LWR Documentation

Release 0.2.0

John Chilton

November 14, 2016

Contents

This project is a Python server application that allows a Galaxy server to run jobs on remote systems (including Windows) without requiring a shared mounted file systems. Unlike traditional Galaxy job runners - input files, scripts, and config files may be transferred to the remote system, the job is executed, and the result downloaded back to the Galaxy server.

Full documentation for the project can be found on Read The Docs.

CHAPTER 1

Configuring Galaxy

Galaxy job runners are configured in Galaxy's `job_conf.xml` file. Some small examples of how to configure this can be found here, but be sure to checkout `job_conf.xml.sample_advanced` in your Galaxy code base or on Bitbucket for complete information.

CHAPTER 2

Downloading LWR

The LWR server application is distributed as a Python project and can be obtained via mercurial from bitbucket.org using the following command:

```
hg clone http://bitbucket.org/jmchilton/lwr
```

CHAPTER 3

LWR Dependencies

Several Python packages must be installed to run the LWR server. These can either be installed into a Python `virtualenv` or into your system wide Python environment using `easy_install`. Instructions for both are outlined below. Additionally, if DRMAA is going to be used to communicate with a cluster, this dependency must be installed as well - again see note below.

3.1 virtualenv

The script `setup_venv.sh` distributed with the LWR server is a short-cut for *nix machines to setup a Python environment (including the installation of virtualenv). Full details for installation suitable for *nix are as follows. These instructions can work for Windows as well but generally the `easy_install` instructions below are more robust for Window's environments.

1. Install virtualenv (if not available):

```
pip install virtualenv
```

2. Create a new Python environment:

```
virtualenv .venv
```

3. Activate environment (varies by OS).

From a Linux or MacOS terminal:

```
. .venv/bin/activate
```

From a Windows terminal:

```
.venv\Scripts\activate
```

4. Install required dependencies into this virtual environment:

```
pip install -r requirements.txt
```

3.2 easy_install

Install python setuptools for your platform, more details on how to do this can be found here.

The `easy_install` command line application will be installed as part of setuptools. Use the following command to install the needed packages via `easy_install`:

```
easy_install paste wsgiutils PasteScript PasteDeploy webob six psutil
```

3.3 DRMAA

If your LWR instance is going to communicate with a cluster via DRMAA, in addition to the above dependencies, a DRMAA library will need to be installed and the python dependency drmaa will need to be installed as well.:

```
. .venv/bin/activate; pip install drmaa
```

or:

```
easy_install drmaa
```

CHAPTER 4

Running the LWR Server Application

4.1 *nix Instructions

The LWR can be started and stopped via the `run.sh` script distributed with the LWR.:

```
./run.sh --daemon
./run.sh --stop-daemon
```

These commands will start and stop the WSGI web server in daemon mode. In this mode, logs are writtin to `paster.log`.

If uWSGI, circus and/or chassuette, are available, more sophisticated web servers can be launched via this `run.sh` command. See the script for more details.

4.2 Alternative Cross Platform Instructions (Windows and *nix)

The `paster` command line application will be installed as part of the previous dependency installation process. This application can be used to start and stop a paste web server running the LWR. This can be done by executing the following command:

The server may be ran as a daemon via the command:

```
paster serve server.ini --daemon
```

When running as daemon, the server may be stopped with the following command:

```
paster serve server.ini --stop-daemon
```

If you setup a virtual environment for the LWR you will need to activate this before executing these commands.

Configuring the LWR Server Application

Rename the `server.ini.sample` file distributed with LWR to `server.ini`, and edit the values therein to configure the server application. Default values are specified for all configuration options that will work if LWR is running on the same host as Galaxy. However, the parameter "host" must be specified for remote submissions to the LWR server to run properly. The `server.ini` file contains documentation for many configuration parameters you may want to modify.

Some advanced configuration topics are discussed below.

5.1 Security

Out of the box the LWR essentially allows anyone with network access to the LWR server to execute arbitrary code and read and write any files the web server can. Hence, in most settings steps should be taken to secure the LWR server.

5.1.1 LWR Web Server

The LWR web server can be configured to use SSL and to require the client (i.e. Galaxy) to pass along a private token authorizing use.

`pyOpenSSL` is required to configure an LWR web server to server content via HTTPS/SSL. This dependency can be difficult to install and seems to be getting more difficult. Under Linux you will want to ensure the needed dependencies to compile pyOpenSSL are available - for instance in a fresh Ubuntu image you will likely need:

```
sudo apt-get install libffi-dev python-dev libssl-dev
```

Then pyOpenSSL can be installed with the following command (be sure to source your virtualenv if setup above):

```
pip install pyOpenSSL
```

Under Windows only older versions for pyOpenSSL are installable via pre- compiled binaries (i.e. using *easy_install*) so it might be good to use non- standard sources such as eGenix.

