Birdhouse Documentation

Release 0.4

Birdhouse

Aug 21, 2017

Contents

Birdhouse is a collection of *Web Processing Service* (WPS) related Python components to support data processing in the climate science community. Many of the *OGC*/*WPS* related software comes from the *GeoPython* project, like PyWPS and OWSLib.

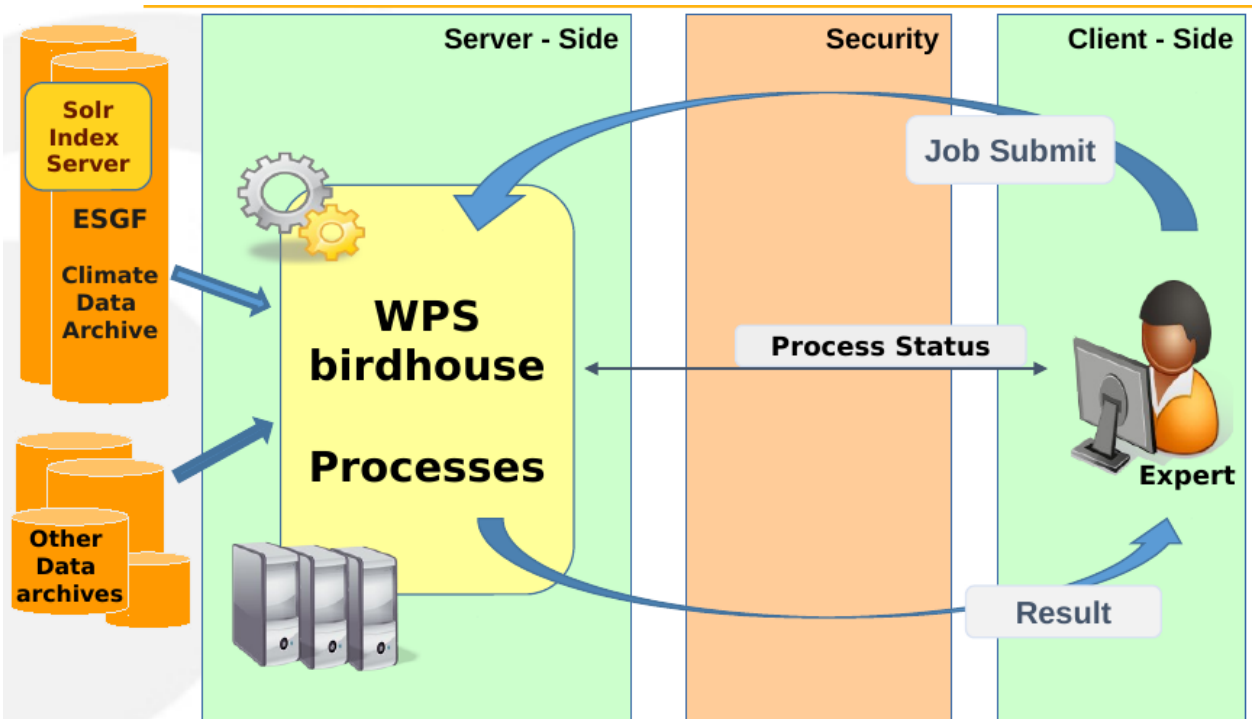
CHAPTER 1

Getting started

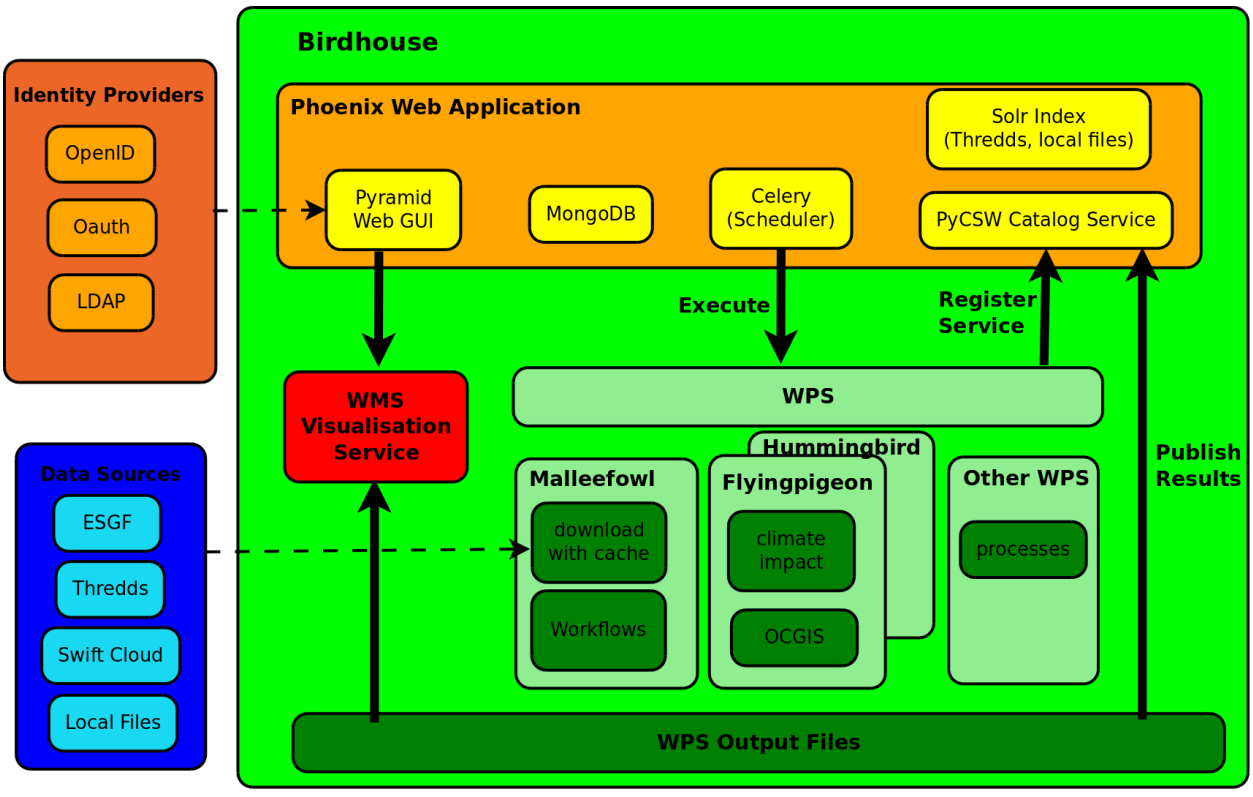
Overview

WPS Use Case

A user runs a WPS processes remotely on a machine with direct access to climate data archives.



Birdhouse with WPS components



ESGF is currently the main climate data resource (but more resources are possible). ESGF Solr-index is used to find ESGF data. The ESGF identity provider with OpenIDs and X509 certificate is used for authentication.

There are several WPS services. Malleefowl is the main one for the Phoenix client. Malleefowl is used to search, download (with caching) ESGF data and to retrieve certificates. Malleefowl has also a workflow engine (dispel4py) to chain WPS processes.

The results of the WPS processes are stored on the file system and are accessible via URL (with a token id).

Results can be shown on a Map using a Web Mapping Service (ncWMS, adagucserver).

The PyCSW Catalog Service is used to register WPS services and also to publish WPS outputs. Published results in the PyCSW can also used as input source for processes again.

WPS serivces can be accessed through web-applications like Phoenix or from scripts.

Birdhouse Architecture

See the Birdhouse Architecture Talk

Projects

Birdhouse is the home of Web Processing Services used in climate science and components to support them (the birds):

WPS client side:

- Phoenix: a web-based WPS client with ESGF data access
- Birdy: a WPS command line tool

WPS supporting services and libraries:

- Twitcher: an OWS Security Proxy
- Malleefowl: access to climate data (ESGF, ...) as a service

WPS services and libraries with algorithms used in climate science analysis:

- Flyingpigeon: services for the climate impact community
- Hummingbird: provides cdo and compliance-checker as a service
- Dodrio: WPS for KIT
- Emu: some example WPS processes for demo

You can find the source code of all birdhouse components on GitHub. *Conda* packages for birdhouse are available on the

birdhouse channel on Binstar.

Docker images with birdhouse components are on Docker Hub

Installation

- *Using birdhouse with Docker*

Birdhouse consists of several components like Malleefowl and Emu. Each of them can be installed individually. The installation is done using the Python-based build system *Buildout*. Most of the dependencies are maintained in the *Anaconda Python distribution*. For convenience, each birdhouse component has a Makefile to ease the installation so you don't need to know how to call the Buildout build tool.

Requirements

Birdhouse uses *Anaconda Python distribution* for most of the dependencies. If Anaconda is not already installed, it will be installed during the installation process. Anaconda has packages for Linux, MacOSX and Windows. But not all packages used by birdhouse are already available in the default package channel of Anaconda. The missing packages are supplied by birdhouse on *Binstar*. But we currently maintain only packages for Linux 64-bit and partly for MacOSX.

So the short answer to the requirements is: **you need a Linux 64-bit installation**.

Birdhouse is currently used on Ubuntu 14.04 and CentOS 6.x. It should also work on Debian, LinuxMint and Fedora.

Birdhouse also installs a few system packages using *apt-get* on Debian based distributions and *yum* on RedHat/CentOS based distributions. For this you need a user account with *sudo* permissions. Installing system packages can be done in a separate step. So your installation user does not need any special permissions. All installed files will go into a birdhouse Anaconda environment in the home folder of the installation user.

Installing from source

The installation of birdhouse components from source is done with some few commands. Here is an example for the Emu WPS service:

```
$ git clone https://github.com/bird-house/emu.git
$ cd emu
$ make clean install
$ make start
$ firefox http://localhost:8094/wps
```

All the birdhouse components follow the same installation pattern. If you want to see all the options of the *Makefile* then type:

```
$ make help
```

You will find more information about these options in the Makefile documentation.

Read the documention of each birdhouse component for the details of the installation and how to configure the components. The birdhouse bootstrap documentation gives some examples of the different ways of making the installation.

On the WPS client side we have:

- Phoenix: a Pyramid web application.
- Birdy: a simple WPS command line tool.

