DAX Documentation

Release 0.11.5

Benjamin Yvernault, Brian Boyd, Stephen Damon, Andrew Plassa

Nov 11, 2019

Contents

1 Installation **3**

1.1 DAX Installation . . . 3
1.1.1 Table of Contents: . . . 3
1.1.2 For Linux user . . . 4
1.1.3 For Mac user . . . 4
1.1.4 Warnings . . . 4
1.1.5 Install for Linux user . . . 5
1.1.6 Install for Mac user . . . 6
1.1.7 No Sudo access . . . 7
1.2 Installation of fs:fsData and proc:genProcData . . . 8
1.2.1 On XNAT VM: . . . 8
1.2.2 ON XNAT webapp: . . . 9
1.3 Source Documentation . . . 10
1.3.1 `dax` – Root package . . . 10
1.3.2 `dax.task` – Task class . . . 10
1.3.3 `dax.spiders` – Spider class . . . 18
1.3.4 `dax.processors` – Processor class . . . 25
1.3.5 `dax.log` – Logging utility . . . 28
1.3.6 `dax.bin` – Responsible for launching, building and updating a Task . . . 28
1.3.7 `dax.XnatUtils` – Collection of utilities for upload/download and general access . . . 30
1.4 DAX Manager . . . 37
1.4.1 Table of Contents: . . . 37
1.5 Contributors . . . 39
1.6 How To Contribute . . . 39
1.7 FAQ . . . 39
1.8 DAX Processors . . . 40
1.8.1 About . . . 40
1.8.2 Processor Repos . . . 40
1.8.3 Overview . . . 41
1.8.4 A “Simple” Example . . . 41
1.8.5 Parts of the Processor YAML . . . 41
1.8.6 inputs . . . 42
1.8.7 Versioning . . . 44
1.8.8 Notes on Singularity run options . . . 44

Python Module Index **45**

Index 47

DAX is Distributed Automation for XNAT

DAX allows you to:

- store analyzed imaging data on XNAT (datatypes)
- extract information from XNAT via scripts (Xnat_tools)
- run pipelines on your data in XNAT via a cluster (processors)

CHAPTER 1

Installation

Install the latest release with pip:

```
pip install dax
```

Contents:

1.1 DAX Installation

1.1.1 Table of Contents:

1. *Requirements*
2. *For Linux user*
3. *For Mac user*
4. *Warnings*
5. *Install dax*
6. *For Linux user*
7. *For Mac user*
8. *No Sudo Access*
9. *Verify the installation*
10. *Programming in python*

Requirements

Requirements for DAX: * Linux or MacOS operating system (has not been tested on windows yet). * Python installed with version 2.7.X * git or pip installed

To check that your python version is 2.7.X:

```
python --version
```

1.1.2 For Linux user

To install pip if you want/don't have it (optional):

```
easy_install pip
```

To install git if you don't have it:

```
apt-get install git
```

1.1.3 For Mac user

If svn command doesn't exist on you mac, install xcode from the Apple Store. Run it and go to Xcode -> Preferences -> Downloads -> Command Line Tools -> Install. Now, you can use svn.

A quick way to check the installation of Xcode and command line developer is to run:

```
xcode-select --install
```

If it asks: "install requested for command line developer tools", do the install.

To install pip, run:

```
sudo easy_install pip
```

If you don't have easy_install, follow the instructions on this link https://pypi.python.org/pypi/setuptools .

To Install git: on this link http://git-scm.com/downloads , click on the Mac Os X button to download the package and install it.

1.1.4 Warnings

Before starting with the different steps, if you see a 'Permission denied' while trying to install the libraries, add sudo in front of the command line. It will ask for your password. This will use the sudo access (http://en.wikipedia.org/wiki/Sudo) when running the command line and you will have the permission to install packages everywhere on your computer.

If you don't have sudo access on your computer, follow the section No Sudo access.

Previously all of the commonly used CLI tools (XnatSwitchProcessStatus, Xnatupload, Xnatdownload, and Xnatinfo etc) were stored under masimatlab. These versions are no longer maintained and the new versions are part of DAX. If you get errors that your versions don't work, you should check your PATH variable

```
echo $PATH
```

If you see a reference to masimatlab/trunk/xnatspiders/Xnat_tools, you should remove this from your path so versions do not conflict. When you install DAX, your environement is set for the new versions (but does not make any changes to the old versions so you need to do this manually).

If you get any nasty traceback errors, you may be missing a required module package. Below is an example:

```
Traceback (most recent call last):
File "/usr/local/bin/fsdownload", line 14, in <module>
  from dax import XnatUtils
File "/Library/Python/2.7/site-packages/dax/__init__.py", line 3, in <module>
  from .launcher import Launcher
File "/Library/Python/2.7/site-packages/dax/launcher.py", line 12, in <module>
  import processors
File "/Library/Python/2.7/site-packages/dax/processors.py", line 4, in <module>
  import task
File "/Library/Python/2.7/site-packages/dax/task.py", line 9, in <module>
  import XnatUtils, bin
File "/Library/Python/2.7/site-packages/dax/bin.py", line 8, in <module>
  import redcap
File "/Library/Python/2.7/site-packages/redcap/__init__.py", line 19, in <module>
  from .project import Project
File "/Library/Python/2.7/site-packages/redcap/project.py", line 10, in <module>
  from .request import RCRequest, RedcapError, RequestException
File "/Library/Python/2.7/site-packages/redcap/request.py", line 18, in <module>
  from requests import post, RequestException
ImportError: No module named requests
```

In this case, the "requests" package is missing. To install, just run "sudo pip install requests". If you get other import errors, they can generally be fixed by running sudo pip install where package name is the last word in the ImportError line.

Install DAX

1.1.5 Install for Linux user

- Install dax (Distributed Automation for XNAT) package:

With pip:

```
sudo pip install dax
#or
pip install https://github.com/VUIIS/dax/archive/master.zip --upgrade
#to get the last version of dax and not the version on pip
```

OR with git:

```
git clone git://github.com/VUIIS/dax
cd dax
sudo python setup.py install
```

- add the XNAT variables to your file ~/.xnat_profile:

Run these commands:

```
echo "export XNAT_USER=XXXXXXXX" >> ~/.xnat_profile
echo "export XNAT_PASS=XXXXXXXX" >> ~/.xnat_profile
echo "export XNAT_HOST=http://XXXXXXXXXXX" >> ~/.xnat_profile
```

Replace the XXXXX by your personal information.

- Last step, you need to check that the file .xnat_profile is called in your .bash_profile.

To do so, use the following command to see the content of your file .bash_profile:

```
cat ~/.bash_profile
```

If you don't see the line "source ~/.xnat_profile" or ". ~/.xnat_profile", your configuration file is not linked to your bash_profile.

To do so, run:

```
echo "source ~/.xnat_profile" >> ~/.bash_profile
```

- Apply the changes:

Run this command:

```
. ~/.xnat_profile
```

You are ready to go.

1.1.6 Install for Mac user

- Install dax (Distributed Automation for XNAT) package:

With pip:

```
sudo pip install dax
# or
pip install https://github.com/VUIIS/dax/archive/master.zip --upgrade
#to get the last version of dax and not the version on pip
```

OR with git:

```
git clone git://github.com/VUIIS/dax
cd dax
sudo python setup.py install
```

- add the XNAT variables to your file ~/.xnat_profile:

Run these commands:

```
echo "export XNAT_USER=XXXXXXXX" >> ~/.xnat_profile
echo "export XNAT_PASS=XXXXXXXX" >> ~/.xnat_profile
echo "export XNAT_HOST=http://xnat.vanderbilt.edu:8080/xnat" >> ~/.xnat_profile
```

Replace the XXXXX by your personal information.

- Last step, you need to check that the file .xnat_profile is called in your .bash_profile.

To do so, use the following command to see the content of your file .bash_profile:

```
cat ~/.bash_profile
```

If you don't see the line "source ~/.xnat_profile" or ". ~/.xnat_profile", your configuration file is not linked to your bash_profile.

To do so, run:

```
echo "source ~/.xnat_profile" >> ~/.bash_profile
```

- Apply the changes:

Run this command:

```
. ~/.xnat_profile
```

You are ready to go.

1.1.7 No Sudo access

If you are not a sudoer on your computer (Linux or MacOS), you can still install dax locally. You need to use git to clone the dax repository and install it locally. Follow the steps below to process with the installation:

```
git clone git://github.com/VUIIS/dax
cd dax
python setup.py install --user
```

You will need to add the local folder of dax/Xnat_tools executables to your PATH:

- For Linux: echo "export PATH=/.local/bin:$PATH">>/.bashrc
- For MacOS: echo "export PATH=~/Library/Python/2.7/bin/:$PATH" >> ~/.profile

If you don't see a line like "source ~/.profile" or ". ~/.profile" (same for .bashrc), your configuration file is not linked to your bash_profile. To do so, run:

```
echo "source ~/.profile" >> ~/.bash_profile
# or for bashrc
echo "source ~/.bashrc" >> ~/.bash_profile
```

Run your configuration file to apply the changes:

```
. ~/.profile
#or for bashrc
. ~/.bashrc
```

Verify the installation

If you want to be sure everything is installed, you can check running those commands:

```
XXXXXXXXX$ python
Python 2.7.1 (r271:86832, Jul 31 2011, 19:30:53)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>import httplib2
>>>import lxml
>>>import pyxnat
>>>import redcap
>>>import dax
```

If you don't have any error, the python packages are all installed properly.

Now you can verify your logins by running:

```
XnatCheckLogin
```

If you see '–>Good login', you are good to go.

You are ready to use the Xnat_tools, dax executables or the spiders.

Programming in python

All the Spiders/DAX package/Xnat_tools are written in python.

''Where can I learn how to program in python?'' If you want to learn how to program in python, here are several links that could help you: * http://www.learnpython.org * https://www.python.org * http://stackoverflow.com * http://google.com

''Where can I program in python?''

- You can use any text Editor that you like to program in python.
- There is an extension for Eclipse for python development called pydev. Here is the link to install pydev on Eclipse and it explains how to create a script : http://www.rose-hulman.edu/class/csse/resources/Eclipse/eclipse-python-configuration.htm
- Atom (https://atom.io) is a nice editor developed by the team who created github.

1.2 Installation of fs:fsData and proc:genProcData

Prerequisites:

- install an XNAT instance https://wiki.xnat.org/documentation/getting-started-with-xnat

1.2.1 On XNAT VM:

1) Make a BACKUP of your $XNAT_HOME, postgres db, and tomcat deployment
2) Stop tomcat
3) Copy plugins to XNAT

Copy the files dax-plugin-fsData-1.0.0.jar and dax-plugin-genProcData-1.4.0.jar to ${XNAT_HOME}/plugins

The jar_files folder is located in dax package at the path dax/dax/xnat_datatypes/jar_files. You can download the files from github repository: https://github.com/VUIIS/dax .

4) Start tomcat and confirm that plugins are installed

1.2.2 ON XNAT webapp:

1) Log onto XNAT as admin
2) click Administer > Data types
3) click Setup Additional Data Type
4) for fs:fsData

4.a) select fs:fsData and valid without adding anything at first.

