	10
	<i>ComplexData description</i>	11
	<i>Bounding box Data description</i>	12
2.3.	Process execution (Execute)	13
2.3.1.	KVP Execute request.....	14
2.3.2.	storeExecuteResponse and status parameters	15
2.3.3.	Lineage.....	16
2.3.4.	Response Document.....	16
2.3.5.	RawResponse.....	17
2.3.6.	XML Execute request.....	17
2.4.	Locale definitions	19
2.5.	WPS Exceptions.....	20
2.6.	SOAP Binding	20
3.	PyWPS – a WPS implementation.....	27
3.1.	Introduction	27
3.2.	PyWPS.....	28
3.2.1.	Installation	28
3.2.2.	Core PyWPS installation	28
3.2.3.	Installing PyWPS code.....	29
3.2.4.	WPS-GRASS-BRIDGE INSTALL (for extra WPS processes)	32
3.2.5.	WPS-Grass-Bridge Add-on	34
3.2.6.	WSDL generation	34
4.	Taverna workbench.....	35
4.1.	OGC proxy	35
4.2.	Using Taverna	37

4.2.1.	WPS-Async call in Taverna.....	48
4.3.	myExperiment.....	50
5.	Document Information.....	52
6.	References.....	53

1. Introduction

This document provides a guide to implementers who wish to use WPS (specifically PyWPS) and chain processes provided by these services within their own systems.

WPS defines the protocol by which service providers can expose their geoprocessing services on the web. It specifies how the service should describe itself, how processes should be called and how they return data; allowing simple web clients to make use of potentially complex services.

Service chaining further extends the power of WPS by allowing users to “chain” the output of one process into another, building a custom processing service to fit their needs. These chained processes could themselves be packaged as WPS processes to make it easy for third parties to use them.

The following sections will explain the basics of the WPS standard, how to implement a WPS server (PyWPS), add contributed processes to it and finally combine these processes in complex workflows using Taverna Workbench.

The first section is aimed at users who have a basic understanding of OGC standards and wish to learn more about WPS. The later sections are aimed at more technical users who might wish to implement these services. These users should be familiar with the operating system on which they plan to install.

2. WPS Overview

Web Processing Service (WPS) is an OGC (Open Geospatial Consortium) standard, defining how a geospatial calculation/model/process can be discovered and executed in a Service Oriented Architecture (SOA) strategy, by using instances exposed via HTTP-GET, HTTP-POST and SOAP internet protocols. In theory WPS should be able to describe any process, run it using pre-defined input/output and report error and status.

As defined in wikipedia¹:

*“... provides rules for standardizing **inputs and outputs** (requests and responses) for **geospatial processing services**, such as polygon overlay. The standard also defines **how a client can request the execution of a process, and how the output from the process is handled**. It defines an **interface that facilitates the publishing of geospatial processes and clients’ discovery of and binding to those processes**. The data required by the WPS can be delivered across a network or they can be available at the server. “*

WPS defines three operations, two for requesting metadata and a third to execute the process:

¹ http://en.wikipedia.org/wiki/Web_Processing_Service

- *GetCapabilities* - Generic WPS service metadata; the list of processes available on the server for instance
- *DescribeProcess* - Full description of a process including input and output parameters
- *Execute* - Execution of a process using supplied inputs, returning outputs as requested

Figure 1 illustrates a simple interaction between a client and a WPS server.

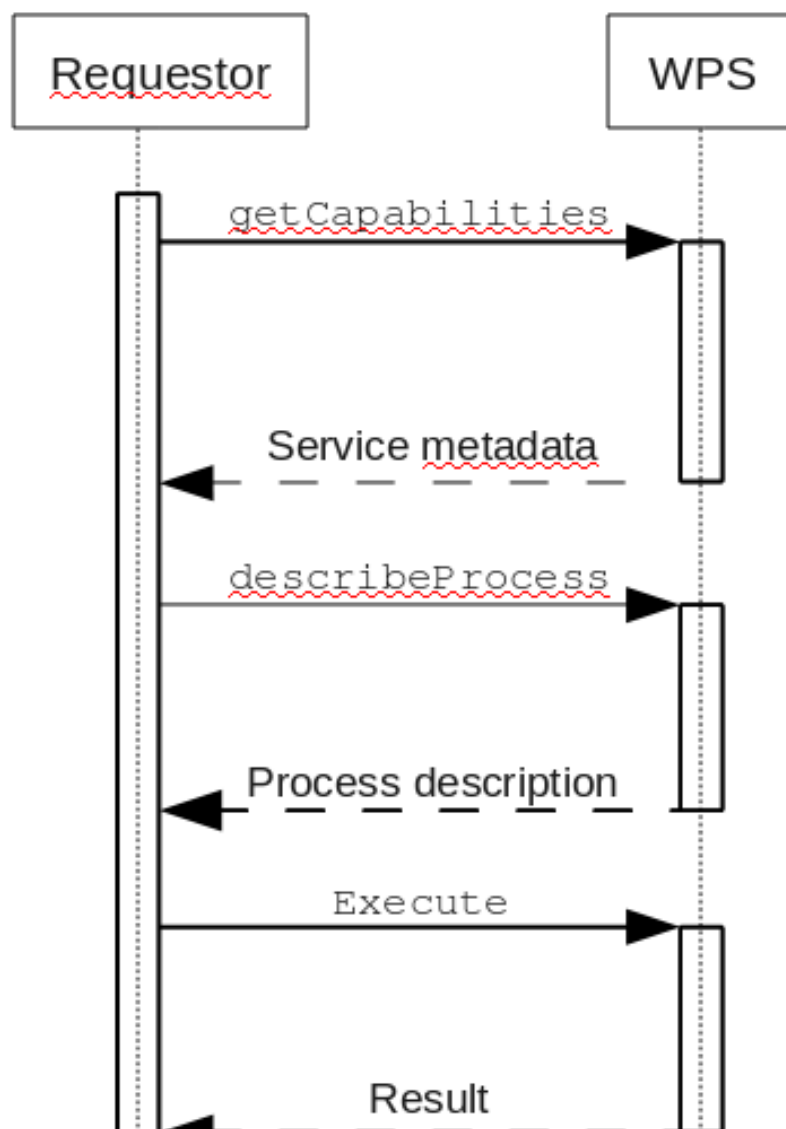


Figure 1 - Basic WPS interaction pattern. In "Towards a Transactional Web Processing Service" [1]

2.1. Server level metadata (GetCapabilities)

Under WPS a number of processes may run on a single server, the *GetCapabilities* request will return metadata about the server and list all processes it provides.

1. Service identification (keywords and abstract describing the service)
2. Service provider (who provides the service and how to contact the person in charge (name, telephone etc.))
3. Operation metadata (HTTP GET and POST description and links to operations)
4. Process offering (List of processes; for each process: abstract, identifier and metadata)
5. Languages supported (Human languages supported by the service, normally the default value is English)
6. WSDL (Web Service Description Language) file location (URL to Web Service Description Language file, allowing for use of service in other SOA structures that support WSDL)

GetCapabilities may be issued as a KVP (Key-Value-Pair) request (using HTTP GET) with 2 mandatory parameters, REQUEST and SERVICE, as indicated in listing 1:

```
http://hostname/wpsInstance?service=WPS&request=GetCapabilities
```

Listing 1 - *GetCapabilities* KVP request.

GetCapabilities may also be issued as an XML request submitted via HTTP POST to the WPS (Listing 2):

```
<?xml version="1.0" encoding="UTF-8"?>
<wps:GetCapabilities xmlns:ows="http://www.opengis.net/ows/1.1"
xmlns:wps="http://www.opengis.net/wps/1.0.0"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
"http://www.opengis.net/ows/1.1 .. %wpsGetCapabilities_request.xsd"
language="en-CA" service="WPS">
  <wps:AcceptVersions>
  <ows:Version>1.0.0</ows:Version>
  </wps:AcceptVersions>
</wps:GetCapabilities>
```

Listing 2 - *GetCapabilities* XML request.

Whichever form it is issued in the request shall return an XML response document in which the client will be informed of the processes being offered by the WPS instance along with a brief description (Listing 3):

```
<wps:Process wps:processVersion="1.0">
  <ows:Identifier>intersectBBOX</ows:Identifier>
  <ows:Title>intersection between 2 bounding boxes</ows:Title>
  <ows:Abstract>intersectBBOX calculates the intersection between 2
  bounding boxes</ows:Abstract>
</wps:Process>
<wps:Process wps:processVersion="2.0">
  <ows:Identifier>ultimatequestionprocess</ows:Identifier>
  <ows:Title>Answer to Life, the Universe and Everything</ows:Title>
  <ows:Abstract>Numerical solution that is the answer to Life,
  Universe and Everything. The process is an improvement to Deep Thought
  computer (therefore version 2.0) since it no longer takes
  7.5 million years, but only a few seconds to give a response,
  with an update of status every 10 seconds.</ows:Abstract>
</wps:Process>
```

Listing 3 - Example of list of processes returned from a *GetCapabilities* request.

2.2.Process level metadata (*DescribeProcess*)

The *DescribeProcess* request provides means for a client to determine what are mandatory, optional, and default parameters for a particular process, as well as the format of the data inputs and outputs.

```
http://hostname/wpsInstance?  
request=describeProcess&  
identifier=reprojectCoords&  
service=WPS&version=1.0.0
```

Listing 4 - Example of a KVP *DescribeProcess* for a process identified as "reprojectCoords".

The SERVICE and VERSION are mandatory parameters in the request. The identifier parameter also allows for a **generic "all" value** that will return an XML response with detail **description for all the services offered by the instance**.

```
<wps:DescribeProcess xmlns="http://www.opengis.net/wps/1.0.0"  
  xmlns:ows="http://www.opengis.net/ows/1.1"  
    xmlns:xlink="http://www.w3.org/1999/xlink"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation=  
      "http://www.opengis.net/wps/1.0.0 ../wpsDescribeProcess_request.xsd"  
  service="WPS" version="1.0.0">  
  <ows:Identifier>reprojectCoords</ows:Identifier>  
</wps:DescribeProcess>
```

Listing 5 - XML *DescribeProcess* request equivalent to the KVP request in listing 4.

The *DescribeProcess* response document will return the following information:

1. Service identification (Title and abstract)
2. Service storage and status supporting
3. Identifier (Service's unique identifier)
4. Data Inputs
5. Data Output

```
<wps:ProcessDescriptions
xsi:schemaLocation="http://www.opengis.net/wps/1.0.0
http://schemas.opengis.net/wps/1.0.0/wpsDescribeProcess_response.xsd"
service="WPS" version="1.0.0" xml:lang="en-CA">
<ProcessDescription wps:processVersion="1.0" storeSupported="true"
statusSupported="true">
<ows:Identifier>reprojectCoords</ows:Identifier>
<ows:Title>reproject coordinates</ows:Title>
<ows:Abstract>reprojectCoords uses gdaltransform to reproject a list of
coordinates from one projection to another</ows:Abstract>
<ows:Metadata></ows:Metadata>
[. . ]
<DataInputs>
<Input>[. . ]</Input>
<Input>[. . ]</Input>[. . ]
</DataInputs>
<ProcessOutputs>
<Output>[. . ]</Output>
```

Listing 6 - Structure of a *describingProcess* response document (some additional output, [..] has been removed for clarity).

2.2.1. Description of Data Input and Process Outputs

Three types of inputs and outputs are defined in the OGC standard:

- LiteralData
- ComplexData
- BoundingBox data

Literal Data

LiteralData can be any character, string, float, date, etc., normally described as Primitive datatype in the W3C XML Schema standard¹. The WPS standard also allows the use of UOM (Unit of Measures), default values and AllowedValues (defining a range of allowed values). In case of missing inputs for an optional input the service will use a default value.

For example a *describeProcess* response may contain the following input description (Listing 7):

¹ <http://www.w3.org/TR/xmlschema-2/#built-in-primitive-datatypes>

```

<Input minOccurs="0" maxOccurs="1"><ows:Identifier>method</ows:Identifier>
  <ows:Title>Interpolation method</ows:Title>
  <ows:Abstract>Interpolation method to be used in a
    dataset of points</ows:Abstract>
  <LiteralData>
  <ows:DataType ows:reference="xsd:string">string</ows:DataType>
  <ows:AllowedValues>
    <ows:Value>idw</ows:Value>
    <ows:Value>kriging</ows:Value>
    <ows:Value>thysson</ows:Value>
  </ows:AllowedValues>
  <DefaultValue>kriging</DefaultValue>

```

Listing 7 - DescribeProcess response describing LiteralData input of type string indicating allowed values and default value.

The XML in Listing 7 defines that input "method" that is of type string, can have one of the following input values: "idw", "kriging", and "thysson" and if no input is present, "kriging" will be used as the default value.

```

<Input minOccurs="0" maxOccurs="1">
  <ows:Identifier>BufferDistance</ows:Identifier>
  <ows:Title>Buffer Distance</ows:Title>
  <ows:Abstract>Distance to be used to calculate buffer.</ows:Abstract>
  <LiteralData>
  <ows:DataType ows:reference="xsd:float">float</ows:DataType>
  <UOMs>
    <Default>
      <ows:UOM>meters</ows:UOM>
    </Default>
    <Supported>
      <ows:UOM>meters</ows:UOM>
      <ows:UOM>feet</ows:UOM>
    </Supported>
  </UOMs>
  <ows:AnyValue/>
  <DefaultValue>100</DefaultValue>
</LiteralData>

```

Listing 8 - DescribeProcess response of a buffer service that indicates the support of different units of measure (UOM): meters or feet.

In this case (Listing 8) we have an input that is a float number and whose units can be in meters or feet, if the input is not specified then use 100.0 meters as default.

ComplexData description

The ComplexData type is used to describe vector, raster (or any other data) . There are two ways to handle complex data:

- Embedded in the XML content, for example a GML content inside the ComplexData element or a binary raster data encoded in base64 [2]
- URL reference to the data or service that will provide data

ComplexData content is loosely defined by the WPS schema. This element's content is defined as "xsd:anyType", no data format or imposition is defined. Despite its loose content ComplexData element may contain a MimeType/Encoding/Schema triplet that describes the XML content. (Listing 9).

```
<Input minOccurs="1" maxOccurs="1">
<ows:Identifier>inputPolygon</ows:Identifier>
<ows:Title>Polygon to be buffered</ows:Title>
<ows:Abstract>GML that describes the polygon to be buffered</ows:Abstract>
<ComplexData>
  <Default>
  <Format>
  <MimeType>application/gml+xml</MimeType>
  <Encoding>UTF-8</Encoding>
  <Schema>http://schemas.opengis.net/gml/3.1.0/base/geometryBasic2d.xsd</Schema>
  </Format>
  </Default>
  <Supported>
  [...]
  </Supported>
</ComplexData>
<Input minOccurs="1" maxOccurs="1">
<ows:Identifier>inputImage</ows:Identifier>
<ows:Title>Input image</ows:Title>
<ComplexData>
  <Default>
  <Format>
  <MimeType>image/tiff</MimeType>
  </Format>
  </Default>
</Input>
```

Listing 9 - "Technical Committee Policies and Procedures: MIME Media Types for GML - OGC 09-144r1" [2].