On the WPS server side we have:

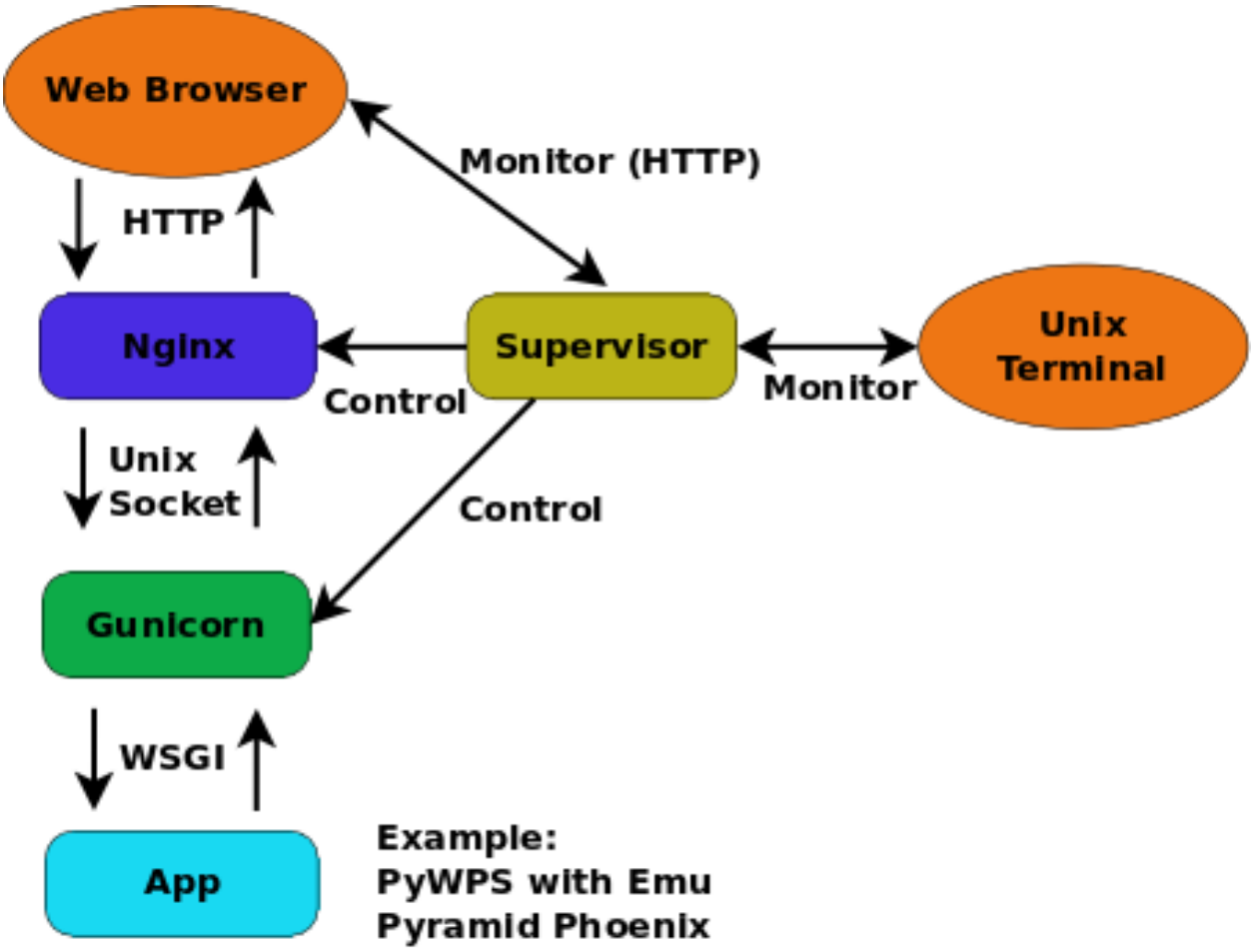
- Malleefowl: provides base WPS services to access data.
- Flyingpigeon: provides WPS services for the climate impact community.
- Hummingbird: provides WPS services for CDO and climate metadata checks.

- Emu: just some WPS processes for testing.

Nginx, gunicorn and supervisor

Birdhouse sets up a *PyWPS* server (and also the Phoenix web application) using *Buildout*. We use the *Gunicorn* HTTP application server (similar to Tomcat for Java servlet applications) to run these web applications with the *WSGI* interface. In front of the Gunicorn application server, we use the *Nginx* HTTP server (similar to the Apache web server). All these web services are started/stopped and monitored by a *Supervisor* service.

See the following image for how this looks like:



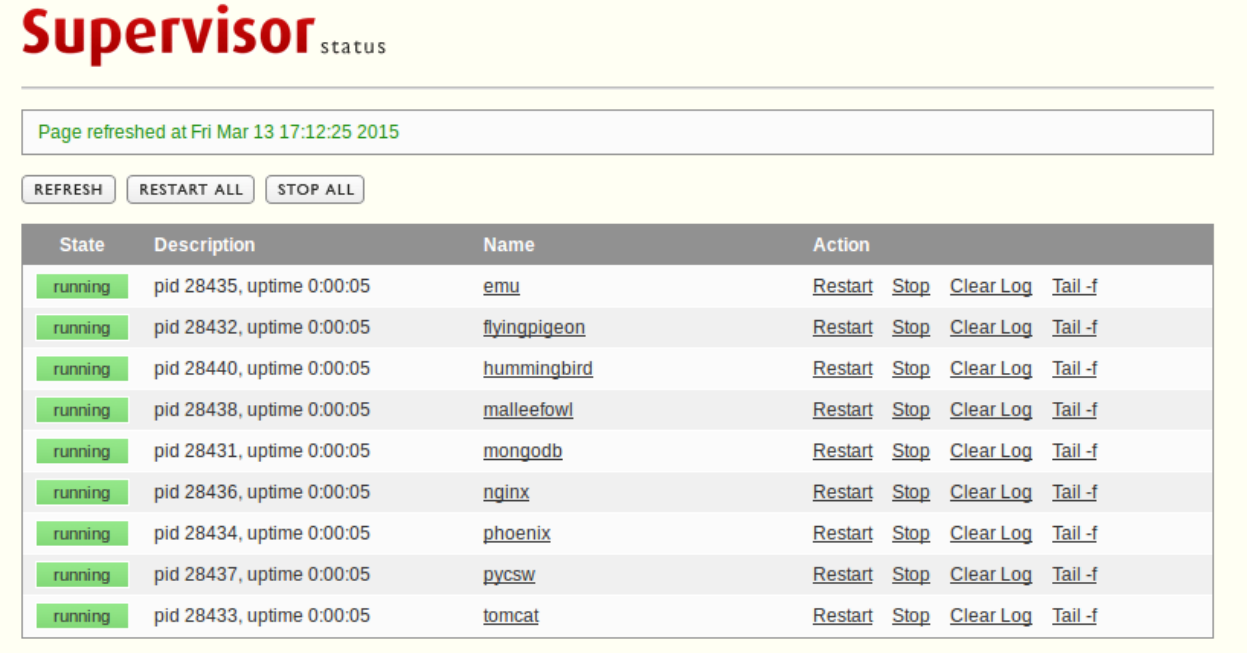
When installing a birdhouse WPS service, you don't need to care about this setup. This is all done by Buildout and using some extensions provided by birdhouse.

The Makefile of a birdhouse application has convenience targets to start/stop a WPS service controlled by the Supervisor and to check the status:

```
$ make start     # start wps service
$ make stop      # stop wps service
$ make status    # show status of wps service
Supervisor status ...
/home/pingu/.conda/envs/birdhouse/bin/supervisorctl status
emu                              RUNNING   pid 25698, uptime 0:00:02
malleefowl                       RUNNING   pid 25702, uptime 0:00:02
mongodb                          RUNNING   pid 25691, uptime 0:00:02
```

```
nginx                            RUNNING   pid 25699, uptime 0:00:02
phoenix                          RUNNING   pid 25694, uptime 0:00:02
pycsw                            RUNNING   pid 25700, uptime 0:00:02
tomcat                           RUNNING   pid 25693, uptime 0:00:02
```

You can also use the Supervisor monitor web service which by default is available on port http://localhost:9001/. The Supervisor monitor app looks like in the following screenshot.



Using birdhouse with Docker

An alternative way to install and deploy birdhouse Web Processing Services is by using *Docker*. The birdhouse WPS servers are available as a Docker image on Docker Hub. See an example on how to use them with the Emu WPS Docker image.

Tutorials

What is WPS?

Geographic Information Processing for the Web *The Web Processing Service (WPS) offers a simple web-based method of finding, accessing, and using all kinds of calculations and models* ([1])

Web Processing Service offers you the following:

- **simple web service** to enable remote call of calcualtions.
- WPS services are **self-describing**.
- WPS is an **interface description**. Several implementations exist.
- WPS is part of the *OGC* **open standards** family: *wms*, wfc, wcs, sos, csw, ...

[1] What is WPS? - http://geoprocessing.info/wpsdoc/Concepts#what

- can be called with **simple HTTP requests** with key/value pairs or
- can be called with HTTP post-requests with XML documents.
- a **lightweight specification**. Comparable to *XML-RPC* ... but XML-RPC is not self-describing.
- can be registered in *Catalog Services*.

You will find further information in the appendix: *WPS Documentation*.

In the following we show an example with a *Word Counter* function which is enabled as a web-service using WPS.

Defining a *Word Counter* function

In the following example we will use the *Word Counter* function:

```
def count_words(file):
    """Calculates number of words in text document.
    Returns JSON document with occurrences of each word.
    """
    return json_doc
```

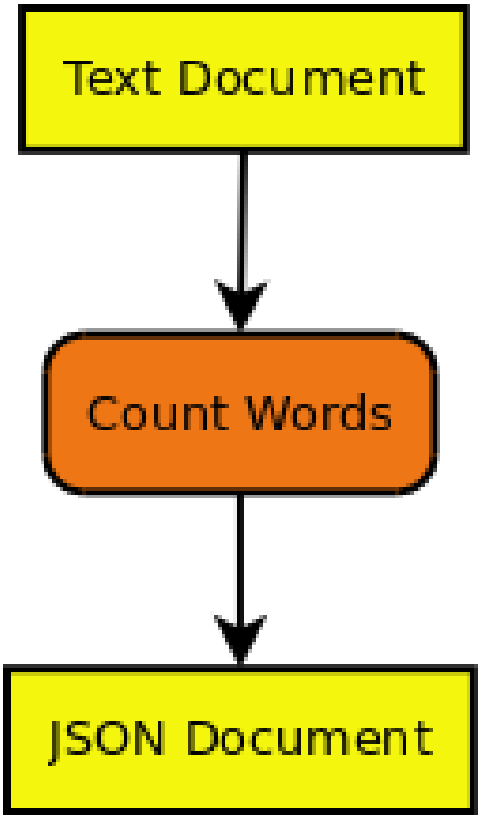
This Python function has the following parts:

- a *name* or *identifier*: count_words
- a *description*: Calculates number of words ...
- *input parameters*: file (mime type *text/plain*)
- *output parameters*: json_doc (mime type *application/json*)

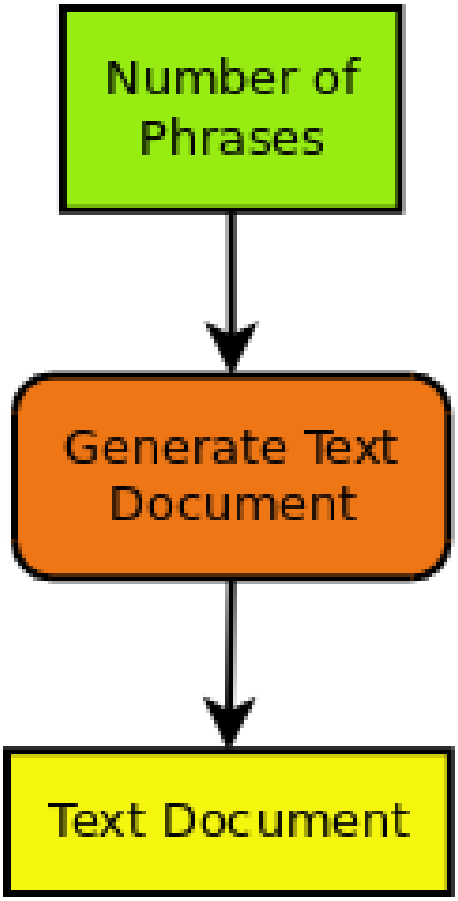
Now, we would like to run this function remotely using a simple web-service. To get this web-service we can use *WPS*. The function parts (name, parameters) are all we need to know to define a WPS process.