4.b) Come back to the new types and edit the fields:

```
enter "FreeSurfer" in both Singular Name and Plural Name field
enter "FS" in Code field
```

4.c) Edit the "Available Report Actions" by adding delete if you want to be able to delete assessor with the following values:

```
Remove Name: delete
Display Name: Delete
Grouping:
Image: delete.gif
Popup:
Secure Access: delete
Feature:
Additional Parameters:
Sequence: 4
```

4.d) click submit and then accept defaults for subsequent screens

5) for proc:genProcData

5.a) select proc:genProcData and valid without adding anything at first.

5.b) Come back to the new types and edit the fields:

```
enter "Processing" in both Singular Name and Plural Name field
enter "Proc" in Code field
```

5.c) Edit the "Available Report Actions" by adding delete if you want to be able to delete assessor with the following values:

```
Remove Name: delete
Display Name: Delete
Grouping:
Image: delete.gif
Popup:
Secure Access: delete
Feature:
Additional Parameters:
Sequence: 4
```

5.d) click submit and then accept defaults for subsequent screens

You are now ready to use the two assessors fs:fsData and proc:genProcData

1.3 Source Documentation

1.3.1 dax – Root package

1.3.2 dax.task – Task class

Task object to generate / manage assessors and cluster.

class dax.task.**Task**(*processor*, *assessor*, *upload_dir*)

Class Task to generate/manage the assessor with the cluster

check_date()

Sets the job created date if the assessor was not made through dax_build

Returns Returns if get_createdate() is != '', sets date otherwise

check_job_usage()

The task has now finished, get the amount of memory used, the amount of walltime used, the jobid of the process, the node the process ran on, and when it started from the scheduler. Set these values on XNAT

Returns None

check_running(*jobid=None*)

Check to see if a job specified by the scheduler ID is still running

Parameters **jobid** – The ID of the job in question assigned by the scheduler.

Returns A String of JOB_RUNNING if the job is running or enqueued and JOB_FAILED if the ready flag (see read_flag_exists) does not exist in the assessor label folder in the upload directory.

commands(*jobdir*)

Call the get_cmds method of the class Processor.

Parameters **jobdir** – Fully qualified path where the job will run on the node. Note that this is likely to start with /tmp on most grids.

Returns A string that makes a command line call to a spider with all args.

get_createdate()

Get the date an assessor was created

Returns String of the date the assessor was created in "%Y-%m-%d" format

get_job_status(*jobid=None*)

Get the status of a job given its jobid as assigned by the scheduler

Parameters **jobid** – job id assigned by the scheduler

Returns string from call to cluster.job_status or UNKNOWN.

get_job_usage()

Get the amount of memory used, the amount of walltime used, the jobid of the process, the node the process ran on, and when it started from the scheduler.

Returns List of strings. Memory used, walltime used, jobid, node used, and start date

get_jobid()
Get the jobid of an assessor as stored on XNAT

Returns string of the jobid

get_jobnode()
Gets the node that a process ran on

Returns String identifying the node that a job ran on

get_jobstartdate()
Get the date that the job started

Returns String of the date that the job started in "%Y-%m-%d" format

get_memused()
Get the amount of memory used for a process

Returns String of how much memory was used

get_processor_name()
Get the name of the Processor for the Task.

Returns String of the Processor name.

get_processor_version()
Get the version of the Processor.

Returns String of the Processor version.

get_qcstatus()
Get the qcstatus of the assessor

Returns A string of the qcstatus for the assessor if it exists. If it does not, it returns DOES_NOT_EXIST. The else case returns an UNKNOWN xsiType with the xsiType of the assessor as stored on XNAT.

get_status()
Get the procstatus of an assessor

Returns The string of the procstatus of the assessor. DOES_NOT_EXIST if the assessor does not exist

get_statuses()
Get the procstatus, qcstatus, and job id of an assessor

Returns Serially ordered strings of the assessor procstatus, qcstatus, then jobid.

get_walltime()
Get the amount of walltime used for a process

Returns String of how much walltime was used for a process

is_open()

Check to see if a task is still in "Open" status as defined in OPEN_STATUS_LIST.

Returns True if the Task is open. False if it is not open

launch(*jobdir*, *job_email=None*, *job_email_options='a'*, *xnat_host=None*, *writeonly=False*, *pbsdir=None*, *force_no_qsub=False*)
Method to launch a job on the grid

Parameters

- **jobdir** – absolute path where the data will be stored on the node
- **job_email** – who to email if the job fails
- **job_email_options** – grid-specific job email options (e.g., fails, starts, exits etc)
- **xnat_host** – set the XNAT_HOST in the PBS job
- **writeonly** – write the job files without submitting them
- **pbsdir** – folder to store the pbs file
- **force_no_qsub** – run the job locally on the computer (serial mode)

Raises cluster.ClusterLaunchException if the jobid is 0 or empty as returned by pbs.submit() method

Returns True if the job failed

outlog_path()
Method to return the path of outlog file for the job

Returns A string that is the absolute path to the OUTLOG file.

pbs_path(*writeonly=False*, *pbsdir=None*)
Method to return the path of the PBS file for the job

Parameters

- **writeonly** – write the job files without submitting them in TRASH
- **pbsdir** – folder to store the pbs file

Returns A string that is the absolute path to the PBS file that will be submitted to the scheduler for execution.

ready_flag_exists()
Method to see if the flag file <UPLOAD_DIR>/<ASSESSOR_LABEL>/READY_TO_UPLOAD.txt exists

Returns True if the file exists. False if the file does not exist.

reproc_processing()
If the procstatus of an assessor is REPROC on XNAT, rerun the assessor.

Returns None

set_createdate(*date_str*)
Set the date of the assessor creation to user passed value

Parameters **date_str** – String of the date in "%Y-%m-%d" format

Returns String of today's date in "%Y-%m-%d" format

set_createdate_today()
Set the date of the assessor creation to today

Returns String of todays date in "%Y-%m-%d" format

set_jobid(*jobid*)
Set the job ID of the assessor on XNAT

Parameters **jobid** – The ID of the process assigned by the grid scheduler

Returns None

set_jobnode(*jobnode*)
Set the value of the the node that the process ran on on the grid

Parameters **jobnode** – String identifying the node the job ran on

Returns None

set_jobstartdate(*date_str*)

Set the date that the job started on the grid based on user passed value

Parameters **date_str** – Datestring in the format "%Y-%m-%d" to set the job starte date to

Returns None

set_jobstartdate_today()
Set the date that the job started on the grid to today

Returns call to set_jobstartdate with today's date

set_launch(*jobid*)
Set the date that the job started and its associated ID on XNAT. Additionally, set the procstatus to JOB_RUNNING

Parameters **jobid** – The ID of the process assigned by the grid scheduler

Returns None

set_memused(*memused*)
Set the amount of memory used for a process

Parameters **memused** – String denoting the amount of memory used

Returns None

set_proc_and_qc_status(*procstatus*, *qcstatus*)
Set the procstatus and qcstatus of the assessor

Parameters

- **procstatus** – String to set the procstatus of the assessor to
- **qcstatus** – String to set the qcstatus of the assessor to

Returns None

set_qcstatus(*qcstatus*)
Set the qcstatus of the assessor

Parameters **qcstatus** – String to set the qcstatus to

Returns None

set_status(*status*)
Set the procstatus of an assessor on XNAT

Parameters **status** – String to set the procstatus of the assessor to

Returns None

set_walltime(*walltime*)
Set the value of walltime used for an assessor on XNAT

Parameters **walltime** – String denoting how much time was used running the process.

Returns None

undo_processing()

Unset the job ID, memory used, walltime, and jobnode information for the assessor on XNAT

Except pyxnat.core.errors.DatabaseError when attempting to delete a resource

Returns None

update_status()
Update the satus of a Task object.

Returns the "new" status (updated) of the Task.

class dax.task.**ClusterTask**(*assr_label*, *upload_dir*, *diskq*)
Class Task to generate/manage the assessor with the cluster

batch_path()
Method to return the path of the PBS file for the job

Returns A string that is the absolute path to the PBS file that will be submitted to the scheduler for execution.

build_commands()
Call the get_cmds method of the class Processor.

Parameters **jobdir** – Fully qualified path where the job will run on the node. Note that this is likely to start with /tmp on most grids.

Returns A string that makes a command line call to a spider with all args.

build_task()
Method to build a job

check_date()
Sets the job created date if the assessor was not made via dax_build

check_job_usage()

The task has now finished, get the amount of memory used, the amount of walltime used, the jobid of the process, the node the process ran on, and when it started from the scheduler. Set these values locally

Returns None

check_running()
Check to see if a job specified by the scheduler ID is still running

Parameters **jobid** – The ID of the job in question assigned by the scheduler.

Returns A String of JOB_RUNNING if the job is running or enqueued and JOB_FAILED if the ready flag (see read_flag_exists) does not exist in the assessor label folder in the upload directory.

commands(*jobdir*)
Call the get_cmds method of the class Processor.

Parameters **jobdir** – Fully qualified path where the job will run on the node. Note that this is likely to start with /tmp on most grids.

Returns A string that makes a command line call to a spider with all args.

get_createdate()
Get the date an assessor was created

Returns String of the date the assessor was created in "%Y-%m-%d" format

get_job_status()
Get the status of a job given its jobid as assigned by the scheduler

Parameters **jobid** – job id assigned by the scheduler

Returns string from call to cluster.job_status or UNKNOWN.

get_job_usage()

Get the amount of memory used, the amount of walltime used, the jobid of the process, the node the process ran on, and when it started from the scheduler.

Returns List of strings. Memory used, walltime used, jobid, node used, and start date

get_jobid()
Get the jobid of an assessor as stored in local cache

Returns string of the jobid

get_jobnode()
Gets the node that a process ran on

Returns String identifying the node that a job ran on

get_jobstartdate()
Get the date that the job started

Returns String of the date that the job started in "%Y-%m-%d" format

get_memused()
Get the amount of memory used for a process

Returns String of how much memory was used

get_processor_name()
Get the name of the Processor for the Task.

Returns String of the Processor name.

get_processor_version()
Get the version of the Processor.

Returns String of the Processor version.

get_qcstatus()
Get the qcstatus

get_status()
Get the procstatus

Returns The string of the procstatus

get_statuses()
Get the procstatus, qcstatus, and job id of an assessor

get_walltime()
Get the amount of walltime used for a process

Returns String of how much walltime was used for a process

is_open()

Check to see if a task is still in "Open" status as defined in OPEN_STATUS_LIST.

