PyWPS

Release 4.2.0

Dec 17, 2018

Contents

PyWPS is a server side implementation of the OGC Web Processing Service (OGC WPS) standard, using the Python programming language. PyWPS is currently supporting WPS 1.0.0. Support for the version 2.0.0. of OGC WPS standard is presently being planned.

Like the bicycle in the logo, PyWPS is:

- simple to maintain
- fast to drive
- able to carry a lot
- easy to hack

Mount your bike and setup your PyWPS instance!

Todo:

- request queue management (probably linked from documentation)
- inputs and outputs IOhandler class description (file, stream, . . .)

CHAPTER 1

Contents:

1.1 OGC Web Processing Service (OGC WPS)

OGC Web Processing Service standard provides rules for standardizing how inputs and outputs (requests and responses) for geospatial processing services. 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.

Note: This description is mainly refering to 1.0.0 version standard, since PyWPS implements this version only. There is also 2.0.0 version, which we are about to implement in near future.

WPS is intended to be state-less protocol (like any OGC services). For every request-response action, the negotiation between the server and the client has to start. There is no official way, how to make the server "remember", what was before, there is no communication history between the server and the client.

1.1.1 Process

A process *p* is a function that for each input returns a corresponding output:

$$p : X \rightarrow Y$$

where *X* denotes the domain of arguments *x* and *Y* denotes the co-domain of values *y*.

Within the specification, process arguments are referred to as *process inputs* and result values are referred to as *process outputs*. Processes that have no process inputs represent value generators that deliver constant or random process outputs.

Process is just some geospatial operation, which has it's in- and outputs and which is deployed on the server. It can be something relatively simple (adding two raster maps together) or very complicated (climate change model). It can take short time (seconds) or long (days) to be calculated. Process is, what you, as PyWPS user, want to expose to other people and let their data processed.

Every process has the following properties:

Identifier Unique process identifier

Title Human readable title

Abstract Longer description of the process, what it does, how is it supposed to be used

And a list of inputs and outputs.

1.1.2 Data inputs and outputs

OGC WPS defines 3 types of data inputs and outputs: *LiteralData*, *ComplexData* and *BoundingBoxData*.

All data types do need to have following properties:

Identifier Unique input identifier

Title Human readable title

Abstract Longer description of data input or output, so that the user could get oriented.

minOccurs Minimal occurrence of the input (e.g. there can be more bands of raster file and they all can be passed as input using the same identifier)

maxOccurs Maxium number of times, the input or output is present

Depending on the data type (Literal, Complex, BoundingBox), other attributes might occur too.

LiteralData

Literal data is any text string, usually short. It's used for passing single parameters like numbers or text parameters. WPS enables to the server, to define *allowedValues* - list or intervals of allowed values, as well as data type (integer, float, string). Additional attributes can be set, such as *units* or *encoding*.

ComplexData

Complex data are usually raster or vector files, but basically any (usually file based) data, which are usually processed (or result of the process). The input can be specified more using *mimeType*, XML *schema* or *encoding* (such as *base64* for raster data.

Note: PyWPS (like every server) supports limited list *mimeTypes*. In case you need some new format, just create pull request in our repository. Refer `pywps.inout.formats.FORMATS` for more details.

Usually, the minimum requirement for input data identification is *mimeType*. That usually is *application/gml+xml* for GML-encoded vector files, *image/tiff; subtype=geotiff* for raster files. The input or output can also be result of any OGC OWS service.

BoundingBoxData

Todo: add reference to OGC OWS Common spec

BoundingBox data are specified in OGC OWS Common specification as two pairs of coordinate (for 2D and 3D space). They can either be encoded in WGS84 or EPSG code can be passed too. They are intended to be used as definition of the target region.

Note: In real life, BoundingBox data are not that commonly used

1.1.3 Passing data to process instance

There are typically 3 approaches to pass the input data from the client to the server:

Data are on the server already In the first case, the data are already stored on the server (from the point of view of the client). This is the simplest case.

Data are send to the server along with the request In this case, the data are directly part of the XML encoded document send via HTTP POST. Some clients/servers are expecting the data to be inserted in *CDATA* section. The data can be text based (JSON), XML based (GML) or even raster based - in this case, they are usually encoded using base64.

Reference link to target service is passed Client does not have to pass the data itself, client can just send reference link to target data service (or file). In such case, for example OGC WFS *GetFeatureType* URL can be passed and server will download the data automatically.

Although this is usually used for *ComplexData* input type, it can be used for literal and bounding box data too.

1.1.4 Sychronous versus asynchronous process request

There are two modes of process instance execution: Synchronous and asynchronous.

Synchronous mode The client sends the *Execute* request to the server and waits with open server connection, till the process is calculated and final response is returned back. This is useful for fast calculations which do not take longer then a couple of seconds (Apache2 httpd server uses 300 seconds as default value for ConnectionTimeout).

Asynchronous mode Client sends the *Execute* request with explicit request for asynchronous mode. If supported by the process (in PyWPS, we have a configuration for that), the server returns back *ProcessAccepted* response immediately with URL, where the client can regularly check for *process execution status*.

Note: As you see, using WPS, the client has to apply *pull* method for the communication with the server. Client has to be the active element in the communication - server is just responding to clients request and is not actively *pushing* any information (like it would if e.g. web sockets would be implemented).

1.1.5 Process status

Process status is generic status of the process instance, reporting to the client, how does the calculation go. There are 4 types of process statuses

ProcessAccepted Process was accepted by the server and the process execution will start soon.

ProcessStarted Process calculation has started. The status also contains report about *percentDone* - calculation progress and *statusMessage* - text reporting current calculation state (example: *"Caculationg buffer"* - 33%).

ProcessFinished Process instance performed the calculation successfully and the final *Execute* response is returned to the client and/or stored on final location

ProcessFailed There was something wrong with the process instance and the server reports *server exception* (see *pywps.exceptions*) along with the message, what could possibly go wrong.

1.1.6 Request encoding, HTTP GET and POST

The request can be encoded either using key-value pairs (KVP) or an XML payload.

Key-value pairs is usually sent via HTTP GET request method encoded directly in the URL. The keys and values are separated with = sign and each pair is separated with *&* sign (with *?* at the beginning of the request. Example could be the *get capabilities reques*:

```
http://server.domain/wps?service=WPS&request=GetCapabilities&version=1.0.0
```

In this example, there are 3 pairs of input parameter: *service*, *request* and *version* with values *WPS*, *GetCapabilities* and *1.0.0* respectively.

XML payload is XML data sent via HTTP POST request method. The XML document can be more rich, having more parameters, better to be parsed in complex structures. The Client can also encode entire datasets to the request, including raster (encoded using base64) or vector data (usually as GML file).:

```
<?xml version="1.0" encoding="UTF-8"?>
<wps:GetCapabilities language="cz" service="WPS" 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 http://schemas.opengis.net/wps/1.0.0/wpsGetCapabilities_request.xsd">
  <wps:AcceptVersions>
    <ows:Version>1.0.0</ows:Version>
  </wps:AcceptVersions>
</wps:GetCapabilities>
```

Note: Even it might be looking more complicated to use XML over KVP, for some complex request it usually is more safe and efficient to use XML encoding. The KVP way, especially for WPS Execute request can be tricky and lead to unpredictable errors.

1.2 PyWPS

Todo:

- how are things organised
- storage
- dblog
- relationship to grass gis

1.2.1 PyWPS philosophy

PyWPS is simple, fast to run, has low requirements on system resources, is modular. PyWPS solves the problem of exposing geospatial calculations to the web, taking care of security, data download, request acceptance, process running and final response construction. Therefore PyWPS has a bicycle in its logo.

1.2.2 Why is PyWPS there

Many scientific researchers and geospatial services provider need to setup system, where the geospatial operations would be calculated on the server, while the system resources could be exposed to clients. PyWPS is here, so that you could set up the server fast, deploy your awesome geospatial calculation and expose it to the world. PyWPS is written in Python with support for many geospatial tools out there, like GRASS GIS, R-Project or GDAL. Python is the most geo-positive scripting language out there, therefore all the best tools have their bindings to Python in their pocket.

1.2.3 PyWPS History

PyWPS started in 2006 as scholarship funded by German Foundation for Environment. During the years, it grow to version 4.0.x. In 2015, we officially entered to OSGeo incubation process. In 2016, Project Steering Committee has started. PyWPS was originally hosted by the Wald server, nowadays, we moved to GeoPython group on GitHub. Since 2016, we also have new domain PyWPS.org.

You can find more at history page.

1.3 Installation

Note: PyWPS is not tested on the MS Windows platform. Please join the development team if you need this platform to be supported. This is mainly because of the lack of a multiprocessing library. It is used to process asynchronous execution, i.e., when making requests storing the response document and updating a status document displaying the progress of execution.

