**Creating a pipeline for DAX**

Friday, 26th May 2017

# Install DAX

First, you need to install the DAX python package on your computer. Please follow the instructions from the GITHUB page: <https://github.com/vuiis/dax> .

The easiest install consists of running the following command:

pip install https://github.com/VUIIS/dax/archive/master.zip --upgrade

It will install the current master branch from DAX project.

# How do you define a pipeline in dax?

DAX pipelines consist of two files: a spider and a processor. You will need to generate both files to finalize your pipelines. Start with the spider and finish by the processor.

The spider defines the pipeline you want to run and specifies the data you need from XNAT, the outputs you want to saved, and the command line to run. Those three steps are sum up below:

1. Downloading the data from XNAT
2. Running your command line on those data
3. Saving the results in a specific folder to be upload to XNAT using dax upload.

The processor file will manage the XNAT aspect of your pipeline. It will allow you to define which data you are looking for, what command line you want to run, and which arguments to use. A processor is linked to a spider and the command line defined in the file should be the Spider command to run.

For more information, you can follow the wiki page on <https://github.com/vuiis/dax/wiki>.

# Generate the SPIDER

## 

## **Naming conventions for Spider:**

Any spider should be name: "**Spider\_ProcName\_vX\_Y\_Z.py**" where the ProcName will refer to the process you are running.

Example: Spider\_dtiQA\_v2\_0\_0.py for dtiQA process, version X=2 Y=0 Z=0.

The assessor label on XNAT are named with the ProcName and version X like: Project-x-subject-x-session-x-scan-x-dtiQA\_v2, in this case, dtiQA\_v2 is the proctype.

To understand the versioning system X\_Y\_Z, you can follow this webpage [http://semver.org](http://semver.org/) .

* X is for major version when you make incompatible API changes.
* Y is for minor version when you add functionality in a backwards-compatible manner.
* Z is for patch version when you make backwards-compatible bug fixes.

**What does that mean?**

This means that any new spider with a version X different will be a new assessor on XNAT. Any changes on Y or Z will not create a new assessor. You will need to rerun the assessor if you want to have the new changes. You can specify the version of the spider in the processor.

Use X=0 for development and go to version X=1 for production. "1.0.0" is the first version for production mode. If no \_vX\_Y\_Z, we assume that it's "1.0.0".

Warning: All spiders should generate a PDF file and be saved on the assessor as a PDF resource.

## **Writing your Spider**

If you want to write your own spider from A to Z in python, you can follow this link:

<https://github.com/VUIIS/dax/wiki/Writing-Spider>

Otherwise, I recommend to use the AutoSpider Class that has been implemented within DAX. It allows the user to forget about the download and upload part of the spider and just write down the running part of it.

After installing DAX package, you will have a new executable on your system called **InitAutoSpider**. This script will generate a folder for your new pipeline and several files within it to help generating your AutoSpider.

You define an Auto spider using three files:

1. Inputs.csv to specify the inputs for the argument parser
2. Outputs.csv to specify the outputs to save
3. call.py (.m, .sh, …) to run your process (matlab, python, bash, ruby)

Each file will be initiate with **InitAutoSpider** and you can edit them to fit your own pipeline. See on DAX wiki for an example. See below for **InitAutoSpider** help message.

|  |
| --- |
| InitAutoSpider --help  usage: InitAutoSpider [-h] -n NAME [-v VERSION] [-d DIRECTORY]  [-e {bash,python,matlab,ruby}]  Start a folder for a new Spider using AutoSpider template and create all the  files needed.  optional arguments:  -h, --help show this help message and exit  -n NAME Name for Spider. e.g. fMRIQA  -v VERSION Spider version, format: X.Y.Z. Default: 1.0.0  -d DIRECTORY Directory where the folder will be created.  -e {bash,python,matlab,ruby}  Type of programming language for executable. |

Generating a spider becomes very easy if you know your inputs, your outputs, and the script you want to run. **InitAutoSpider** will ask if you want to specify your inputs and outputs. You can follow the instruction prompted by the script to write down those files. A genspider.sh file can then be run to create your Spider\_{ProcName}\_vX\_Y\_Z.py.

Example of InitAutoSpider:

|  |
| --- |
| XXXX@XXXX:~$ InitAutoSpider -n Test -v 2.0.0 -d . -e python  Do you want to write the spider inputs? [yes/no] yes  -- inputs.csv --  The csv file contains three values per line. The format is:  <VARIABLE\_NAME>,<TYPE>,<DESCRIPTION>  Those values are added to the argparser and copy the inputs for the process.  VARIABLE\_NAME will be used in your call.py as follow ${{VARIABLE\_NAME}}. It can  only contains underscores as specific characters.  TYPE can take the following values:  FILE : full path to the input file to copied. XNAT path or local path.  DIR : full path to a directory that will be copied.  PATH : full path for an executable or a resource. Won't be copied.  STRING : regular string  INT : regular integer  DESCRIPTION will be set in the argparser for the AutoSpider.  Examples:  t1\_file,FILE,T1 nifti file path  t2\_id,STRING,Scan ID of the T2  mat\_file,PATH,Path to matlab script  How many inputs do you want to set? [INT] 1  -- setting input 1 --  1 - What is the variable name? in\_file  1 - What type of data? [Select integer: 1.FILE / 2.DIR / 3.PATH / 4.STRING / 5.INT] 1  1 - What is the description? Input NIFTI file path  Is this input required? [yes/no] yes  Do you want to write the spider outputs? [yes/no] yes  -- outputs.csv --  The csv file contains three values per line. The format is:  <PATH>,<TYPE>,<RESOURCE>  Those values are used to copy the output to the upload folder.  PATH corresponds to the path of the resource to copy from the AutoSpider's  jobdir. E.G: DATA/\*.nii.gz  TYPE can take the following values:  FILE : full path to the input file to copied. XNAT path or local path.  DIR : full path to a directory that will be copied.  RESOURCE represents the name of the XNAT resource to store the PATH on the XNAT  assessor.It can only contains underscores as specific characters.  Examples:  DATA,DIR,DATA  DATA/alff.pdf,FILE,PDF  NII/\*.nii.gz,FILE,NII  How many outputs do you want to set? [INT] 1  -- setting output 1 --  1 - What is the output path in the jobdir? report.pdf  1 - What type of output? [Select integer: 1.FILE / 2.DIR] 1  1 - What is the resource name? PDF  Is this output required? [yes/no] yes  Folder /Users/byvernault/Test/v2.0.0 ready.  XXXX@XXXX:~$ ls Test/v2.0.0/  call.py genspider.sh inputs.csv outputs.csv test.sh |

## **Writing Inputs.csv**

The inputs.csv file will define the inputs of your spider. Each line corresponds to one input to the spider. This input will be added to the argument parser and will represent your variable name in your call.xx script. For example:

|  |
| --- |
| gad,FILE,GAD scan image file  vessel,FILE,Vessel scan image file  exe,PATH,Path to Niftypipe python script perform\_vessels2gad\_registration.py  omp,STRING,number of core use by reg\_aladin,f  working\_dir,PATH,working directory for temp files,f |

Where for example gad, vessel, exe, omp, and working\_dir are the variables names, FILE, PATH, STRING, are the types, and the last values are the description. A fourth value can be added if the inputs are not required (‘,f’).

The type for the inputs needs to be one of those values:

* FILE : full path to the input file to copied. XNAT path or local path.
* DIR : full path to a directory that will be copied.
* PATH : full path for an executable or a resource. Won't be copied.
* STRING : regular string
* INT : regular integer

## **Writing Outputs.csv**

The outputs.csv file will define the outputs of your spider. Each line corresponds to a resource on XNAT assessors. It needs to be a file in your job directory, the folder given to the –d option. For example:

|  |
| --- |
| \*\_res.nii.gz,FILE,OUTPUT  \*.txt,FILE,TRANS  vessels2gad.pdf,FILE,PDF |

Where \*\_res.nii.gz is the path for the outputs (file or folder), FILE (or FOLDER) is the type of outputs, and OUTPUT (or the third value) is the name of the resource on the assessor. A fourth value can be added if the outputs might not exist (‘,F’).

## **Writing call.EXT**

The call file representing the process to run can be written in three languages:

* matlab with the extension ‘.m’
* python with the extension ‘.py’
* bash with the extension ‘.sh’
* ruby with the extension ‘.rb’

You are free to write this file as you wish. You can get your inputs values by using the following template:

${word}

where word can be replaced with the variable names defined in the first column in your inputs.csv file.

In the case of our example above, we can use ${omp} to get the number of cores required by the user.

Example for Vessels2Gad\_Reg:

|  |
| --- |
| import os  import glob  from dax import spiders, XnatUtils  JOB\_DIR = '${temp\_dir}'  GAD\_FILE = '${gad}'  VESSEL\_FILE = '${vessel}'  NIFTYPIPE\_EXE = '${exe}'  OPENMP\_CORE = '${omp}'  WORKING\_DIR = '${working\_dir}'  EXE\_CMD = '''{exe\_path} \  --ref {gad} \  --vessels {vessels} \  --output {output} \  --no\_qsub \  --remove\_tmp \  --n\_procs 1 \  {omp} \  {wdir}'''  OMP = '''--openmp\_core {number\_core}'''  WDIR = '''--working\_dir '{working\_dir}' '''  def main():  """ Main function."""  if WORKING\_DIR != 'None' and not os.path.exists(WORKING\_DIR):  os.makedirs(WORKING\_DIR)  if XnatUtils.executable\_exists(NIFTYPIPE\_EXE):  \_omp = ''  \_wd = ''  if OPENMP\_CORE is not None and OPENMP\_CORE != 'None':  \_omp = OMP.format(number\_core=OPENMP\_CORE)  if WORKING\_DIR is not None and WORKING\_DIR != 'None':  \_wd = WDIR.format(working\_dir=WORKING\_DIR)  cmd = EXE\_CMD.format(exe\_path=NIFTYPIPE\_EXE,  gad=GAD\_FILE,  vessels=VESSEL\_FILE,  output=JOB\_DIR,  omp=\_omp,  wdir=\_wd)  os.system(cmd)  make\_pdf()  else:  raise Exception("Error: executable %s not found" % (NIFTYPIPE\_EXE))  def make\_pdf():  """Method to make the PDF for the spider. """  # PDF pages:  pdf = os.path.join(JOB\_DIR, 'vessels2gad.pdf')  # Images outputs:  images = [GAD\_FILE]  images.extend(glob.glob(os.path.join(JOB\_DIR, '\*.nii.gz')))  labels = dict()  for ind in range(len(images)):  if ind == 0:  labels[str(ind)] = 'Reference'  else:  labels[str(ind)] = 'Vessel {0}'.format(str(ind))  spiders.plot\_images(pdf, 1, images, 'Vessel to Gad Registration Pipeline',  image\_labels=labels)  if \_\_name\_\_ == '\_\_main\_\_':  main() |