Returns True if the Task is open. False if it is not open

launch(*force_no_qsub=False*)
Method to launch a job on the grid

Raises cluster.ClusterLaunchException if the jobid is 0 or empty as returned by pbs.submit() method

Returns True if the job failed

outlog_path()
Method to return the path of outlog file for the job

Returns A string that is the absolute path to the OUTLOG file.

reproc_processing()

Raises NotImplementedError

Returns None

set_createdate(*date_str*)
Set the date of the assessor creation to user passed value

Parameters **date_str** – String of the date in "%Y-%m-%d" format

Returns String of today's date in "%Y-%m-%d" format

set_createdate_today()
Set the date of the assessor creation to today

Returns String of todays date in "%Y-%m-%d" format

set_jobid(*jobid*)
Set the job ID of the assessor

Parameters **jobid** – The ID of the process assigned by the grid scheduler

Returns None

set_jobnode(*jobnode*)
Set the value of the the node that the process ran on on the grid

Parameters **jobnode** – String identifying the node the job ran on

Returns None

set_jobstartdate(*date_str*)

Set the date that the job started on the grid based on user passed value

Parameters **date_str** – Datestring in the format "%Y-%m-%d" to set the job starte date to

Returns None

set_launch(*jobid*)
Set the date that the job started and its associated ID. Additionally, set the procstatus to JOB_RUNNING

Parameters **jobid** – The ID of the process assigned by the grid scheduler

Returns None

set_memused(*memused*)
Set the amount of memory used for a process

Parameters **memused** – String denoting the amount of memory used

Returns None

set_proc_and_qc_status(*procstatus*, *qcstatus*)
Set the procstatus and qcstatus of the assessor

set_qcstatus(*qcstatus*)
Set the qcstatus of the assessor

Parameters **qcstatus** – String to set the qcstatus to

Returns None

set_status(*status*)
Set the procstatus of an assessor on XNAT

Parameters **status** – String to set the procstatus of the assessor to

Returns None

set_walltime(*walltime*)
Set the value of walltime used for an assessor

Parameters **walltime** – String denoting how much time was used running the process.

Returns None

undo_processing()

Unset the job ID, memory used, walltime, and jobnode information for the assessor on XNAT

Except pyxnat.core.errors.DatabaseError when attempting to delete a resource

Returns None

update_status()
Update the status of a Cluster Task object.

Returns the "new" status (updated) of the Task.

upload_outlog_dir()
Method to return the path of outlog file for the job

Returns A string that is the absolute path to the OUTLOG file.

upload_pbs_dir()
Method to return the path of dir for the PBS

Returns A string that is the directory path for the PBS dir

class dax.task.**XnatTask**(*processor*, *assessor*, *upload_dir*, *diskq*)
Class Task to generate/manage the assessor with the cluster

batch_path()
Method to return the path of the PBS file for the job

Returns A string that is the absolute path to the PBS file that will be submitted to the scheduler for execution.

build_commands(*assr*, *jobdir*)
Call the build_cmds method of the class Processor.

Parameters **jobdir** – Fully qualified path where the job will run on the node. Note that this is likely to start with /tmp on most grids.

Returns A string that makes a command line call to a spider with all args.

build_task(*assr*, *jobdir*, *job_email=None*, *job_email_options='a'*, *xnat_host=None*)
Method to build a job

check_job_usage()

The task has now finished, get the amount of memory used, the amount of walltime used, the jobid of the process, the node the process ran on, and when it started from the scheduler. Set these values on XNAT

Returns None

check_running()
Check to see if a job specified by the scheduler ID is still running

Parameters **jobid** – The ID of the job in question assigned by the scheduler.

Returns A String of JOB_RUNNING if the job is running or enqueued and JOB_FAILED if the ready flag (see read_flag_exists) does not exist in the assessor label folder in the upload directory.

get_job_status()
Get the status of a job given its jobid as assigned by the scheduler

Parameters **jobid** – job id assigned by the scheduler

Returns string from call to cluster.job_status or UNKNOWN.

launch()
Method to launch a job on the grid

outlog_path()
Method to return the path of outlog file for the job

Returns A string that is the absolute path to the OUTLOG file.

set_launch(*jobid*)
Set the date that the job started and its associated ID on XNAT. Additionally, set the procstatus to JOB_RUNNING

Parameters **jobid** – The ID of the process assigned by the grid scheduler

Returns None

update_status()
Update the satus of an XNAT Task object.

Returns the "new" status (updated) of the Task.

1.3.3 dax.spiders – Spider class

Title: spiders.py Author: Benjamin Yvernault contact: b.yvernault@ucl.ac.uk Purpose:

Spider base class and class for Scan and Session spider Spider name must be: Spider_[name]_v[version].py Utils for spiders

class dax.spiders.**Spider**(*spider_path*, *jobdir*, *xnat_project*, *xnat_subject*, *xnat_session*, *xnat_host=None*, *xnat_user=None*, *xnat_pass=None*, *suffix=''*, *subdir=True*, *skip_finish=False*)
Base class for spider

check_executable(*executable*, *name*, *version_opt='–version'*)
Method to check the executable.

Parameters

- **executable** – executable path
- **name** – name of Executable

Returns Complete path to the executable

define_spider_process_handler()

Define the SpiderProcessHandler so the file(s) and PDF are checked for existence and uploaded to the upload_dir accordingly.

Implemented in derived classes.

Raises NotImplementedError() if not overridden.

Returns None

download(*obj_label*, *resource*, *folder*)

Return a python list of the files downloaded for the scan's resource

example:

download(scan_id, "DICOM", "/Users/test")

or download(assessor_label, "DATA", "/Users/test")

Parameters

- **obj_label** – xnat object label (scan ID or assessor label)
- **resource** – folder name under the xnat object
- **folder** – download directory

Returns python list of files downloaded

download_inputs()
Download inputs data from XNAT define in self.inputs.

self.inputs = list of data dictionary with keys define below keys:

'type': 'scan' or 'assessor' or 'subject' or 'project' or 'session' 'label': label on XNAT (not needed for session/subject/project) 'resource': name of resource to download or list of resources 'dir': directory to download files into (optional)

- for assessor only if not giving the label but just proctype 'scan': id of the scan for the assessor (if None, sessionAssessor)

self.data = list of dictionary with keys define below: 'label': label on XNAT 'files': list of files downloaded

set self.data, a python list of the data downloaded.

end()
Finish the script by sending the end of script flag and cleaning folder

Parameters **jobdir** – directory for the spider

Returns None

finish()
Method to copy the results in the Spider Results folder dax.RESULTS_DIR Implemented in derived class objects.

Raises NotImplementedError if not overriden by user

Returns None

static get_data_dict(*otype*, *label*, *resource*, *directory*, *scan=None*)
Create a data_dict for self.inputs from user need.

get_exe_version(*executable*, *version_opt='–version'*)
Method to check the executable.

Parameters

- **executable** – executable to run
- **version_opt** – options to get the version of the executable

Returns version

get_xnat_dict(*data_dict*, *resource*)
Return a OrderedDict dictionary with XNAT information.

keys: project subject experiment scan resource assessor out/resource (for assessor)

has_spider_handler()

Check to see that the SpiderProcessHandler is defined. If it is not, call define_spider_process_handler

Returns None

merge_pdf_pages(*pdf_pages*, *pdf_final*)
Concatenate all pdf pages in the list into a final pdf.

See function at the end of the file.

plot_images_page(*pdf_path*, *page_index*, *nii_images*, *title*, *image_labels*, *slices=None*, *cmap='gray'*, *vmins=None*, *vmaxs=None*, *volume_ind=None*, *orient='ax'*)
Plot list of images (3D-4D) on a figure (PDF page).

See function at the end of the file.

plot_stats_page(*pdf_path, page_index, stats_dict, title, tables_number=3, columns_header=['Header', 'Value'], limit_size_text_column1=30, limit_size_text_column2=10*)
Generate pdf report of stats information from a csv/txt.

See function at the end of the file.

pre_run()
Pre-Run method to download and organise inputs for the pipeline Implemented in derived class objects.

Raises NotImplementedError if not overridden.

Returns None

print_args(*argument_parse*)
print arguments given to the Spider

Parameters **argument_parse** – argument parser

Returns None

print_end()
Last print statement to give the time and date at the end of the spider

Returns None

print_err(*err_message*)
Print error message using time writer

Parameters **err_message** – error message displayed for the user

Returns None

print_info(*author*, *email*)
Print information on the spider using time writer

Parameters

- **author** – author of the spider
- **email** – email of the author

Returns None

print_init(*argument_parse*, *author*, *email*)
Print a message to display information on the init parameters, author, email, and arguments using time writer

Parameters

- **argument_parse** – argument parser
- **author** – author of the spider
- **email** – email of the author

Returns None

print_msg(*message*)
Print message using time writer

Parameters **message** – string displayed for the user

Returns None

run()
Runs the "core" or "image processing process" of the pipeline Implemented in derived class objects.

Raises NotImplementedError if not overridden.

Returns None

run_cmd_args()
Run a command line via os.system() with arguments set in self.cmd_args

cmd_args is a dictionary: exe: executable to use (matlab, python, sh) template: string defining the command line with argument args: dictionary with:

key = argument value = value to set

filename: name for the file if written into a file (optional)

Returns True if succeeded, False otherwise

run_system_cmd(*cmd*)
Run system command line via os.system()

Parameters **cmd** – command to run

Returns True if succeeded, False otherwise

select_obj(*intf*, *obj_label*, *resource*)
Select scan or assessor resource

Parameters

- **obj_label** – xnat object label (scan ID or assessor label)
- **resource** – folder name under the xnat object

return pyxnat object

static select_str(*xnat_dict*)
Return string for pyxnat to select object from python dict

Parameters **tmp_dict** – python dictionary with xnat information keys = ["project", "subject", "experiement", "scan", "resource"]

or

keys = ["project", "subject", "experiement", "assessor", "out/resource"]

Return string string path to select pyxnat object

upload(*fpath*, *resource*)
Upload files to the queue on the cluster to be upload to XNAT by DAX E.g: spider.upload("/Users/DATA/", "DATA")

spider.upload("/Users/stats_dir/statistical_measures.txt", "STATS")

Parameters

- **fpath** – path to the folder/file to be uploaded
- **resource** – folder name to upload to on the assessor

Raises ValueError if the file to upload does not exist

Returns None

upload_dict(*files_dict*)

upload files to the queue on the cluster to be upload to XNAT by DAX following the files python dictionary: {resource_name : fpath}

E.g: fdict = {"DATA" ["/Users/DATA/", "PDF": "/Users/PDF/report.pdf"}] spider.upload_dict(fdict)

Parameters **files_dict** – python dictionary containing the pair resource/fpath

Raises ValueError if the filepath is not a string or a list

Returns None

class dax.spiders.**ScanSpider**(*spider_path*, *jobdir*, *xnat_project*, *xnat_subject*, *xnat_session*, *xnat_scan*, *xnat_host=None*, *xnat_user=None*, *xnat_pass=None*, *suffix=''*, *subdir=True*, *skip_finish=False*)
Derived class for scan-spider

define_spider_process_handler()

Define the SpiderProcessHandler for the end of scan spider using the init attributes about XNAT

Returns None

finish()
Method to copy the results in the Spider Results folder dax.RESULTS_DIR Implemented in derived class objects.

Raises NotImplementedError if not overriden by user

Returns None

pre_run()
Pre-Run method to download and organise inputs for the pipeline Implemented in derived class objects.

Raises NotImplementedError if not overridden.

Returns None

run()
Runs the "core" or "image processing process" of the pipeline Implemented in derived class objects.

Raises NotImplementedError if not overridden.