1.3.1 Dependencies and requirements

PyWPS runs on Python 2.7, 3.3 or higher. PyWPS is currently tested and developed on Linux (mostly Ubuntu). In the documentation we take this distribution as reference.

Prior to installing PyWPS, Git and the Python bindings for GDAL must be installed in the system. In Debian based systems these packages can be installed with a tool like *apt*:

```
$ sudo apt-get install git python-gdal
```

Alternatively, if GDAL is already installed on your system you can install the GDAL Python bindings via pip with:

```
$ pip install GDAL==1.10.0 --global-option=build_ext --global-option="-I/usr/include/
↪gdal"
```

1.3.2 Download and install

Using pip The easiest way to install PyWPS is using the Python Package Index (PIP). It fetches the source code from the repository and installs it automatically in the system. This might require superuser permissions (e.g. *sudo* in Debian based systems):

```
$ sudo pip install -e git+https://github.com/geopython/pywps.git@master#egg=pywps-
↪dev
```

Todo:

- document Debian / Ubuntu package support

Manual installation Manual installation of PyWPS requires downloading the source code followed by usage of the *setup.py* script. An example again for Debian based systems (note the usage of *sudo* for install):

```
$ tar zxf pywps-x.y.z.tar.gz
$ cd pywps-x.y.z/
```

Then install the package dependencies using pip:

```
$ pip install -r requirements.txt
$ pip install -r requirements-gdal.txt  # for GDAL Python bindings (if python-
↪gdal is not already installed by `apt-get`)
$ pip install -r requirements-dev.txt  # for developer tasks
```

To install PyWPS system-wide run:

```
$ sudo python setup.py install
```

For Developers Installation of the source code using Git and Python's virtualenv tool:

```
$ virtualenv my-pywps-env
$ cd my-pywps-env
$ source bin/activate
$ git clone https://github.com/geopython/pywps.git
$ cd pywps
```

Then install the package dependencies using pip as described in the Manual installation section. To install PyWPS:

```
$ python setup.py install
```

Note that installing PyWPS via a virtualenv environment keeps the installation of PyWPS and its dependencies isolated to the virtual environment and does not affect other parts of the system. This installation option is handy for development and / or users who may not have system-wide administration privileges.

1.3.3 The Flask service and its sample processes

To use PyWPS the user must code processes and publish them through a service. An example service is available that makes up a good starting point for first time users. It launches a very simple built-in server (relying on the Flask Python Microframework), which is good enough for testing but probably not appropriate for production. This example service can be cloned directly into the user area:

```
$ git clone https://github.com/geopython/pywps-flask.git
```

It may be run right away through the *demo.py* script. First time users should start by studying the structure of this project and then code their own processes.

There is also an example service

Full more details please consult the *Processes* section. The example service contains some basic processes too, so you could get started with some examples (like *area*, *buffer*, *feature_count* and *grassbuffer*). These processes are to be taken just as inspiration and code documentation - most of them do not make any sense (e.g. *sayhello*).

1.4 Configuration

PyWPS is configured using a configuration file. The file uses the ConfigParser format, with interpolation initialised using *os.environ*.

New in version 4.0.0.

Warning: Compatibility with PyWPS 3.x: major changes have been made to the config file in order to allow for shared configurations with PyCSW and other projects.

The configuration file has several sections:

- *metadata:main* for the server metadata inputs
- *server* for server configuration
- *processing* for processing backend configuration
- *logging* for logging configuration
- *grass* for *optional* configuration to support GRASS GIS

PyWPS ships with a sample configuration file (`default-sample.cfg`). A similar file is also available in the *flask* service as described in *The Flask service and its sample processes* section.

Copy the file to `default.cfg` and edit the following:

1.4.1 [metadata:main]

The *[metadata:main]* section was designed according to the PyCSW project configuration file.

identification_title the title of the service

identification_abstract some descriptive text about the service

identification_keywords comma delimited list of keywords about the service

identification_keywords_type keyword type as per the ISO 19115 MD_KeywordTypeCode codelist). Accepted values are `discipline`, `temporal`, `place`, `theme`, `stratum`

identification_fees fees associated with the service

identification_accessconstraints access constraints associated with the service

provider_name the name of the service provider

provider_url the URL of the service provider

contact_name the name of the provider contact

contact_position the position title of the provider contact

contact_address the address of the provider contact

contact_city the city of the provider contact

contact_stateorprovince the province or territory of the provider contact

contact_postalcode the postal code of the provider contact

contact_country the country of the provider contact

contact_phone the phone number of the provider contact

contact_fax the facsimile number of the provider contact

contact_email the email address of the provider contact

contact_url the URL to more information about the provider contact

contact_hours the hours of service to contact the provider

contact_instructions the how to contact the provider contact

contact_role the role of the provider contact as per the ISO 19115 CI_RoleCode codelist). Accepted values are `author`, `processor`, `publisher`, `custodian`, `pointOfContact`, `distributor`, `user`, `resourceProvider`, `originator`, `owner`, `principalInvestigator`

1.4.2 [server]

url the URL of the WPS service endpoint

language the ISO 639-1 language and ISO 3166-1 alpha2 country code of the service (e.g. `en-CA`, `fr-CA`, `en-US`)

encoding the content type encoding (e.g. `ISO-8859-1`, see https://docs.python.org/2/library/codecs.html#standard-encodings). Default value is 'UTF-8'

parallelprocesses maximum number of parallel running processes - set this number carefully. The effective number of parallel running processes is limited by the number of cores in the processor of the hosting machine. As well, speed and response time of hard drives impact ultimate processing performance. A reasonable number of parallel running processes is not higher than the number of processor cores.

maxrequestsize maximal request size. 0 for no limit

maxprocesses maximal number of requests being stored in queue, waiting till they can be processed (see `parallelprocesses` configuration option).

workdir a directory to store all temporary files (which should be always deleted, once the process is finished).

outputpath server path where to store output files.

outputurl corresponding URL

allowedinputpaths server paths which are allowed to be used by file URLs. A list of paths must be seperated by *:*.

Example: */var/lib/pywps/downloads:/var/lib/pywps/public*

By default no input paths are allowed.

cleantempdir flag to enable removal of process temporary workdir after process has finished.

Default = *true*.

Note: *outputpath* and *outputurl* must correspond. *outputpath* is the name of the resulting target directory, where all output data files are stored (with unique names). *outputurl* is the corresponding full URL, which is targeting to *outputpath* directory.

Example: *outputpath=/var/www/wps/outputs* shall correspond with *outputurl=http://foo.bar/wps/outputs*

1.4.3 [processing]

mode the mode/backend used for processing. Possible values are: *default*, *multiprocessing* and *scheduler*. *default* is the same as *multiprocessing* and is the default value ... all processes are executed using the Python multiprocessing module on the same machine as the PyWPS service. *scheduler* is used to enable the job scheduler extension and process execution is delegated to a configured scheduler system like Slurm and Grid Engine.

path path to the PyWPS *joblauncher* executable. This option is only used for the *scheduler* backend and is by default set automatically: *os.path.dirname(os.path.realpath(sys.argv[0]))*

1.4.4 [logging]

level the logging level (see http://docs.python.org/library/logging.html#logging-levels)

format the format string used by the logging *:Formatter:* (see https://docs.python.org/3/library/logging.html#logging.Formatter). For example: `%(asctime)s] [%(levelname)s] %(message)s`.

file the full file path to the log file for being able to see possible error messages.

database Connection string to database where the login about requests/responses is to be stored. We are using SQLAlchemy please use the configuration string. The default is SQLite3 *:memory:* object.

1.4.5 [grass]

gisbase directory of the GRASS GIS instalation, refered as GISBASE

Sample file

```
[server]
encoding=utf-8
language=en-US
url=http://localhost/wps
maxoperations=30
maxinputparamlength=1024
maxsingleinputsize=
maxrequestsize=3mb
temp_path=/tmp/pywps/
processes_path=
outputurl=/data/
outputpath=/tmp/outputs/
workdir=
allowedinputpaths=/tmp

[metadata:main]
identification_title=PyWPS Processing Service
identification_abstract=PyWPS is an implementation of the Web Processing Service␣
↪standard from the Open Geospatial Consortium. PyWPS is written in Python.
identification_keywords=PyWPS,WPS,OGC,processing
identification_keywords_type=theme
identification_fees=NONE
identification_accessconstraints=NONE
provider_name=Organization Name
```

(continues on next page)

(continued from previous page)

```
provider_url=http://pywps.org/
contact_name=Lastname, Firstname
contact_position=Position Title
contact_address=Mailing Address
contact_city=City
contact_stateorprovince=Administrative Area
contact_postalcode=Zip or Postal Code
contact_country=Country
contact_phone=+xx-xxx-xxx-xxxx
contact_fax=+xx-xxx-xxx-xxxx
contact_email=Email Address
contact_url=Contact URL
contact_hours=Hours of Service
contact_instructions=During hours of service.  Off on weekends.
contact_role=pointOfContact

[processing]
mode=default

[logging]
level=INFO
file=logs/pywps.log
database=sqlite:///logs/pywps-logs.sqlite3
format=%(asctime)s] [%(levelname)s] file=%(pathname)s line=%(lineno)s module=
↪%(module)s function=%(funcName)s %(message)s

[grass]
gisbase=/usr/local/grass-7.3.svn/
```

1.5 Processes

New in version 4.0.0.