WPS definition of *Word Counter*

To add a new proccess you need to define the input and output parameters. For the *Word Counter* process this looks like the following.



Here is another example for a *Text Generator* process. We will use it later for chaining processes.



There are two types of input/output parameters:

- Literal Parameters (green): these are simple data types like integer, boolean, string, ...
- Complex Parameters (yellow): these are documents with a mime-type (xml, cvs, jpg, netcdf, ...) provided as URL or directly.

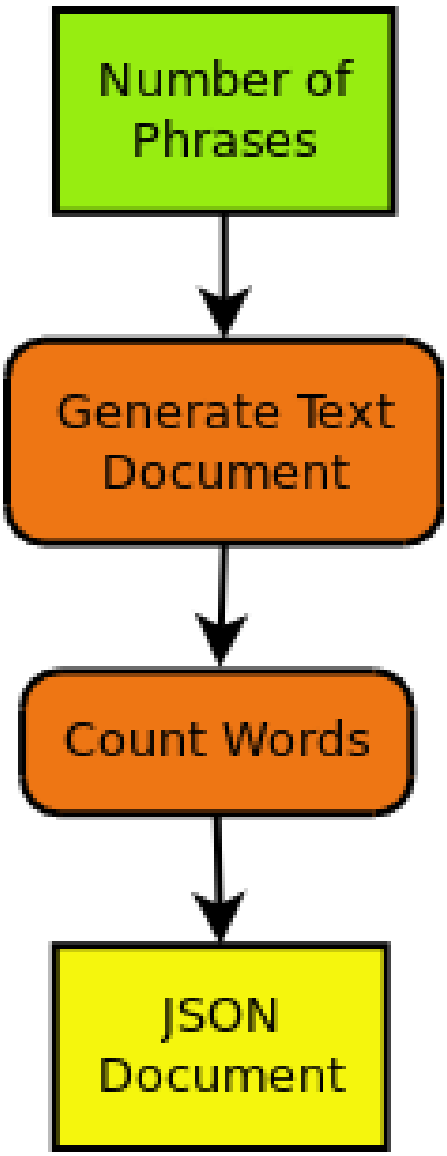
An input/output parameter has:

- a name or *identifier*
- a descriptive *title*
- an *abstract* giving a description of the parameter
- *multiplicity* ... how often can this parameter occur: optional, once, many ...
- in case of literal parameters a list of *allowed values*.

For more details see the following WPS Tutorial.

Chaining WPS processes

If you know the input/output parameters of processes you can chain processes. For example we will chain a *Text Generator* process to our *Word Counter* process.



The *Text document* output of the *Text Generator* process becomes the input of *Word Counter* process.

You can chain process manually by calling them one after the other. The WPS specification allows you to also chain process with a single WPS request. To get even more flexibility (using if-clauses, loops, monitoring ...) you can also use a *workflow engine* (*Taverna*, *VisTrails*, *Dispel4py*, ...).

You will find more details about chaining in the GeoProcessing document and the GeoServer Tutorial.

WPS process implementation with PyWPS

There are several WPS implementations available (*GeoServer*, *COWS*, ...). In birdhouse, we use the Python implementation *PyWPS*. In *PyWPS* the *Word Counter* process could look like the following:

```
from pywps.Process import WPSProcess
class Process(WPSProcess):

  def __init__(self):
    ##
```

```
    # Process initialization
    WPSProcess.__init__(self,
        identifier = "wordcount",
        title="Word Counter",
        abstract="""Counts words in text document.""",
        )

    ##
    # Adding process inputs

    self.text = self.addComplexInput(identifier="text",
              title="Text Document",
              formats = [{'mimeType':'text/plain'}])


    ##
    # Adding process outputs

    self.output = self.addComplexOutput(identifier = "output",
            title="Word count result")

    ##
    # Execution part of the process
    def execute(self):

        # count words and save result
        self.output.setValue( count_words( self.text.getValue() )  )

        return
```

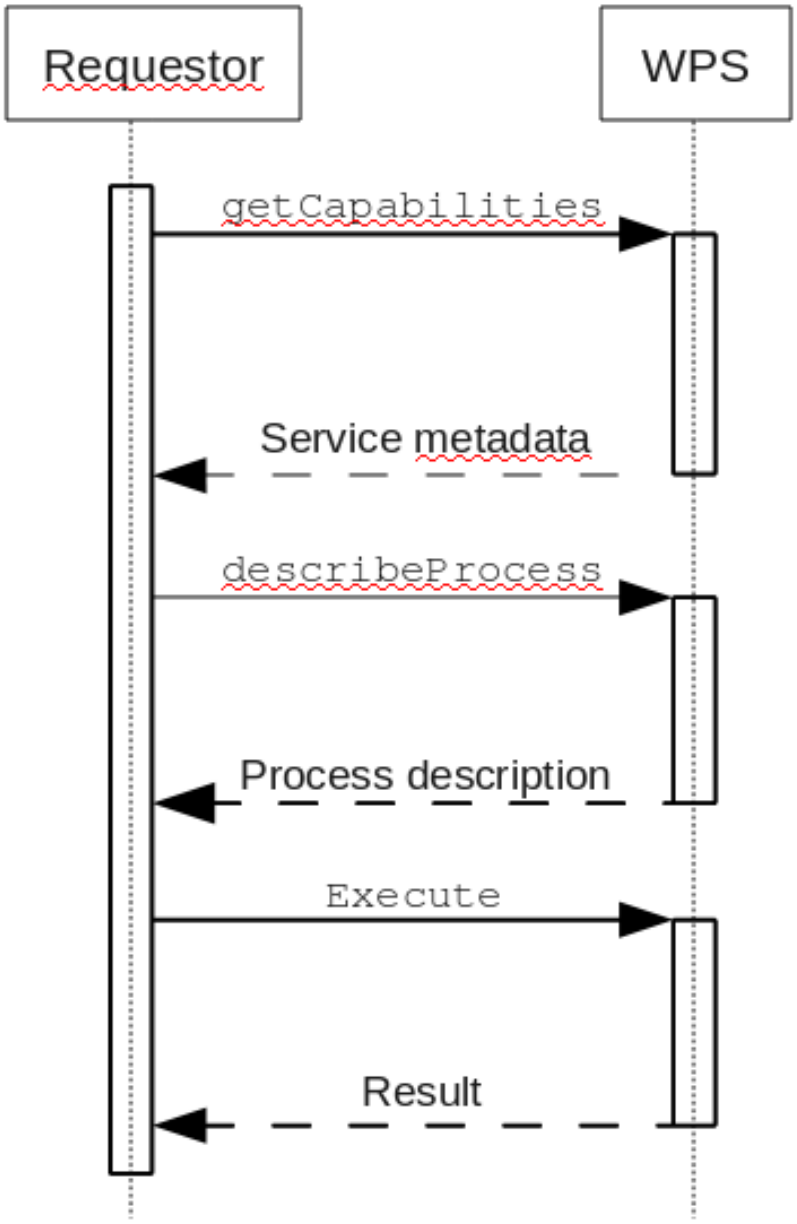
You can see the definition of the input and output parameters and the `execute()` method where the real `count_words()` function is called. You will find more details about implementing a WPS process in the PyWPS Tutorial.

Using WPS

A WPS service has three operations:

- *GetCapabilities*: which processes are available
- *DescribeProcess*: what are the input and output parameters of a specific process
- *Execute*: run a process with parameters.

The following diagram shows these operations:



To call these process one can use simple HTTP request with key/value pairs:

- *GetCapabilites* request:

```
http://localhost:8094/wps?&request=GetCapabilities&service=WPS&version=1.0.0
```

- *DescribeProcess* request for *wordcount* process:

```
http://localhost:8094/wps?&request=DescribeProcess&service=WPS&version=1.0.0&
↪identifier=wordcount
```

- *Exceute* request:

```
http://localhost:8094/wps?request=Execute&service=WPS&version=1.0.0&
↪identifier=wordcount
                        &DataInputs=text=http://birdhouse.readthedocs.org/en/
↪latest/index.html
```

A process can be run *synchronously* or *asynchronously*:

- sync: You make a HTTP request and you need to wait until the request returns with a response (or timeout). This is only useful for short-running processes.
- async: You make a HTTP request and you get immediately a response document. This document gives you a link to a status document which you need to poll until the process has finished.

Processes can be run with simple HTTP get-requests (as shown above) and also with HTTP post-requests. In the later case XML documents are exchanged with the communication details (process, parameters, ...).

For more details see the following WPS Tutorial.

There are also some IPython notebooks which show the usage of WPS.

Calling *Word Counter* with Birdy

Now, we are using Birdy wps command line client to access the *wordcount* process.

Which proccess are available (*GetCapabilities*):

```
$ birdy -h
usage: birdy [-h] <command> [<args>]

optional arguments:
  -h, --help            show this help message and exit

command:
  List of available commands (wps processes)

  {chomsky,helloworld,inout,ultimatequestionprocess,wordcount}
                        Run "birdy <command> -h" to get additional help.
```

What input and output parameters does *wordcount* have (*DescribeProcess*):

```
$ birdy wordcount -h
usage: birdy wordcount [-h] --text [TEXT] [--output [{output} [{output} ...]]]

optional arguments:
  -h, --help            show this help message and exit
  --text [TEXT]         Text document: URL of text document, mime
                        types=text/plain
  --output [{output} [{output} ...]]
                        Output: output=Word count result, mime
                        types=text/plain (default: all outputs)
```

Run *wordcount* with a text document (*Execute*):

```
$ birdy wordcount --text http://birdhouse.readthedocs.org/en/latest/index.html
Execution status: ProcessAccepted
Execution status: ProcessSucceeded
Output:
output=http://localhost:8090/wpsoutputs/emu/output-37445d08-cf0f-11e4-ab7e-
↪68f72837e1b4.txt
```

Using Let's encrypt to generate a certificte

One can use the Let's Encrypt service to generate automatically a valid x509 certificate for web-services.