Listing 9, describes two complex inputs that are mandatory (`minOccurs="1"`), that will be identified as `inputPolygon` and `inputImage`. ComplexData `inputPolygon` is a GML3.1.0 content defined in schema `geometryBasic2d.xsd` and encoded in UTF-8, ComplexData `inputImage` is a tiff image.

Bounding box Data description

Bounding Box, or just BBOX is the third data type and it is used to describe a bounding box area. BBOX is defined in the document "Web Services Common Specification 06-121r3" [3]. The input description must state the default coordinate reference system (CRS) (normally a URI of the EPSG code system) and other CRS supported.

```
<Input>
<ows:Identifier>bboxInput</ows:Identifier>
  <ows:Title>bounding box of dummy polygon</ows:Title>
  <ows:Abstract>Bounding box of dummy polygon to be used for fast
    polygon interception calculation</ows:Abstract>
<BoundingBoxData>
  <Default>
<CRS>urn:ogc:def:crs:EPSG:6.6:4326</CRS>
  </Default>
  <Supported>
  <CRSsType>
    <CRS>urn:ogc:def:crs:EPSG:6.6:4326</CRS>
    <CRS>urn:ogc:def:crs:EPSG:6.6:4979</CRS>
  </CRSsType>
  </Supported>
```

Listing 10 - Bounding Box Input description. Coordinate reference system (CRS) is defined as URI.

Listing 10 indicates a BBOX input where the default CSR is in geographical projection (lat/long, EPSG:4326) and optionally EPSG:4979 could also be used (which is an alias to EPSG:4326).

A KVP (see below) Execute request with a BBOX input would be like:

```
..&bboxInput=71.63.41.75,-70.78.42.90,urn:ogc:def:crs:EPSG:6.6:4326
```

2.3. Process execution (Execute)

The Execute request is the request that will launch the specified process implemented by the WPS instance. In an Execute request a client must specify:

1. Process identifier
2. Input values as defines in the *DescribeProcess*
3. Version and language
4. Type of Output either:
 - Stored in the server
 - Contained inside the XML response
 - Raw response of single output, dump the result to the client (for example an image)
5. If the server shall return a status document (synchronous or asynchronous call).
6. If the input data should be returned in the response document (lineage).

2.3.1. KVP Execute request

Execute requests tend to be complex (since they have to define I/O and process execution) and should be done in XML. Nevertheless KVP and HTTP GET can be used to run an *Execute* request and to understand its purpose.

For example the following KVP Execute request executes a service that generates a buffer around a polygon.

```
http://hostname/wpsInstance?&
request=Execute&
identifier=bufferPolygon&
datainputs=bufferDistance=35;
    inputPolygon=http%3A%2F%2Flocaldomain%2Fpolygon.gml
service=WPS&version=1.0.0&
storeExecuteResponse=true&
lineage=true&
status=true
```

Listing 11 - Execute request

The SERVICE and VERSION are mandatory parameters like in *DescribeProcess*. DATAINPUTS contains the inputs that have the following identification: **bufferDistance** and **inputPolygon**

The KVP DATAINPUTS should be encoded using the following rules:

1. A semicolon (;) shall be used to separate one input from the next
2. An equal sign (=) shall be used to separate an input name from its value and attributes, and an attribute name from its value
3. An at-symbol (@) shall be used to separate an input value from its attributes and one attribute from another.
4. All field values and attribute values shall be encoded using the standard Internet practice for encoding URLs (http://en.wikipedia.org/wiki/Url_encoding).

Based on Listing 7 and the *DescribeProcess* XML response, the *bufferDistance* description in the *Execute* could be:

```
...bufferDistance=35@datatype=xsd:integer@uom=meter
```

Listing 12 - Execute request with data inputs values and attributes for literal data

As indicated in the ComplexData description an input can be embedded in the *Execute* request or referenced as in URL that will be used by the WPS to fetch the data content and process it, an extended example of `inputPolygon` based on Listing 7:

```
inputPolygon=http%3A%2F%2Ffoo%2Flocaldomain%2Fpolygon.gml@
format=application%2Fgml%2Bxml@encoding=UTF-8@
schema=geometryBasic2d.xsd
```

Listing 13 - Execute request with data inputs values and attributes for complex data (XML). URL in quoted format.

Or in a human friendly form:

```
inputPolygon=http://localdomain/polygon.gml@
format=application/gml+xml@
encoding=UTF-8@schema=geometryBasic2d.xsd
```

Listing 14 - Execute request with data inputs values and attributes for complex data (XML). URL in human-friendly format.

2.3.2. `storeExecuteResponse` and status parameters

Geo-processes are computationally intensive, and might require a considerable amount of time to run, eventually reaching a point where they take more time than allowed for an HTTP connection to remain open (while doing the execute request). WPS uses a pull methodology¹ to deal with time-consuming processes. It's up to the client to initiate the Execute request and then to constantly query (pull) the server for an update/content.

The pull strategy is defined by two parameters during the Execute request: **`storeExecuteResponse`** and **`status`**.

If **`storeExecuteResponse=true`**, the server will store the execute response document in a web accessible URL that the client will use to pull the document. When making the request the service shall respond immediately with an URL to the client. This URL will be used to determine the status of the process and/or pull the process output when concluded.

If **`status=true`**, then the server will constantly update the stored document with on-going reports on the status of execution (for example percentage concluded). If **`status=false`** then the service shall not update the execute response document until the process either completes successfully or fails. In case **`status=true`** and **`storeExecuteResponse=false`** then a service exception shall be reported.

¹ http://en.wikipedia.org/wiki/Pull_technology

Listing 10 (above) defines: **storeExecuteResponse=true&status=true**, in this case the client will receive an URL with the document location, and its content will be updated as the process runs in the server.

2.3.3. Lineage

A response execute document may contain the input used to run the WPS request. This is normally referred as lineage. To request a lineage structure in the response document the KVP GET should contain **lineage=true**.

Lineage is an independent KVP parameter that is NOT associated with responseDocument properties.

2.3.4. Response Document

The sort of response document from an execute request can be manipulated during the client request. A client may request for only a specific output mimetype encoding schema or for the output to be given as reference (an URL pointing to the output).

In a KVP request, the parameter RESPONSEDOCUMENT will define the sort of output returned:

```
http://hostname/wpsInstance?&
request=Execute&
identifier=bufferPolygon&
datainputs=bufferDistance=35;
inputPolygon=http%3A%2F%2Flocaldomain%2Fpoylgon.gml
service=WPS&version=1.0.0&
storeExecuteResponse=true&
lineage=true&
status=true&
```

Listing 15 - Example of a request for a specific output (identifier as buffer) as reference (URL).

The responsedocument syntax follows generic datainputs encoding rules. If asReference=true then the output will be returned as a reference (URL) that the client may use to pull the output.

Other output attributes can be used to manipulate an output, for example a buffer result could be returned as an XML structure or converted into a raster image (if allowed by the service), see listings 16 and 17.

```
responsedocument=buffer=@mimetype=image/tiff
```

Listing 16 - Example of a request for image in TIFF format

Or

```
responsedocument=buffer=@mimetype=application/gml+xml
```

Listing 17 - Example of a request for vector data in GML format

2.3.5. RawResponse

Optionally a single output can be outputted as a raw response, meaning a raw response for an output that will generate a GeoTIFF file will be the image itself without any WPS Response document. The raw response is obtained by using parameter RAWDATAOUTPUT **instead of** RESPONSEDOCUMENT (Listing 18).

```
http://hostname/wpsInstance?&
request=Execute&
identifier=bufferPolygon&
datainputs=bufferDistance=35;
inputPolygon=http%3A%2F%2Flocaldomain%2Fpoylgon.gml
service=WPS&version=1.0.0&
storeExecuteResponse=true&
lineage=true&
status=true&
rawdataoutput=buffer=@mimetype=image/tiff
```

Listing 18 - Raw output request for a buffer in TIFF format.

2.3.6. XML Execute request

XML Execute request mimics the KVP Execute request, with identical attributes and functionalities. The XML document can be divided into 3 sections:

storageExecuteDocument and status are defined as attributes of the response document:

1. Service identification, language and version request.
2. Data inputs
3. Structure of response document

Listing 19 is an example of an *Execute* request for a process named *bufferProcess*

```
<wps:Execute service="WPS" version="1.0.0" language="en-CA"
xmlns:wps="http://www.opengis.net/wps/1.0.0"
xmlns:ows="http://www.opengis.net/ows/1.1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.opengis.net/wps/1.0.0/wpsExecute_request.xsd">
<ows:Identifier>bufferProcess</ows:Identifier>
```

Listing 19 - XML section with service identification and Execute request to WPS instance.

Data inputs follow the ComplexData description, allowing for embedded data or URL reference to be used. The usage of `xlink:href` allows for the usage of unquoted URLs.

```
<wps>DataInputs>
<wps:Input>
<ows:Identifier>inputPolygon</ows:Identifier>
<ows:Title>Input polygon for buffer calculation</ows:Title>
  <wps:Reference xlink:href="http://localdomain/polygon.gml" />
</wps:Input>
```

Listing 20 - Example of a complexData input passed by Reference.

Data and Reference elements are used to determine if the content is embedded or passed as reference.

```
<wps>DataInputs>
<wps:Input>
<ows:Identifier>inputPolygon</ows:Identifier>
<ows:Title>Input polygon for buffer calculation</ows:Title>
  <wps>Data>
    <wps:ComplexData mimeType="application/xml+gml" encoding="UTF-8"
schema="http://schemas.opengis.net/gml/3.1.1/base/geometryBasic2d.xsd">
      <gml:Polygon>
        <gml:outerBoundaryIs>
          <gml:LinearRing>
            <gml:coordinates>96331.843041083571734,202504.810803000174928...
              96331.843041083571734,202504.810803000174928
            </gml:coordinates>
          </gml:LinearRing>
        </gml:outerBoundaryIs>
      </gml:Polygon>
```

Listing 21 - Example of complexData inputs embedded in the WPS Execute request.

The third section of the Execute request is optional and if present, it defines what the response document should contain. If the **status=true** or **storeExecuteResponse=true** (see Listing 22), data format or lineage should be returned.

```
<wps:ResponseForm>
  <wps:ResponseDocument lineage="true"
    storeExecuteResponse="true" status="true">
    <wps:Output asReference="true" mimeType="application/xml+gml">
      <ows:Identifier>buffer</ows:Identifier>
    </wps:Output>
  </wps:ResponseDocument>
```

Listing 22 - Response document section of WPS Execute request.

2.4. Locale definitions

WPS offers multilingual capabilities in metadata description. A *GetCapabilities* request shall return the supported languages in the server.

```
<wps:Languages>
  <wps:Default>
    <ows:Language>en-CA</ows:Language>
  </wps:Default>
  <wps:Supported>
    <ows:Language>en-CA</ows:Language>
    <ows:Language>fr-CA</ows:Language>
    <ows:Language>fr-FR</ows:Language>
    <ows:Language>es-ES</ows:Language>
    <ows:Language>pt-PT</ows:Language>
    <ows:Language>pt-BR</ows:Language>
  </wps:Supported>
```

Listing 23 - Example of languages supported by a multilingual server.

Language code description follows standard RFC 4646. The client may use the attribute LANGUAGE with the code language to obtain metadata description (*describeProcess*) in the desired language.

2.5.WPS Exceptions

WPS execution may encounter errors that will be returned to the client, using a specific exception code according to the error. Some errors may indicate a locator parameter of the error.

```
<?xml version="1.0" encoding="utf-8"?>
<ows:ExceptionReport version="1.0.0"
xmlns:ows="http://www.opengis.net/ows/1.1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.opengis.net/ows/1.1
http://schemas.opengis.net/ows/1.1.0/owsExceptionReport.xsd">
  <ows:Exception exceptionCode="InvalidParameterValue"
    locator="NonExistingService"/>
```

Listing 24 - WPS exception of a DescribeProcess request using a non-existing service (request=DescribeProcess&identifier=NonExistingService&service=WPS&version=1.0.0).

WPS instances have a minimum of 4 possible exceptionCodes (WPS defines 8 exceptionCodes in total) [4]:

- MissingParameterValue; The request does not include a parameter value or a default cannot be found
- InvalidParameterValue; The request contains an invalid parameter value
- NoApplicableCode; Generic exception, no other code could be applied
- NotEnoughStorage; The server does not have enough space available

2.6.SOAP Binding

SOAP (Simple Object Access Protocol) is a messaging framework, meaning, a structured way to pass, explain and process a message. The expression "SOAP message" is normally referring to some sort of XML content that is associated with a SOAP header than in turn is wrapped by a SOAP envelope. This SOAP structure is normally transported by HTTP, nevertheless nothing prevents the use of protocols like: FTP, SSH, SMTP, etc.

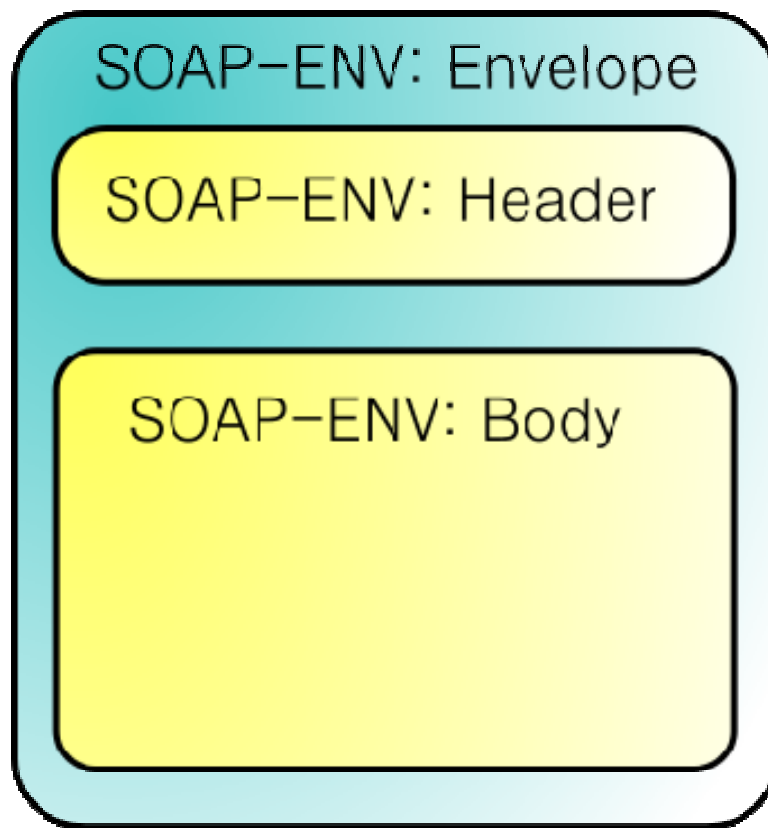


Figure 2 - Generic SOAP structure.