Todo:

- Input validation
- IOHandler

PyWPS works with processes and services. A process is a Python *Class* containing an *handler* method and a list of inputs and outputs. A PyWPS service instance is then a collection of selected processes.

PyWPS does not ship with any processes predefined - it's on you, as user of PyWPS to set up the processes of your choice. PyWPS is here to help you publishing your awesome geospatial operation on the web - it takes care of communication and security, you then have to add the content.

Note: There are some example processes in the PyWPS-Flask project.

1.5.1 Writing a Process

Note: At this place, you should prepare your environment for final *Deployment to a production server*. At least, you should create a single directory with your processes, which is typically named *processes*:

```
$ mkdir processes
```

In this directory, we will create single python scripts containing processes.

Processes can be located *anywhere in the system* as long as their location is identified in the `PYTHONPATH` environment variable, and can be imported in the final server instance.

A processes is coded as a class inheriting from *pywps.Process*. In the PyWPS-Flask server they are kept inside the *processes* folder, usually in separated files.

The instance of a *Process* needs following attributes to be configured:

identifier unique identifier of the process

title corresponding title

inputs list of process inputs

outputs list of process outputs

handler method which recieves *pywps.app.WPSRequest* and *pywps.response.WPSResponse* as inputs.

1.5.2 Example vector buffer process

As an example, we will create a *buffer* process - which will take a vector file as the input, create specified the buffer around the data (using Shapely), and return back the result.

Therefore, the process will have two inputs:

- *ComplexData* input - the vector file
- *LiteralData* input - the buffer size

And it will have one output:

- *ComplexData* output - the final buffer

The process can be called *demobuffer* and we can now start coding it:

```
$ cd processes
$ $EDITOR demobuffer.py
```

At the beginning, we have to import the required classes and modules

Here is a very basic example:

```
28 from pywps import Process, LiteralInput, ComplexOutput, ComplexInput, Format
29 from pywps.app.Common import Metadata
30 from pywps.validator.mode import MODE
31 from pywps.inout.formats import FORMATS
```

As the next step, we define a list of inputs. The first input is *pywps.ComplexInput* with the identifier *vector*, title *Vector map* and there is only one allowed format: GML.

The next input is *pywps.LiteralInput*, with the identifier *size* and the data type set to *float*:

```
inpt_vector = ComplexInput(
    'vector',
    'Vector map',
    supported_formats=[Format('application/gml+xml')],
    mode=MODE.STRICT
)
```

Next we define the output *output* as *pywps.ComplexOutput*. This output supports GML format only.

```
out_output = ComplexOutput(
    'output',
    'HelloWorld Output',
    supported_formats=[Format('application/gml+xml')]
```

Next we create a new list variables for inputs and outputs.

```
inputs = [inpt_vector, inpt_size]
```

Next we define the *handler* method. In it, *geospatial analysis may happen*. The method gets a *pywps.app.WPSRequest* and a *pywps.response.WPSResponse* object as parameters. In our case, we calculate the buffer around each vector feature using GDAL/OGR library. We will not got much into the details, what you should note is how to get input data from the *pywps.app.WPSRequest* object and how to set data as outputs in the *pywps.response.WPSResponse* object.

```
@staticmethod
def _handler(request, response):
    """Handler method - this method obtains request object and response
    object and creates the buffer
    """

    from osgeo import ogr

    # obtaining input with identifier 'vector' as file name
    input_file = request.inputs['vector'][0].file

    # obtaining input with identifier 'size' as data directly
    size = request.inputs['size'][0].data

    # open file the "gdal way"
    input_source = ogr.Open(input_file)
    input_layer = input_source.GetLayer()
    layer_name = input_layer.GetName()

    # create output file
    driver = ogr.GetDriverByName('GML')
    output_source = driver.CreateDataSource(
        layer_name,
        ["XSISCHEMAURI=http://schemas.opengis.net/gml/2.1.2/feature.xsd"])
    output_layer = output_source.CreateLayer(layer_name, None, ogr.wkbUnknown)

    # get feature count
    count = input_layer.GetFeatureCount()
    index = 0
```

(continues on next page)

(continued from previous page)

```
97
98      # make buffer for each feature
99      while index < count:
100
101         response._update_status(WPS_STATUS.STARTED, 'Buffering feature %s' % index,␣
    ↪float(index) / count)
102
103         # get the geometry
104         input_feature = input_layer.GetNextFeature()
105         input_geometry = input_feature.GetGeometryRef()
106
107         # make the buffer
108         buffer_geometry = input_geometry.Buffer(float(size))
109
110         # create output feature to the file
111         output_feature = ogr.Feature(feature_def=output_layer.GetLayerDefn())
112         output_feature.SetGeometryDirectly(buffer_geometry)
113         output_layer.CreateFeature(output_feature)
114         output_feature.Destroy()
115         index += 1
116
117     # set output format
118     response.outputs['output'].data_format = FORMATS.GML
119
120     # set output data as file name
121     response.outputs['output'].file = layer_name
122
123     return response
```

At the end, we put everything together and create new a *DemoBuffer* class with handler, inputs and outputs. It's based on *pywps.Process*:

```
51  class DemoBuffer(Process):
52      def __init__(self):
53
54          super(DemoBuffer, self).__init__(
55              _handler,
56              identifier='demobuffer',
57              version='1.0.0',
58              title='Buffer',
59              abstract='This process demonstrates, how to create any process in PyWPS␣
    ↪environment',
60              metadata=[Metadata('process metadata 1', 'http://example.org/1'),
61                        Metadata('process metadata 2', 'http://example.org/2')],
62              inputs=inputs,
63              outputs=outputs,
64              store_supported=True,
65              status_supported=True
66          )
```

1.5.3 Declaring inputs and outputs

Clients need to know which inputs the processes expects. They can be declared as `pywps.Input` objects in the *Process* class declaration:

```
from pywps import Process, LiteralInput, LiteralOutput

class FooProcess(Process):
    def __init__(self):
        inputs = [
            LiteralInput('foo', data_type='string'),
            ComplexInput('bar', [Format('text/xml')])
        ]
        outputs = [
            LiteralOutput('foo_output', data_type='string'),
            ComplexOutput('bar_output', [Format('JSON')])
        ]

        super(FooProcess, self).__init__(
            ...
            inputs=inputs,
            outputs=outputs
        )
        ...
```

Note: A more generic description can be found in *OGC Web Processing Service (OGC WPS)* chapter.

LiteralData

- *LiteralInput*
- *LiteralOutput*

A simple value embedded in the request. The first argument is a name. The second argument is the type, one of *string*, *float*, *integer* or *boolean*.

ComplexData

- *ComplexInput*
- *ComplexOutput*

A large data object, for example a layer. ComplexData do have a *format* attribute as one of their key properties. It's either a list of supported formats or a single (already selected) format. It shall be an instance of the `pywps.inout.formats.Format` class.

ComplexData `Format` and input validation

The ComplexData needs as one of its parameters a list of supported data formats. They are derived from the *Format* class. A *Format* instance needs, among others, a *mime_type* parameter, a *validate* method – which is used for input data validation – and also a *mode* parameter – defining how strict the validation should be (see *pywps.validator.mode.MODE*).

The *Validate* method is up to you, the user, to code. It requires two input paramers - *data_input* (a *ComplexInput* object), and *mode*. This methid must return a *boolean* value indicating whether the input data are considered valid or not for given *mode*. You can draw inspiration from the *pywps.validator.complexvalidator.validategml()* method.

The good news is: there are already predefined validation methods for the ESRI Shapefile, GML and GeoJSON formats, using GDAL/OGR. There is also an XML Schema validaton and a JSON schema validator - you just have to pick the propper supported formats from the *pywps.inout.formats.FORMATS* list and set the validation mode to your *ComplexInput* object.

Even better news is: you can define custom validation functions and validate input data according to your needs.