Debian/Ubuntu

Instructions on: https://certbot.eff.org/#ubuntutyakkety-nginx

Enable certbot ubuntu repo:

```
$ sudo apt-get update
$ sudo apt-get install software-properties-common
$ sudo add-apt-repository ppa:certbot/certbot
$ sudo apt-get update
```

Install certbot for nginx:

```
$ sudo apt-get install python-certbot-nginx
```

Links

- https://letsencrypt.org/

External Tutorials

- PyWPS 4.0.0 Slides
- PyWPS Tutorial
- PyWPS on OSGeo Live
- PyWPS Course with OpenLayers
- Emu Example with Docker
- Phoenix Tutorial
- Flyingpigeon Tutorial
- Example with Birdy WPS command line tool
- Conda for Data Science
- Reaching Deployment Nirvana using Buildout

IPython Notebooks

IPython Notebooks with examples on howto use a Web Processing Service. You need a running Emu WPS service with some test process to run the notebook examples:

- Tutorial on nbviewer:
- Tutorial Notebooks on GitHub
- Flyingpigeon Notebooks on GitHub

Administrator Guide

Files and Folders

Birdhouse is a framework with several compartments. They can be installed according to the specefic needs of the user. Here is a short overview in order of the most important files and folders:

Environment

Three folder locations have to be pointed out:

- **repository clones:** The fetched code by `git clone`. It is recommended to store the repositories in `~/birdhouse`
- **anaconda**: By default, the installation process creates a folder `~/anaconda` for general anaconda-specific software (see also anaconda).
- **conda environments:** All birds (repositories) are built with their own environment to avoid missmatch of dependencies.

By default, the conda environments are in `~/.conda/envs/`.

To change the default settings, create a `Makefile.config` with:

```
$ cp Makefile.config.example Makefile.config
```

and change the paths accordingly to your needs.

Furthermore, in `environment.yml`, the conda packages can be defined. It is recommended to pin the version. The bird-specific packages are defined here, while in `requirements/conda_pinned`, general versions are set.

There are **log files** situated at:: `~/birdhouse/var/log/pywps/`

Set up a birdhouse ecosystem server

If you are already familliar with installing single standalone WPS (follow the installation guides in the documentations of e.g. emu), then you are ready to set up a birdhouse containing flyingpigeon (providing scientific analyses methods), malleefowl (to search and fetch data) and the pheonix (a graphic interface for a web browser including a WMS).

general remarks:

Check the *Requirements* of your system!
The installation is done as **normal user**, root rights are causing conflicts.

clone the repositories from gitHub:

It is recommended to collect the repositories in a seperate folder (e.g. birdhouse, but can have a name of your choice):

```
$ mkdir birdhouse
$ cd birdhouse
```

- **fetch the source code:**

```
$ git clone https://github.com/bird-house/flyingpigeon.git
$ git clone https://github.com/bird-house/pyramid-phoenix.git
$ git clone https://github.com/bird-house/malleefowl.git
```

- **phoenix password**

To be able to log into the Phoenix GUI once the services are running, it is necessary to generate a password: go into the pyramid-phoenix folder and run:

```
$ make passwd
```

This will automatically write a password hash into pyramid-phoenix/custom.cfg

- **installation**

You can run the installation with default settings. It will create an anaconda environment into your HOME direcory and deploy all required software dependecies there. *read the ''changing the default configuration' first if you would like to change the defaults.*

In **all** of the tree folders (malleefowl, flyingpigeon and pyramid-phoenix) run:

```
$ make install
```

This installation will take some minutes to fetch all dependencies and install them into seperate conda environments. With the default settings, the installation creates the following folders:

```
$ ls ~/anaconda/
```

contains general software required by anaconda:

```
$ ls ~/.conda/envs/
```

contains the seperate environments of the birds for their specific software dependencies:

```
$ ls ~/birdhouse/var/
```

the local cache for fetched input files, output files and logs. This folder is growing (while fetching files and storing job outputs) under productive usage of birdhouse.

- **start the services**

in **one** of the birds run:

```
$ make start
```

or:

```
$ make restart
```

and to check if the services are running, run:

```
$ make status
```

- **launching the Phoenix GUI**

If the services are running, you can launch the GUI in a common web browser. By default, phoenix is set to port 8081:

```
firefox http://localhost:8081
```

or:

```
firefox https://localhost:8443/
```

Now you can log in (upper right corner) with your Phoenix password created previously. Phoenix is just a graphical interface with no more function than looking nice ;-).

- **register a service in the GUI**

Your first administrator step is to register flyingpigeon as a service. For that, log in with your phoenix password. In the upper right corner is a tool symbol to open the 'settings'. Click on 'Services' and the 'Register a Service'.

flyingpigeon is per default at port 8093.

the appropriate url is:

```
http://localhost:8093/wps
```

Provide service title and name as you like: Service Title: Flyingpigeon Service Name: flyingpigeon

check 'Service Type' : 'Web Processing Service' (default) and register.

Optionally, you can check 'Public access?', to allow unregistered users to launch jobs. (**NOT recommended**)

- **launching a job**

Now your birdhouse ecosysem is set up. The also installed malleefowl is already running in the background and will do a lot of work silently. Ther is **no need to register malleefowl** manually!

Launching a job can be performed as a process (Process menu) or with the wizard. To get familliar with the processes provided by each of the birds, read the approriate documentation for each of the services listed in the overview:

- **changing the default configuration:**

The default configuration can be changed by creating a Makefile.config file. There is an example provided to be used:

```
$ cp Makefile.config.example Makefile.config
```

and set the appropriate path. You have to **do this in all** bird repositories.

Furthermore, you might change the hostname (to provide your service to the outside), ESGF-node connection, the port or the log-level for more/less information in the administrator logfiles. Here is an example pyramid-phoenix/custom.cfg:

```
[settings]
hostname = localhost
http-port = 8081
https-port = 8443
log-level = DEBUG
# run 'make passwd' and to generate password hash
phoenix-password = sha256:513....
# generate secret
# python -c "import os; print(''.join('%02x' % ord(x) for x in os.urandom(16)))"
phoenix-secret = d5e8417....30
esgf-search-url = https://esgf-data.dkrz.de/esg-search
wps-url = http://localhost:8091/wps
```

- **Administration HELP:**

In case of questions or trouble shooting, feel welcome to join the birdhouse chat and get into contact with the developers directly:

Birdhouse-Chatroom

Backups

See the mongodb documentation on how to backup the database. With the following command you can make a dump of the `users` collection of the Phoenix database:

```
$ mongodump --port 27027 --db phoenix_db --collection users
```

Developer Guide

- *Python Packaging*
- *Python Code Style*
  - *Atom*
  - *Sublime*
  - *PyCharm*
  - *Kate*
  - *Emacs*
  - *Vim*
  - *Spyder*
- *Coding Style using EditorConfig*

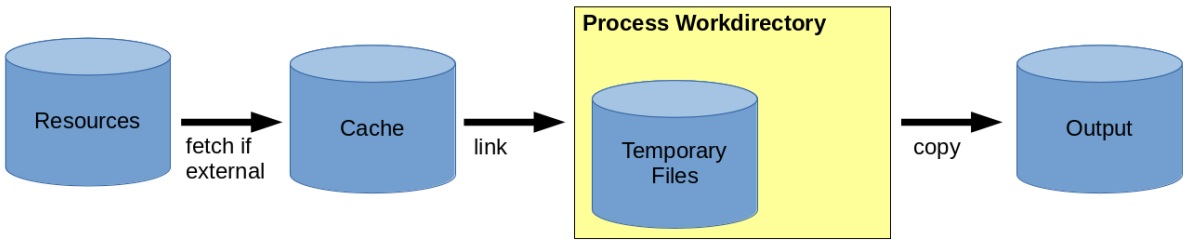
Writing a WPS process

In birdhouse, we are using the *PyWPS* implementation of a *Web Processing Service*. Writing a WPS process in birdhouse is the same as in PyWPS. The PyWPS documentation has a tutorial on writing a process. *Please* follow this PyWPS tutorial.

To get started more easily, you can install Emu with some example processes for PyWPS.

Data production

WPS is designed to reduce data transport and enables data processing close to the data archive. Nevertheless, files are stored within birdhouse in a structured way. For designing a WPS process or process chain, the location of input, output and temporary files are illustrated as follows:



Resources, which are already on the local disc system (output by other processes or as locally stored data archives), are linked into the cache simply with a soft link to avoid data transport and disc space usage.

The locations are defined as follows:

- **Resources:** Any kind of accessable data such as ESGF, thredd server or files stored on the server-side disc system.
- **Cache:** `~/birdhouse/var/lib/pywps/cache/` The cache is for external data which are not located on the server side. The files of the cache are separated by the birds performing the data fetch and keep the folder structure of the original data archive. Once a file is already in the cache, the data will not be refetched if a second request is made. The cache can be seen as a local data archive. Under productive usage of birdhouse, this folder is growing, since all requested external data are stored here.