SOAP in WPS is normally used as simple wrapper (encapsulation) around the XML requests (following the SOAP document-style encoding), and currently different WPS frameworks have different implementation levels.

A WPS Getcapabilities request using SOAP 1.1 would be as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<!--
Equivalent GET request is
http://foo.bar/foo?Service=WPS&Version=1.0.0&Request=GetCapabilities
&Language=en-CA
-->
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <wps:GetCapabilities xmlns:ows="http://www.opengis.net/ows/1.1"
      xmlns:wps="http://www.opengis.net/wps/1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.opengis.net/wps/1.0.0
                          ../wpsGetCapabilities_request.xsd"
      language="en-CA" service="WPS">
      <wps:AcceptVersions>
        <ows:Version>1.0.0</ows:Version>
      </wps:AcceptVersions>
    </wps:GetCapabilities>
  </SOAP-ENV:Body>

```

Listing 25 - Example of a getCapabilities request using SOAP1.1 wrapper.

The same request can be wrapped in SOAP 1.2 wrapper (both versions are supported)

```

<?xml version="1.0" encoding="UTF-8"?>
<!--
Equivalent GET request is
http://foo.bar/foo?Service=WPS&Version=1.0.0&Request=GetCapabilities
&Language=en-CA
-->
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Body>
    <wps:GetCapabilities xmlns:ows="http://www.opengis.net/ows/1.1"
      xmlns:wps="http://www.opengis.net/wps/1.0.0"
      xmlns:xlink="http://www.w3.org/1999/xlink"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.opengis.net/wps/1.0.0
                          ../wpsGetCapabilities_request.xsd"
      language="en-CA" service="WPS">
      <wps:AcceptVersions>
        <ows:Version>1.0.0</ows:Version>
      </wps:AcceptVersions>
    </wps:GetCapabilities>
  </soap:Body>

```

Listing 26 - Example of a GetCapabilities request using SOAP1.2 wrapper.

SOAP header is an extension mechanism that provides a way to pass information in SOAP messages that is not application payload (therefore WPS request parameters can not be expressed in the SOAP-header). Such "control" information includes, for example, passing directives or contextual

information related to the processing of the message. This allows a SOAP message to be extended in an application-specific manner (from: <http://www.w3.org/TR/soap12-part0/#L1161>). An example of the extension mechanism is a basic authentication using the SOAP header:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <h:BasicAuth xmlns:h="http://soap-authentication.org/basic/2001/10/"
      SOAP-ENV:mustUnderstand="true">
      <h:Name>bacon</h:Name>
      <h:Password>eggs</h:Password>
    </h:BasicAuth>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <wps:GetCapabilities> .....</wps:GetCapabilities>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 27 - Example of a SOAP1.1 header authentication structure. Source: <http://www.whitemesa.com/soapauth.html>

The **mustUnderstand="true"** (or **mustUnderstand="1"**) mandates that the containing header block must be parsed and understood by the system, in this case the authentication has to be processed before any the SOAP Body content is processed.

SOAP support is applicable to all WPS operations (*GetCapabilities*, *DescribeProcess* and *Execute*), but *Execute* request only allow SOAP usage when the output is a ResponseDocument, a request for a specific RawDataOutput shall generate a SOAP error.

The *Execute* request may exist in two forms:

- Complete XML request as described above (listings 19-22).
- Process name and I/O encoded as an element in the SOAP body.

The process name and I/O encoded as elements in the SOAP body facilitates the usage of WPS inside WSDL. The process name shall be turned into an element in the SOAP body by prepending the text "ExecuteProcess_" and the I/O names shall be turned into elements, e.g:

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Body>
    <ExecuteProcess_BufferProcess>
      <InputPolygon>http://rsg.pml.ac.uk/wps/testdata/simplePoly.gml</InputPolygon>
      <BufferDistance>400</BufferDistance>
    </ExecuteProcess_DummyProcess>
  </soap:Body>
</soap:Envelope>
```

Listing 28 - Example of a WPS request with process name an I/O encoded inside the SOAP Body content.

The WPS 1.0.0 specification document is not clear in some aspects of the SOAP Body encoding, for example there is no information on how to encode WPS attributes (e.g status) or how to deal with async calls to the service.

The SOAP standard supports fault reporting by including a `Fault` element inside the `Body` element. This `Fault` element contains mandatory `Code` and `Reason` elements, these mandatory elements contain the error code and an extended explanation (respectively). The `Code` element reports a specific SOAP error defined in the specification, in case of WPS this value is "soap:Server". A WPS `ExceptionReport` shall be contained inside an optional `Detail` element (Listing 29).

```
<soap:Envelope xmlns:soap=http://www.w3.org/2003/05/soap-envelope
  xmlns:ows=http://www.opengis.net/ows/1.2>
  <soap:Body>
    <soap:Fault>
      <soap:Code>
        <soap:Value>soap:Server</soap:Value>
      </soap:Code>
      <soap:Reason>
        <soap:Text>A server exception was encountered.</soap:Text>
      </soap:Reason>
      <soap:Detail>
        <ows:ExceptionReport>
          ...
        </ows:ExceptionReport>
      </soap:Detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Listing 29 - SOAP1.2 fault exception containing a WPS Exception Report.

WSDL stands for Web Services Description Language (the name changed with the release of WSDL 2.0, previously it was Web Services Definition Language, see Figure 3 for a comparison). WSDL is a document written in XML that describes a Web service. It specifies the location of the service and the operations (or methods) the service exposes basically its an XML document that contains more or less the same information as *GetCapabilities/DescribeProcess*, but in a generalist way without the use of OGC's nomenclature and standards. The problem with WSDL is that we have to "explain" everything to the system, meaning we have to explain all WPS structures.

In theory having a WPS instance providing WSDL information should allow it to be used as a web service in any programming environment or orchestration language like BPEL (*Business Process Execution Language*).

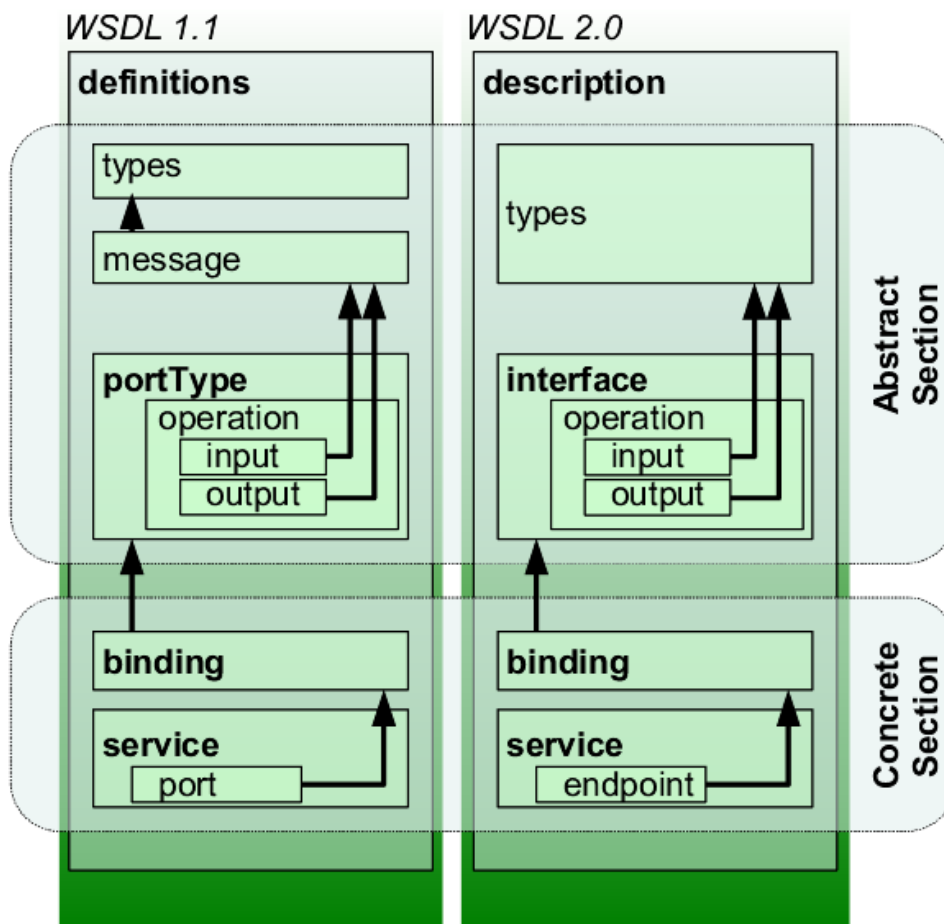


Figure 3 - Representation of concepts defined by WSDL 1.1 and WSDL 2.0 documents.

WSDL implementation is described in the WPS spec as a "best practices" avoiding the specification of WSDL contents and interaction with WPS attribute and I/O. Therefore different WPS frameworks have different WSDL implementations strategies.

A WPS supporting WSDL shall return a generic WSDL document, describing all processes, when submitting the following KVP request:

```
http://hostname/wpsInstance?WSDL
```

Listing 30 - WSDL document request. Generic request (all processes)

In addition each individual service may contain its own WSDL description that is returned when submitting the following KVP request:

```
http://hostname/wpsInstance/identifier[/service.soap]?WSDL
```

Listing 31 - WSDL document request. Process specific

Where **identifier** is the WPS process identifier and the **[/service.soap]** is an optional URL section that points to the root of the HTTP SOAP requests.

3. PyWPS – a WPS implementation

3.1. Introduction

Plymouth Marine Laboratory (PML), and its fellow NETMAR partners, has implemented the Python WPS implementation (PyWPS) and it has actively supported new developments (SOAP/WSDL) contained in a separate project branch¹. In the near future the SOAP-WSDL branch will be merged with the project's trunk code and released as an official PyWPS release.

Code development was done in the NETMAR project, and had the following objectives:

- Improvement of WPS-WSDL interface and SOAP support
- Automatic conversion of WPS processes into WSDL
- Easy integration with Taverna-workbench
- Support for asynchronous services in WSDL

The automatic conversion of WPS processes into WSDL was a successful strategy that allowed GRASS GIS modules to be exported into a WSDL document with help of the WPS-GRASS-Bridge project [5]. The work carried out by NETMAR allowed PyWPS services to be easily integrated into the Taverna workflow system providing a ready built chaining environment.

PML also supports PyWPS by maintaining a Wiki with the latest PyWPS tutorials and information.

The following sections will explain how to install a PyWPS server along with the WPS-GRASS-Bridge and the WSDL integration needed for it to be accessed as a SOAP service by Taverna.

¹ <https://svn.wald.intevation.org/svn/pywps/branches/pywps-3.2-soap>

3.2. PyWPS

PyWPS Web Processing Service 3.2: is a Python program which implements the OGC WPS 1.0.0 standard. PyWPS can be run in Windows and Linux, but the Windows implementation lacks asynchronous capabilities due to problems forking Python processes in Windows. PyWPS doesn't provide out of the box processes since it is mainly focused on providing a framework within which developers can develop their own processes. Nevertheless, PyWPS supports easy-to-do connections to GRASS¹, GDAL² and Rpy³.

PyWPS is normally run as a CGI script but it can also be run using mod_python and support for mod_wsgi is being finalized. Since the code is written in standard Python it can also be ported as a Javalet (using Jython [6]) and run from Tomcat application server. PyWPS is lightweight, with only 5000 lines of code, making it a good candidate for testing new concepts.

3.2.1. Installation

PyWPS follows the default Python installation procedure using the setup.py script. Further configuration is necessary to deal with server paths and process folder location.

WPS-GRASS-Bridge requires GRASS 7.0, available via SVN⁴, to run the served modules and to generate their WPS description based in XML metadata.

Installation examples will use Ubuntu 10.04.

3.2.2. Core PyWPS installation

PyWPS-3.2-SOAP requires the following debian packages:

- python-lxml
- python-magic
- apache2
- python-mapscript (optional)

To download the code it is necessary to install the SVN package

- subversion

¹ <http://www.sciencedirect.com/science/article/pii/S1364815211002775>

² <http://www.gdal.org>

³ <http://rpy.sourceforge.net/rpy2.html>

⁴ <https://svn.osgeo.org/grass/grass-web/trunk>

Packages can be installed using GUIs available in the distribution, or by running the apt-get install command:

```
~#>sudo apt-get install subversion
```

3.2.3. Installing PyWPS code

Code is downloaded using the svn command:

```
~#>cd ~
~#>svn co https://svn.wald.intevation.org/svn/pywps/branches/pywps-3.2-soap pywps
```

Installation follows the default python procedure:

```
~#>cd pywps
~#>sudo python setup.py install
```

The directory, `/usr/local/bin`, will contain a `wps.py` script that can be called to check if the `pywps` module is operational

```
~#>wps.py
PyWPS NoApplicableCode: Locator: None; Value: No query string found.
Content-Type: application/xml
<?xml version="1.0" encoding="utf-8"?>
<ows:ExceptionReport version="1.0.0" xmlns:ows="http://www.opengis.net/ows/1.1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.opengis.net/ows/1.1
http://schemas.opengis.net/ows/1.1.0...tionReport.xsd">
<ows:Exception exceptionCode="NoApplicableCode">
<ows:ExceptionText>'No query string found.'</ows:ExceptionText>
</ows:Exception>
</ows:ExceptionReport>
```

PyWPS will report an exception since the request lacks any operations/inputs. No configuration file is being used therefore no processes have been assigned to the WPS instance.

PyWPS contains a default configuration file that needs to be configured: `pywps.cfg`. The file may have a global or local version, the global location is added to `/etc`, while the local file should be defined in the wrapper script (see later). Normally `pywps` will locate the configuration file by searching the `PYWPS_CFG` environment variable.

PyWPS package provides an example file that can be used as a starting template.