Returns None

class dax.spiders.**SessionSpider**(*spider_path*, *jobdir*, *xnat_project*, *xnat_subject*, *xnat_session*, *xnat_host=None*, *xnat_user=None*, *xnat_pass=None*, *suffix=''*, *subdir=True*, *skip_finish=False*)
Derived class for session-spider

define_spider_process_handler()

Define the SpiderProcessHandler for the end of session spider using the init attributes about XNAT

Returns None

finish()
Method to copy the results in the Spider Results folder dax.RESULTS_DIR Implemented in derived class objects.

Raises NotImplementedError if not overriden by user

Returns None

pre_run()
Pre-Run method to download and organise inputs for the pipeline Implemented in derived class objects.

Raises NotImplementedError if not overridden.

Returns None

run()
Runs the "core" or "image processing process" of the pipeline Implemented in derived class objects.

Raises NotImplementedError if not overridden.

Returns None

class dax.spiders.**AutoSpider**(*name*, *params*, *outputs*, *template*, *version=None*, *exe_lang=None*)
Class for Autospider

copy_input(*src*, *input_name*)
Copy inputs or download from XNAT.

copy_inputs()
Copy the inputs data for AutoSpider.

copy_local_input(*src*, *input_name*)
Copy local inputs.

copy_xnat_input(*src*, *input_name*)
Copy xnat inputs.

download_xnat_file(*src*, *dst*)
Download XNAT specific file.

download_xnat_resource(*src*, *dst*)
Download XNAT complete resource.

end()
Finish the script by sending the end of script flag and cleaning folder :return: None

finish()
finish method to copy the results.

get_argparser()
Get argparser for the AutoSpider.

go()
Main method for AutoSpider.

is_xnat_uri(*uri*)
Check if uri is xnat or local.

pre_run()
Pre-Run method to download and organise inputs for the pipeline Implemented in derived class objects.

print_args(*argument_parse*)
print arguments given to the Spider

Parameters **argument_parse** – argument parser

Returns None

print_end()
Last print statement

Returns None

run()
Run method to execute the template for AutoSpider.

class dax.spiders.**TimedWriter**(*name=None*, *use_date=False*)
Class to automatically write timed output message

Args: name - Names to write with output (default=None)

Examples: >>>a = Time_Writer() >>>a("this is a test") [00d 00h 00m 00s] this is a test >>>sleep(60) >>>a("this is a test") [00d 00h 01m 00s] this is a test

Written by Andrew Plassard (Vanderbilt)

print_stderr_message(*text*)
Prints a timed message to stderr

Parameters **text** – The text to print

Returns None

print_timed_message(*text*, *pipe=<open file '<stdout>', mode 'w'>*)
Prints a timed message

Parameters

- **text** – text to print

- **pipe** – pipe to write to. defaults to sys.stdout

Returns None

1.3.4 dax.processors – Processor class

Processor class define for Scan and Session.

class dax.processors.**Processor**(*walltime_str*, *memreq_mb*, *spider_path*, *version=None*, *ppn=1*, *env=None*, *suffix_proc=''*, *xsitype='proc:genProcData'*, *job_template=None*)

Base class for processor

build_cmds(*cobj*, *dir*)

Build the commands that will go in the PBS/SLURM script :raises: NotImplementedError if not overridden from base class. :return: None

default_settings_spider(*spider_path*)

Get the default spider version and name

Parameters **spider_path** – Fully qualified path and file of the spider

Returns None

get_assessor_input_types()

Enumerate the assessor input types for this. The default implementation returns an empty collection; override this method if you are inheriting from a non-yaml processor. :return: a list of input assessor types

get_proctype()

Return the processor name for this processor. Override this method if you are inheriting from a non-yaml processor. :return: the name of the processor type

has_inputs()

Check to see if the spider has all the inputs necessary to run.

Raises NotImplementedError if user does not override

Returns None

set_spider_settings(*spider_path*, *version*)

Method to set the spider version, path, and name from filepath

Parameters

- **spider_path** – Fully qualified path and file of the spider
- **version** – version of the spider

Returns None

should_run()

Responsible for determining if the assessor should shouw up in session.

Raises NotImplementedError if not overridden.

Returns None

class dax.processors.**ScanProcessor**(*scan_types*, *walltime_str*, *memreq_mb*, *spider_path*, *version=None*, *ppn=1*, *env=None*, *suffix_proc=''*, *full_regex=False*, *job_template=None*)

Scan Processor class for processor on a scan on XNAT

get_assessor(*cscan*)

Returns the assessor object depending on cscan and the assessor label.

Parameters **cscan** – CachedImageScan object from XnatUtils

Returns String of the assessor label

get_assessor_name(*cscan*)
Returns the label of the assessor

Parameters **cscan** – CachedImageScan object from XnatUtils

Returns String of the assessor label

get_task(*cscan*, *upload_dir*)
Get the Task object

Parameters

- **cscan** – CachedImageScan object from XnatUtils
- **upload_dir** – the directory to put the processed data when the process is done

Returns Task object

has_inputs()

Method to check and see that the process has all of the inputs that it needs to run.

Raises NotImplementedError if not overridden.

Returns None

should_run(*scan_dict*)
Method to see if the assessor should appear in the session.

Parameters **scan_dict** – Dictionary of information about the scan

Returns True if it should run, false if it shouldn't

class dax.processors.**SessionProcessor**(*walltime_str*, *memreq_mb*, *spider_path*, *version=None*, *ppn=1*, *env=None*, *suffix_proc=''*, *job_template=None*)
Session Processor class for processor on a session on XNAT

get_assessor(*csess*)
Returns the assessor object depending on csess and the assessor label.

Parameters **csess** – CachedImageSession object from XnatUtils

Returns String of the assessor label

get_assessor_name(*csess*)
Returns the label of the assessor

Parameters **csess** – CachedImageSession object from XnatUtils

Returns String of the assessor label

get_task(*csess*, *upload_dir*)
Return the Task object

Parameters

- **csess** – CachedImageSession from XnatUtils
- **upload_dir** – directory to put the data after run on the node

Returns Task object of the assessor

has_inputs()
Check to see that the session has the required inputs to run.

Raises NotImplementedError if not overriden from base class.

Returns None

should_run(*session_dict*)

By definition, this should always run, so it just returns true with no checks

Parameters **session_dict** – Dictionary of session information for XnatUtils.list_experiments()

Returns True

class dax.processors.**AutoProcessor**(*xnat*, *yaml_source*, *user_inputs=None*)
Auto Processor class for AutoSpider using YAML files

get_assessor_input_types()
Enumerate the assessor input types for this. The default implementation returns an empty collection; override this method if you are inheriting from a non-yaml processor. :return: a list of input assessor types

get_cmds(*assr*, *jobdir*)
Method to generate the spider command for cluster job.

Parameters

- **assessor** – pyxnat assessor object
- **jobdir** – jobdir where the job's output will be generated

Returns command to execute the spider in the job script

get_proctype()
Return the processor name for this processor. Override this method if you are inheriting from a non-yaml processor. :return: the name of the processor type

has_inputs(*cobj*)
Method to check the inputs.

By definition: status = 0 -> NEED_INPUTS, for session asr inputs and resources status = 1 -> NEED_TO_RUN status = -1 -> NO_DATA, for scan primary input isn't usable qcstatus needs a value only when -1 or 0.

You need to set qcstatus to a short string that explain why it's no ready to run. e.g: No NIFTI

Parameters **cobj** – cached object define in dax.XnatUtils (Session or Scan) (see XnatUtils in dax for information)

Returns status, qcstatus

parse_session(*csess*, *sessions*)
Method to run the processor parser on this session, in order to calculate the pattern matches for this processor and the sessions provided :param csess: the active session. For non-longitudinal studies, this is the session that the pattern matching is performed on. For longitudinal studies, this is the 'current' session from which all prior sessions are numbered for the purposes of pattern matching :param sessions: the full, time-ordered list of sessions that should be considered for longitudinal studies. :return: None

should_run(*obj_dict*)
Method to see if the assessor should appear in the session.

Parameters **obj_dict** – Dictionary of information about the scan or sesion

Returns True if it should run, false if it shouldn't

1.3.5 `dax.log` – Logging utility

`dax.log.`**`setup_critical_logger`**(*name*, *logfile*)
Sets up the critical logger

Parameters

- **`name`** – Name of the logger
- **`logfile`** – file to store the log to. sys.stdout if no file define

Returns logger object

`dax.log.`**`setup_debug_logger`**(*name*, *logfile*)
Sets up the debug logger

Parameters

- **`name`** – Name of the logger
- **`logfile`** – file to store the log to. sys.stdout if no file define

Returns logger object

`dax.log.`**`setup_error_logger`**(*name*, *logfile*)
Sets up the error logger

Parameters

- **`name`** – Name of the logger
- **`logfile`** – file to store the log to. sys.stdout if no file define

Returns logger object

`dax.log.`**`setup_info_logger`**(*name*, *logfile*)
Sets up the info logger

Parameters

- **`name`** – Name of the logger
- **`logfile`** – file to store the log to. sys.stdout if no file define

Returns logger object

`dax.log.`**`setup_warning_logger`**(*name*, *logfile*)
Sets up the warning logger

Parameters

- **`name`** – Name of the logger
- **`logfile`** – file to store the log to. sys.stdout if no file define

Returns logger object

1.3.6 `dax.bin` – Responsible for launching, building and updating a Task

File containing functions called by dax executables

`dax.bin.`**`build`**(*settings_path*, *logfile*, *debug*, *projects=None*, *sessions=None*, *mod_delta=None*, *proj_lastrun=None*)

Method that is responsible for running all modules and putting assessors into the database

Parameters

- **settings_path** – Path to the project settings file
- **logfile** – Full file of the file used to log to
- **debug** – Should debug mode be used
- **projects** – Project(s) that need to be built
- **sessions** – Session(s) that need to be built

Returns None

dax.bin.**check_default_keys**(*yaml_file*, *doc*)
Static method to raise error if key not found in dictionary from yaml file. :param yaml_file: path to yaml file defining the processor :param doc: doc dictionary extracted from the yaml file

dax.bin.**launch_jobs**(*settings_path*, *logfile*, *debug*, *projects=None*, *sessions=None*, *writeonly=False*, *pbsdir=None*, *force_no_qsub=False*)
Method to launch jobs on the grid

Parameters

- **settings_path** – Path to the project settings file
- **logfile** – Full file of the file used to log to
- **debug** – Should debug mode be used
- **projects** – Project(s) that need to be launched
- **sessions** – Session(s) that need to be updated
- **writeonly** – write the job files without submitting them
- **pbsdir** – folder to store the pbs file
- **force_no_qsub** – run the job locally on the computer (serial mode)

Returns None

dax.bin.**load_from_file**(*filepath*, *args*, *logger*, *singularity_imagedir=None*)
Check if a file exists and if it's a python file :param filepath: path to the file to test :return: True the file pass the test, False otherwise

dax.bin.**pi_from_project**(*project*)
Get the last name of PI who owns the project on XNAT

Parameters **project** – String of the ID of project on XNAT.

Returns String of the PIs last name

dax.bin.**raise_yaml_error_if_no_key**(*doc*, *yaml_file*, *key*)
Method to raise an execption if the key is not in the dict :param doc: dict to check :param yaml_file: YAMLfile path :param key: key to search

dax.bin.**read_yaml_settings**(*yaml_file*, *logger*)
Method to read the settings yaml file and generate the launcher object.