- **Working directory:** `~/birdhouse/var/lib/pywps/tmp/` Each process is running in a temporary folder (= working directory) which is removed after the process is successfully executed. Like the cache, the working directories are separated by birds. Resource files are linked into the directory.
- **Output files:** `~/birdhouse/var/lib/pywps/outputs/` The output files are also stored in output folders separated by the birds producing the files. In the case of flyingpigeon, you can get the paths with:

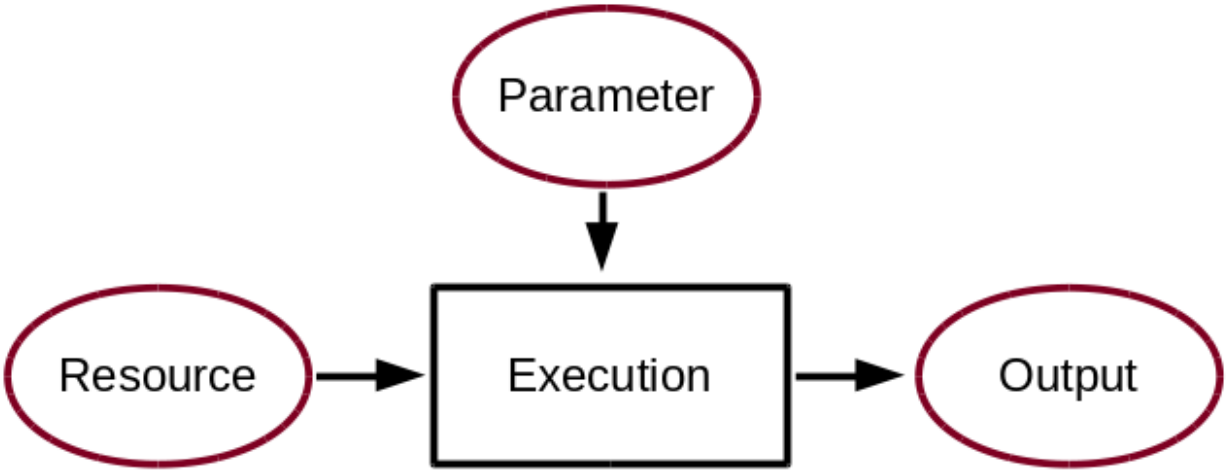
```
from flyingpigeon import config

output_path = config.output_path()         # returns the output folder path
outputUrl_path = config.outputUrl_path()   # returns the URL address of the output␣
↪folder
```

And in some special cases, static files are used (e.g. html files to provide general information). These files are located in the repository. In the case of flyingpigeon, they are located at: `./flyingpigeon/flyingpigeon/static/`

and copied during the installation (or update) to: `~/birdhouse/var/www/`

Designing a process

For designing a process it is necessary to know some basic concepts about how data are produced in birdhouse. The following are some basic explanations to help in developing appropriate processes to provide a scientific method as a service. The word **process** is used in the same sense as in the OGC standard: *for any algorithm, calculation or model that either generates new data or transforms some input data into output data*, and can be illustrated as follows:



The specific nature of web processing services is that processes can be described in a standardised way (see: *Writing a WPS process*). In the flyingpigeon repository, the process descriptions are located in:

```
./flyingpigeon/flyingpigeon/processes
```

As part of the process description there is an **execute** function:

```
def execute(self):
    # here starts the actual data processing
    import pythonlib
    from flyingpigeon import aflyingpigeonlib as afl

    result = afl.nicefunction(indata, parameter1=argument1, parameter2=argument2)
```

```
    self.output.setValue( result )
```

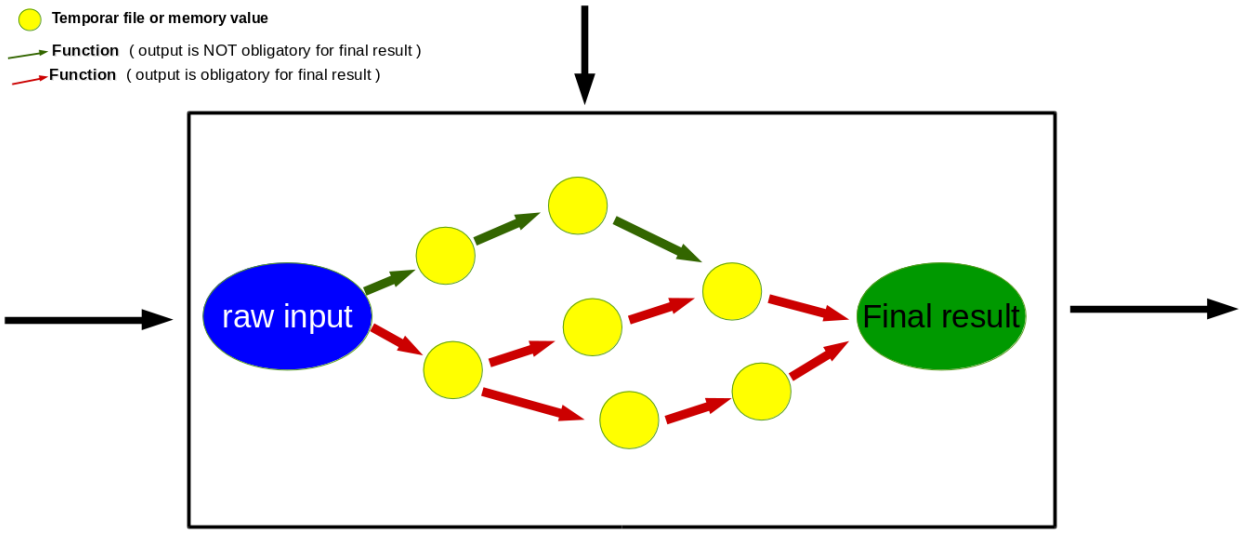
It is a recommended practice to separate the functions (the actual data processing) from the process description. This creates modularity and enables multiple usage of functions when designing several processes. The modules in flyingpigeon are located here:

```
./flyingpigeon/flyingpigeon
```

Generally, the execution of a process contains several processing steps, where temporary files and memory values are generated. Birdhouse runs each job in a separate folder, by default situated in:

```
~/birdhouse/var/lib/pywps/tmp/
```

This tmp folder is removed after job is successfully executed. To reuse temporary files, it is necessary to declare them as output files. Furthermore, during execution, there are steps which are necessary to be successfully performed and a result is called back. If this particular step fails, the whole process should exit with an appropriate error message, while in other cases it is not relevent for producing the final result. The following image shows a theoretical chain of functions:



In practice, the functions should be encapsulated in **try** and **except** calls and appropriate information given to the log file or shown as a status message:

```
from pywps.Process import WPSProcess
import logging
logger = logging.getLogger(__name__)

# set a status message
self.status.set('execution started at : %s ' % dt.now(),5)

try:
    self.status.set('the process is doing something : %s '  % dt.now(),10)
    result = 42
    logger.info('found the answer of life')
except:
    msg = 'This failed but is obligatory for the output. The process stops now!'
    logger.error(msg)
    raise Exception(msg)

```

```
try:
    self.status.set('the process is doing something else : %s '  % dt.now(),20)
    interesting = True
    # or generate a temporary file
    logger.info(' Thanks for reading the guidelines ')
except:
    msg = 'This failed but is not obligatory for the output. The process will␣
↪continue.'
    logger.debug(msg)

try:
    self.status.set('the process is doing something else : %s '  % dt.now(),20)
    interesting = True
    # or generate a temporary file
    logger.info(' Take your time to understand enverything ')
except:
    msg = 'This failed. The process will continue but writes out the reason of the␣
↪failture'
    logger.exception(msg)


try:
    self.status.set('the process is doing something else : %s '  % dt.now(),20)
    interesting = True
    # or generate a temporary file
    logger.info(' This is the right way to do it  ')
except:
    msg = 'Here comes a warning: Are you sure this is the right way to do it??'
    logger.warn(msg)
```

The log file then looks like:

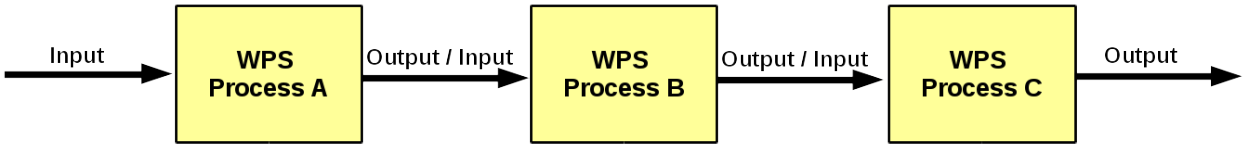
```
tail -f  ~/birdhouse/var/log/pywps/flyingpigeon.log

PyWPS [2016-09-14 11:49:13,819] INFO: Start ocgis module call function
PyWPS [2016-09-14 11:49:13,820] INFO: Execute ocgis module call function
PyWPS [2016-09-14 11:49:13,828] DEBUG: input has Lambert_Conformal projection and can␣
↪not subsetted with geom
PyWPS [2016-09-14 11:49:13,828] DEBUG: failed for point ['2.356138', ' 48.846450']␣
↪Validation failed on the parameter "uri" with the message: Cannot be None
PyWPS [2016-09-14 11:49:13,993] INFO: Start ocgis module call function
PyWPS [2016-09-14 11:49:13,994] INFO: Execute ocgis module call function
PyWPS [2016-09-14 11:49:14,029] INFO: OcgOperations set
PyWPS [2016-09-14 11:49:14,349] INFO: tas as variable dedected
PyWPS [2016-09-14 11:49:14,349] INFO: data_mb  = 0.0417938232422 ; memory_limit =␣
↪1660.33984375
PyWPS [2016-09-14 11:49:14,349] INFO: ocgis module call as ops.execute()
PyWPS [2016-09-14 11:49:16,648] INFO: Succeeded with ocgis module call function
```

Logging information is written to the logfile depending on the 'log-level' settings in ~/custom.cfg

Another point to think about when designing a process is the possibility of chaining processes together. The result of a process can be a final result or be used as an input for another process. Chaining processes is a common practice but depends on the user you are designing the service for. Technically, for the development of WPS process chaining, here are a few summary points:

- the functional code should be modular and provide an interface/method for each single task
- provide a wps process for each task

- wps processes can be chained, manually or programmatically, to run a complete workflow
- wps chaining can be done manually, with workflow tools, direct wps chaining or with code scripts
- a complete workflow chain could also be started by a wps process.



In birdhouse, restflow and dispel4py are integrated, and WPS chaining is used in the wizard of phoenix. This WPS chain fetches data and runs a process (selected by the user) with the fetched data : http://pyramid-phoenix.readthedocs.io/en/latest/user_guide.html#wizard

Here is a tutorial to follow: *Chaining WPS processes*.

or:

http://birdhouse.readthedocs.io/en/latest/appendix.html#scientific-workflow-tools

Writing Documentation

Documentation is written in ReStructuredText and generated with Sphinx. The birdhouse components use the Buildout recipe birdhousebuilder.recipe.sphinx which sets up Sphinx and a minimal `docs` folder. With `make docs` the documentation is generated locally. The documentation is published to Read the Docs with each commit to the *master* branch. The API reference is generated automatically using the Sphinx plugin AutoAPI.

- http://sphinx-doc.org/tutorial.html
- http://quick-sphinx-tutorial.readthedocs.io/en/latest/

Using Anaconda in birdhouse

The installation of the birdhouse components and especially the processes involve many software dependencies. The core dependencies are of course the WPS-related packages like *PyWPS* and *OWSLib* from the *GeoPython* project. But most dependencies come from the processes themselves served by the WPS, such as *numpy*, *R*, *NetCDF*, *CDO*, *matplotlib*, *ncl*, *cdat*, and many more.

The aim of birdhouse is to take care of all these dependencies so that the user does not need to install them manually. If these dependencies were only *pure* Python packages, then using the *Buildout* build tool, together with the Python package index *PyPi*, would be sufficient. But many Python packages have *C* extensions and there are also non-Python packages that need to be installed like *R* and *NetCDF*.