```
~#> sudo cp ~/pywps-3.2-soap/pywps/default.cfg /etc/pywps.cfg
```

The `/etc/pywps.cfg` file contains some mandatory parameters that need to be configured. The parameters relate to server paths and process locations. Assuming the default paths defined by the Apache2 configuration file (`/etc/apache2/sites-available/default`) and that `pywps` processes directory is also inside the `cgi-bin/` folder:

```
[server]
maxoperations=0
maxinputparamlength=1024
maxfilesize=300mb
tempPath=/tmp
processesPath=/usr/lib/cgi-bin/processes
outputUrl=http://localhost/wpsoutputs
outputPath=/var/www/wpsoutputs
logFile=/var/log/apache2/pywps.log
logLevel=DEBUG
```

The log file should be set allowing for apache2 to write into it

```
~:~#>sudo touch /var/log/apache2/pywps.log
(apache2 user:group defined in /etc/apache2/envvars)
~:~#>sudo chown www-data:www-data /var/log/apache2/pywps.log
~:~#>sudo chmod 744 /var/log/apache2/pywps.log
```

The `/var/log/apache2` directory may have permission restrictions and the `pywps.log` file may not have access permission. It's advisable to change directory permission:

```
~:~#>sudo chmod a+x /var/log/apache2
```

PyWPS requires that processes are contained inside a specific folder, following Python's module structure.

```
~:~#>sudo mkdir /usr/lib/cgi-bin/processes
```

and another folder to store the WPS's output

```
~:~#>sudo chown -R www-data:www-data /var/www/wpsoutptus
```

Processes are simple scripts that need to be structured as a Python module (for more information read the README file in the `/pywps/processes` directory).

PyWPS provides a small set of processes as examples. In this example we will copy the processes to the server's `cgi-bin/process` folder

```
~:~#>sudo cp ~/pywps-3.2-soap/pywps/processes/* /usr/lib/cgi-bin/processes/
```

Additional processes may be coded by the user or by importing other tools (e.g GRASS-GIS, GDAL R)

PyWPS follows a wrapper strategy, meaning multiple access points (wrappers) to one WPS API. Each wrapper file may contain the location of diverse configuration files, system variables and process paths. Using Nano editor (or vim) the user must compose a wrapper file named `pywps.cgi` as follows:

```
#!/bin/sh
export PYWPS_CFG=/etc/pywps.cfg
export PYWPS_PROCESSES=/usr/lib/cgi-bin/processes
/usr/local/bin/wps.py $1
```

The file requires the following permissions:

```
~:~#> sudo chown www-data:www-data /usr/lib/cgi-bin/pywps.cgi
~:~#>sudo chmod 744 /usr/lib/cgi-bin/pywps.cgi
```

Opening an browser and making the following request should generate a WPS reply:

<http://localhost/cgi-bin/pywps/cgi?request=getCapabilities&service=WPS>

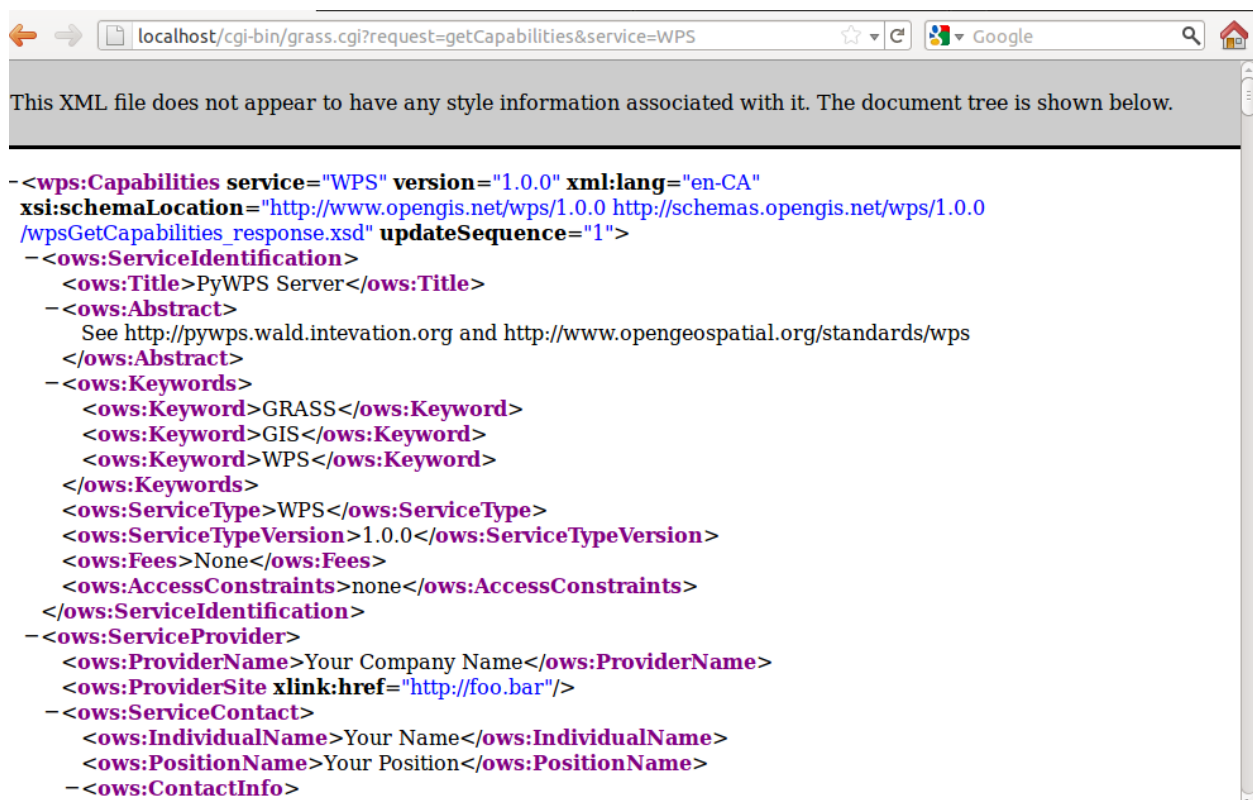


Figure 4 - WPS response to a `getCapabilities` request.

3.2.4. WPS-GRASS-BRIDGE INSTALL (for extra WPS processes)

WPS-GRASS-Bridge requires GRASS-GIS version 7.0 (this version is under active development and may contain some bugs) and the PyXB package.

Download pyXB version 1.1.3

```
:~#> svn co http://sourceforge.net/projects/pyxb pyxb
```

Standard python install:

```
:~#>cd pyxb/
```

```
:~/pyxb#> sudo python setup.py install
```

Download the WPS-Bridge-GRASS:

```
:~#>svn checkout http://wps-grass-bridge.googlecode.com/svn/trunk wps-grass-bridge
```

Get WPS-Bridge-GRASS code is also from SVN (revision 82)

WPS-Bridge-GRASS doesn't require installation, only the configuration of file `GlobalGrassSettings.py` containing parameters pointing to folders (mainly the GRASS location, log paths, temporary folder and `GrassAddonPath`).

The `GrassAddonPath` points to a folder inside the WPS-GRASS-Bridge that contain new processes based in Python, supporting generic raster math calculations (e.g `r.math`, `r.add` etc):

Example of `GlobalGrassSettings.py`

```
WORKDIR="/tmp"
```

```
OUTPUTDIR="/tmp"
```

```
LOGFILE="/var/log/grass-bridge/logfile.txt"
```

```
LOGFILE_MODULE_STDOUT="/var/log/grass-bridge/logfile_module_stdout.txt"
```

```
LOGFILE_MODULE_STDERR="/var/log/grass-bridge/logfile_module_stderr.txt"
```

```
GRASS_GIS_BASE="/usr/local/grass-7.0.svn"
```

```
GRASS_ADDON_PATH="/home/pywps/wps-grass-bridge/gms/Testing/Python/GrassAddons"
```

```
GRASS_VERSION="7.0.svn"
```

The log file needs to have RW permissions

```
:~#>sudo mkdir /var/log/grass-bridge
```

```
:~#>cd /var/log/grass-bridge
```

```
:~#>sudo sh -c "touch logfile.txt && touch logfile_module_stdout.txt && touch logfile_module_stderr.txt"
```

```
:~#>sudo chmod 666 *
```

The WPS-GRASS-Bridge can be tested by calling the command the `PyWPSGrassModuleStarter` that will test the `v.voronoi` process:

```
:~#>cd ~/wps-grass-bridge
~/wps-grass-bridge#>python PyWPSGrassModuleStarter.py
```

Note: The `PyWPSGrassModuleStarter.py` needs write permission, since it saves inputs files into the folder containing the script.

As good practice it is advisable to regenerate the XML GRASS process description, and then the Python process.

```
:~#>cd ~/wps-grass-bridge/gms/Testing/Python/XML
~/wps-grass-bridge/gms/Testing/Python/XML#>grass70 -text (enter the grass bash)
~/wps-grass-bridge/gms/Testing/Python/XML (GRASS70)#>sh create_xml.sh
~/wps-grass-bridge/gms/Testing/Python/XML (GRASS70)#>cd -
(to go to the last path (wps-grass-bridge))
~/wps-grass-bridge/#> sh create_pywps_processes.sh
```

The `create_pywps_processes.sh` script will generate the pyWPS processes that will be stored in the `wps-grass-bridge/pywps_services` directory.

A new wrapper script should be created in the `cgi-bin` directory, referencing the `pywps_services/` folder. The wrapper will be named `grass.cgi`.

```
#!/bin/sh
export PYWPS_CFG=/etc/pywps.cfg
export PYWPS_PROCESSES=/home/pywps/wps-grass-bridge/pywps_services
python /usr/local/bin/wps.py $1
```

The WPS-GRASS-Bridge processes can now be tested from the browser. The following URL will call the `v.delaunay` module, generating delaunay polygons from random points.

[http://localhost/cgi-bin/grass.cgi?request=Execute&service=WPS&version=1.0.0&identifier=v.delaunay&datainputs=\[input=http://rsg.pml.ac.uk/wps/testdata/random_points.gml\]](http://localhost/cgi-bin/grass.cgi?request=Execute&service=WPS&version=1.0.0&identifier=v.delaunay&datainputs=[input=http://rsg.pml.ac.uk/wps/testdata/random_points.gml])

The PyWPS log will contain all the loaded processes from `/pywps_service`, it also contains Warnings reporting any processes that couldn't be loaded, eg:

```
PyWPS [2011-10-13 22:07:07,326] WARNING: Could not import process
[<class r_terraflow.r_terraflow at 0x14fd598>]: NameError("global
name 'inf' is not defined")
```

3.2.5. WPS-Grass-Bridge Add-on

WPS-GRASS-Bridge provides a special wrapper script for `r.mapcalc`, the wrapper script is called `r.math`. The `r.math` script is located in the `~/src/wps-grass-bridge/gms/Testing/Python/GrassAddons` directory, the script has 4 other siblings: `r.add`, `r.div`, `r.mult` and `r.sub` that perform addition, division, multiplication and subtraction operations (respectively) on input raster images.

These add-on script are based on python's `grass` module, WPS-GRASS-Bridge automatically sets the Python path and loads the modules, in case of problem the Python path can be added to the wrapper script by adding it as environment variable:

```
#!/bin/sh
export PYTHONPATH=$PYTHONPATH:/usr/local/grass-7.0.svn/etc/python
export PYWPS_CFG=/etc/pywps_grass.cfg
export PYWPS_PROCESSES=/home/pywps/wps-grass-bridge/pywps_services
/usr/local/bin/wps.py $1
```

The GRASS Python module will have problems while loading, reporting a "Unable to load GDAL library", this maybe caused by a naming problem in the GDAL library, since it is installed as `libgdal1.8.0.so`, while python module will search for a `libgdal.so`. The solution is to make a symbolic link to the `gdal` library.

```
~/#>cd /usr/lib
:/usr/lib#>sudo ln -s libgdal1.8.0.so libgdal.so
:/usr/lib#>sudo ldconfig
```

3.2.6. WSDL generation

WSDL generation is automatic and transparent, without any need of configuration by the user. Any WPS process is represented as 1 or 2 WSDL processes according to the following decision tree:

- WPS process with only synchronous support:
 - 1 WSDL service identified as `ExecuteProcess_<service name>`
- WPS with asynchronous support:
 - 1 WSDL service identified as `ExecuteProcess_<service_name>` returning the service's final result
 - 1 WSDL service identified as `ExecuteProcessAsync_<service_name>` returning the service's status URL

The WSDL description only requires that the PyWPS configuration file properly indicates the server adress (`/etc/pywps.cfg`). If `serveraddress` is incorrect the WSDL document will contain an incorrect location attribute in the address element, and a WPS Exception will be raised as result.

[wps]

:

```
serveraddress=http://localhost/cgi-bin/pywps.cgi
```

:

It takes 1.75 min to generate a dynamic WSDL file using the request, <http://localhost/cgi-bin/grass.cgi?WSDL>, as PyWPS has to generate a `describeProcess&identifier=all` XML document (a document with complete process description with +/- 137 GRASS processes) and then apply an XSLT to convert it into WSDL.

The proper strategy is to request the document once, saved in the `/cgi-bin` (eg. `grass.wsdl`) and use it as reference point. Another strategy would be to split the `pywps_service` folder into a vector and raster folder and build wrapper scripts for such folder, diminishing the number of processes in a WPS instance.

4. Taverna workbench

Taverna workbench is a desktop application for designing and executing workflows. The desktop application includes a workflow engine that orchestrates services based in SOAP, WSDL and RESTful protocols, the engine is also integrated in the Taverna server and which allows for remote execution of workflows.

Taverna requires the `graphviz` package:

```
~/#>sudo apt-get install graphviz-dev graphviz
```

Taverna installation instructions and packages can be found here: [7]

Taverna is downloaded as package that is decompressed into a folder and run from the terminal with command:

```
~/taverna-worbench-2.3.0#>./taverna.sh
```

It's advisable to change the JAVA memory options in the `taverna.sh` script. The script contains the following execution line:

```
exec "$javabin" -Xmx400m -XX:MaxPermSize=140m
```

The memory usage could be increased as follows:

```
exec "$javabin" -Xmx800m -XX:MaxPermSize=280m
```

4.1.OGC proxy

W3C has introduced some changes to XML schemas and has introduced a "tar-pit" policy around some XML supporting servers, causing problems when fetching schemas and XML contain. The use

of catalogues is not possible in the current version of Taverna 2.3, and direct access to XML content is mandatory. OGC hasn't updated the service schemas to reflect the W3C changes, and therefore some validation problems may occur.

The OGC proxy is a reverse file proxy script written in Python, that proxy will serve the W3C/OGC schemas from local files. The file serving is transparent, meaning an URL request to the schema will be redirected to the file location of the schema and served. The proxy is configured to work only in specific IP numbers (therefore W3C and OGC URLs) by automatic configuration of IPTables.

Download proxy contents from PyWPS SVN:

```
~/#>svn co https://svn.wald.intevation.org/svn/pywps/branches/pywps-3.2-soap/ogcproxy/  
ogcproxy
```

The necessary W3C schemas and all OGC schemas are contained in 2 compressed packages.