BoundingBoxData

- *BoundingBoxInput*
- *BoundingBoxOutput*

BoundingBoxData contain information about the bounding box of the desired area and coordinate reference system. Interesting attributes of the BoundingBoxData are:

crs current coordinate reference system

dimensions number of dimensions

ll pair of coordinates (or triplet) of the lower-left corner

ur pair of coordinates (or triplet) of the upper-right corner

1.5.4 Accessing the inputs and outputs in the *handler* method

Handlers receive as input argument a `WPSRequest` object. Input values are found in the *inputs* dictionary:

```
@staticmethod
def _handler(request, response):
    name = request.inputs['name'][0].data
    response.outputs['output'].data = 'Hello world %s!' % name
    return response
```

inputs is a plain Python dictionary. Most of the inputs and outputs are derived from the `IOHandler` class. This enables the user to access the data in four different ways:

input.file Returns a file name - you can access the data using the name of the file stored on the hard drive.

input.url Return a link to the resource using either the `file://` or `http://` scheme. The target of the url is not downloaded to the PyWPS server until its content is explicitly accessed through either one of the `file`, `data` or `stream` attributes.

input.data Is the direct link to the data themselves. No need to create a file object on the hard drive or opening the file and closing it - PyWPS will do everything for you.

input.stream Provides the IOStream of the data. No need for opening the file, you just have to *read()* the data.

Because there could be multiple input values with the same identifier, the inputs are accessed with an index. For example:

```
request.inputs['file_input'][0].file
request.inputs['data_input'][0].data
request.inputs['stream_input'][0].stream
url_input = request.inputs['url_input'][0]
```

As mentioned, if an input is a link to a remote file (an `http` address), accessing the `url` attribute simply returns the url's string, but accessing any other attribute triggers the file's download:

```
url_input.url # returns the link as a string (no download)
url_input.file # downloads target and returns the local path
url_input.data # returns the content of the local copy
```

PyWPS will persistently transform the input (and output) data to the desired form. You can also set the data for your *Output* object like *output.data = 1* or *output.file = "myfile.json"* - it works the same way. However, once the source type is set, it cannot be changed. That is, a *ComplexOutput* whose `data` attribute has been set once has read-only access to the three other attributes (`file`, `stream` and `url`), while the `data` attribute can be freely modified.

1.5.5 Progress and status report

OGC WPS standard enables asynchronous process execution call, that is in particular useful, when the process execution takes longer time - process instance is set to background and WPS Execute Response document with *ProcessAccepted* messag is returned immediately to the client. The client has to check *statusLocation* URL, where the current status report is deployed, say every n-seconds or n-minutes (depends on calculation time). Content of the response is usually *percentDone* information about the progress along with *statusMessage* text information, what is currently happening.

You can set process status any time in the *handler* using the `WPSResponse.update_status()` function.

1.5.6 Returning large data

WPS allows for a clever method of returning a large data file: instead of embedding the data in the response, it can be saved separately, and a URL is returned from where the data can be downloaded. In the current implementation, PyWPS saves the file in a folder specified in the configuration passed by the service (or in a default location). The URL returned is embedded in the XML response.

This behaviour can be requested either by using a GET:

```
...ResponseDocument=output=@asReference=true...
```

Or a POST request:

```
...
<wps:ResponseForm>
    <wps:ResponseDocument>
        <wps:Output asReference="true">
            <ows:Identifier>output</ows:Identifier>
            <ows:Title>Some Output</ows:Title>
        </wps:Output>
    </wps:ResponseDocument>
</wps:ResponseForm>
...
```

output is the identifier of the output the user wishes to have stored and accessible from a URL. The user may request as many outputs by reference as needed, but only *one* may be requested in RAW format.

1.5.7 Process deployment

In order for clients to invoke processes, a PyWPS `Service` class must be present with the ability to listen for requests. An instance of this class must created, receiving instances of all the desired processes classes.

In the *flask* example service the `Service` class instance is created in the `Server` class. `Server` is a development server that relies on Flask. The publication of processes is encapsulated in *demo.py*, where a main method passes a list of processes instances to the `Server` class:

```
from pywps import Service
from processes.helloworld import HelloWorld
from processes.demobuffer import DemoBuffer

...
processes = [ DemoBuffer(), ... ]

server = Server(processes=processes)

...
```

1.5.8 Running the dev server

The *The Flask service and its sample processes* server is a WSGI application that accepts incoming *Execute* requests and calls the appropriate process to handle them. It also answers *GetCapabilities* and *DescribeProcess* requests based on the process identifier and their inputs and outputs.

A host, a port, a config file and the processes can be passed as arguments to the `Server` constructor. **host** and **port** will be **prioritised** if passed to the constructor, otherwise the contents of the config file (*pywps.cfg*) are used.

Use the *run* method to start the server:

```
...
s = Server(host='localhost', processes=processes, config_file=config_file)
s.run()
...
```

To make the server visible from another computer, replace `localhost` with `0.0.0.0`.

1.5.9 Automated process documentation

A *Process* can be automatically documented with Sphinx using the *autoprocess* directive. The *Process* object is instantiated and its content examined to create, behind the scenes, a docstring in the Numpy format. This lets developers embed the documentation directly in the code instead of having to describe each process manually. For example:

```
.. autoprocess:: pywps.tests.DocExampleProcess
   :docstring:
   :skiplines: 1
```

would yield

class pywps.tests.**DocExampleProcess**

doc_example_process_identifier Process title (v4.0)

Multiline process abstract.

Parameters

- **literal_input** (*integer, optional, units:[meters, feet]*) – Literal input value abstract.

- **date_input** (*{'2000-01-01', '2018-01-01'}*) – The title is shown when no abstract is provided.
- **complex_input** (*application/json*, *application/x-netcdf*) – Complex input abstract.
- **bb_input** (*[EPSG:4326]*) – BoundingBox input title (EPSG.io)

Returns

- **literal_output** (*boolean*) – Boolean output abstract.
- **complex_output** (*text/plain*) – Complex output
- **bb_output** (*[EPSG:4326]*) – BoundingBox output title

References

- PyWPS docs
- NumPy docstring conventions

Notes

This is additional documentation that can be added following the Numpy docstring convention.

The `docstring` option fetches the *`Process`* docstring and appends it after the Reference section. The first lines of this docstring can be skipped using the `skiplines` option.

To use the *autoprocess* directive, first add *'sphinx.ext.napoleon'* and *'pywps.ext_autodoc'* to the list of extensions in the Sphinx configuration file `conf.py`. Then, insert *autoprocess* directives in your documentation source files, just as you would use an *autoclass* directive, and build the documentation.

Note that for input and output parameters, the *title* is displayed only if no *abstract* is defined. In other words, if both *title* and *abstract* are given, only the *abstract* will be included in the documentation to avoid redundancy.

1.6 Deployment to a production server

As already described in the *Installation* section, no specific deployment procedures are for PyWPS when using flask-based server. But this formula is not intended to be used in a production environment. For production, sudo service apache2 restartApache httpd or nginx servers are more advised. PyWPS is runs as a WSGI application on those servers. PyWPS relies on the Werkzeug library for this purpose.

1.6.1 Deploying an individual PyWPS instance

PyWPS should be installed in your computer (as per the *Installation* section). As a following step, you can now create several instances of your WPS server.

It is advisable for each PyWPS instance to have its own directory, where the WSGI file along with available processes should reside. Therefore create a new directory for the PyWPS instance:

```
$ sudo mkdir /path/to/pywps/

# create a directory for your processes too
$ sudo mkdir /path/to/pywps/processes
```

Note: In this configuration example it is assumed that there is only one instance of PyWPS on the server.

Each instance is represented by a single *WSGI* script (written in Python), which:

1. Loads the configuration files
2. Serves processes
3. Takes care about maximum number of concurrent processes and similar

1.6.2 Creating a PyWPS *WSGI* instance

An example WSGI script is distributed along with the pywps-flask service, as described in the *Installation* section. The script is actually straightforward - in fact, it's a just wrapper around the PyWPS server with a list of processes and configuration files passed as arguments. Here is an example of a PyWPS WSGI script:

```
$ $EDITOR /path/to/pywps/pywps.wsgi
```

```
#!/usr/bin/env python3

from pywps.app.Service import Service

# processes need to be installed in PYTHON_PATH
from processes.sleep import Sleep
from processes.ultimate_question import UltimateQuestion
from processes.centroids import Centroids
from processes.sayhello import SayHello
from processes.feature_count import FeatureCount
from processes.buffer import Buffer
from processes.area import Area

processes = [
    FeatureCount(),
    SayHello(),
    Centroids(),
    UltimateQuestion(),
    Sleep(),
    Buffer(),
    Area()
]

# Service accepts two parameters:
# 1 - list of process instances
# 2 - list of configuration files
application = Service(
    processes,
    ['/path/to/pywps/pywps.cfg']
)
```

Note: The WSGI script is assuming that there are already some processes at hand that can be directly included. Also it assumes, that the configuration file already exists - which is not the case yet.

The Configuration is described in next chapter (*Configuration*), as well as process creation and deployment (*Processes*).