In this situation, the *Anaconda Python distribution* is helpful. Anaconda already has a lot of Python-related packages available for different platforms (Linux, MacOSX, Windows), and there is no compilation needed on the installation host. Anaconda makes it easy to build own packages (*conda recipes*) and upload them to the free *Anaconda Server*.

Conda recipes by birdhouse

Birdhouse uses *Anaconda* to maintain package dependencies. Anaconda allows you to write your own conda recipes. In birdhouse, we have written several conda recipes for the packages that were not available on Anaconda. These

additional conda recipes by birdhouse are available on GitHub.

Anaconda provides a free *Anaconda Server*. Here you can upload your built conda packages for different platforms (Linux, MacOX, Windows). These packages are then available for installation with the *conda* installer.

Birdhouse has an organisation where all conda packages are collected which are built from the conda recipes on GitHub. These packages can be installed with the *conda* installer using the *birdhouse* channel. For example, if you are already using Anaconda, you can install *PyWPS* with the following command:

```
$ conda install --channel birdhouse pywps
```

Building conda packages

You can build packages locally and upload them to the *Anaconda Server*:

The Anaconda builds are using Docker images. The Anaconda docker image for Linux-64 is available on *Docker Hub*. But sometimes the docker image for Linux-64 provided by Anaconda fails for some packages. That is why birdhouse has in addition its own Linux-64 build image which is based on the Anaconda image. The Dockerfile for this image is on GitHub.

Warning: When you build conda packages for Linux-64, you need to be very careful to ensure that these packages will run on most Linux distributions (like CentOS, Debian, Ubuntu, ...). Our experience is that packages tjat build on CentOS 6.x will also run on recent Debian/Ubuntu distributions. The Docker build images are also CentOS 6.x based.

Note: You can build a conda package with the provided docker image for Linux-64. See the readme on how to use it.

Note: For future conda packages, one should use the community-driven conda-forge channel.

Using conda

See the conda documentation.

Anaconda alternatives

If Anaconda is not available, one could also provide these packages from source and compile them on each installation host. Buildout does provide ways to do so, but an initial installation with most of the software used in climate science could *easily take hours*.

Alternative package managers to Anaconda are for example *Homebrew* (MacOSX only) and *Linuxbrew* (a fork of Homebrew for Linux).

Using Buildout in birdhouse

Birdhouse uses the *Buildout* build tool to install and configure all birdhouse components (*Phoenix*, *Malleefowl*, *Emu*...). The main configuration file is `buildout.cfg` which is in the root folder of the application. As an example, have a look at the buildout.cfg from Emu.

Before building an application with Buildout, you have an initial bootstrap step:

```
$ python bootstrap-buildout.py -c buildout.cfg
```

This will generate the `bin/buildout` script. Now you can build the application:

```
$ bin/buildout -c buildout.cfg
```

The default configuration in the `buildout.cfg` should always work to run your application on `localhost` with default ports. You can customize the configuration by editing the `custom.cfg` which extends and overwrites the settings of `buildout.cfg`. You may have a look at the custom.cfg example of Emu. So, instead of using `buildout.cfg`, you should use `custom.cfg` for the build:

```
$ bin/buildout -c custom.cfg
```

For convenience, birdhouse has a Makefile which hides all these steps. If you want to build an application, you just need to run:

```
$ make install
```

See the Makefile example of Emu For more details, see the *Installation* section and the Makefile documentation.

Buildout recipes by birdhouse

Buildout has a plugin mechanism to extend the build tool functionality with recipes. Buildout can handle Python dependencies on its own. But in birdhouse, we install most dependencies with Anaconda. We are using a Buildout extension to install conda packages with Buildout. Buildout does use these Python packages instead of downloading them from *PyPi*. There is also a set of recipes to set up Web Processing Services with *PyWPS*, *Nginx*, *Gunicorn* and *Supervisor*. All these Buildout recipes are on GitHub and can be found on PyPi.

Here is the list of currently-used Buildout recipes by birdhouse:

- birdhousebuilder.recipe.conda: A Buildout recipe to install Anaconda packages.
- birdhousebuilder.recipe.pywps: A Buildout recipe to install and configure PyWPS Web Processing Service with Anaconda.
- birdhousebuilder.recipe.pycsw: A Buildout recipe to install and configure pycsw Catalog Service (CSW) with Anaconda.
- birdhousebuilder.recipe.nginx: A Buildout recipe to install and configure Nginx with Anaconda.
- birdhousebuilder.recipe.supervisor: A Buildout recipe to install and configure supervisor for Anaconda.
- birdhousebuilder.recipe.docker: A Buildout recipe to generate a Dockerfile for birdhouse applications.
- birdhousebuilder.recipe.sphinx: A Buildout recipe to generate documentation with Sphinx.
- birdhousebuilder.recipe.ncwms: A Buildout recipe to install and configure ncWMS2 Web Map Service.
- birdhousebuilder.recipe.adagucserver: A Buildout recipe to install and configure Adagucserver Web Map Service.

Python Packaging

Links:

- https://packaging.python.org/

Example:

```
$ python setup.py sdist
$ python setup.py bdist_wheel
$ python setup.py register -r pypi
$ twine upload dist/*
```

Check the rst docs in the long_description of `setup.py`:

- https://github.com/collective/collective.checkdocs

Example:

```
$ python setup.py checkdocs
```

Python Code Style

Birdhouse uses PEP8 checks to ensure a consistent coding style. Currently the following PEP8 rules are enabled in `setup.cfg`:

```
[flake8]
ignore=F401,E402
max-line-length=120
exclude=tests
```

See the flake8 documentation on how to configure further options.

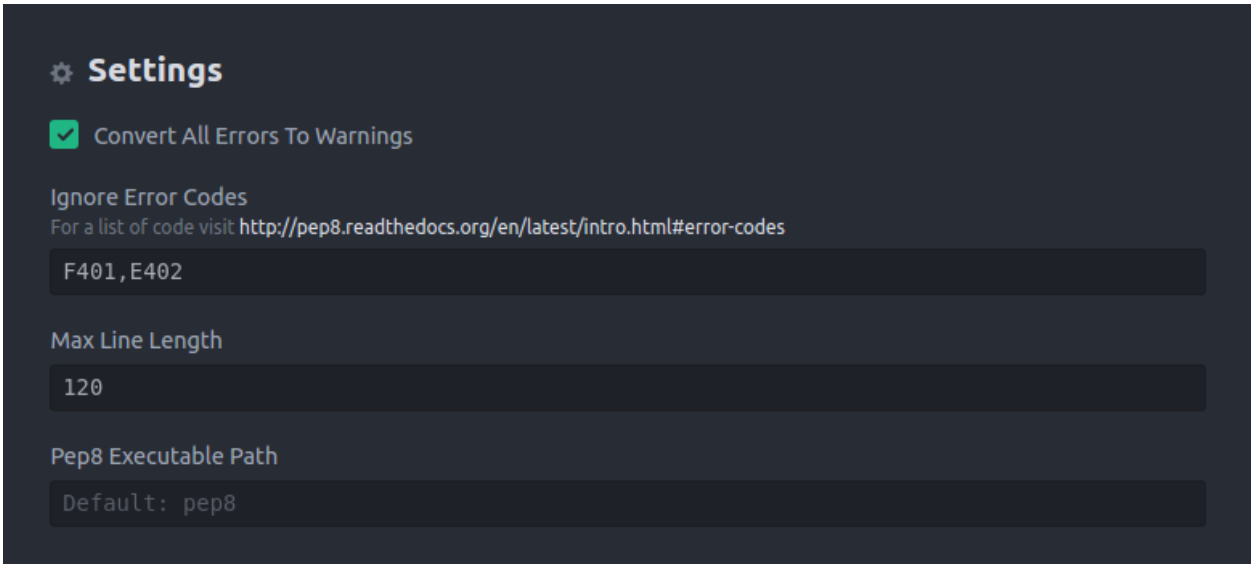
To check the coding style run `flake8`:

```
$ flake8 emu    # emu is the folder with python code
# or
$ make pep8     # make calls flake8
```

To make it easier to write code according to the PEP8 rules enable PEP8 checking in your editor. In the following we give examples how to enable code checking for different editors.

Atom

- Homepage: https://atom.io/
- PEP8 Atom Plugin: https://github.com/AtomLinter/linter-pep8



Sublime

- Install package control if you don't already have it: https://packagecontrol.io/installation
- Follow the instructions here to install Python PEP8 Autoformat: https://packagecontrol.io/packages/Python%20PEP8%20Autoformat
- Edit the settings to conform to the values used in birdhouse, if necessary
- To show the ruler and make wordwrap default, open Preferences → Settings—User and use the following rules

```
{
 // set vertical rulers in specified columns.
 "rulers": [79],

 // turn on word wrap for source and text
 // default value is "auto", which means off for source and on for text
 "word_wrap": true,

 // set word wrapping at this column
 // default value is 0, meaning wrapping occurs at window width
 "wrap_width": 79
 }
```

PyCharm

TODO

Kate

TODO

Emacs

TODO

Vim

TODO

Spyder

TODO

Coding Style using EditorConfig

EditorConfig is used to keep consistent coding styles between different editors. The configuration is on github in the top level directory `.editorconfig`. See the EditorConfig used in Birdhouse. Check the EditorConfig page on how to activate it for your editor.

Community

There are numerous ways to interact with the Birdhouse community, for example join the chat or the mailing list.

Mailing list

- News are published to the wps mailing list (Archive).
- Technical discussions take place in wps-dev mailing list (Archive).

Feel free to register.

Chat-room

If you want to have a quick chat with one of the developers, or just follow the discussions, feel welcome to join the Gitter chat room

Wiki

The birdhouse wiki provides an area for supporting information that frequently changes and / or is outside the scope of the formal documentation.

Meetings

The wiki is used to organize and document birdhouse meetings.

Contributing

The Birdhouse project openly welcomes contributions (bug reports, bug fixes, code enhancements/features, etc.). This document will outline some guidelines on contributing to birdhouse. As well, the birdhouse *Community* is a great place to get an idea of how to connect and participate in birdhouse community and development.

Code of Conduct

Contributors to this project are expected to act respectfully toward others in accordance with the OSGeo Code of Conduct.

Source code

The source code of all birdhouse components is available on Github.

Issue tracker

Please use the issue tracker on GitHub for the corresponding birdhouse component.

WPS client side:

- Phoenix web application
- Birdy command line WPS client

WPS server side:

- Flyingpigeon WPS for climate impact
- Hummingbird WPS processes for cdo and compliance checking
- Emu WPS processes for demo and testing
- Malleefowl WPS base processes to access data

WPS Security:

- Twitcher, an WPS security proxy

Website development

The birdhouse website is on http://bird-house.github.io/. The HTML pages are maintained on GitHub.

