PyWPS

Release 4.0.0rc2

Apr 17, 2017

Contents

Note: Please be awere that PyWPS-4 is still in pre-release state, there is no stable release yet.

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.

PyWPS has a bicycle in it's logo, because:

- It's simple to maintain
- It's fast to drive
- It can carry a lot
- It's easy to hack

Mount your bike and setup & configure your PyWPS instance!

Todo

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

CHAPTER 1

Contents:

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.

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

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 list of in- and outputs.

Data in- and outputs

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

All data types do need to have following attributes:

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

Passing data to process instance

There are 3 typical ways, how 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.

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).

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.

Request encoding, HTTP GET and POST

The request can be encoded either using Key-value-pairs (KVP) or as the XML-formatted document.

Key-value-pair 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 is document sent via HTTP POST request method. The XML document can be more rich, having more parameters, better to be parsed in complex structures. Client can also encode whole 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.

PyWPS

Todo

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

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.

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.

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.

Installation

Note: PyWPS-4 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.

Dependencies and requirements

PyWPS-4 runs on Python 2.7, 3.3 or newer. PyWPS is currently tested and developed on Linux (mostly Ubuntu). In the documentation we take this distribution as reference.

Prior to installing PyWPS-4, 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 install git python-gdal
```

Download and install

Using PIP The easiest way to install PyWPS-4 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
```

Manual installation In alternative PyWPS-4 can be installed manually. It requires the cloning of the source code from the repository and then the usage of the *setup.py* script. An example again for Debian based systems (note the usage of *sudo* for install):

```
$ git clone https://github.com/geopython/pywps.git pywps-4

$ cd pywps-4/
```

Then install the package dependencies using pip:

```
$ pip install -r requirements.txt
$ pip install -r requirements-dev.txt  # for developer tasks
```

To install PyWPS system-wide run:

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

The demo service and its sample processes

To use PyWPS-4 the user must code processes and publish them through a service. A demo service is available that makes up a good starting point for first time users. This launches a very simple built-in server (relying on flask), which is good enough for testing but probably not appropriate for production. It can be cloned directly into the user area:

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

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

Full more details please consult the *Processes* section. The *demo* 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*).

Configuration

PyWPS is configured using a configuration file. The file uses the ConfigParser format.

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 3 sections:

- *metadata:main* for the server metadata inputs
- *server* for server configuration
- *loggging* 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 *demo* service as described in *The demo service and its sample processes* section.

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

[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`

[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

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

Note: *outputpath* and *outputurl* must corespond. *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*

[logging]

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

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.

[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/
logfile=
loglevel=INFO
logdatabase=
workdir=

[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
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

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

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-Demo project.

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 `Process`. In the PyWPS-Demo 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.app.WPSResponse` as inputs.

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:

```
from pywps import Process, LiteralInput, ComplexOutput, ComplexInput, Format
from pywps.validator.mode import MODE
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
)

inpt_size = LiteralInput('size', 'Buffer size', data_type='float')
```

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]
outputs = [out_output]
```

Next we define the *handler* method. In it, *geospatial analysis may happen*. The method gets a `pywps.app.WPSRequest` and a `pywps.app.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.app.WPSResponse` object.

```
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

    # make buffer for each feature
    while index < count:

        response.update_status('Buffering feature %s' % index, float(index)/count)

        # get the geometry
        input_feature = input_layer.GetNextFeature()
        input_geometry = input_feature.GetGeometryRef()

        # make the buffer
        buffer_geometry = input_geometry.Buffer(
```

```
                float(size)
        )

        # create output feature to the file
        output_feature = ogr.Feature(feature_def=output_layer.GetLayerDefn())
        output_feature.SetGeometryDirectly(buffer_geometry)
        output_layer.CreateFeature(output_feature)
        output_feature.Destroy()
        index += 1

    # set output format
    response.outputs['output'].output_format = FORMATS.GML

    # set output data as file name
    response.outputs['output'].file = layer_name

    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`:

```
class DemoBuffer(Process):
    def __init__(self):

        super(DemoBuffer, self).__init__(
            _handler,
            identifier='demobuffer',
            version='1.0.0',
            title='Buffer',
            abstract='This process demonstrates, how to create any process in PyWPS
↪environment',
            inputs=inputs,
            outputs=outputs,
            store_supported=True,
            status_supported=True
        )
```

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

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 3 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.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.

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.

Example:

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

Because there could be multiple input values with the same identifier, the inputs are accessed with an index. For *LiteralInput*, the value is a string. For *ComplexInput*, the value is an open file object, with a *mime_type* attribute:

```
@staticmethod
def handler(request, response):
    layer_file = request.inputs['layer'][0].file
    mime_type = layer_file.mime_type
    bytes = layer_file.read()
    msg = ("You gave me a file of type %s and size %d"
           % (mime_type, len(bytes)))
    response.outputs['output'].data = msg
    return response
```

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.

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-4 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.

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 *demo* 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)

...
```

Running the dev server

The *The demo 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='0.0.0.0', processes=processes, config_file=config_file)

s.run()
...
```

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

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, Apache 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.

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

Creating a PyWPS *WSGI* instance

An example WSGI script is distributed along with PyWPS-Demo 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*).

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-4
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
```

Deployment on nginx

Note: We are currently missing documentation about *nginx*. Please help documenting the deployment of PyWPS to nginx.

You should be able to deploy PyWPS on nginx as a standard WSGI application. The best documentation is probably to be found at Readthedocs.

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"?>
<!-- PyWPS 4.0.0-alpha2 -->
<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.

Migrating from PyWP 3.x to 4.x

TODO

PyWPS and external tools

GRASS GIS

Todo

How to setup and get GRASS GIS up and running with PyWPS and example process

PyWPS API Doc

Process

Inputs and outputs

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

LiteralData

ComplexData

BoundingBoxData

Request and response objects

Refer *Exceptions* for their description.

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.

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. 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.

2. Fork and clone the PyWPS repository

When you start modifying to 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.

You can finally 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.

3. Commit and pull request

If your modification to 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
```

Finally you an create a pull request. This it 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 if your pull request. They review your contribution and hopefully accept merging it to the code base.

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
```

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

```
git fetch upstream
```

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

```
git pull upstream master
```

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.

Exceptions

PyWPS-4 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:

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

E

P