1.6.3 Deployment on Apache2 httpd server

First, the WSGI module must be installed and enabled:

```
$ sudo apt-get install libapache2-mod-wsgi
$ sudo a2enmod wsgi
```

You then can edit your site configuration file (*/etc/apache2/sites-enabled/yoursite.conf*) and add the following:

```
# PyWPS
WSGIDaemonProcess pywps home=/path/to/pywps user=www-data group=www-data processes=2␣
↪threads=5
WSGIScriptAlias /pywps /path/to/pywps/pywps.wsgi process-group=pywps

<Directory /path/to/pywps/>
    WSGIScriptReloading On
    WSGIProcessGroup pywps
    WSGIApplicationGroup %{GLOBAL}
    Require all granted
</Directory>
```

Note: *WSGIScriptAlias* points to the *pywps.wsgi* script created before - it will be available under the url http://localhost/pywps

Note: Please make sure that the *logs*, *workdir*, and *outputpath* directories are writeable to the Apache user. The *outputpath* directory need also be accessible from the URL mentioned in *outputurl* configuration.

And of course restart the server:

```
$ sudo service apache2 restart
```

1.6.4 Deployment on Nginx-Gunicorn

Note: We will use Greenunicorn for pyWPS deployment, since it is a very simple to configurate server.

For difference between WSGI server consult: WSGI comparison.

uWSGU is more popular than gunicorn, best documentation is probably to be found at Readthedocs.

We need nginx and gunicorn server:

```
$ apt install nginx-full
$ apt install gunicorn3
```

It is assumed that PyWPS is installed in your system (if not see: ref:*installation*) and we will use pywps-flask as installation example.

First, cloning the pywps-flask example to the root / (you need to be sudoer or root to run the examples):

```
$ cd /
$ git clone https://github.com/geopython/pywps-flask.git
```

Second, preparing the WSGI script for gunicorn. It is necessary that the WSGI script located in the pywps-flask service is identified as a python module by gunicorn, this is done by creating a link with .py extention to the wsgi file:

```
$ cd /pywps-flask/wsgi
$ ln -s ./pywps.wsgi ./pywps_app.py
```

Gunicorn can already be tested by setting python path on the command options:

```
$ gunicorn3 -b 127.0.0.1:8081  --workers $((2*`nproc --all`)) --log-syslog  --
↪pythonpath /pywps-flask wsgi.pywps_app:application
```

The command will start a gunicorn instance on the localhost IP and port 8081, logging to systlog (/var/log/syslog), using pywps process folder /pywps-flask/processes and loading module wsgi.pywps_app and object/function application for WSGI.

Note: Gunicorn uses a prefork model where the master process forks processes (workers) that willl accept incomming connections. The –workers flag sets the number of processes, the default values is 1 but the recomended value is 2 or 4 times the number of CPU cores.

Next step is to configure NGINX, by pointing to the WSGI server by changing the location paths of the default site file but editing file /etc/nginx/sites-enabled as follows::

```
server {
     listen 80 default_server;
     listen [::]:80 default_server;
     server_name _;

     #better to redirect / to wps application
     location / {
     return 301 /wps;
     }

     location /wps {
             # with try_files active there will be problems
             #try_files $uri $uri/ =404;

             proxy_set_header Host $host;
             proxy_redirect          off;
             proxy_set_header        X-NginX-Proxy true;
             proxy_set_header X-Real-IP $remote_addr;
             proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
             proxy_pass http://127.0.0.1:8081;
             }

}
```

It is likely that part of the proxy configuration is already set on the file /etc/nginx/proxy.conf. Of course the necessatyrestart of nginx

```
$ service nginx restart
```

The service will now be available on the IP of the server or localhost

```
http://localhost/wps?request=GetCapabilities&service=wps
```

The current gunicorn instance was launched by the user. In a production server it is necessary to set gunicorn as a service

On ubuntu 16.04 the systemcltd system requires a service file that will start the gunicorn3 service. The service file (/lib/systemd/system/gunicorn.service) has to be configure as follows:

```
[Unit]
Description=gunicorn3 daemon
After=network.target

[Service]
User=www-data
Group=www-data
PIDFile=/var/run/gunicorn3.pid
Environment=WORKERS=3
ExecStart=/usr/bin/gunicorn3 -b 127.0.0.1:8081   --preload --workers $WORKERS --log-
↪syslog --pythonpath /pywps-flask wsgi.pywps_app:application
ExecReload=/bin/kill -s HUP $MAINPID
ExecStop=/bin/kill -s TERM $MAINPID

[Install]
WantedBy=multi-user.target
```

And then enable the service and then reload the systemctl daemon:

```
$ systemctl enable gunicorn3.service
$ systemctl daemon-reload
$ systemctl restart gunicorn3.service
```

And to check that everything is ok:

```
$ systemctl status gunicorn3.service
```

Note: Todo NGIX + uWSGI

1.6.5 Testing the deployment of a PyWPS instance

Note: For the purpose of this documentation, it is assumed that you've installed PyWPS using the *localhost* server domain name.

As stated, before, PyWPS should be available at http://localhost/pywps, we now can visit the url (or use *wget*):

```
# the --content-error parameter makes sure, error response is displayed
$ wget --content-error -O - "http://localhost/pywps"
```

The result should be an XML-encoded error message.

```
<?xml version="1.0" encoding="UTF-8"?>
<ows:ExceptionReport 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" version="1.0.0">
    <ows:Exception exceptionCode="MissingParameterValue" locator="service">
        <ows:ExceptionText>service</ows:ExceptionText>
    </ows:Exception>
</ows:ExceptionReport>
```

The server responded with the *pywps.exceptions.MissingParameterValue* exception, telling us that the parameter *service* was not set. This is compliant with the OGC WPS standard, since each request mast have at least the *service* and *request* parameters. We can say for now, that this PyWPS instance is properly deployed on the server, since it returns proper exception report.

We now have to configure the instance by editing the *pywps.cfg* file and adding some processes.

1.7 Migrating from PyWPS 3.x to 4.x

The basic concept of PyWPS 3.x and 4.x remains the same: You deploy PyWPS once and can have many instances with set of processes. It's good practice to store processes in single files, although it's not required.

Note: Unluckily, there is not automatic tool for conversion of processes nor compatibility module. If you would like to sponsor development of such module, please contact Project Steering Committee via PyWPS mailing list or members of PSC directly.

1.7.1 Configuration file

Configuration file format remains the same (it's the one used by configparser module). The sections are shift a bit, so they are more alike another GeoPython project - pycsw.

See section *Configuration*.

1.7.2 Single process definition

The main principle remains the same between 3.x and 4.x branches: You have to define process class *class* and it's *__init__* method with inputs and outputs.

The former *execute()* method can now be any function and is assigned as *handler* attribute.

handler function get's two arguments: *request* and *response*. In *requests*, all input data are stored, *response* will have output data assinged.

The main difference between 3.x and 4.x is, *every input is list of inputs*. The reason for such behaviour is, that you, as user of PyWPS define input defined by type and identifier. When PyWPS process is turned to running job, there can be usually *more then one input with same identifier* defined. Therefore instead of calling:

```
def execute(self):

        ...

        # 3.x inputs
        input = self.my_input.getValue()
```

you shall use first index of an input list:

```
def handler(request, response):

        ...

        # 4.X inputs
        input = request.inputs['my_input'][0].file
```

1.7.3 Inputs and outputs data manipulation

Btw, PyWPS Inputs do now have *file*, *data*, *url* and *stream* attributes. They are transparently converting one data-representation-type to another. You can read input data from file-like object using *stream* or get directly the data into variable with *input.data*. You can also save output data directly using *output.data = { ..... }*.

See more in *Processes*

1.8 Deployment

While PyWPS 3.x was usually deployed as CGI application, PyWPS 4.x is configured as *WSGI* application. PyWPS 4.x is distributed without any processes or sample deploy script. We provide such example in our pywps-flask project.

Note: PYWPS_PROCESSES environment variable is gone, you have to assing processes to deploy script manually (or semi-automatically).

For deployment script, standard WSGI application as used by flask microframework has to be defined, which get's two parameters: list of processes and configuration files:

```
from pywps.app.Service import Service
from processes.buffer import Buffer

processes = [Buffer()]

application = Service(processes, ['wps.cfg'])
```

Those 4 lines of code do deploy PyWPS with Buffer process. This gives you more flexible way, how to define processes, since you can pass new variables and config values to each process class instance during it's definition.

1.9 Sample processes

For sample processes, please refer to pywps-flask project on GITHub.

1.10 Needed steps summarization

1. Fix configuration file
2. Every processes needs new class and inputs and outputs definition
3. In *execute* method, you just need to review inputs and outputs data assignment, but the core of the method should remain the same.
4. Replace shell or python-based CGI script with Flask-based WSGI script

1.11 PyWPS and external tools

1.11.1 GRASS GIS

PyWPS can handle all the management needed to setup temporal GRASS GIS environemtn (GRASS DBASE, Location and Mapset) for you. You just need to configure it in the *pywps.Process*, using the parameter `grass_location`, which can have 2 possible values:

epsg:[EPSG_CODE] New temporal location is created using the EPSG code given. PyWPS will create temporal directory as GRASS Location and remove it after the WPS Execute response is constructed.