Documentation

The documentation is created with Sphinx and is automatically published to ReadTheDocs with GitHub webhooks.

The main documentation (which you are reading now) is the starting point to get an overview of what birdhouse provides. Each birdhouse component comes with its own Sphinx documentation and is referenced by the main birdhouse document.

Frequently Asked Questions

General Questions

What is "birdhouse"?

Birdhouse is collection of Python packages to make the usage of Web Processing Services (WPS) easy. The available packages are used in the climate science community.

What is "WPS"?

The very short answer WPS is the acronym for Web Processing Service.

The sligthly longer answer So, let's say you have a function (maybe written in Python) which might calculate the **"summer days in Finland since 1990"**. Then this function has probably input parameters (region, from-date, to-date, NetCDF files, ...) and an output (or even more ...) which might be just an integer number or a text document or even a nice diagram. Now, you would like to provide this function as a web service, so that other people can call it with just a simple URL like:

```
http://myhost/wps/identifier=summer_days&region=finland&from=1990
```

... ok ... then you should have a deeper look at this *WPS* thing.

Getting Help

Glossary

Anaconda

Anaconda Python distribution Python distribution for large-scale data processing, predictive analytics, and scientific computing. https://www.continuum.io/

Binstar

Anaconda Server

Anaconda cloud Binstar is a service that allows you to create and manage public and private *Anaconda* package repositories. https://anaconda.org/ https://docs.continuum.io/

Bokeh Bokeh is a Python interactive visualization library that targets modern web browsers for presentation. Its goal is to provide elegant, concise construction of novel graphics in the style of D3.js, but also deliver this capability with high-performance interactivity over very large or streaming datasets. http://bokeh.pydata.org/en/latest/

Buildout *Buildout* is a Python-based build system for creating, assembling and deploying applications from multiple parts, some of which may be non-Python-based. It lets you create a buildout configuration and reproduce the same software later. http://www.buildout.org/en/latest/

CDO

Climate Data Operators *CDO* is a collection of command line Operators to manipulate and analyse Climate and NWP model Data. https://code.zmaw.de/projects/cdo

cfchecker The NetCDF Climate Forcast Conventions compliance checker. https://pypi.python.org/pypi/cfchecker

climate indice A climate index is a calculated value that can be used to describe the state and the changes in the climate system. http://icclim.readthedocs.io/en/latest/intro.html#climate-indices-label

CMIP5 In climatology, the Coupled Model Intercomparison Project (CMIP) is a framework and the analog of the Atmospheric Model Intercomparison Project (AMIP) for global coupled ocean-atmosphere general circulation models. https://en.wikipedia.org/wiki/Coupled_model_intercomparison_project

Conda The *conda* command is the primary interface for managing Anaconda installations. http://conda.pydata.org/docs/index.html

CORDEX The CORDEX vision is to advance and coordinate the science and application of regional climate downscaling through global partnerships. http://www.cordex.org/

COWS The COWS Web Processing Service (WPS) is a *generic* web service and offline processing tool developed within the Centre for Environmental Data Archival (CEDA). http://cows.ceda.ac.uk/cows_wps.html

CSW

Catalog Service *Catalog Service for the Web (CSW)*, sometimes seen as Catalog Service - Web, is a standard for exposing a catalogue of geospatial records in XML on the Internet (over HTTP). The catalogue is made up of records that describe geospatial data (e.g. KML), geospatial services (e.g. WMS), and related resources. https://en.wikipedia.org/wiki/Catalog_Service_for_the_Web

Dispel4py *Dispel4Py* is a Python library for describing abstract workflows for distributed data-intensive applications. http://www2.epcc.ed.ac.uk/~amrey/VERCE/Dispel4Py/index.html

Docker *Docker* - An open platform for distributed applications for developers and sysadmins. https://www.docker.com/

Docker Hub Docker Hub manages the lifecycle of distributed apps with cloud services for building and sharing containers and automating workflows. https://hub.docker.com/

Emu *Emu* is a Python package with some test proccess for *Web Processing Services*. http://emu.readthedocs.io/en/latest/

ESGF

Earth System Grid Federation An open source effort providing a robust, distributed data and computation platform, enabling world wide access to Peta/Exa-scale scientific data. http://esgf.llnl.gov/

GeoPython GitHub organisation of Python projects related to geospatial. https://geopython.github.io/

GeoServer GeoServer is an open source software server written in Java that allows users to share and edit geospatial data. http://docs.geoserver.org/stable/en/user/index.html

GitHub GitHub is a web-based Git repository hosting service. https://github.com/ https://en.wikipedia.org/wiki/GitHub

Gunicorn Gunicorn *Green Unicorn* is a Python WSGI HTTP Server for UNIX. http://gunicorn.org/

Homebrew The missing package manager for OS X. http://brew.sh/

ICCLIM

Indice Calculation CLIMate *ICCLIM* (Indice Calculation CLIMate) is a Python library for computing a number of *climate indices*. http://icclim.readthedocs.io/en/latest/

Linuxbrew Linuxbrew is a fork of Homebrew, the Mac OS package manager, for Linux. http://brew.sh/linuxbrew/

Malleefowl *Malleefowl* is a Python package to simplify the usage of *Web Processing Services*. http://malleefowl.readthedocs.io/en/latest/

NetCDF NetCDF (Network Common Data Form) is a set of software libraries and self-describing, machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data. https://en.wikipedia.org/wiki/NetCDF

Nginx nginx [engine x] is an HTTP and reverse proxy server. http://nginx.org/

ocgis

OpenClimateGIS *OpenClimateGIS* (OCGIS) is a Python package designed for geospatial manipulation, subsetting, computation, and translation of climate datasets stored in local *NetCDF* files or files served through *THREDDS* data servers. https://www.earthsystemcog.org/projects/openclimategis/ https://github.com/NCPP/ocgis

OGC

Open Geospatial Consortium The *Open Geospatial Consortium (OGC)* is an international voluntary consensus standards organization, originated in 1994. https://en.wikipedia.org/wiki/Open_Geospatial_Consortium, http://www.opengeospatial.org/standards/wps

OpenID OpenID (OID) is an open standard and decentralized protocol by the non-profit OpenID Foundation that allows users to be authenticated by certain co-operating sites (known as Relying Parties or RP) using a third party service. https://en.wikipedia.org/wiki/OpenID, http://openid.net/

OWSLib OWSLib is a Python package for client programming with *Open Geospatial Consortium* web service interface standards, and their related content models. OWSLib has *WPS* client library which is used in Birdhouse to access WPS services. http://geopython.github.io/OWSLib/, http://geopython.github.io/OWSLib/#wps

Phoenix Pyramid *Phoenix* is a web-application build with the Python web-framework pyramid. Phoenix has a user interface to make it easier to interact with *Web Processing Services*. http://pyramid-phoenix.readthedocs.io/en/latest

PyCSW pycsw is an *OGC* CSW server implementation written in Python. Started in 2010 (more formally announced in 2011), pycsw allows for the publishing and discovery of geospatial metadata, providing a standards-based metadata and catalogue component of spatial data infrastructures. http://pycsw.org/, https://github.com/geopython/pycsw

PyPi

Python Package Index The Python Package Index is a repository of software for the Python programming language. https://pypi.python.org/pypi

Pyramid Pyramid is a Python web framework. http://www.pylonsproject.org/

PyWPS *Python Web Processing Service* is an implementation of the *Web processing Service* standard from *Open Geospatial Consortium*. http://pywps.org/

RestFlow *RestFlow* is a dataflow programming language and runtime engine designed to make it easy for scientists to build and execute computational pipelines. https://github.com/restflow-org/restflow/wiki

Supervisor Supervisor is a client/server system that allows its users to monitor and control a number of processes on UNIX-like operating systems. http://supervisord.org/

Taverna *Taverna* is an open source and domain-independent Workflow Management System – a suite of tools used to design and execute scientific workflows. http://www.taverna.org.uk/

TDS

THREDDS The THREDDS Data Server (TDS) is a web server that provides metadata and data access for scientific datasets, using a variety of remote data access protocols. http://www.unidata.ucar.edu/software/thredds/current/tds/

VisTrails *VisTrails* is an open-source scientific workflow and provenance management system that supports data exploration and visualization. http://www.vistrails.org/index.php/Main_Page

WMS

Web Mapping Service A Web Map Service (WMS) is a standard protocol for serving georeferenced map images over the Internet that are generated by a map server using data from a GIS database. https://en.wikipedia.org/wiki/Web_Map_Service

Workflow

Workflow Management System A workflow management system (WfMS) is a software system for the set-up, performance and monitoring of a defined sequence of tasks, arranged as a workflow. https://en.wikipedia.org/wiki/Workflow_management_system

WPS

Web Processing Service WPS is an open standard to search and run processes with a simple web-based interface. See: *What is WPS?*.

WSGI WSGI is an interface specification by which server and application communicate. http://wsgi.tutorial.codepoint.net/

x509 In cryptography, X.509 is an ITU-T standard for a public key infrastructure (PKI) and Privilege Management Infrastructure (PMI). https://en.wikipedia.org/wiki/X.509

XML-RPC It's a spec and a set of implementations that allow software running on disparate operating systems, running in different environments to make procedure calls over the Internet. http://xmlrpc.scripting.com/default.html

Release Notes

Bonn (August 2016)

- Leaflet map with time-dimension plugin.
- using twitcher security proxy.
- using conda environments for each birdhouse compartment.
- using ansible to deploy birdhouse compartments.
- added weather-regimes and analogs detection processes.
- allow upload of files to processes.
- updated Phoenix user interface.