```
~/#>tar -xvf w3c.tar.gz && tar -xvf ogc.tar.gz
```

Note: The ogc.tar.gz contains all the OGC schemas, but the majority will not be used by the WPS.

The current folder should have the following directories:

```
w3c/  
ogc/
```

The src/ogcproxy.py should be configured to point to the folder containing the schemas.

On line 19:

```
#Add here HOST and local path to folder with content, change  
/users/rsg/jmdj/ogc{w3c}, to folder with ogc/w3c content  
cachedHost={"schemas.opengis.net":"/home/pywps/ogc", "www.w3.org":"/home/py  
wps/w3c"}
```

The cachedHost Python dictionary should be changed to point to the folder with the schemas, the dictionary follows the structure URL as key ==> Folder path as value.

The ipDic contains the IPs or URLs defined in the cachedHost dictionary. There is no need to change them unless the W3C and OGC sites change locations

```
ipDic={"schemas.opengis.net":"66.244.86.50", "www.w3.org":"128.30.52.37"}
```

The proxy script is initiated by running it as sudo

```
~/#>cd ~/ogcproxy/src/  
~/ogcproxy/src/#>sudo python ogcproxy.py
```

An iptable listing should output the following:

```
~/#> sudo iptables -L -t nat  
Chain PREROUTING (policy ACCEPT)  
target    prot opt source                destination  
Chain OUTPUT (policy ACCEPT)  
target    prot opt source                destination
```

```
REDIRECT tcp -- anywhere          hans-moleman.w3.org tcp dpt:http
/* "set by ogcproxy.py PID:17010" */ redir ports 8000
REDIRECT tcp -- anywhere          www.opengeospatial.org tcp
dpt:http /* "set by ogcproxy.py PID:17010" */ redir ports 8000
Chain POSTROUTING (policy ACCEPT)
target    prot opt source                destination
```

The OUTPUT chain will redirect any traffic going to the W3C/OGC into port 8000 where the reverse proxy is listening, and serving the files. The proxy is shutdown using the kill command with the PID that is indicated in the iptables rule information:

```
:~/#>sudo kill -9 17010
```

4.2.Using Taverna

Taverna workbench has a friendly-user interface divided in 3 sections (Figure 5):

- Service Panel
- Workflow editor
- Explorer

The user starts by indicating the WSDL location (obtained from the WPS service using a getCapabilities request), and introducing it in the "WSDL import service". Figures 6 and 7.

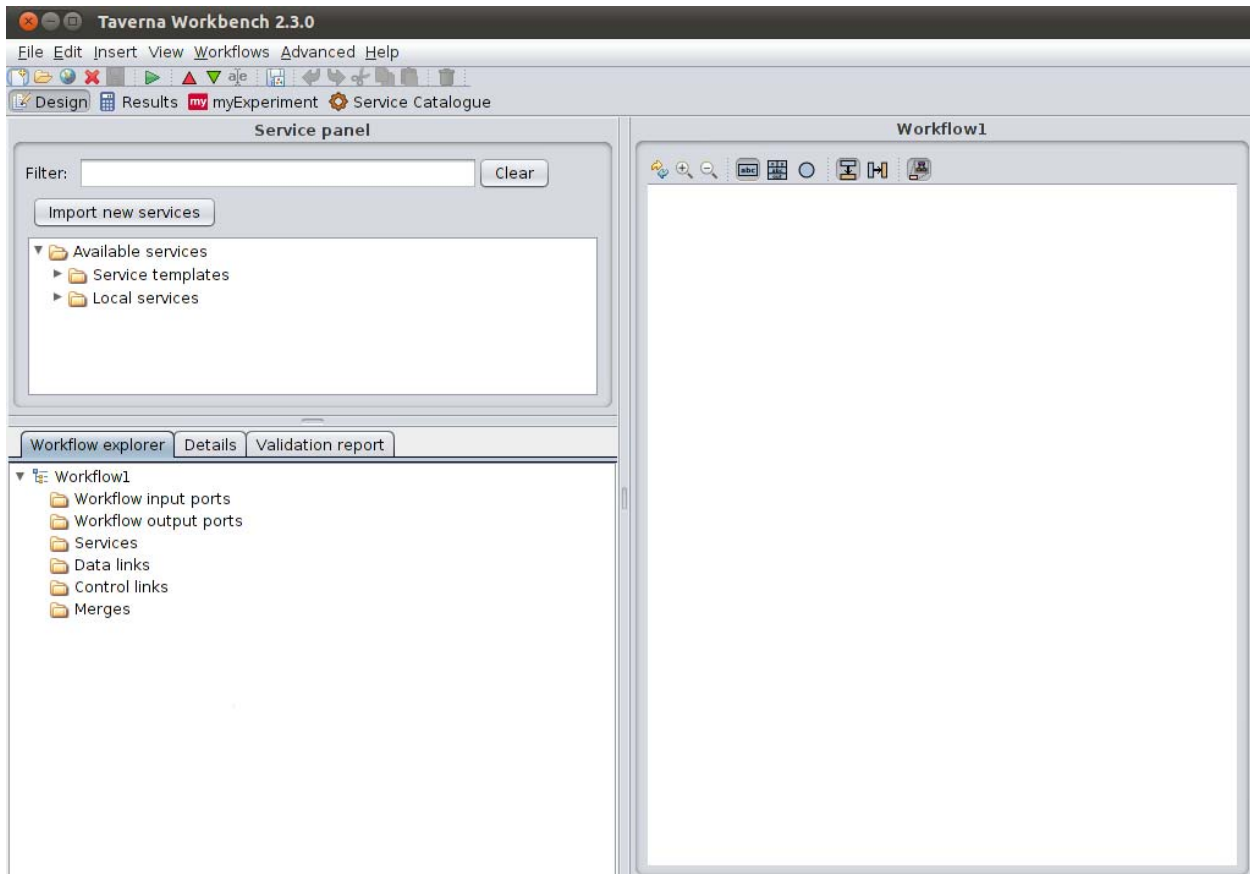


Figure 5 - Taverna-workbench GUI

The Taverna-workbench consumes the WSDL service description provided by the WPS instance.

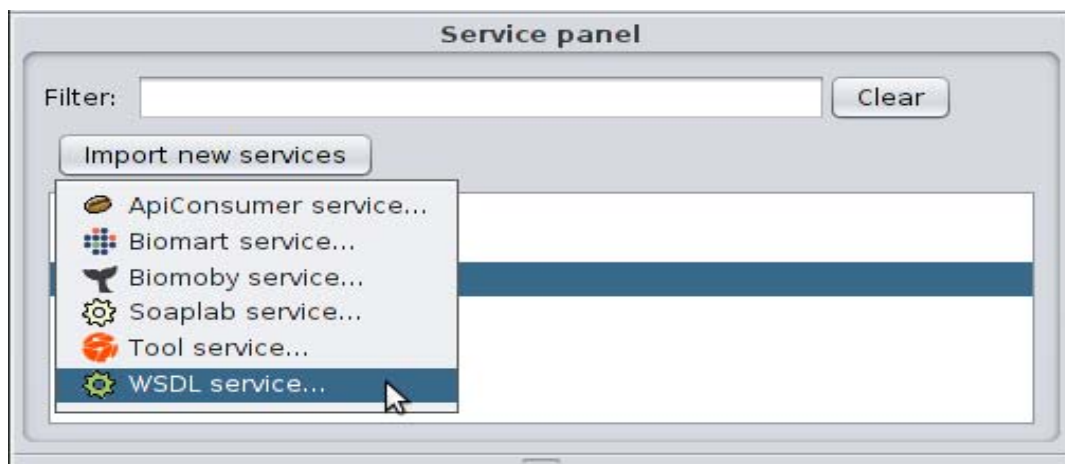


Figure 6 - WSDL service import.

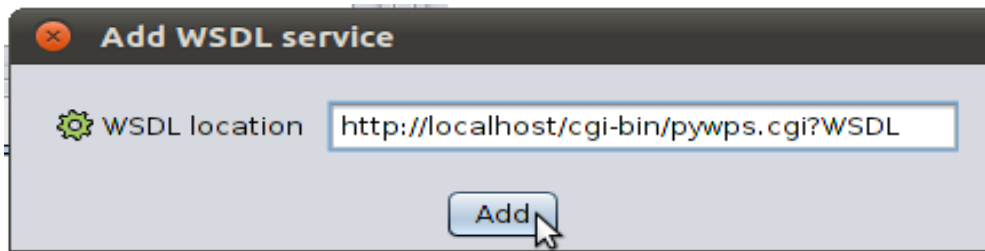


Figure 7 - Adding service URL (example).

The Service is then populated with a process list (Figure 8), in this case processes from GRASS-GIS.

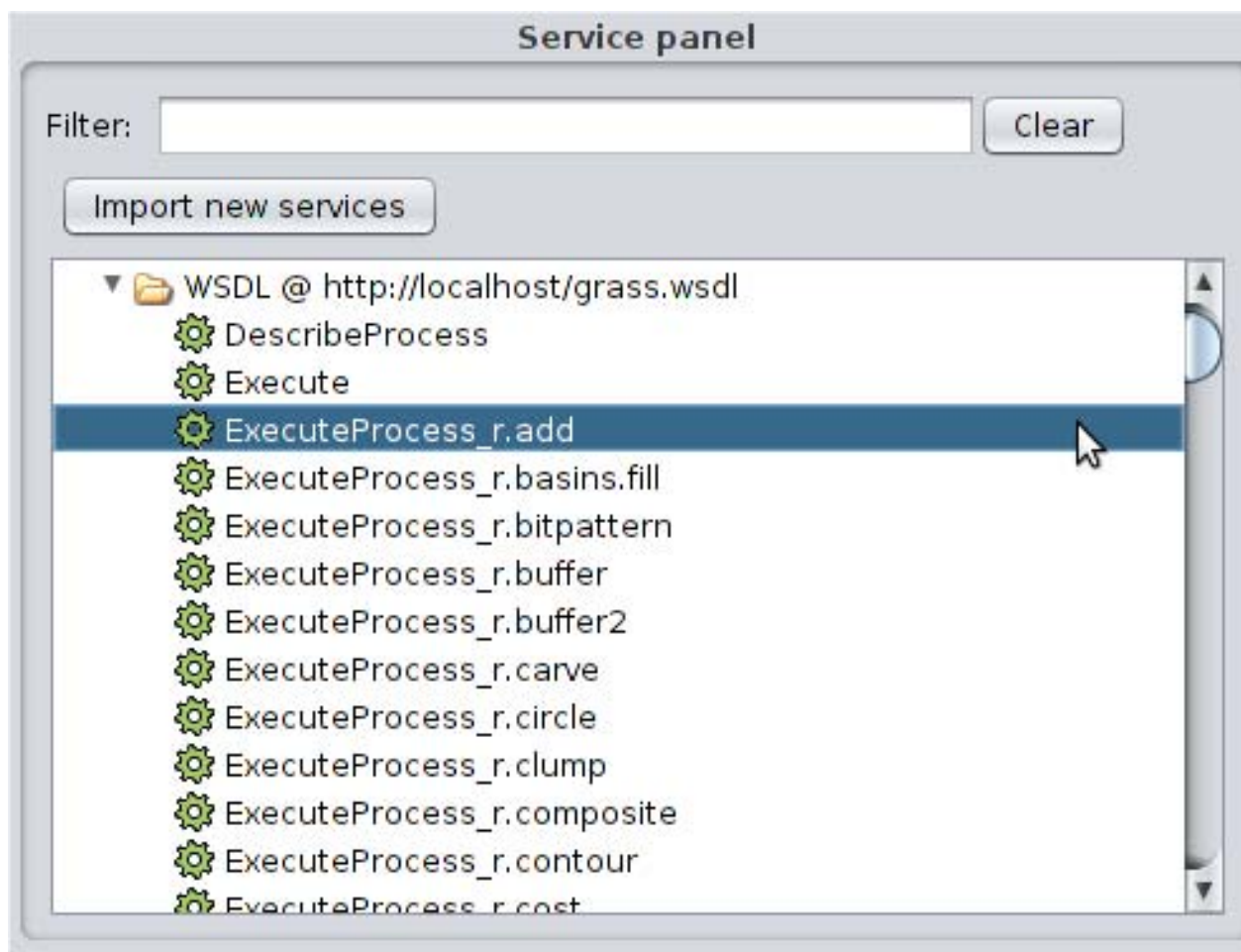


Figure 8 - Service panel populated with several WPS processes.

A process can then be dragged/dropped into the workflow editor on the right. The process will appear as a box wrapped in 2 I/O boxes (DataInputs and ProcessOutputs) (Figure 9).

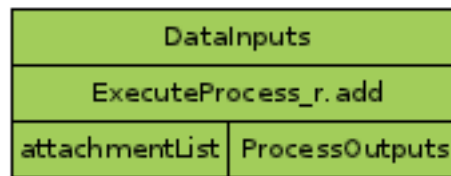


Figure 9 - Process representation.

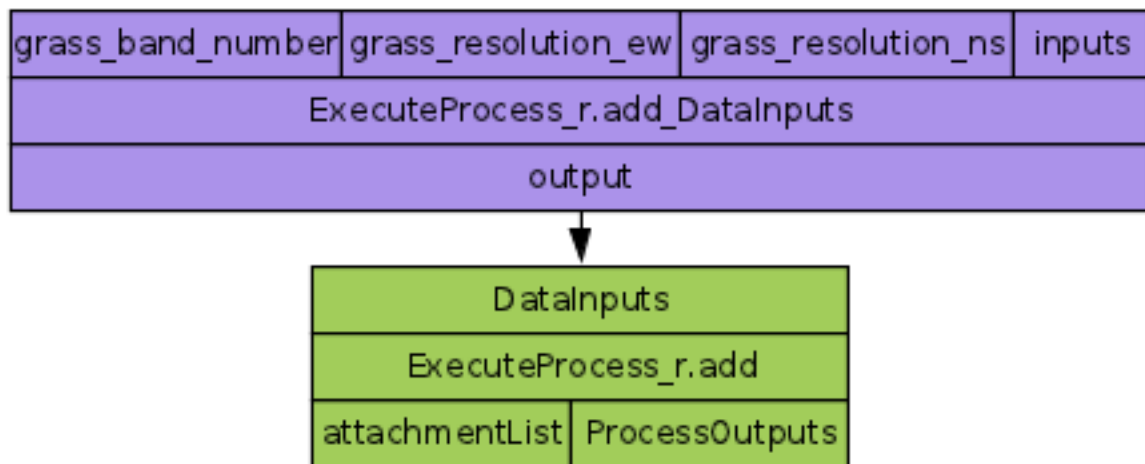


Figure 10 - Process representation with inputs contained inside the DataInputs element.

The service representation follows the WPS standard as DataInputs is the parent element of inputs and ProcessOutputs is the parent element of processes' output (Figure 10).

By positioning the mouse over the process name and right click, the service menu is open and the Add XML input Splitter (Figure 11) should be clicked to show the actual inputs expected in the DataInputs.

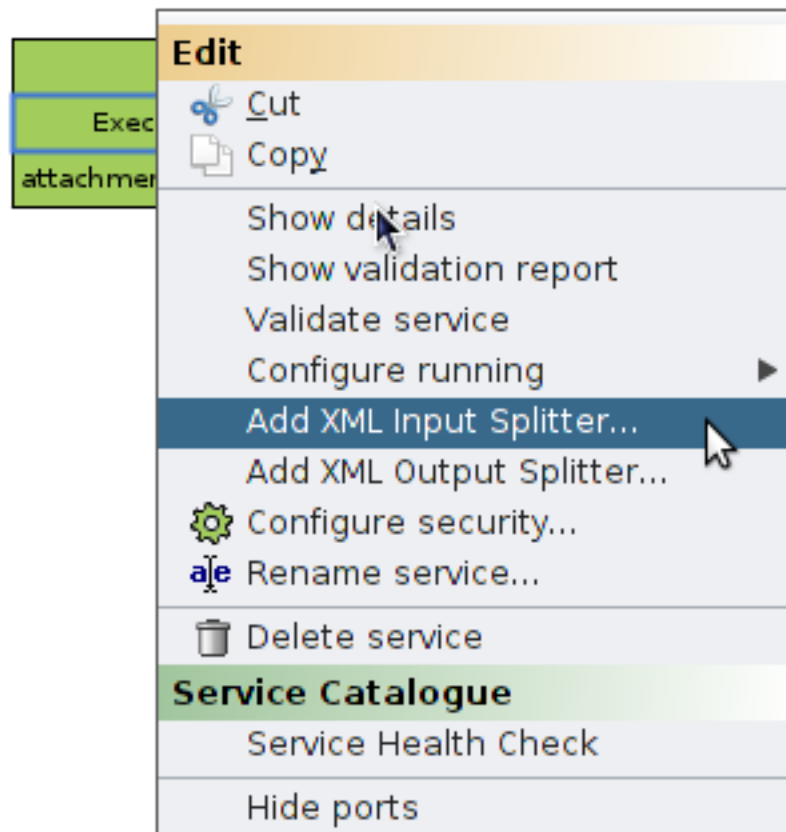


Figure 11 - Adding XML input splitter to determine the WPS's inputs

ProcessesOutputs are handled in the same way. Taverna assumes that DataInputs is the process Input and ProcessOutputs is the output and in turn these I/O are processed by a pre (or post) process that assembles the XML content.

The workflow will not be complete without "Workflow input ports" and "Workflow output ports", these boxes represent the actual I/O of the workflow and the input one will query the user for input locations. The workflow I/O are added by clicking the red and green triangle located in the tool bar (Figure 12).



Figure 12 -Toolbar icons. Red triangle for input, inverted green triangle for output and play symbol to launch workflow execution.

Connecting the several components a final workflow would look as follows (Figure 13):

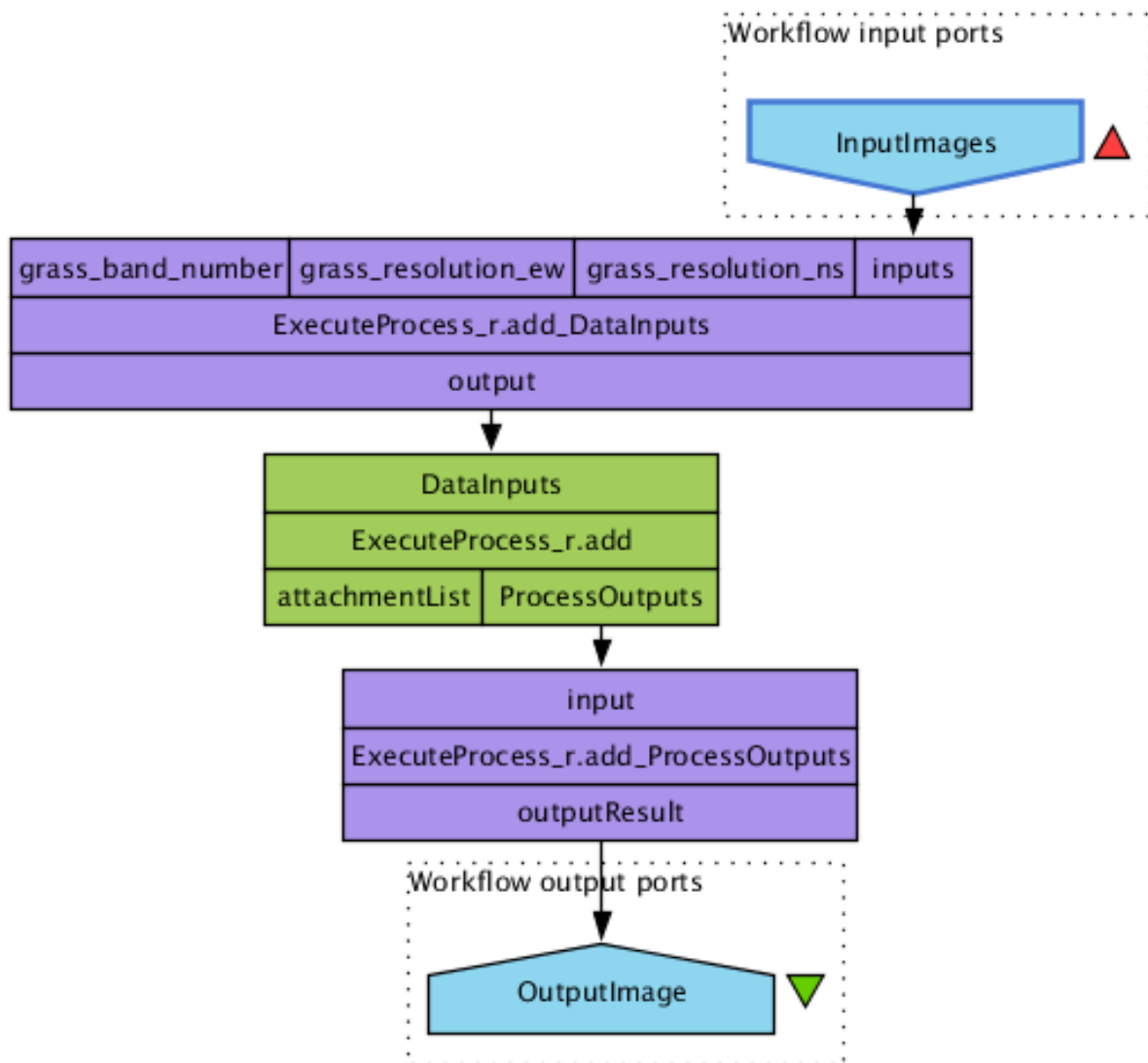


Figure 13 - Complete workflow calling the r.add process.

The r.add process is provided by GRASS-Bridge-WPS and its "inputs" content is described as follows:

```
<Input minOccurs="1" maxOccurs="1024">
  <ows:Identifier>inputs</ows:Identifier>
  <ows:Title>Raster maps to add.</ows:Title>
  <ComplexData>
    <Default>
      <Format>
        <MimeType>image/tiff</MimeType>
      </Format>
    </Default>
    <Supported>
      <Format>
        <MimeType>image/tiff</MimeType>
      </Format>
      <Format>
        <MimeType>image/geotiff</MimeType>
      </Format>
    </Supported>
  </ComplexData>
</Input>
```

Listing 32- r.add process input definitions.

The input is a multiple input that can range from 1 till 1024 images. In Taverna this sort of input is referred to as "list input" (Figure 14 ,15 and 16).

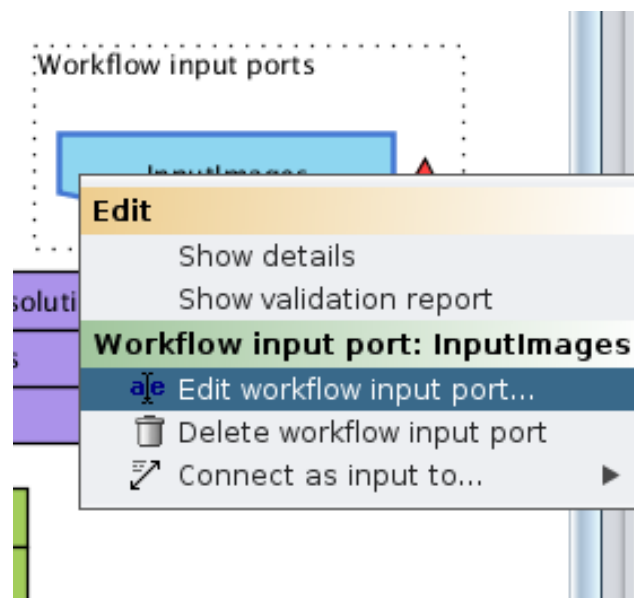


Figure 14 - Input port configuration.

By right click on the input port then "Edit workflow input port" and then select "list of depth", it's possible to use WPS's multi input capabilities.

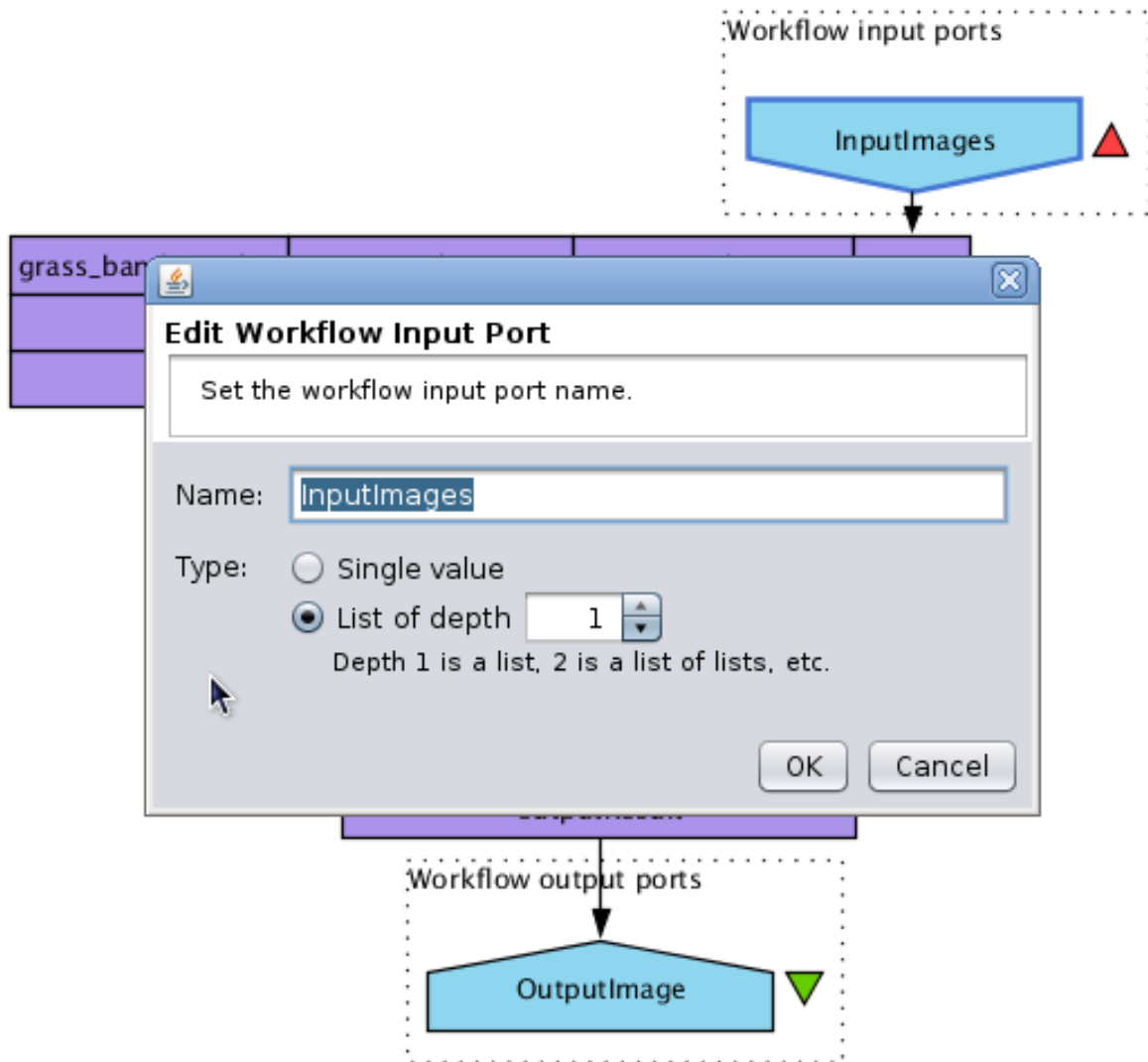


Figure 15 - List select. Multiple input WPS.

To run the workflow the user can use Ctrl+R or click on the run button in the toolbar, this will open the input request port.

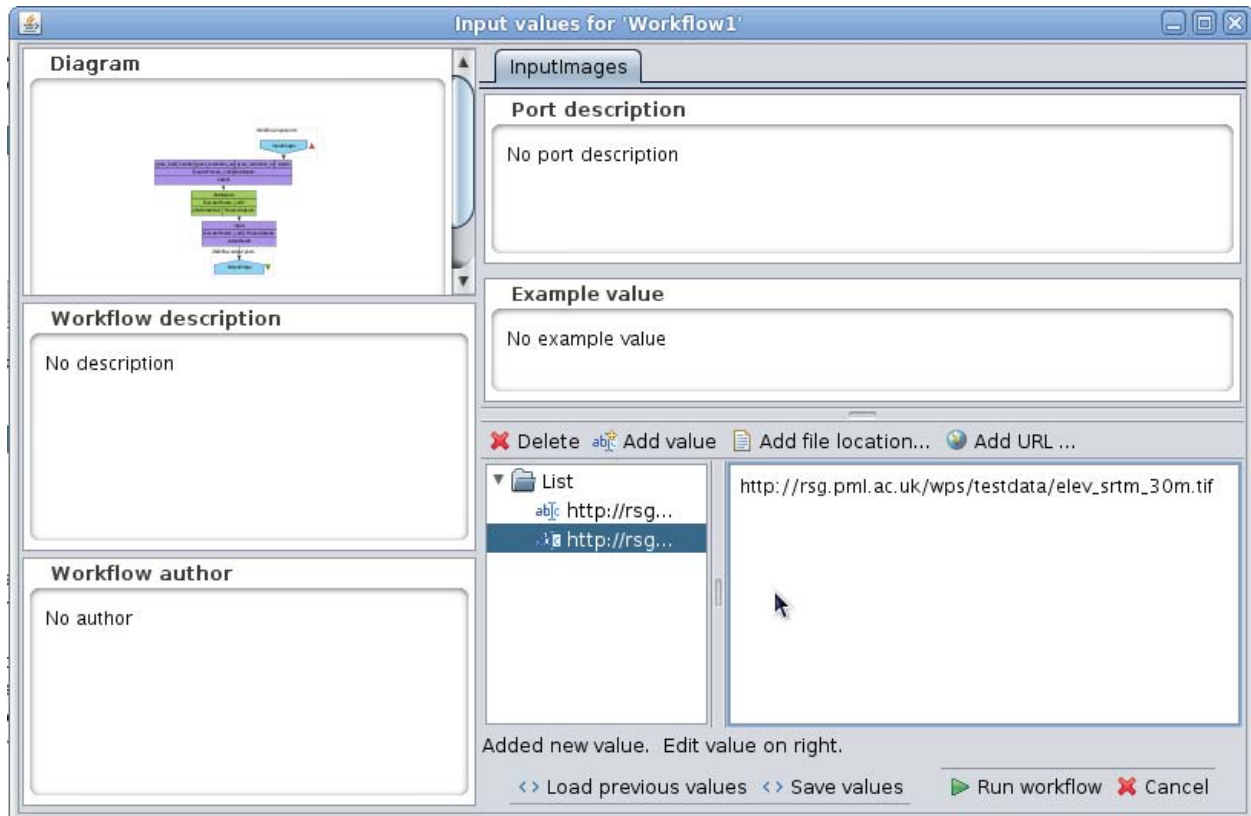


Figure 16 - Input dialogue GUI containing two inputs (elev_srtm_30m.tif file).

The user can then add the inputs to be sent to the service. In this example 2 identical URLs will be sent as input (Figure 16). After setting the inputs and clicking on the "Run workflow" the user will be conducted to the Results view, there is is possible to check the workflow as it is run and its final output, as follows (Figure 17):

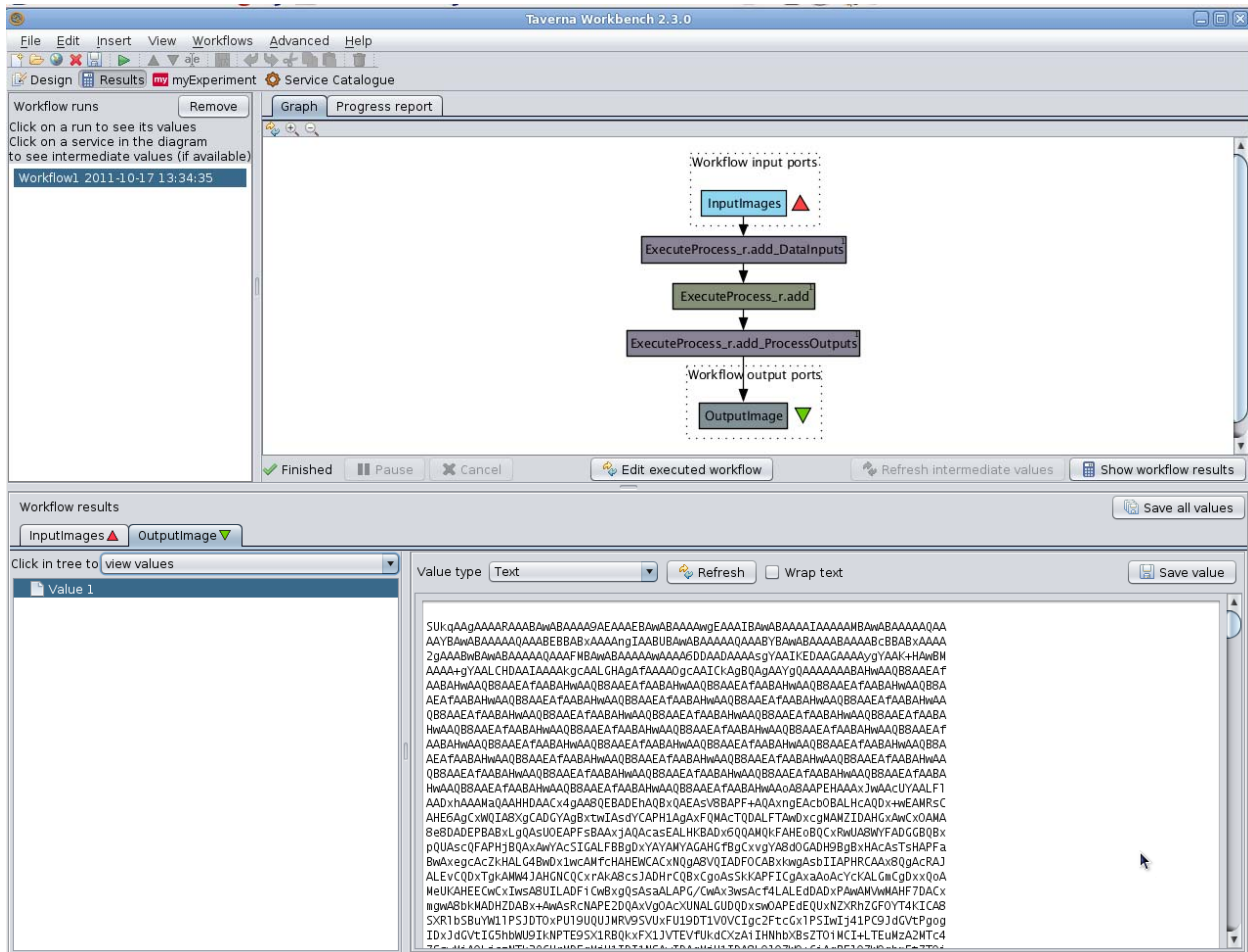


Figure - 17- Workflow output. Output image is in base64 code, therefore identified by Taverna as a text (string) output.

The output is located in the lower right section. The presented result in this example is a base64 encoded image, hence the long char string (Figure 17).

Taverna provides local services (located in the "Local Services" of the service panel) that can decode base64 into a binary format for display, this service in conjunction with another service that transforms GeoTiff into PNG allow for the construction of the following workflow (Figure 18):

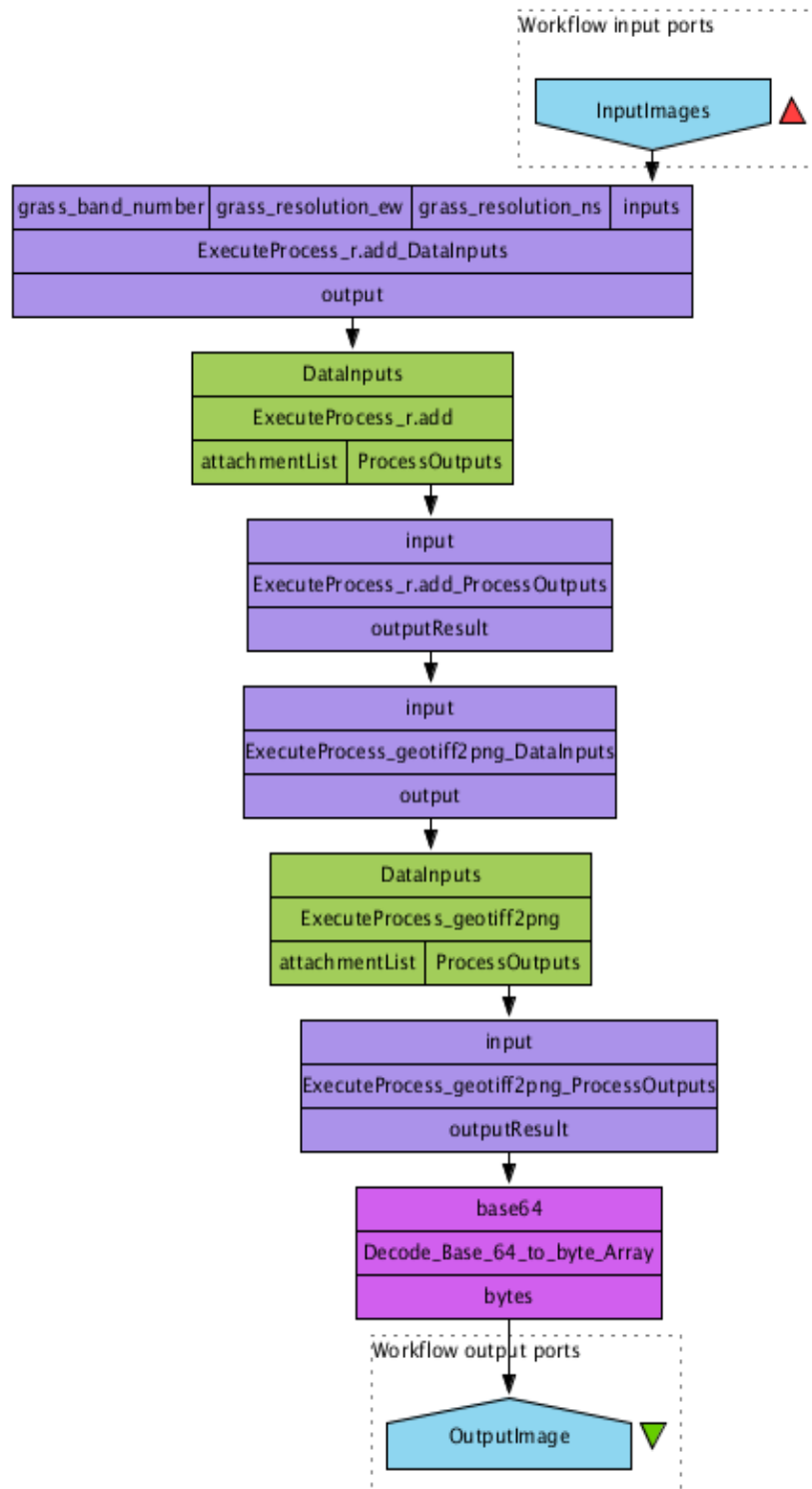


Figure 18 - Extended workflow with image transformation (base64 encoding into binary).

That when run will have the following output (Figure 19):

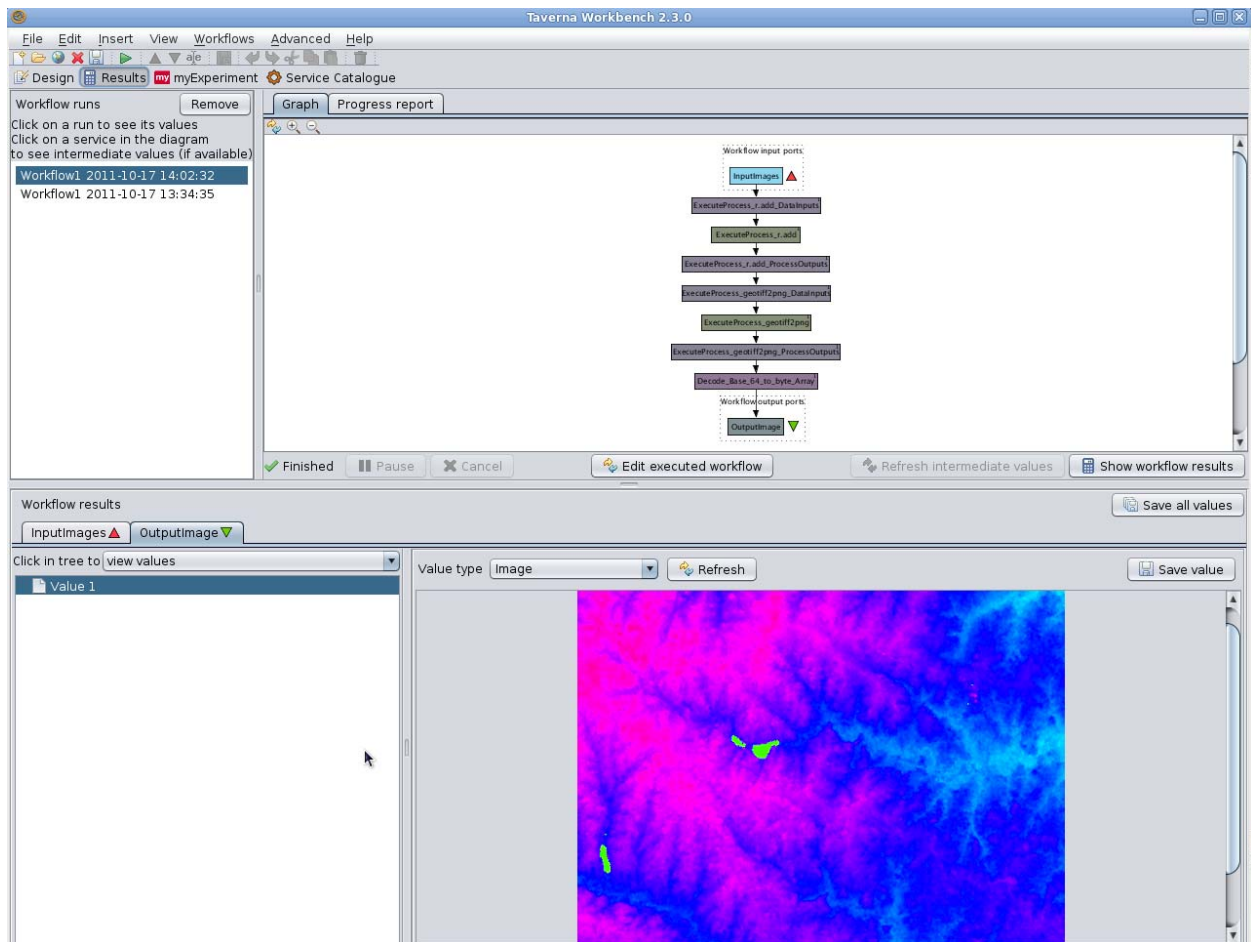


Figure 19 - Image PNG displayed in Taverna.