/path/to/grassdbase/location/ Existing absolute path to GRASS Location directory. PyWPS will create temporal GRASS Mapset direcetory and remove it after the WPS Exceute response is constructed.

Then you can use Python - GRASS interfaces in the execute method, to make the work.

Note: Even PyWPS supports GRASS integration, the data still need to be imported using GRASS modules `v.in.*` or `r.in.*` and also they have to be exported manually at the end.

```
def execute(request, response):
        from grass.script import core as grass
        grass.run_command('v.in.ogr', input=request.inputs["input"][0].file,
        ...)
        ...
        grass.run_command('v.out.ogr', input="myvector", ...)
```

Also do not forget to set `gisbase` *Configuration* option.

1.11.2 OpenLayers WPS client

1.11.3 ZOO-Project

ZOO-Project provides both a server (C) and client (JavaScript) framework.

1.11.4 QGIS WPS Client

The QGIS WPS client provides a plugin with WPS support for the QGIS Desktop GIS.

1.12 Extensions

PyWPS has extensions to enhance its usability in special uses cases, for example to run Web Processing Services at High Performance Compute (HPC) centers. These extensions are disabled by default. They need a modified configuration and have additional software packages. The extensions are:

- Using batch job schedulers (distributed resource management) at HPC compute centers.
- Using container solutions like Docker in a cloud computing infrastructure.

1.12.1 Job Scheduler Extension

By default PyWPS executes all processes on the same machine as the PyWPS service is running on. Using the PyWPS scheduler extension it becomes possible to delegate the execution of asynchronous processes to a scheduler system like Slurm, Grid Engine and TORQUE. By enabling this extension one can handle the processing workload using an existing scheduler system commonly found at High Performance Compute (HPC) centers.

Note: The PyWPS process implementations are not changed by using the scheduler extension.

To activate this extension you need to edit the `pywps.cfg` configuration file and make the following changes:

```
[processing]
mode = scheduler
```

The scheduler extension uses the DRMAA library to talk to the different scheduler systems. Install the additional Python dependencies using pip:

```
$ pip install -r requirements-processing.txt  # drmaa
```

If you are using the conda package manager you can install the dependencies with:

```
$ conda install drmaa dill
```

The package dill is an enhanced version of the Python pickle module for serializing and de-serializing Python objects.

Warning: In addition you need to install and configure the drmaa modules for your scheduler system on the machine PyWPS is running on. Follow the instructions given in the DRMAA documentation and by your scheduler system installation guide.

Note: See an **example** on how to use this extension with a Slurm batch system in a docker demo.

Note: COWS WPS has a scheduler extension for Sun Grid Engine (SGE).

Interactions of PyWPS with a scheduler system

The PyWPS scheduler extension uses the Python dill library to dump and load the processing job to/from filesystem. The batch script executed on the scheduler system calls the PyWPS `joblauncher` script with the dumped job status and executes the job (no WPS service running on scheduler). The job status is updated on the filesystem. Both the PyWPS service and the `joblauncher` script use the same PyWPS configuration. The scheduler assumes that the PyWPS server has a shared filesystem with the scheduler system so that XML status documents and WPS outputs can be found at the same file location. See the interaction diagram how the communication between PyWPS and the scheduler works.

The following image shows an example of using the scheduler extension with Slurm.

1.12.2 Docker Container Extension

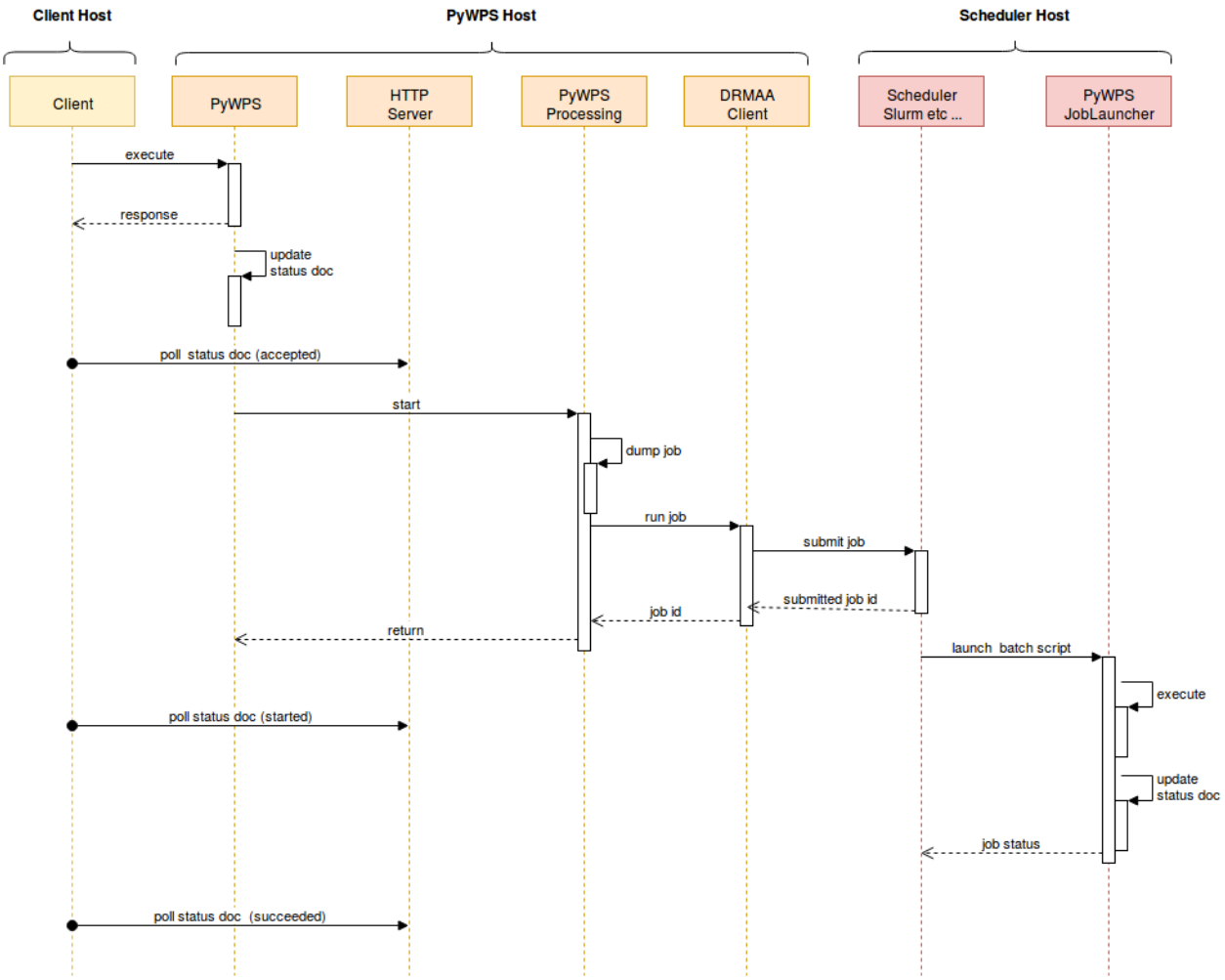

Fig. 1: Interaction diagram for PyWPS scheduler extension.

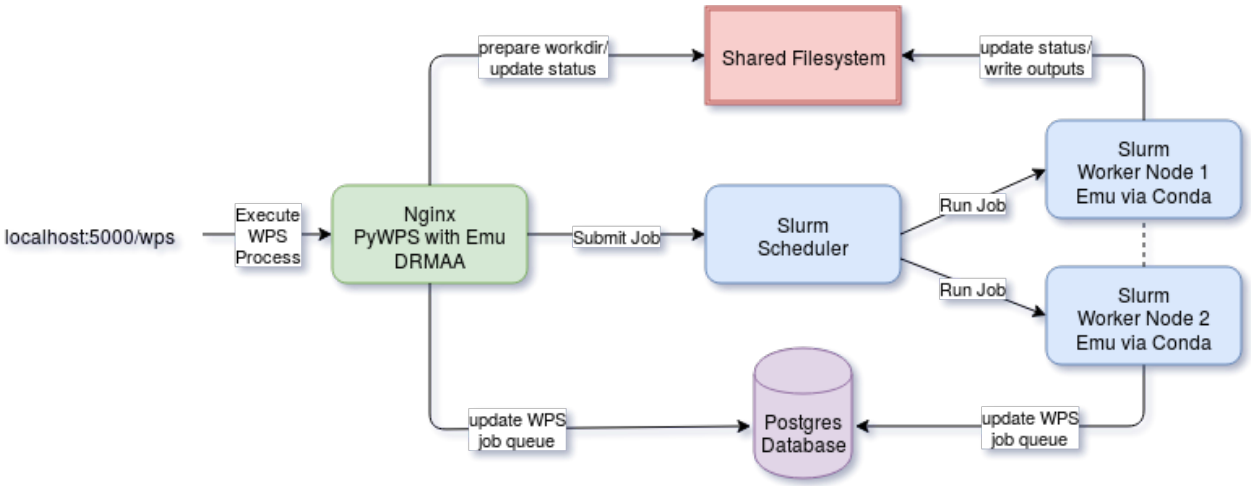

Fig. 2: Example of PyWPS scheduler extension usage with Slurm.