Parameters **yaml_file** – path to yaml file defining the settings

Returns launcher object

dax.bin.**set_logger**(*logfile*, *debug*)
Set the logging depth

Parameters

- **logfile** – File to log output to
- **debug** – Should debug depth be used?

Returns logger object

dax.bin.**update_tasks**(*settings_path*, *logfile*, *debug*, *projects=None*, *sessions=None*)
Method that is responsible for updating a Task.

Parameters

- **settings_path** – Path to the project settings file
- **logfile** – Full file of the file used to log to
- **debug** – Should debug mode be used
- **projects** – Project(s) that need to be launched
- **sessions** – Session(s) that need to be updated

Returns None

1.3.7 dax.XnatUtils – Collection of utilities for upload/download and general access

XnatUtils contains useful function to interface with XNAT using Pyxnat.

The functions are several categories:

1) Class Specific to XNAT and Spiders: InterfaceTemp to create an interface with XNAT using a tempfolder AssessorHandler to handle assessor label string and access object SpiderProcessHandler to handle results at the end of any spider

2) Methods to query XNAT database and get XNAT object :

3) Methods to access/check objects on XNAT

4) Methods to Download / Upload data to XNAT

5) Other Methods

6) Cached Class for DAX

7) Old download functions still used in some spiders

class dax.XnatUtils.**InterfaceTemp**(*xnat_host=None*, *xnat_user=None*, *xnat_pass=None*, *temp_dir=None*)

Extends the pyxnat.Interface class to make a temporary directory, write the cache to it and then blow it away on the Interface.disconnect call() NOTE: This is deprecated in pyxnat 1.0.0.0

Using netrc to get username password if not given.

authenticate()
Authenticate to XNAT.

Connect to XNAT and try to Disconnect the JSESSION before reconnecting. Raise XnatAuthentificationError if it failes.

Returns True or False

connect()
Connect to XNAT.

disconnect()
Disconnect the JSESSION and blow away the cache.

Returns None

get_project_assessors(*projectid*)
List all the assessors that you have access to based on passed project.

Parameters **projectid** – ID of a project on XNAT

Returns List of all the assessors for the project

get_project_scans(*project_id*, *include_shared=True*)
List all the scans that you have access to based on passed project.

Parameters

- **intf** – pyxnat.Interface object
- **projectid** – ID of a project on XNAT
- **include_shared** – include the shared data in this project

Returns List of all the scans for the project

get_scans(*projectid*, *subjectid*, *sessionid*)

List all the scans that you have access to based on passed session/subject/project.

Parameters

- **intf** – pyxnat.Interface object
- **projectid** – ID of a project on XNAT
- **subjectid** – ID/label of a subject
- **sessionid** – ID/label of a session

Returns List of all the scans

get_session_resources(*projectid*, *subjectid*, *sessionid*)

Gets a list of all of the resources for a session associated to a subject/project requested by the user

Parameters

- **intf** – pyxnat.Interface object
- **projectid** – ID of a project on XNAT
- **subjectid** – ID/label of a subject
- **sessionid** – ID/label of a session to get resources for

Returns List of resources for the session

get_sessions(*projectid=None*, *subjectid=None*)

List all the sessions either:

1) that you have access to

or

2) in a single project (and single subject) based on kargs

Parameters

- **intf** – pyxnat.Interface object
- **projectid** – ID of a project on XNAT
- **subjectid** – ID/label of a subject

Returns List of sessions

class dax.XnatUtils.**AssessorHandler**(*label*)

Class to intelligently deal with the Assessor labels. Make the splitting of the strings easier.

get_proctype()

Get the proctype from the assessor label

Returns The proctype for the assessor

get_project_id()

Get the project ID from the assessor label

Returns The XNAT project label

get_scan_id()

Get the scan ID from teh assessor label

Returns The scan id for the assessor label

get_session_label()

Get the session label from the assessor label

Returns The XNAT session label

get_subject_label()

Get the subject label from the assessor label

Returns The XNAT subject label

is_valid()

Check to see if we have a valid assessor label (aka not None)

Returns True if valid, False if not valid

select_assessor(*intf*)

Run Interface.select() on the assessor label

Parameters **intf** – pyxnat.Interface object

Returns The pyxnat EObject of the assessor

class dax.XnatUtils.**SpiderProcessHandler**(*script_name*, *suffix*, *project=None*, *subject=None*, *experiment=None*, *scan=None*, *alabel=None*, *assessor_handler=None*, *time_writer=None*, *host=None*)

Class to handle the uploading of results for a spider.

add_file(*filepath*, *resource*)

Add a file in the assessor in the upload directory based on the resource name as will be seen on XNAT

Parameters

- **filepath** – Full path to a file to upload

- **resource** – The resource name it should appear under in XNAT

Returns None

add_folder(*folderpath*, *resource_name=None*)
Add a folder to the assessor in the upload directory.

Parameters

- **folderpath** – Full path to the folder to upoad
- **resource_name** – Resource name chosen (if different than basename)

Raises

- **shutil.Error** – Directories are the same
- **OSError** – The directory doesn't exist

Returns None

add_pdf(*filepath*)
Add the PDF and run ps2pdf on the file if it ends with .ps

Parameters **filepath** – Full path to the PDF/PS file

Returns None

add_snapshot(*snapshot*)
Add in the snapshots (for quick viewing on XNAT)

Parameters **snapshot** – Full path to the snapshot file

Returns None

clean(*directory*)
Clean directory if no error and pdf created

Parameters **directory** – directory to be cleaned

done()

Create a flag file that the assessor is ready to be uploaded and set the status as READY_TO_UPLOAD

Returns None

file_exists(*fpath*)
Check to see if a file exists

Parameters **fpath** – full path to a file to assert it exists

Returns True if it exists, False if it doesn't

folder_exists(*fpath*)
Check to see if a folder exists

Parameters **fpath** – Full path to a folder to assert it exists

Returns True if it exists, False if it doesn't

print_copying_statement(*label*, *src*, *dest*)
Print a line that data is being copied to the upload directory

Parameters

- **label** – The XNAT resource label

- **src** – Source directory or file
- **dest** – Destination directory or file

Returns None

print_err(*msg*)
Print error message using time writer if set, print otherwise

Parameters **msg** – Message to print

Returns None

print_msg(*msg*)
Prints a message using TimedWriter or print

Parameters **msg** – Message to print

Returns None

set_assessor_status(*status*)
Set the status of the assessor based on passed value

Parameters **status** – Value to set the procstatus to

Except All catchable errors.

Returns None

set_error()
Set the flag for the error to 1

Returns None

class dax.XnatUtils.**CachedImageSession**(*intf*, *proj*, *subj*, *sess*)
Enumeration for assessors function, to control what assessors are returned

assessors(*select=(0,)*)
Get a list of CachedImageAssessor objects for the XNAT session

Returns List of CachedImageAssessor objects for the session.

full_object()
Return a the full pyxnat Session object of this sessions

Returns pyxnat Session object

get(*name*)
Get the value of a variable name in the session

Parameters **name** – The variable name that you want to get the value of

Returns The value of the variable or '' if not found.

get_resources()

Return a list of dictionaries that correspond to the information for each resource

Returns List of dictionaries

has_shared_project()
Get the project if shared.

Returns project_shared_id if shared, None otherwise

info()
Get a dictionary of lots of variables that correspond to the session

Returns Dictionary of variables

label()
Get the label of the session

Returns String of the session label

resources()
Get a list of CachedResource objects for the session

Returns List of CachedResource objects for the session

scans()
Get a list of CachedImageScan objects for the XNAT session

Returns List of CachedImageScan objects for the session.

session()
Get the session associated with this object :return: session asscoiated with this object

class dax.XnatUtils.**CachedImageScan**(*intf*, *scan_element*, *parent*)
Class to cache the XML information for a scan on XNAT

get(*name*)
Get the value of a variable associated with a scan.

Parameters **name** – Name of the variable to get the value of

Returns Value of the variable if it exists, or '' otherwise.

get_resources()
Get a list of dictionaries of info for each CachedResource.

Returns List of dictionaries of infor for each CachedResource.

info()
Get lots of variables assocaited with this scan.

Returns Dictionary of infomation about the scan.

label()
Get the ID of the scan

Returns String of the scan ID

parent()
Get the parent of the scan

Returns XML String of the scan parent

resources()
Get a list of the CachedResource (s) associated with this scan.

Returns List of the CachedResource (s) associated with this scan.

session()
Get the session associated with this object :return: session asscoiated with this object

class dax.XnatUtils.**CachedImageAssessor**(*intf*, *assr_element*, *parent*)
Class to cache the XML information for an assessor on XNAT

get(*name*)
Get the value of a variable associated with the assessor

Parameters **name** – Variable name to get the value of

Returns Value of the variable, otherwise ''.

get_in_resources()

Get a list of dictionaries of info for the CachedResource objects for "in" type

Returns List of dictionaries of info for the CachedResource objects for "in" type

get_out_resources()

Get a list of dictionaries of info for the CachedResource objects for "out" type

Returns List of dictionaries of info for the CachedResource objects for "out" type

get_resources()
Makes a call to get_out_resources.

Returns List of dictionaries of info for the CachedResource objects for "out" type

in_resources()
Get a list of CachedResource objects for "in" type

Returns List of CachedResource objects for "in" type

info()
Get a dictionary of information associated with the assessor

Returns None

label()
Get the label of the assessor

Returns String of the assessor label

out_resources()
Get a list of CachedResource objects for "out" type

Returns List of CachedResource objects for "out" type

parent()
Get the parent element of the assessor (session)

Returns The session element XML string

class dax.XnatUtils.**CachedResource**(*element*, *parent*)
Class to cache resource XML info on XNAT

get(*name*)
Get the value of a variable associated with the resource

Parameters **name** – Variable name to get the value of

Returns The value of the variable, '' otherwise.

info()
Get a dictionary of information relating to the resource

Returns dictionary of information about the resource.

label()
Get the label of the resource

Returns String of the label of the resource

parent()
Get the resource parent XML string

Returns The resource parent XML string

1.4 DAX Manager

1.4.1 Table of Contents:

1. *About*
2. *How to set it up*
3. *How to add a Module*
4. *How to add a Process*