Taverna contains extensive tools and functionalities that help workflow development. It's advisable to consult Taverna's web site and read the tutorials and examples [7].

4.2.1. WPS-Async call in Taverna

Async calls to WPS services return an URL pointing to a document with information concerning the service status and then it's final result.

The user would have to select the Asynchronous service, that is indicated with prefix "ExecuteProcessAsync_" and that outputs a "statusURLResult".

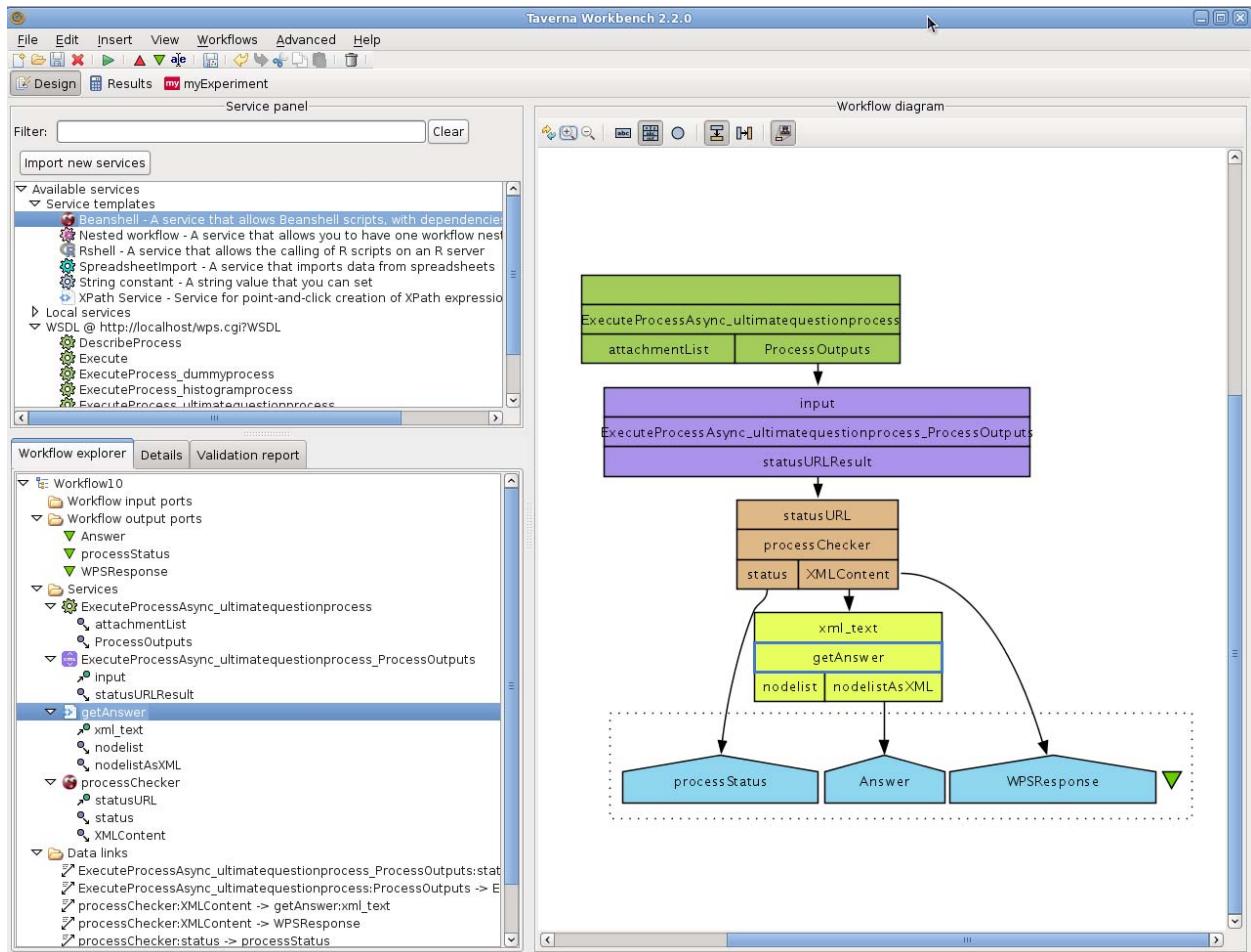


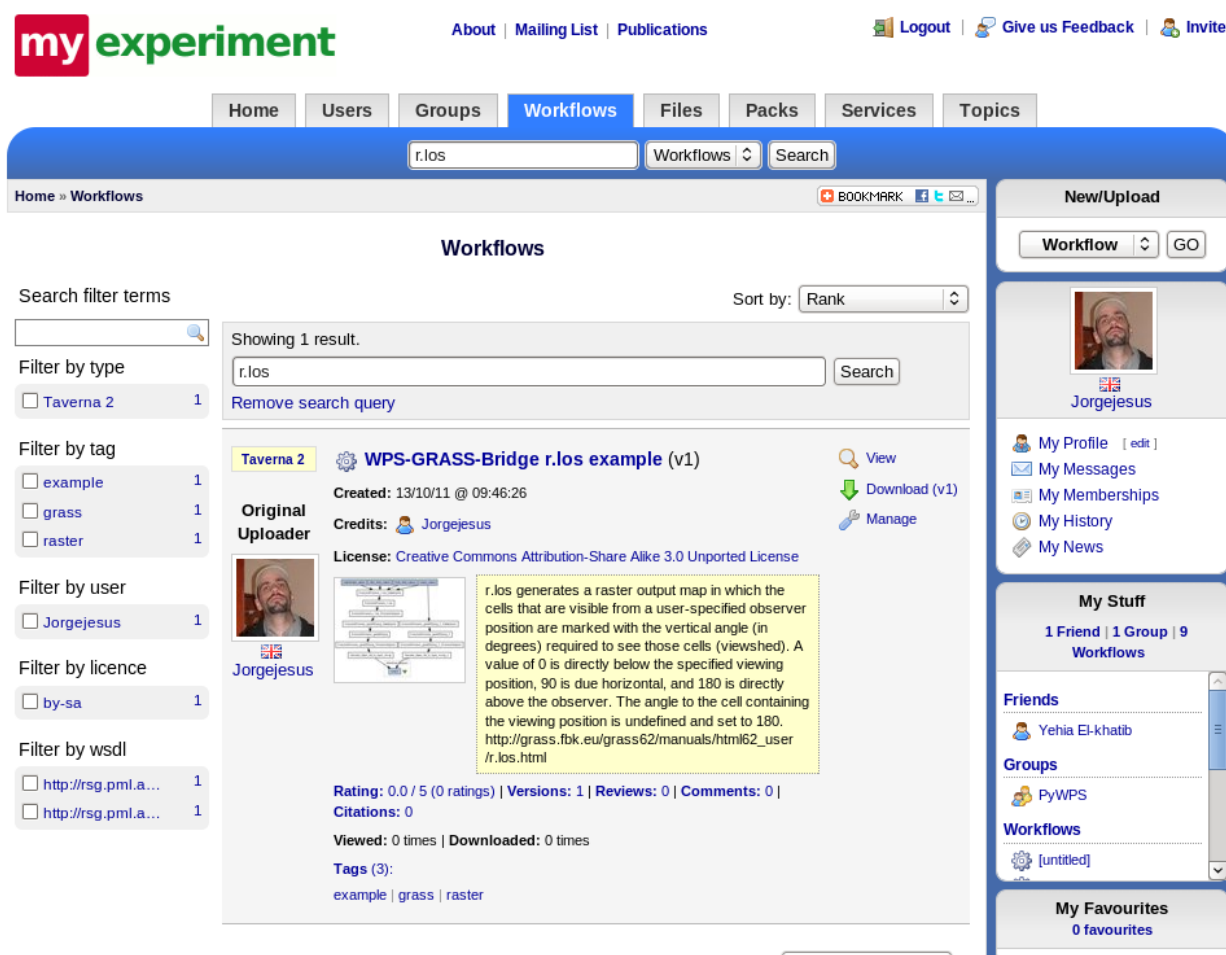
Figure 20 - Example of an asyn call to the "ultimatequestion process" that takes 25 seconds to complete.

The statusURLresult would be then passed to a local process (programmed in BeanShell) that would query the status URL until a output is present and passed to the following process (or output box) (Figure 20).

For an extended tutorial and complete example please see [8]:

4.3.myExperiment

myExperiment [9] is a social web site that allows for researchers to share their scientific workflows, any workflow developed in Taverna using WPS-WSDL can be uploaded to such platform and downloaded/consulted by other peers:



The screenshot displays the myExperiment website interface. At the top, there is a navigation bar with the myExperiment logo and links for 'About', 'Mailing List', 'Publications', 'Logout', 'Give us Feedback', and 'Invite'. Below this is a secondary navigation bar with tabs for 'Home', 'Users', 'Groups', 'Workflows', 'Files', 'Packs', 'Services', and 'Topics'. The 'Workflows' tab is selected. A search bar at the top contains the query 'r.los'. The main content area shows a search result for 'WPS-GRASS-Bridge r.los example (v1)' by user 'Jorgejesus'. The workflow details include its creation date (13/10/11 @ 09:46:26), credits to 'Jorgejesus', and a Creative Commons Attribution-Share Alike 3.0 Unported License. A description explains that the 'r.los' module generates a raster output map where cells visible from a user-specified observer position are marked with a vertical angle (in degrees) required to see those cells (viewshed). The description also includes a URL: http://grass.fbk.eu/grass62/manuals/html62_user/r.los.html. The workflow has a rating of 0.0/5 (0 ratings), 1 version, 0 reviews, 0 comments, and 0 citations. It has been viewed 0 times and downloaded 0 times. The tags are 'example', 'grass', and 'raster'. On the right side, there is a sidebar with sections for 'New/Upload', 'My Profile' (with a link to 'edit'), 'My Messages', 'My Memberships', 'My History', 'My News', 'My Stuff' (1 Friend, 1 Group, 9 Workflows), 'Friends' (Yehia El-khatib), 'Groups' (PyWPS), 'Workflows' ([untitled]), and 'My Favourites' (0 favourites).