Todo: This extension is on our wish list. In can be used to encapsulate and control the execution of a process. It enhances also the use case of Web Processing Services in a cloud computing infrastructure.

1.13 PyWPS API Doc

1.13.1 Process

class pywps.**Process**(*handler*, *identifier*, *title*, *abstract=''*, *keywords=[]*, *profile=[]*, *metadata=[]*, *inputs=[]*, *outputs=[]*, *version='None'*, *store_supported=False*, *status_supported=False*, *grass_location=None*)

Parameters

- **handler** – A callable that gets invoked for each incoming request. It should accept a single *pywps.app.WPSRequest* argument and return a pywps.app.WPSResponse object.
- **identifier** (*string*) – Name of this process.
- **title** (*string*) – Human readable title of process.
- **abstract** (*string*) – Brief narrative description of the process.
- **keywords** (*list*) – Keywords that characterize a process.
- **inputs** – List of inputs accepted by this process. They should be *LiteralInput* and *ComplexInput* and *BoundingBoxInput* objects.
- **outputs** – List of outputs returned by this process. They should be *LiteralOutput* and *ComplexOutput* and *BoundingBoxOutput* objects.
- **metadata** – List of metadata advertised by this process. They should be pywps.app.Common.Metadata objects.

1.13.2 Inputs and outputs

class pywps.validator.mode.**MODE**

Validation mode enumeration

NONE = 0

SIMPLE = 1

STRICT = 2

VERYSTRICT = 3

Most of the inputs nad outputs are derived from the *IOHandler* class

class pywps.inout.basic.**IOHandler**(*workdir=None*, *mode=0*)

Base IO handling class subclassed by specialized versions: FileHandler, UrlHandler, DataHandler, etc.

If the specialized handling class is not known when the object is created, instantiate the object with IOHandler. The first time the *file*, *url* or *data* attribute is set, the associated subclass will be automatically registered. Once set, the specialized subclass cannot be switched.

Parameters

- **workdir** – working directory, to save temporal file objects in.
- **mode** – MODE validation mode.

file [str] Filename on the local disk.

url [str] Link to an online resource.

stream [FileIO] A readable object.

data [object] A native python object (integer, string, float, etc)

base64 [str] A base 64 encoding of the data.

```
>>> # setting up
>>> import os
>>> from io import RawIOBase
>>> from io import FileIO
>>>
>>> ioh_file = IOHandler(workdir=tmp)
>>> assert isinstance(ioh_file, IOHandler)
>>>
>>> # Create test file input
>>> fileobj = open(os.path.join(tmp, 'myfile.txt'), 'w')
>>> fileobj.write('ASDF ASFADSF ASF ASF ASDF ASFASF')
>>> fileobj.close()
>>>
>>> # testing file object on input
>>> ioh_file.file = fileobj.name
>>> assert isinstance(ioh_file, FileHandler
>>> assert ioh_file.file == fileobj.name
>>> assert isinstance(ioh_file.stream, RawIOBase)
>>> # skipped assert isinstance(ioh_file.memory_object, POSH)
>>>
>>> # testing stream object on input
>>> ioh_stream = IOHandler(workdir=tmp)
>>> assert ioh_stream.workdir == tmp
>>> ioh_stream.stream = FileIO(fileobj.name,'r')
>>> assert isinstance(ioh_stream, StreamHandler)
>>> assert open(ioh_stream.file).read() == ioh_file.stream.read()
>>> assert isinstance(ioh_stream.stream, RawIOBase)
```

LiteralData

class pywps.**LiteralInput**(*identifier*, *title*, *data_type='integer'*, *abstract=''*, *keywords=[]*, *metadata=[]*, *uoms=None*, *min_occurs=1*, *max_occurs=1*, *mode=1*, *allowed_values=<class 'pywps.inout.literaltypes.AnyValue'>*, *default=None*, *default_type=3*)

Parameters

- **identifier** (*str*) – The name of this input.
- **title** (*str*) – Title of the input
- **data_type** (*pywps.inout.literaltypes.LITERAL_DATA_TYPES*) – data type
- **abstract** (*str*) – Input abstract

- **keywords** (*list*) – Keywords that characterize this input.
- **metadata** (*list*) – TODO
- **uoms** (*str*) – units
- **min_occurs** (*int*) – minimum occurence
- **max_occurs** (*int*) – maximum occurence
- **mode** (pywps.validator.mode.MODE) – validation mode (none to strict)
- **allowed_values** (pywps.inout.literaltypes.AnyValue) – or *pywps.inout.literaltypes.AllowedValue* object
- **metadata** – List of metadata advertised by this process. They should be pywps.app.Common.Metadata objects.

class pywps.**LiteralOutput**(*identifier*, *title*, *data_type='string'*, *abstract=''*, *keywords=[]*, *metadata=[]*, *uoms=None*, *mode=1*)

Parameters

- **identifier** – The name of this output.
- **title** (*str*) – Title of the input
- **data_type** (*pywps.inout.literaltypes.LITERAL_DATA_TYPES*) – data type
- **abstract** (*str*) – Input abstract
- **uoms** (*str*) – units
- **mode** (pywps.validator.mode.MODE) – validation mode (none to strict)
- **metadata** – List of metadata advertised by this process. They should be pywps.app.Common.Metadata objects.

class pywps.inout.literaltypes.**AnyValue**

Any value for literal input

class pywps.inout.literaltypes.**AllowedValue**(*allowed_type='value'*, *value=None*, *minval=None*, *maxval=None*, *spacing=None*, *range_closure='closed'*)

Allowed value parameters the values are evaluated in literal validator functions

Parameters

- **allowed_type** (*pywps.validator.allowed_value.ALLOWEDVALUETYPE*) – VALUE or RANGE
- **value** – single value
- **minval** – minimal value in case of Range
- **maxval** – maximal value in case of Range
- **spacing** – spacing in case of Range
- **range_closure** (*pywps.input.literaltypes.RANGECLOSURETYPE*) –

pywps.inout.literaltypes.**LITERAL_DATA_TYPES = ('float', 'boolean', 'integer', 'string', 'po**

tuple() -> empty tuple tuple(iterable) -> tuple initialized from iterable's items

If the argument is a tuple, the return value is the same object.

ComplexData

class pywps.**ComplexInput**(*identifier*, *title*, *supported_formats*, *data_format=None*, *abstract=''*, *keywords=[]*, *metadata=[]*, *min_occurs=1*, *max_occurs=1*, *mode=0*, *default=None*, *default_type=3*)

Complex data input

Parameters

- **identifier** (*str*) – The name of this input.
- **title** (*str*) – Title of the input
- **supported_formats** (*pywps.inout.formats.Format*) – List of supported formats
- **data_format** (*pywps.inout.formats.Format*) – default data format
- **abstract** (*str*) – Input abstract
- **keywords** (*list*) – Keywords that characterize this input.
- **metadata** (*list*) – TODO
- **min_occurs** (*int*) – minimum occurrence
- **max_occurs** (*int*) – maximum occurrence
- **mode** (pywps.validator.mode.MODE) – validation mode (none to strict)

class pywps.**ComplexOutput**(*identifier*, *title*, *supported_formats=None*, *abstract=''*, *keywords=[]*, *metadata=None*, *as_reference=False*, *mode=0*)

Parameters

- **identifier** – The name of this output.
- **title** – Readable form of the output name.
- **supported_formats** (*(pywps.inout.formats.Format,)*) – List of supported formats. The first format in the list will be used as the default.
- **abstract** (*str*) – Description of the output
- **mode** (pywps.validator.mode.MODE) – validation mode (none to strict)
- **metadata** – List of metadata advertised by this process. They should be pywps.app.Common.Metadata objects.

class pywps.**Format**(*mime_type*, *schema=None*, *encoding=None*, *validate=<function emptyvalidator>*, *mode=1*, *extension=None*)

Input/output format specification

Predefined Formats are stored in *pywps.inout.formats.FORMATS*

Parameters

- **mime_type** (*str*) – mimetype definition
- **schema** (*str*) – xml schema definition
- **encoding** (*str*) – base64 or not
- **validate** (*function*) – function, which will perform validation. e.g.
- **mode** (*number*) – validation mode
- **extension** (*str*) – file extension

pywps.inout.formats.**FORMATS**
FORMATS(GEOJSON, JSON, SHP, GML, KML, KMZ, GEOTIFF, WCS, WCS100, WCS110, WCS20, WFS, WFS100, WFS110, WFS20, WMS, WMS130, WMS110, WMS100, TEXT, DODS, NETCDF, LAZ, LAS) List of out of the box supported formats. User can add custom formats to the array.

pywps.validator.complexvalidator.**validategml**(*data_input*, *mode*)
GML validation function

Parameters

- **data_input** – ComplexInput
- **mode** (pywps.validator.mode.MODE) –

This function validates GML input based on given validation mode. Following happens, if *mode* parameter is given:

MODE.NONE it will return always *True*

MODE.SIMPLE the mimetype will be checked

MODE.STRICT GDAL/OGR is used for getting the proper format.