Paris (October 2015)

- updated documents on readthedocs
- OAuth2 used for login with GitHub, Ceda, ...
- LDAP support for login
- using ncWMS and adagucwms
- register and use Thredds catalogs as data source

- publish local netcdf files and Thredds catalogs to birdhouse Solr
- qualtiy check processes added (cfchecker, qa-dkrz)
- generation of docker images for each birdhouse component
- using dispel4py as workflow engine in Malleefowl
- using Celery task scheduler/queue to run and monitor WPS processes
- improved Phoenix web client
- using birdy wps command line client

Paris (September 2014)

- Phoenix UI as WPS client with ESGF faceted search component and a wizard to chain WPS processes
- PyWPS based processing backend with supporting processes of Malleefowl
- WMS service (inculded in Thredds) for visualization of NetCDF files
- OGC CSW catalog service for published results and OGC WPS services
- ESGF data access with wget and OpenID
- Caching of accessed files from ESGF Nodes and Catalog Service
- WPS processes: cdo, climate-indices, ensemble data visualization, demo processes
- IPython environment for WPS processes
- initial unit tests for WPS processes
- Workflow engine Restflow for running processing chains. Currently there is only a simple workflow used: get data with wget - process data.
- Installation based on anaconda and buildout
- buildout recipes (birdhousebuilder) available on PyPI to simplify installation and configuration of multiple WPS server
- Monitoring of all used services (WPS, WMS, CSW, Phoenix) with supervisor
- moved source code and documentation to birdhouse on GitHub

Helsinki (May 2014)

- presentation of birdhouse at EGI, Helsinki
- stabilized birdhouse and CSC processes
- updated documenation and tutorials

Vienna (April 2014)

- presentation of birdhouse at EGU, Vienna
- “quality check” workflow for CORDEX data

Roadmap

Milestone December 2015

- prototype for wps security proxy
- update ncWMS2 and adagucserver wms
- update sphinx with api references
- improved birdy command line (https, argcomplete)
- caching of wps requests
- deployment with docker using docker-compose
- minimal bird example and skeleton function
- unit tests with sample netcdf data
- wps decorator
- enable wps for apache-climate processes
- try sci-wms web map service

Milestone March 2015

- move docs to readthedocs
- birdhouse overview
- presentation at LSDMA in Berlin

Long-term TODO List

Security

- using OAuth for login
- secure WPS service:
  - wps client and services should not be changed
  - using OAuth Token generation
  - Token should be part of the url `http://localhost/wps/emu/auhbgt3n` or `http://localhost/wps/emu?request=getcapabilities&token=auhbgt3n`
  - using a security proxy service in front of WPS servers.
  - *GetCapabilities* and *DescribeProcess* should be available without a security token.

Data Sources

- OpenStack
  - using python swift client
- PyCSW:

  - already there but needs to be refactored
  - CSW is used for publishing results
- ESGF/Thredds:
  - opendap without aggregations (mostly not available)
- Observational Climate Data:
  - which are available for public access and usage (license issuses)
- local file archives:
  - make them searchable ... pattern matching ... index service ...
- CERA climate database
- OGC data services like WCS and SOS, ...

Web Processing Service

- usage of other WPS implementations: COWS, GeoServer, Zoo, ...
  - process integration interface (with python decorators) which generates the integration code for other WPS services.
- extensions: cancel (comes with wps 2.0), dry-run, ... cows and maybe geoserver have some of these
- caching process execution: cows has cachings ... but should be independent of the wps implementation

Deployment

- deployment with saltstack and/or docker ...

Highload Processing

- integration of scheduler ... slurm ... (cows has an example for that)
- using load balancing ...

Docs & Testing

- tests:
  - improved unit tests
  - continous integration with github + travisCI + binstar + docker
  - complete install tests with docker builds
- complete sphinx documentation
- need a better overview of the components
- simple understable image of what WPS is good for

License

Birdhouse is Open Source and released under the Apache License, Version 2.0.

Useful Links

WPS Documentation

- What is WPS?
- WPS on OSGeo Live
- WPS tutorial
- OGC Web Processing Service Standard
- PyWPS Wiki
- GeoServer tutorial

Talks:

- The WPS 2.0 standard (preliminary information)
- WPS Application Patterns
- Using WPS (PyWPS) with Taverna Orchestration
- Pywps a tutorial for beginners and developers
- Zoo presentation foss4g.jp-2011

WPS Software

WPS Server Software:

- *PyWPS*
- GeoServer - http://docs.geoserver.org/stable/en/user/services/wps/index.html

- Zoo - `http://www.zoo-project.org/`
- *COWS*
- Deegree - http://www.deegree.org/
- 52 North - http://52north.org/communities/geoprocessing/wps/

WPS Client Software:

- *OWSLib Python Client*
- OpenLayers WPS Plugin - http://dev.openlayers.org/docs/files/OpenLayers/WPSClient-js.html
- GeoTools WPS Module - http://docs.geotools.org/latest/userguide/unsupported/wps.html
- 52 North Java Client - http://52north.org/communities/geoprocessing/wps/index.html
- 52 North Javascript Client - http://geoprocessing.demo.52north.org:8080
- WPS Javascript Client by Boundless - https://github.com/boundlessgeo/wps-gui

QGIS Desktop GIS with wps plugins:

- http://www.qgis.org/en/site/
- http://plugins.qgis.org/plugins/wps/
- http://geolabs.fr/plugins.xml

uDig Desktop GIS with wps plugins:

- http://udig.refractions.net/
- https://udig.github.io/docs/user/reference/Using%20the%20WPS%20plugin.html
- https://github.com/52North/uDig-WPS-plugin (outdated)

WMS Software

WMS server:

- ncWMS2 - http://reading-escience-centre.github.io/edal-java/
- adaguc - http://adaguc.knmi.nl/
- sci-wms - http://sci-wms.github.io/sci-wms/

WMS clients:

- OpenLayers - http://openlayers.org/
- **Leaflet - http://leafletjs.com/**
  - time dimension - http://apps.socib.es/Leaflet.TimeDimension/examples/
- GeoExt - http://geoext.github.io/geoext2/

Scientific Workflow Tools

Workflow Engines:

- *Dispel4py*
- *RestFlow*
- *Taverna*

- *VisTrails*
- Kepler - https://kepler-project.org/
- KNIME - http://www.knime.org/

Taverna with WPS:

- `http://rsg.pml.ac.uk/wps/generic.cgi?request=GetCapabilities&service=WPS`
- https://www.youtube.com/watch?v=JNAtoOejVIo
- https://taverna.incubator.apache.org/introduction/services-in-taverna.html
- https://github.com/myGrid/small-area-estimator
- http://comments.gmane.org/gmane.science.biology.informatics.taverna.user/1415
- http://dev.mygrid.org.uk/wiki/display/developer/SCUFL2

VisTrails with WPS:

- https://github.com/ict4eo/eo4vistrails
- http://proj.badc.rl.ac.uk/cows/wiki/CowsWps/CDOWPSWorkingGroup/WPSAndWorkflows
- `http://www.kitware.com/source/home/post/105`

Kepler with WPS:

- https://kepler-project.org/users/sample-workflows

Workflows with PyWPS:

- https://github.com/AnnaHomolka/PyWPS/blob/master/doc/tutorial_process_chaining.pdf

Other Workflow Engines:

- `http://www.yawlfoundation.org/`
- https://en.wikipedia.org/wiki/Scientific_workflow_system
- http://airavata.apache.org/
- http://search.cpan.org/~nuffin/Class-Workflow-0.11/

Scientific Python

- Anaconda - https://www.continuum.io/downloads

Completely free enterprise-ready Python distribution for large-scale data processing, predictive analytics, and scientific computing

- pandas - http://pandas.pydata.org/

Python Data Analysis Library

Python in Climate Science

- OpenClimateGIS - https://earthsystemcog.org/projects/openclimategis/

OpenClimateGIS is a Python package designed for geospatial manipulation, subsetting, computation, and translation of climate datasets stored in local NetCDF files or files served through THREDDS data servers. [..]

- ICCLIM (i see clim ...) - https://github.com/cerfacs-globc/icclim

Python library for climate indices calculation. Documentation at http://icclim.readthedocs.io/en/latest/

Python Web Frameworks and Utils

- Pyramid - http://www.pylonsproject.org/
- Authomatic - http://peterhudec.github.io/authomatic/
- Bootstrap - http://getbootstrap.com/
- Bootstrap Tutorial - http://www.w3schools.com/bootstrap/default.asp
- Deform - https://github.com/Pylons/deform
- Deform with Bootstrap demo - http://deform2demo.repoze.org/
- Colander - http://docs.pylonsproject.org/projects/colander/en/latest/index.html
- TinyMCE - https://www.tinymce.com/
- Font Awesome - http://fontawesome.io/
- Leaflet - http://leafletjs.com/
- Leaflet TimeDimension - http://apps.socib.es/Leaflet.TimeDimension/examples/

Example WPS Services

List of available Web Processing Services:

- Zoo WPS for PublicaMundi project - `http://zoo.dev.publicamundi.eu/cgi-bin/zoo_loader.cgi?service=WPS&version=1.0.0&request=GetCapabilities`
- GeoServer Demo WPS - `http://demo.opengeo.org/geoserver/wps?request=GetCapabilities&service=WPS`
- USGS Geo Data Portal- `http://cida.usgs.gov/climate/gdp/process/WebProcessingService`
- KNMI climate4impact Portal - `http://climate4impact.eu//impactportal/WPS?request=GetCapabilities&service=WPS`
- BADC CEDA - `http://ceda-wps2.badc.rl.ac.uk/wps?request=GetCapabilities&service=WPS`
- delatres - `http://dtvirt5.deltares.nl/wps/?Request=GetCapabilities&Service=WPS`
- 52 North - `http://geoprocessing.demo.52north.org:8080/52n-wps-webapp-3.3.1/WebProcessingService?Request=GetCapabilities&Service=WPS`
- 52 North - `http://geoprocessing.demo.52north.org:8080/52n-wps-webapp-3.3.1-gt/WebProcessingService?Request=GetCapabilities&Service=WPS`
- ZOO Demo WPS - `http://zoo-project.org/cgi-bin/zoo_loader3.cgi?Request=GetCapabilities&Service=WPS`
- British Antarctic Survey WPS for Meteorological Data - `http://sosmet.nerc-bas.ac.uk:8080/wpsmet/WebProcessingService?Request=GetCapabilities&Service=WPS`
- PyWPS Demo - `http://apps.esdi-humboldt.cz/pywps/?request=GetCapabilities&service=WPS&vers0.0`

Alternatives to WPS

- XML-RPC: Simple cross-platform distributed computing, based on the standards of the Internet. - http://xmlrpc.scripting.com/
- Swagger is a simple yet powerful representation of your RESTful API. - http://swagger.io/

Related Projects

- http://geopython.github.io/
- http://geonode.org/
- http://esgf.llnl.gov/
- http://climate4impact.eu/impactportal/general/index.jsp
- http://adaguc.knmi.nl/
- http://wps-web1.ceda.ac.uk/ui/home
- `https://freva.met.fu-berlin.de/`
- https://climate.apache.org/

CHAPTER 2

Presentations & Blog Posts

- UNCCC Subgroup 2017 at Kigali
- AGU 2016 at San Francisco
- ESGF F2F 2016 at Washington
- FOSS4G 2016 at Bonn
- EGU 2016 at Vienna
- ICRC-CORDEX 2016
- Model Animation LSCE
- Talk on USGS WebEx 2016/02/18
- Paris Coding Spring 2015 at IPSL
- EGI Community Forum 2014 at Helsinki
- Prag
- CSC 2.0 Hamburg
- Vienna
- LSDMA

CHAPTER 3

License Agreement

Birdhouse is Open Source and released under the *Apache License, Version 2.0*.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index

T

V

W

X