Figure 21 - Example of workflow search in myExperiment, using "r.los" as word query.

Figure 21, shows a workflow search, in this search the keyword was "r.los" that is a GRASS module that generates a raster output map in which the cells that are visible from a user-specified observer position are marked with the vertical angle (in degrees) required to see those cells (viewshed).

The r.los workflow example can be downloaded directly into Taverna or explored as a PNG in the site (Figure 22).

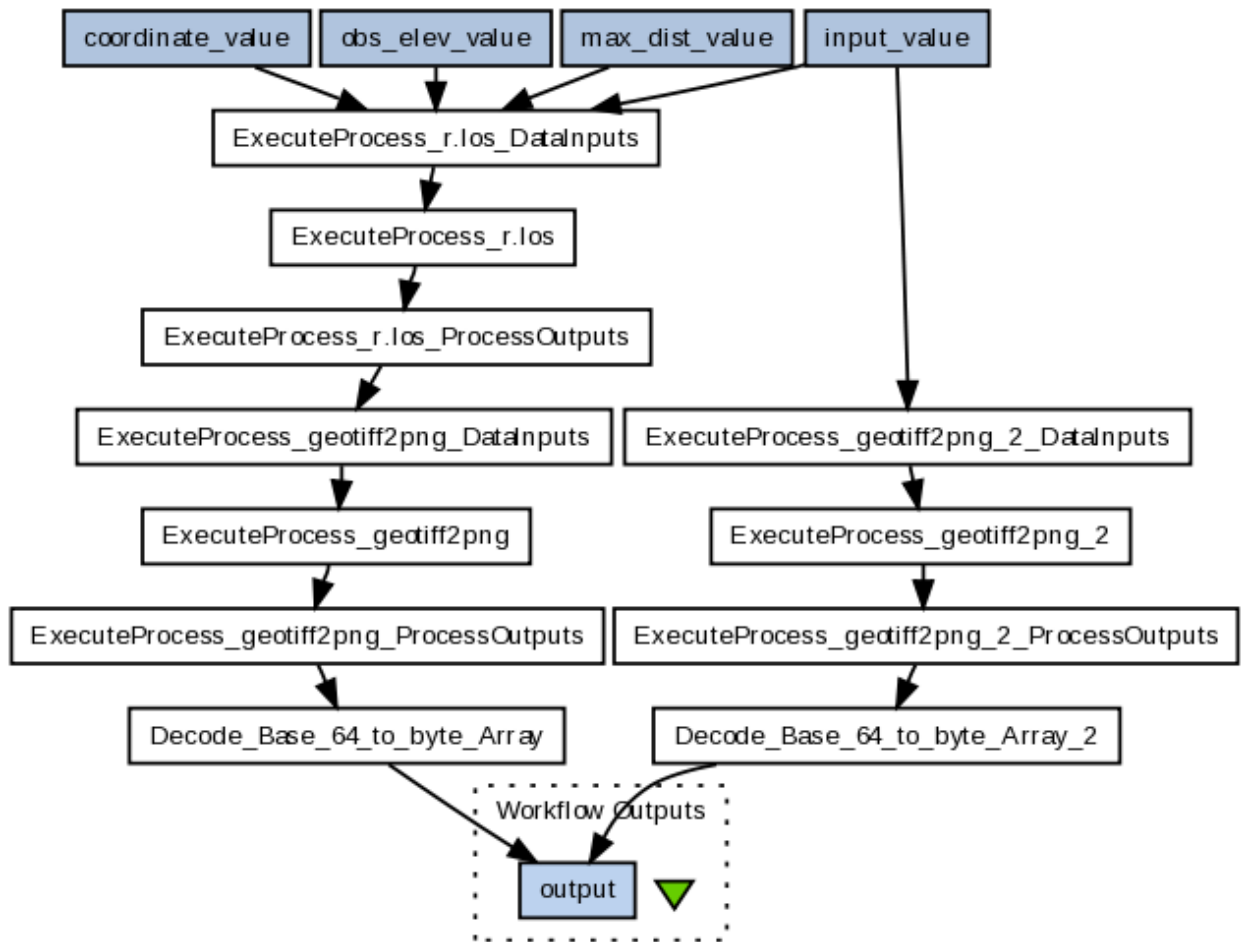


Figure 22 - Generic representation of a workflow containing the r.los web service.

Acknowledgements

This cookbook was written for the community under the auspices of the NETMAR (Open Service Network for Marine Environmental Data) project. NETMAR is partially funded by the European Commission under Theme ICT-2009.6.4 ICT for environmental services and climate change adaptation of the Information & Communication Technologies FP7 Programme.

This document is also available as part of the OGC GEOSS tutorial. All graphical material is licensed under Creative Commons 2.0. All software packages are under LGPL or MIT Licence and can be freely distributed and/or modified.

5. Document Information

Author Dr. Jorge Samuel Mendes de Jesus

Contact jmdj@pml.ac.uk

Version 1.0

Date 2011 December 21

Revisions

6. References

- [1] Schaeffer, B. "Towards a Transactional Web Processing Service". Proceedings of the GI-Days, Muenster. 2008.
- [2] Open Geospatial Consortium Inc. "Technical Committee Policies and Procedures: MIME Media Types for GML - document: OGC 09-144r1 version 1.0". Editor: Clements Portele. 2010.
- [3] Open Geospatial Consortium Inc. "Web Services Common Specification. document: OGC 06-121r3 version 1.1.0 with Corrigendum 1". Editor: Arliss Whiteside. 2007.
- [4] Open Geospatial Consortium Inc. "OpenGIS Web Processing Service. document: OGC 05-007r7". Editor: Peter Schut. 2007.
- [5] de Jesus, J., Walker, P., Grant, M., and Groom, S. "WPS orchestration using the Taverna workbench: The eScience approach". Computers & Geosciences (in press). 2011.
- [6] Juneau, J., Baker, J., Soto, L., Ng, V., and Wierzbicki, F. "The definitive guide to Jython: Python for the Java platform". Springer. 2010
- [7] <http://www.taverna.org.uk/download/workbench/2-3/>
- [8] http://wiki.rsg.pml.ac.uk/pywps/Async_Request
- [9] <http://www.myexperiment.org/>