About

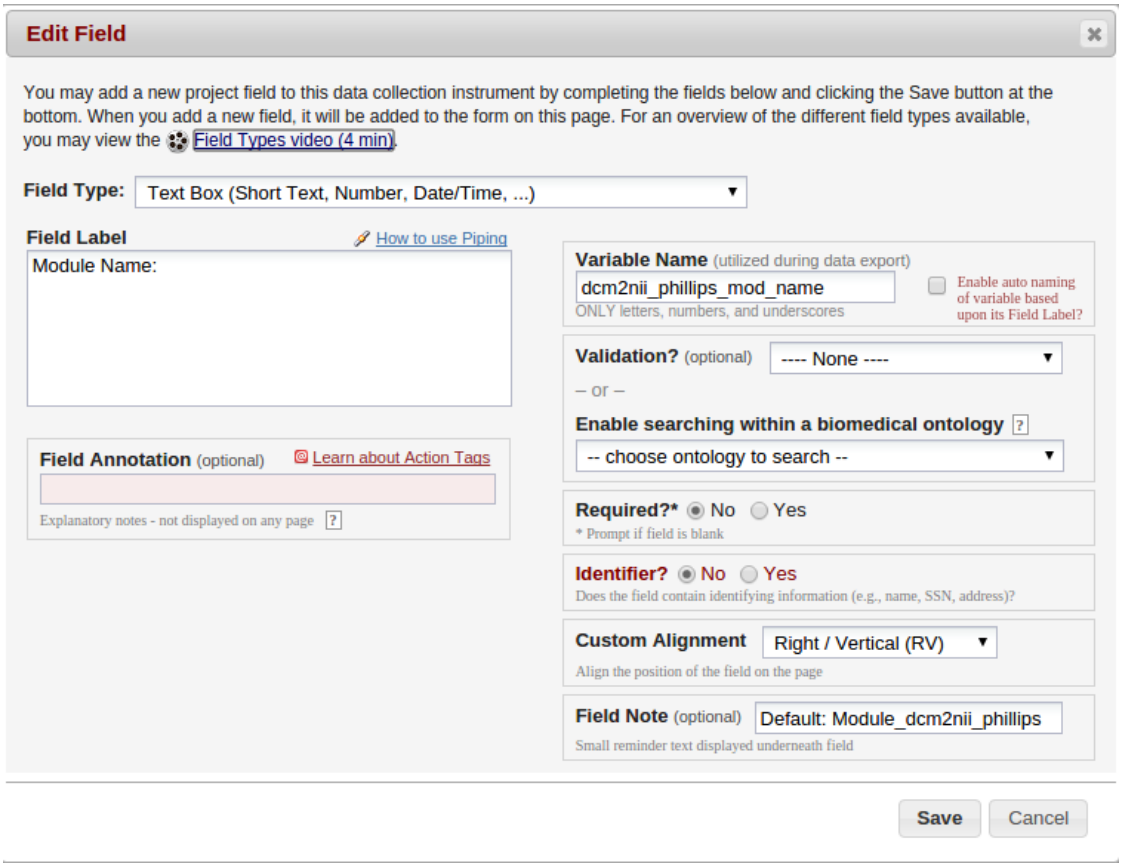
DAX Manager is a non-required tool hosted in REDCap which allows you to quickly generate settings files that can be launched with DAX. This alleviates the need to manual write settings files and makes updating scan types, walltimes, etc a much quicker and streamlined process.

How to set it up

The main instrument should be called General and contains a lot of standard variables that are required for DAX to interface with DAX Manager appropriately. For convenience, a copy of the latest data dictionary has been included and can be downloaded here for reference. It is suggested to use this version even if you do not plan on running all of the spiders because it is currently being used in production `files/dax_manager/XNATProjectSettings_DataDictionary_2016-01-21.csv`.

How to add a Module

Variables used in a module must all start with the text immediately AFTER Module. For example, consider "Module dcm2nii philips". All of the variables for this module must start with "**dcm2nii_philips_**". One required variable is the "on" variable. This variable, again, in the case of "Module dcm2nii philips", would be called "dcm2nii_philips_on". This is used to check to see if the module related to this record in REDCap should be run for your project or not. It must also be of the yes/no REDCap type. If you do not have this variable included, you will get errors when you run dax_manager. The second required variable is the "Module name" variable. In the case of "Module dcm2nii philips", this variable is called "dcm2nii_philips_mod_name". This relates to the class name of the python module file. This information is stored in the REDCap "Field Note" (See below).



This variable must be a REDCap Text Box type (as do all other variables at this point). This must be entered in the following format: "Default: <Module_Class_Name>". All other variables that are used must also start with the "**dcm2nii_philips_**" prefix and must match those of the module init.

Additionally, for the sake of user-friendliness, all variables should use REDCap's branching logic to only appear if the module is "on". It is important to note that in all cases, the REDCap "Field Label" is not used in any automated fashion, but should be something obvious to the users.

How to add a Process

Just like in the case of Modules, Processes follow a close formatting pattern. Similarly, all process variables should start with the text immediately after "Process ". For this example, consider "Process Multi_Atlas". Just like in the case of the modules, the first variable should be a REDCap yes/no and should be called "multi_atlas_on". The remainder of the variables should all be of REDCap type "Text Box". The next required variable is the "Processor Name" variable which must be labeled with the "<Process Name>_proc_name" suffix. In the case of "Process Multi_Atlas", this is called "multi_atlas_proc_name". Just like in the case of the Module, the class name of the processor should be entered in the REDCap Field Note after "Default: ".

There are several other required variables which will be enumerated below (suffix listed first):

1. _suffix_proc - Used to determine what the processor suffix (if any should be)
2. _version - The version of the spider (1.0.0, 2.0.1 etc)
3. _walltime - The amount of walltime to use for the spider when executed on the grid
4. _mem_mb - The amount of ram to request for the job to run. Note this should be in megabytes

5. _scan_types - If writing a ScanProcessor, this is required. If writing a SessionProcessor, this is not required. This, in the case of a ScanProcessor, is used to filter out the scan types that the processor will accept to run the spider on.

Just like in the case of a Module, all variables other than the "on" variable should use REDCap branching logic to only be visible when the process is "on".

1.5 Contributors

DAX is a multi-institution collaborative effort of the following labs:

MASI at Vanderbilt University, Nashville, Tennessee, USA

Center for Cognitive Medicine at Vanderbilt University, Nashville, Tennessee, USA

TIG at UCL (University College London), London, UK

1.6 How To Contribute

We encourage all collaborations! However, we follow a pull-request work flow to help facilitate a simplified code-review process. If you would like to contribute, we kindly request that any of your work be done in a branch. Rules for branching and merging are outlined below:

1. Branches - The scope of your branch should be narrow. Do not make a branch only for changing documentation, and then refactor how task.py works. These should be two totally separate branches.
2. Testing - You should test your branch before making a pull request. Do not make a pull request with untested code.
3. Committing - Use helpful commit messages. Do not use messages like "updates", "bug fix", and "updated a few files" etc. Please make these commit messages at least somewhat helpful. Use lots of commits, do not make 1 bulk commit of all of the changes that you make. This practice makes it hard for others to review.
4. Pull request - When you are ready to make a pull request, please try to itemize all of the changes that you made in at least moderate depth. This will alert everyone reviewing the code of possible things to check to make sure that you didn't break anything.
5. Merging - Do NOT merge your own pull request. Contributors should review each and every pull request before merging into the master branch. Please allow at least a few days before commenting and asking for status. If the depth of changes is deep, please allow at least a few weeks.
6. Master branch - NEVER commit to the master branch directly unless there is a serious bug fix.

If you are unfamiliar with branches in github, please see the link below:

Working with Branches

1.7 FAQ

These FAQs assume that you have read the XNAT documentation and or are familiar with navigating through the web UI. If you are not, you can read the XNAT documentation here.

1. **What is DAX?** DAX is an open source project that uses the pyxnat wrapper for the REST api to automate pipeline running on a DRMAA complaint grid.

2. **What are Modules?** Modules are a special class in DAX. They represent, generally, a task that should not be preformed on the grid. The purpose for this was to not fill up the grid queue with jobs that take 20-30 seconds. Examples of such tasks could be converting a DICOM to a NIfTI file, changing the scan type, archiving a session from the prearchive, or performing skull-stripping. As you can see, these tasks can all be considered "light-weight" and thus probably dont have a place on the grid.
3. **What are Spiders?** Spiders are a python script. The purpose of the script is to download data from XNAT, run an image processing pipeline, and then prepare the data to be uploaded to XNAT. Spiders are run on the grid because they can take hours to days.
4. **My assessor says "NO_DATA". What does that mean?** An assessor procstatus of NO_DATA means that the job will never run, but the assessor is showing up to remind you that you set this spider to always run. For example, if you have a process that runs a pipeline and the can types don't exist in the session, the status would be NO_DATA. However, if at some later point you upload these scans back to the session, you will need to change the procstatus of the corresponding assessor to NO_DATA. This will not automatically be done for you.
5. **My assessor says "NEED_INPUTS". What does that mean?** An assessor procstatus of NEED_INPUTS means that something required for the job to run does not exist yet. Or more simply, the run dependencies have not yet been met. Such dependencies could be another assessor being completed and QA'd, waiting for a manually labeled ROI to be uploaded to a resource, or a custom conversion of an EDAT file.
6. **My assessor says "JOB_FAILED". What does that mean?** An assessor procstatus means that somehow your job failed on the grid. There are many different reasons why this could have happened. Your best bet is to consult the OUTLOG resource of the assesor. This will be the full log of what was printed to STDOUT and STDERR. If the OUTLOG resource doesn't exist yet, it has not yet been uploaded, but wil be automatically uploaded shortly.
7. **How do I know the EXACT command line call that was made?** The PBS resource contains the script that was submitted to the grid scheduler for execution. You can view this file for the exact command line call(s) that were executed on the grid.
8. **I think I found a bug, what should I do?** The easiest way to get a bug fixed is to post as much information as you can on the DAX github issue tracker. If possible, please post the command line call you made (with any sensitive information removed) and the stack trace or error log in question.
9. **I have an idea of something I want to add. How do I go about adding it?** Great! We'd love to see what you have to include! Please read the guidelines on how to contribute

1.8 DAX Processors

1.8.1 About

DAX pipelines are defined by creating YAML text files. If you are not familiar with YAML, start here: https://learnxinyminutes.com/docs/yaml/.

A processor YAML file defines the Environment, Inputs, Commands, and Outputs of your pipeline.

1.8.2 Processor Repos

There are several existing processors that can be used without modification. The processors in these repositories can also provide valuable examples.

https://github.com/bud42/dax-processors

https://github.com/MASILab/yaml_processors

1.8.3 Overview

The processor file defines how a script to run a pipeline should be created. DAX will use the processor to generate scripts to be submitted to your cluster as jobs. The script will contain the commands to download the inputs from XNAT, run the pipeline, and prepare the results to be uploaded back to XNAT (the actual uploading is performed by DAX via *dax upload*).

1.8.4 A "Simple" Example

```
---
moreauto: true
inputs:
  default:
    container_path: MRIQA_v1.0.0.simg
  xnat:
    scans:
      - name: scan_t1
        types: MPRAGE
        resources:
          - resource: NIFTI
            ftype: FILE
            varname: t1_nifti
outputs:
  - path: stats.txt
    type: FILE
    resource: STATS
  - path: report.pdf
    type: FILE
    resource: PDF
  - path: DATA
    type: DIR
    resource: DATA
command: >-
  singularity
  run
  --bind $INDIR:/INPUTS
  --bind $OUTDIR:/OUTPUTS
  {container_path}
  --t1_nifti /INPUTS/{t1_nifti}
attrs:
  walltime: '36:00:00'
  memory: 8192
```

1.8.5 Parts of the Processor YAML

All processor YAML files should start with these two lines:

```
---
moreauto: true
```

The primary components of a processor YAML file are:

- inputs
- outputs

- command
- attrs

Each of these components is required.