# TestING YOUR SPIDER

In the folder generated by **InitAutoSpider**, you will find a test.sh file. A good practice is to edit that file and write down your command line for your spider to test it. Anybody that wants to test your spider can then see / run this file.

# GEnerating the processor Yaml

When you finish generating your spider and you tested it, you can now generate your processor file. This file can be a regular processor developed in python but it will require a lof of programing from the user point of you or it can be a YAML processor. You can read more about the YAML format by following the link on Wikipedia <https://en.wikipedia.org/wiki/YAML>.

An assessor on XNAT correspond to one task to run on your cluster. Those tasks will be launched on the cluster by the processor which will compute the command line to run in the PBS file (or equivalent file).

For example, the Vessels2Gad\_Reg processor file:

|  |
| --- |
| ---  inputs:  default:  spiderpath: /home/Spider\_Vessels2Gad\_Reg\_v1\_0\_0.py  working\_dir:  nipype\_exe: perform\_vessels2gad\_registration.py  ppn: 4  xnat:  scans:  - scan1:  types: 1mm Ax FSPG + Gad  resources:  - resource: NIFTI  varname: gad  - scan2:  types: Ax Inhance 3D MR\*  resources:  - resource: NIFTI  varname: vessel  vars:  # id: vesselsids  # any scan attrs: the variable name  #assessors:  # - assessor1:  # proctypes: Gif\_Parcellation\_v1  # needqc: False  # resources:  # - resource1: SEG  # name: parcellation  # vars:  # procversion: gifversion  command: python {spiderpath} --exe {nipype\_exe} --gad {gad} --vessel {vessel} --omp {ppn}  attrs:  # suffix:  xsitype: proc:genProcData  walltime: 01:00:00  memory: 4048  ppn: 4  type: scan # session  scan\_nb: scan2 # not used if session |

In this YAML file, there are several levels. First level consists of three tags:

* ‘inputs’ that will define the inputs for the command line
* ‘command’ that represent the command to run in your job generated by the processor
* ‘attrs’ that are predefined values for a processor. For example, the walltime or the memory for the job on the cluster.

Then each level can be divided into other layers. ‘Inputs’ are divided into two:

* ‘default’ inputs that are not related to XNAT. The argname should be the argparser option name in your spider. You can add it as a tag in your command line if it’s required for the spider to run. If not, without a value, it won’t add it to the command line. For example above, ‘working\_dir’ is not set in the command and won’t be use except if someone add a value. In that case, the processor will add to your command line: --working\_dir {value}
* ‘xnat’ inputs that are related to XNAT and are divided into two groups:
  + ‘scans’
  + ‘assessors’

Either you can be using scans resources, or assessors’ resources, or even both. Each group will be a list of objects define by a number (scan1, scan2, scan3, or assessor1, assessor2, assessor3) and some other tags:

* ‘types’ or ‘proctypes’ for the type of the object (scan or assessor)
* ‘needqc’ could be False or True to specify if you want to wait for a good qc before using those scans or assessors.
* ‘resources’ which is a list of resources associated to the object. You just need to specify the resource you want to look for and the variable you will set it to.
* ‘vars’ where you can access any other tags that you want to add to your command line (attributes on the object: variable name)

The command should represent the spider command you want to run. Each attribute specified in {} marks should correspond to an input specified in inputs. You don’t have to specify all of them especially if there might be no value for it.

The ‘attrs’ part won’t have more tags in it. ‘Type’ represents the type of processor, either on a scan or on a session. If it’s on a scan, you need to have the ‘scan\_nb’ tag to specify which scan we are looking for to generate those assessors.

Warning: If an input defines in ‘default’ does not have a value, it won’t put it in the command line except if you specify its variable name in the ‘command’.

If you need to look for examples, please look at the ucl-processing on cmiclab.