MODE.VERYSTRICT the lxml.etree is used along with given input *schema* and the GML file is properly validated against given schema.

BoundingBoxData

class pywps.**BoundingBoxInput**(*identifier*, *title*, *crss*, *abstract=''*, *keywords=[]*, *dimensions=2*, *metadata=[]*, *min_occurs=1*, *max_occurs=1*, *mode=0*, *default=None*, *default_type=3*)

Parameters

- **identifier** (*string*) – The name of this input.
- **title** (*string*) – Human readable title
- **abstract** (*string*) – Longer text description
- **crss** – List of supported coordinate reference system (e.g. ['EPSG:4326'])
- **keywords** (*list*) – Keywords that characterize this input.
- **dimensions** (*int*) – 2 or 3
- **metadata** (*list*) – TODO
- **min_occurs** (*int*) – how many times this input occurs
- **max_occurs** (*int*) – how many times this input occurs
- **metadata** – List of metadata advertised by this process. They should be pywps.app.Common.Metadata objects.

class pywps.**BoundingBoxOutput**(*identifier*, *title*, *crss*, *abstract=''*, *keywords=[]*, *dimensions=2*, *metadata=[]*, *min_occurs='1'*, *max_occurs='1'*, *as_reference=False*, *mode=0*)

Parameters

- **identifier** – The name of this input.
- **title** (*str*) – Title of the input
- **abstract** (*str*) – Input abstract

- **crss** – List of supported coordinate reference system (e.g. ['EPSG:4326'])
- **dimensions** (*int*) – number of dimensions (2 or 3)
- **min_occurs** (*int*) – minimum occurence
- **max_occurs** (*int*) – maximum occurence
- **mode** (pywps.validator.mode.MODE) – validation mode (none to strict)
- **metadata** – List of metadata advertised by this process. They should be pywps.app.Common.Metadata objects.

Request and response objects

pywps.response.status.**WPS_STATUS**
WPSStatus(UNKNOWN, ACCEPTED, STARTED, PAUSED, SUCCEEDED, FAILED) Process status information

class pywps.app.**WPSRequest**(*http_request=None*)

operation
Type of operation requested by the client. Can be *getcapabilities*, *describeprocess* or *execute*.

http_request
Original Werkzeug HTTPRequest object.

inputs
A MultiDict object containing input values sent by the client.

check_accepted_versions(*acceptedversions*)

Parameters **acceptedversions** – string

check_and_set_language(*language*)
set this.language

check_and_set_version(*version*)
set this.version

json
Return JSON encoded representation of the request

class pywps.response.**WPSResponse**(*wps_request*, *uuid=None*, *version='1.0.0'*)

status
Information about currently running process status pywps.response.status.STATUS

Processing

pywps.processing.**Process**(*process*, *wps_request*, *wps_response*)
Factory method (looking like a class) to return the configured processing class.

Returns instance of *pywps.processing.Processing*

class pywps.processing.**Processing**(*process*, *wps_request*, *wps_response*)
Processing is an interface for running jobs.

class pywps.processing.**Job**(*process*, *wps_request*, *wps_response*)
Job represents a processing job.

Refer *Exceptions* for their description.

1.14 Developers Guide

If you identify a bug in the PyWPS code base and want to fix it, if you would like to add further functionality, or if you wish to expand the documentation, you are welcomed to contribute such changes. However, contributions to the code base must follow an orderly process, described below. This facilitates both the work on your contribution as its review.

1.14.1 0. GitHub account

The PyWPS source code is hosted at GitHub, therefore you need an account to contribute. If you do not have one, you can follow these instructions.

1.14.2 1. Open a new issue

The first action to take is to clearly identify the goal of your contribution. Be it a bug fix, a new feature or documentation, a clear record must be left for future tracking. This is made by opening an issue at the GitHub issue tracker. In this new issue you should identify not only the subject or goal, but also a draft of the changes you expect to achieve. For example:

> **Title**: Process class must be magic
>
> **Description**: The Process class must start performing some magics. Give it a magic wand.

1.14.3 2. Fork and clone the PyWPS repository

When you start modifying the code, there is always the possibility for something to go wrong, rendering PyWPS unusable. The first action to avoid such a situation is to create a development sand box. In GitHub this can easily be made by creating a fork of the main PyWPS repository. Access the PyWPS code repository and click the *Fork* button. This action creates a copy of the repository associated with your GitHub user. For more details please read the forking guide.

Now you can clone this forked repository into your development environment, issuing a command like:

```
git clone https://github.com/<github-user>/PyWPS.git pywps
```

Where you should replace <*github-user*> with your GitHub user name.

Before you start coding ensure you are working on the *master* branch:

> git checkout master

For the moment, this is the main development branch in the PyWPS project.

You can now start programming your new feature, or fixing that bug you found. Keep in mind that PyWPS depends on a few libraries, refer to the *Installation* section to make sure you have all of them installed.

1.14.4 3. Commit and pull request

If your modification to the code is relatively small and can be included in a single *commit* then all you need to is reference the issue in the **commit** message, e.g.:

```
git commit -m "Fixes #107"
```

Where *107* is the number of the issue you opened initially in the PyWPS issue tracker. Please refer to the guide on closing issues with commits messages. Then you push the changes to your forked repository, issuing a command like:

```
git push origin master
```

Once again, make sure your commit(s) are pushed to the *master* branch.

Finally you can create a pull request. This is a formal request to merge your contribution with the code base; it is fully managed by GitHub and greatly facilitates the review process. You do so by accessing the repository associated with your user and clicking the *New pull request* button. Make sure your contribution is not creating conflicts and click *Create pull request*. If needed, there is also a guide on pull requests.

If you contribution is more substantial, and composed of multiple commits, then you must identify the issue it closes in the pull request itself. Check out this guide for the details.

The members of the PyWPS PSC are then notified of your pull request. They review your contribution and hopefully accept merging it to the code base.

1.14.5 4. Updating local repository

Later on, if you wish to make further contributions, you must make sure to be working with the very latest version of the code base. You can add another *remote* reference in your local repository pointing to the main PyWPS repository:

```
git remote add upstream https://github.com/geopython/PyWPS
```

You can use the *fetch* command to update your local repository metadata:

```
git fetch upstream
```

Then you use a *pull* command to merge the latest *commits* into your local repository:

```
git pull upstream master
```

1.14.6 5. Help and discussion

If you have any doubts or questions about this contribution process or about the code please use the PyWPS mailing list or the PyWPS Gitter . This is also the right place to propose and discuss the changes you intend to introduce.

1.15 Exceptions

PyWPS will throw exceptions based on the error occurred. The exceptions will point out what is missing or what went wrong as accurately as possible.

Here is the list of Exceptions and HTTP error codes associated with them:

class pywps.exceptions.**NoApplicableCode**(*description*, *locator=''*, *code=400*)

No applicable code exception implementation

also

Base exception class

class pywps.exceptions.**InvalidParameterValue**(*description*, *locator=''*, *code=400*)
Invalid parameter value exception implementation

class pywps.exceptions.**MissingParameterValue**(*description*, *locator=''*, *code=400*)
Missing parameter value exception implementation

class pywps.exceptions.**FileSizeExceeded**(*description*, *locator=''*, *code=400*)
File size exceeded exception implementation

class pywps.exceptions.**VersionNegotiationFailed**(*description*, *locator=''*, *code=400*)
Version negotiation exception implementation

class pywps.exceptions.**OperationNotSupported**(*description*, *locator=''*, *code=400*)
Operation not supported exception implementation

class pywps.exceptions.**StorageNotSupported**(*description*, *locator=''*, *code=400*)
Storage not supported exception implementation

class pywps.exceptions.**NotEnoughStorage**(*description*, *locator=''*, *code=400*)
Storage not supported exception implementation

1.16 Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

V

W