1.8.6 inputs

The **inputs** section defines the files and parameters to be prepared for the pipeline. Currently, the only subsections of inputs supported are **defaults** and **xnat**.

The **defaults** subsection can contain paths to local resources such as singularity containers, local codebases, local data to be used by the pipeline. It can essentially contain any value that needs to be passed directly to the **command** template (see below).

The **xnat** section defines the files, directories or values that are extracted from XNAT and passed to the command. Currently, the subsections of **xnat** that are supported are **scans**, **assessors**, **attrs**, and **filters**. Each of these subsections contains an array with a specific set of fields for each item in the array.

xnat scans

Each **xnat scans** item requires a **types** field. The **types** field is used to match against the scan type attribute on XNAT. The value can be a single string or a comma-separated list. Wildcards are also supported.

By default, any scan that matches will be included. You can exclude scans with a quality of *unusable* on XNAT by including the field **needs_qc** with value of *True*. The default is to run anything, i.e. value of *False*. Note that questionable is treated the same as *usable*, so they'll always run.

The **resources** subsection of each xnat scan should contain a list of resources to download from the matched scan. Each resource requires fields for **ftype** and **var**.

ftype specifies what type to downloaded from the resource, either *FILE*, *DIR*, or *DIRJ*. *FILE* will download individual files from the resource. *DIR* will download the whole directory from the resource with the hierarchy maintained. *DIRJ* will also download the directory but strips extraneous intermediate directories from the produced path as implemented by the *-j* flag of unzip.

The **var** field defines the tag to be replaced in the **command** string template (see below).

Optional fields for a resource are **fmatch** and **fcount**. **fmatch** defines a regular expression to apply to filter the list of filenames in the resource. **fcount** can be used to limit the number of files matched. By default, only 1 file is downloaded.

xnat assessors

Each xnat assessor item requires a **proctype** field. The **proctype** field is used to match against the assessor proctype attribute on XNAT. The value can be a single string or a comma-separated list. Wildcards are also supported.

By default, any assessor that matches **proctype** will be included. If you want to only run if an assessor is "good", you set **needs_qc** to *True*, This will not include assessors with an XNAT qcstatus of "NEEDS_QA". It will run on "Passed", "Good", etc. A qcstatus that's "bad" or "Failed" will also be excluded.

The **resources** subsection of each xnat assessor should contain a list of resources to download from the matched scan. Each resource requires fields for **ftype** and **var**.

The **ftype** specifies what type to downloaded from the resource, either *FILE*, *DIR*, or *DIRJ*. *FILE* will download individual files from the resource. *DIR* will download the whole directory from the resource with the hierarchy maintained. *DIRJ* will also download the directory but strips extraneous intermediate directories from the produced path as impelemented by the "-j" flag of unzip.

The **var** field defines the tag to be replaced in the **command** string template (see below).

Optional fields for a resource are fmatch, fdest and fcount. fmatch defines a regular expression to apply to filter the list of filenames in the resource. fcount can be used to limit the number of files matched. By default, only 1 file is downloaded. The inputs for some containers are expected to be in specific locations with specific filenames. This is accomplished using the **fdest** field. The file or directory gets copied to /INPUTS and renamed to the name specified in **fdest**.

xnat attrs

You can evaluate attributes at the subject, session, or scan level. Any fields that are accessible via the XNAT API can be queried. Each **attrs** item should contain a **varname**, **object**, and **attr**. **varname** specifies the tag to be replaced in the **command** string template. **object** is the XNAT object type to query and can be either *subject*, *session*, or *scan*. **attr** is the XNAT field to query. If the object type is *scan*, then a scan name from the xnat scans section must be included with the **ref** field.

For example:

```
attrs:
    - varname: project
      object: session
      attr: project
```

This will extract the value of the project attribute from the session object and replace {project} in the command template.

xnat filters

filters allows you to filter a subset of the cartesian product of the matched scans and assessors. Currently, the only filter implemented is a match filter. It will only create the assessors where the specified list of inputs match. This is used when you want to link a set of assessors that all use the same initial scan as input.

For example:

```
filters:
    - type: match
      inputs: scan_t1,assr_freesurfer/scan_t1
```

This will tell DAX to only run this pipeline where the value for scan_t1 and assr_freesurfer/scan_t1 are the same scan.

outputs

The **outputs** section defines a list files or directories to be uploaded to XNAT upon completion of the pipeline. Each output item must contain fields **path**, **type**, and **resource**. The **path** value contains the local relative path of the file or directory to be uploaded. The type of the path should either be *FILE* or *DIR*. The **resource** is the name of resource of the assessor created on XNAT where the output is to be uploaded.

For every processor, a *PDF* output with **resource** named PDF is required and must be of type *FILE*.

command

The **command** field defines a string template that is formatted using the values from **inputs**.

Each tag specified inside curly braces ("{}"") corresponds to a field in the **defaults** input section, or to a **var** field from a resource on an input or to a **varname** in the xnat attrs section.

Not all **var** must be used.

attrs

The **attrs** section defines miscellaneous other attributes including cluster parameters. These values replace tags in the jobtemplate.

jobtemplate

The **jobtemplate** is a text file that contains a template to create a batch job script.

1.8.7 Versioning

By default, name and version are parsed from the container file name, based on the format: <NAME>_v<major.minor.revision>.simg where<NAME>_v<major> is the proctype.

The YAML file can override these by using any of the top level fields **procversion**, **procname**, and/or **proctype**. **procversion** specifies the major.minor.revision, e.g. *1.0.2*. **procname** specifies the name only without version, e.g. mprage. **proctype** is the name and major version, e.g. *mprage_v1*. If only **procname** is specified, the version is parsed from the container name. If only **procversion** is specified, the name is parsed from the container name. If **proctype** is specified, it will override everything else to determine proctype.

1.8.8 Notes on Singularity run options

–cleanenv avoids env confusion. However we need to avoid –contain for the most part, because it removes access to temp space on the host that many spiders will need, e.g. Freesurfer and /dev/shm. For compiled Matlab spiders (at least), we need to provide –home $INDIR to avoid .mcrCache collisions in temp space when multiple spiders are running.

Python Module Index

d

dax, 10
dax.bin, 28
dax.log, 28
dax.processors, 25
dax.spiders, 18
dax.task, 10
dax.XnatUtils, 30

Index

A

add_file() (*dax.XnatUtils.SpiderProcessHandler method*), 32
add_folder() (*dax.XnatUtils.SpiderProcessHandler method*), 33
add_pdf() (*dax.XnatUtils.SpiderProcessHandler method*), 33
add_snapshot() (*dax.XnatUtils.SpiderProcessHandler method*), 33
AssessorHandler (*class in dax.XnatUtils*), 32
assessors() (*dax.XnatUtils.CachedImageSession method*), 34
authenticate() (*dax.XnatUtils.InterfaceTemp method*), 30
AutoProcessor (*class in dax.processors*), 27
AutoSpider (*class in dax.spiders*), 23

B

batch_path() (*dax.task.ClusterTask method*), 14
batch_path() (*dax.task.XnatTask method*), 17
build() (*in module dax.bin*), 28
build_cmds() (*dax.processors.Processor method*), 25
build_commands() (*dax.task.ClusterTask method*), 14
build_commands() (*dax.task.XnatTask method*), 17
build_task() (*dax.task.ClusterTask method*), 14
build_task() (*dax.task.XnatTask method*), 18

C

CachedImageAssessor (*class in dax.XnatUtils*), 35
CachedImageScan (*class in dax.XnatUtils*), 35
CachedImageSession (*class in dax.XnatUtils*), 34
CachedResource (*class in dax.XnatUtils*), 36
check_date() (*dax.task.ClusterTask method*), 14
check_date() (*dax.task.Task method*), 10
check_default_keys() (*in module dax.bin*), 29
check_executable() (*dax.spiders.Spider method*), 18
check_job_usage() (*dax.task.ClusterTask method*), 14
check_job_usage() (*dax.task.Task method*), 10
check_job_usage() (*dax.task.XnatTask method*), 18
check_running() (*dax.task.ClusterTask method*), 14
check_running() (*dax.task.Task method*), 10
check_running() (*dax.task.XnatTask method*), 18
clean() (*dax.XnatUtils.SpiderProcessHandler method*), 33
ClusterTask (*class in dax.task*), 14
commands() (*dax.task.ClusterTask method*), 14
commands() (*dax.task.Task method*), 10
connect() (*dax.XnatUtils.InterfaceTemp method*), 30
copy_input() (*dax.spiders.AutoSpider method*), 23
copy_inputs() (*dax.spiders.AutoSpider method*), 23
copy_local_input() (*dax.spiders.AutoSpider method*), 23
copy_xnat_input() (*dax.spiders.AutoSpider method*), 24

D

dax (*module*), 10
dax.bin (*module*), 28
dax.log (*module*), 28
dax.processors (*module*), 25
dax.spiders (*module*), 18
dax.task (*module*), 10
dax.XnatUtils (*module*), 30
default_settings_spider() (*dax.processors.Processor method*), 25
define_spider_process_handler() (*dax.spiders.ScanSpider method*), 22
define_spider_process_handler() (*dax.spiders.SessionSpider method*), 23
define_spider_process_handler() (*dax.spiders.Spider method*), 19
disconnect() (*dax.XnatUtils.InterfaceTemp method*), 31
done() (*dax.XnatUtils.SpiderProcessHandler method*), 33

download() (*dax.spiders.Spider method*), 19
download_inputs() (*dax.spiders.Spider method*), 19
download_xnat_file() (*dax.spiders.AutoSpider method*), 24
download_xnat_resource() (*dax.spiders.AutoSpider method*), 24

E

end() (*dax.spiders.AutoSpider method*), 24
end() (*dax.spiders.Spider method*), 19

F

file_exists() (*dax.XnatUtils.SpiderProcessHandler method*), 33
finish() (*dax.spiders.AutoSpider method*), 24
finish() (*dax.spiders.ScanSpider method*), 22
finish() (*dax.spiders.SessionSpider method*), 23
finish() (*dax.spiders.Spider method*), 19
folder_exists() (*dax.XnatUtils.SpiderProcessHandler method*), 33
full_object() (*dax.XnatUtils.CachedImageSession method*), 34

G

get() (*dax.XnatUtils.CachedImageAssessor method*), 35
get() (*dax.XnatUtils.CachedImageScan method*), 35
get() (*dax.XnatUtils.CachedImageSession method*), 34
get() (*dax.XnatUtils.CachedResource method*), 36
get_argparser() (*dax.spiders.AutoSpider method*), 24
get_assessor() (*dax.processors.ScanProcessor method*), 25
get_assessor() (*dax.processors.SessionProcessor method*), 26
get_assessor_input_types() (*dax.processors.AutoProcessor method*), 27
get_assessor_input_types() (*dax.processors.Processor method*), 25
get_assessor_name() (*dax.processors.ScanProcessor method*), 26
get_assessor_name() (*dax.processors.SessionProcessor method*), 26
get_cmds() (*dax.processors.AutoProcessor method*), 27
get_createdate() (*dax.task.ClusterTask method*), 14
get_createdate() (*dax.task.Task method*), 10
get_data_dict() (*dax.spiders.Spider static method*), 20
get_exe_version() (*dax.spiders.Spider method*), 20
get_in_resources() (*dax.XnatUtils.CachedImageAssessor method*), 35
get_job_status() (*dax.task.ClusterTask method*), 15
get_job_status() (*dax.task.Task method*), 10
get_job_status() (*dax.task.XnatTask method*), 18
get_job_usage() (*dax.task.ClusterTask method*), 15
get_job_usage() (*dax.task.Task method*), 10
get_jobid() (*dax.task.ClusterTask method*), 15
get_jobid() (*dax.task.Task method*), 11
get_jobnode() (*dax.task.ClusterTask method*), 15
get_jobnode() (*dax.task.Task method*), 11
get_jobstartdate() (*dax.task.ClusterTask method*), 15
get_jobstartdate() (*dax.task.Task method*), 11
get_memused() (*dax.task.ClusterTask method*), 15
get_memused() (*dax.task.Task method*), 11
get_out_resources() (*dax.XnatUtils.CachedImageAssessor method*), 36
get_processor_name() (*dax.task.ClusterTask method*), 15
get_processor_name() (*dax.task.Task method*), 11
get_processor_version() (*dax.task.ClusterTask method*), 15
get_processor_version() (*dax.task.Task method*), 11
get_proctype() (*dax.processors.AutoProcessor method*), 27
get_proctype() (*dax.processors.Processor method*), 25
get_proctype() (*dax.XnatUtils.AssessorHandler method*), 32
get_project_assessors() (*dax.XnatUtils.InterfaceTemp method*), 31
get_project_id() (*dax.XnatUtils.AssessorHandler method*), 32
get_project_scans() (*dax.XnatUtils.InterfaceTemp method*), 31
get_qcstatus() (*dax.task.ClusterTask method*), 15
get_qcstatus() (*dax.task.Task method*), 11
get_resources() (*dax.XnatUtils.CachedImageAssessor method*), 36
get_resources() (*dax.XnatUtils.CachedImageScan method*), 35
get_resources() (*dax.XnatUtils.CachedImageSession method*), 34
get_scan_id() (*dax.XnatUtils.AssessorHandler method*), 32
get_scans() (*dax.XnatUtils.InterfaceTemp method*), 31

get_session_label() (*dax.XnatUtils.AssessorHandler method*), 32
get_session_resources() (*dax.XnatUtils.InterfaceTemp method*), 31
get_sessions() (*dax.XnatUtils.InterfaceTemp method*), 31
get_status() (*dax.task.ClusterTask method*), 15
get_status() (*dax.task.Task method*), 11
get_statuses() (*dax.task.ClusterTask method*), 15
get_statuses() (*dax.task.Task method*), 11
get_subject_label() (*dax.XnatUtils.AssessorHandler method*), 32
get_task() (*dax.processors.ScanProcessor method*), 26
get_task() (*dax.processors.SessionProcessor method*), 26
get_walltime() (*dax.task.ClusterTask method*), 15
get_walltime() (*dax.task.Task method*), 11
get_xnat_dict() (*dax.spiders.Spider method*), 20
go() (*dax.spiders.AutoSpider method*), 24

H

has_inputs() (*dax.processors.AutoProcessor method*), 27
has_inputs() (*dax.processors.Processor method*), 25
has_inputs() (*dax.processors.ScanProcessor method*), 26
has_inputs() (*dax.processors.SessionProcessor method*), 26
has_shared_project() (*dax.XnatUtils.CachedImageSession method*), 34
has_spider_handler() (*dax.spiders.Spider method*), 20

I

in_resources() (*dax.XnatUtils.CachedImageAssessor method*), 36
info() (*dax.XnatUtils.CachedImageAssessor method*), 36
info() (*dax.XnatUtils.CachedImageScan method*), 35
info() (*dax.XnatUtils.CachedImageSession method*), 34
info() (*dax.XnatUtils.CachedResource method*), 36
InterfaceTemp (*class in dax.XnatUtils*), 30
is_open() (*dax.task.ClusterTask method*), 15
is_open() (*dax.task.Task method*), 11
is_valid() (*dax.XnatUtils.AssessorHandler method*), 32
is_xnat_uri() (*dax.spiders.AutoSpider method*), 24

L

label() (*dax.XnatUtils.CachedImageAssessor method*), 36
label() (*dax.XnatUtils.CachedImageScan method*), 35
label() (*dax.XnatUtils.CachedImageSession method*), 35
label() (*dax.XnatUtils.CachedResource method*), 36
launch() (*dax.task.ClusterTask method*), 16
launch() (*dax.task.Task method*), 11
launch() (*dax.task.XnatTask method*), 18
launch_jobs() (*in module dax.bin*), 29
load_from_file() (*in module dax.bin*), 29

M

merge_pdf_pages() (*dax.spiders.Spider method*), 20

O

out_resources() (*dax.XnatUtils.CachedImageAssessor method*), 36
outlog_path() (*dax.task.ClusterTask method*), 16
outlog_path() (*dax.task.Task method*), 12
outlog_path() (*dax.task.XnatTask method*), 18

P

parent() (*dax.XnatUtils.CachedImageAssessor method*), 36
parent() (*dax.XnatUtils.CachedImageScan method*), 35
parent() (*dax.XnatUtils.CachedResource method*), 36
parse_session() (*dax.processors.AutoProcessor method*), 27
pbs_path() (*dax.task.Task method*), 12
pi_from_project() (*in module dax.bin*), 29
plot_images_page() (*dax.spiders.Spider method*), 20
plot_stats_page() (*dax.spiders.Spider method*), 20
pre_run() (*dax.spiders.AutoSpider method*), 24
pre_run() (*dax.spiders.ScanSpider method*), 23
pre_run() (*dax.spiders.SessionSpider method*), 23
pre_run() (*dax.spiders.Spider method*), 20
print_args() (*dax.spiders.AutoSpider method*), 24
print_args() (*dax.spiders.Spider method*), 20
print_copying_statement() (*dax.XnatUtils.SpiderProcessHandler method*), 33
print_end() (*dax.spiders.AutoSpider method*), 24
print_end() (*dax.spiders.Spider method*), 20
print_err() (*dax.spiders.Spider method*), 21
print_err() (*dax.XnatUtils.SpiderProcessHandler method*), 34
print_info() (*dax.spiders.Spider method*), 21

print_init() (*dax.spiders.Spider method*), 21
print_msg() (*dax.spiders.Spider method*), 21
print_msg() (*dax.XnatUtils.SpiderProcessHandler method*), 34
print_stderr_message() (*dax.spiders.TimedWriter method*), 24
print_timed_message() (*dax.spiders.TimedWriter method*), 24
Processor (*class in dax.processors*), 25

R

raise_yaml_error_if_no_key() (*in module dax.bin*), 29
read_yaml_settings() (*in module dax.bin*), 29
ready_flag_exists() (*dax.task.Task method*), 12
reproc_processing() (*dax.task.ClusterTask method*), 16
reproc_processing() (*dax.task.Task method*), 12
resources() (*dax.XnatUtils.CachedImageScan method*), 35
resources() (*dax.XnatUtils.CachedImageSession method*), 35
run() (*dax.spiders.AutoSpider method*), 24
run() (*dax.spiders.ScanSpider method*), 23
run() (*dax.spiders.SessionSpider method*), 23
run() (*dax.spiders.Spider method*), 21
run_cmd_args() (*dax.spiders.Spider method*), 21
run_system_cmd() (*dax.spiders.Spider method*), 21

S

ScanProcessor (*class in dax.processors*), 25
scans() (*dax.XnatUtils.CachedImageSession method*), 35
ScanSpider (*class in dax.spiders*), 22
select_assessor() (*dax.XnatUtils.AssessorHandler method*), 32
select_obj() (*dax.spiders.Spider method*), 22
select_str() (*dax.spiders.Spider static method*), 22
session() (*dax.XnatUtils.CachedImageScan method*), 35
session() (*dax.XnatUtils.CachedImageSession method*), 35
SessionProcessor (*class in dax.processors*), 26
SessionSpider (*class in dax.spiders*), 23
set_assessor_status() (*dax.XnatUtils.SpiderProcessHandler method*), 34
set_createdate() (*dax.task.ClusterTask method*), 16
set_createdate() (*dax.task.Task method*), 12
set_createdate_today() (*dax.task.ClusterTask method*), 16
set_createdate_today() (*dax.task.Task method*), 12
set_error() (*dax.XnatUtils.SpiderProcessHandler method*), 34
set_jobid() (*dax.task.ClusterTask method*), 16
set_jobid() (*dax.task.Task method*), 12
set_jobnode() (*dax.task.ClusterTask method*), 16
set_jobnode() (*dax.task.Task method*), 13
set_jobstartdate() (*dax.task.ClusterTask method*), 16
set_jobstartdate() (*dax.task.Task method*), 13
set_jobstartdate_today() (*dax.task.Task method*), 13
set_launch() (*dax.task.ClusterTask method*), 16
set_launch() (*dax.task.Task method*), 13
set_launch() (*dax.task.XnatTask method*), 18
set_logger() (*in module dax.bin*), 29
set_memused() (*dax.task.ClusterTask method*), 16
set_memused() (*dax.task.Task method*), 13
set_proc_and_qc_status() (*dax.task.ClusterTask method*), 17
set_proc_and_qc_status() (*dax.task.Task method*), 13
set_qcstatus() (*dax.task.ClusterTask method*), 17
set_qcstatus() (*dax.task.Task method*), 13
set_spider_settings() (*dax.processors.Processor method*), 25
set_status() (*dax.task.ClusterTask method*), 17
set_status() (*dax.task.Task method*), 13
set_walltime() (*dax.task.ClusterTask method*), 17
set_walltime() (*dax.task.Task method*), 13
setup_critical_logger() (*in module dax.log*), 28
setup_debug_logger() (*in module dax.log*), 28
setup_error_logger() (*in module dax.log*), 28
setup_info_logger() (*in module dax.log*), 28
setup_warning_logger() (*in module dax.log*), 28
should_run() (*dax.processors.AutoProcessor method*), 27
should_run() (*dax.processors.Processor method*), 25
should_run() (*dax.processors.ScanProcessor method*), 26
should_run() (*dax.processors.SessionProcessor method*), 27
Spider (*class in dax.spiders*), 18
SpiderProcessHandler (*class in dax.XnatUtils*), 32

T

Task (*class in dax.task*), 10
TimedWriter (*class in dax.spiders*), 24

U

undo_processing() (*dax.task.ClusterTask method*),

17
undo_processing() (*dax.task.Task method*), 14
update_status() (*dax.task.ClusterTask method*), 17
update_status() (*dax.task.Task method*), 14
update_status() (*dax.task.XnatTask method*), 18
update_tasks() (*in module dax.bin*), 30
upload() (*dax.spiders.Spider method*), 22
upload_dict() (*dax.spiders.Spider method*), 22
upload_outlog_dir() (*dax.task.ClusterTask method*), 17
upload_pbs_dir() (*dax.task.ClusterTask method*), 17

X

XnatTask (*class in dax.task*), 17