Once installed, you will need to set the option `ssl_pem` in `server.ini`. This parameter should reference an OpenSSL certificate file for use by the Python paste server. This parameter can be set to `*` to automatically generate such a certificate. Such a certificate can manually be generated by the following method:

```
$ openssl genrsa 1024 > host.key
$ chmod 400 host.key
$ openssl req -new -x509 -nodes -sha1 -days 365  \
          -key host.key > host.cert
```

```
$ cat host.cert host.key > host.pem
$ chmod 400 host.pem
```

More information can be found in the paste httpserver documentation.

Finally, in order to force Galaxy to authorize itself, you will want to specify a private token - by simply setting `private_key` to some long random string in `server.ini`.

Once SSL has been enabled and a private token configured, Galaxy job destinations should include a `private_token` parameter to authenticate these jobs.

5.1.2 LWR Message Queue

If LWR is processing Galaxy requests via a message queue instead of a web server the underlying security mechanisms of the message queue should be used to secure the LWR communication - configuring SSL with the LWR and a `private_token` above are not required.

This will likely consist of setting some combination of `amqp_connect_ssl_ca_certs`, `amqp_connect_ssl_keyfile`, `amqp_connect_ssl_certfile`, `amqp_connect_ssl_cert_reqs`, in LWR's `server.ini` file. See `server.ini.sample` for more details and the Kombo documentation for even more information.

5.2 Customizing the LWR Environment

In more sophisticated deployments, the LWR's environment will need to be tweaked - for instance to define a `DRMAA_LIBRARY_PATH` environment variable for the `drmaa` Python module or to define the location to a find a location of Galaxy (via `GALAXY_HOME`) if certain Galaxy tools require it or if Galaxy metadata is being set by the LWR. The recommend way to do this is to copy `local_env.sh.sample` to `local_env.sh` and customize it.

This file of deployment specific environment tweaks will be source by `run.sh` if it exists as well as by other LWR scripts in more advanced usage scenarios.

5.3 Job Managers (Queues)

By default the LWR will maintain its own queue of jobs. While ideal for simple deployments such as those targetting a single Windows instance, if the LWR is going to be used on more sophisticate clusters, it can be configured to maintain multiple such queues with different properties or to delegate to external job queues (via DRMAA, qsub/qstat CLI commands, or Condor).

For more information on configured external job managers, see the job managers documentation.

Warning: If you are using DRMAA, be sure to define `DRMAA_LIBRARY_PATH` in `local_env.sh` defined above.

5.4 Galaxy Tools

Some Galaxy tool wrappers require a copy of the Galaxy codebase itself to run. Such tools will not run under Windows, but on *nix hosts the LWR can be configured to add the required Galaxy code a jobs `PYTHON_PATH` by setting `GALAXY_HOME` environment variable in the LWR's `local_env.sh` file (described above).

5.5 Caching (Experimental)

LWR and its clients can be configured to cache job input files. For some workflows this can result in a significant decrease in data transfer and greater throughput. On the LWR side - the property `file_cache_dir` in `server.ini` must be set. See Galaxy's job_conf.xml for information on configuring the client.

More discussion on this can be found in this galaxy-dev mailing list thread and future plans and progress can be tracked on this Trello card.

5.6 Message Queue (Experimental)

Galaxy and the LWR can be configured to communicate via a message queue instead of an LWR web server. In this mode, the LWR will download files from and upload files to Galaxy instead of the inverse - this may be very advantageous if the LWR needs to be deployed behind a firewall or if the Galaxy server is already setup (via proxy web server) for large file transfers.

To bind the LWR server to a message queue, one needs to first ensure the `kombu` Python dependency is installed (`pip install kombu`). Once this available, simply set the `message_queue_url` property in `server.ini` to the correct URL of your configured AMQP endpoint.

Configuring your AMQP compatible message queue is beyond the scope of this document - see RabbitMQ for instance for more details (other MQs should work also).

CHAPTER 6

Testing

A simple sanity test can be run against a running LWR server by executing the following command (replace the URL command with the URL of your running LWR application):

```
python run_client_tests.py --url=http://localhost:8913
```

CHAPTER 7

Development

This project is distributed with unit and integration tests (many of which will not run under Windows), the following command will install the needed python components to run these tests.:

```
pip install -r dev-requirements.txt
```

The following command will then run these tests:

```
nosetests
```

The following command will then produce a coverage report corresponding to this test and place it in the coverage_html_report subdirectory of this project.:

```
coverage html
```

CHAPTER 8

Job Managers

By default the LWR will maintain its own queue of jobs. Alternatively, the LWR can be configured to maintain multiple such queues with different properties or to delegate to external job queues (via DRMAA, qsub/qstat CLI commands, or Condor).

To change the default configuration, rename the file `job_managers.ini.sample` distributed with the LWR to `job_managers.ini` and modify it to reflect your desired configuration, and finally uncomment the line `#job_managers_config = job_managers.ini` in `server.ini`.

Likely the cleanest way to interface with an external queueing system is going to be DRMAA. In this case, one should likely copy `local_env.sh.sample` to `local_env.sh` and update it to set `DRMAA_LIBRARY_PATH` to point to the correct `libdrmaa.so` file. Also, the Python `drmaa` module must be installed (see more information about *drmaa dependency <https://lwr.readthedocs.org/#job-managers>*).

8.1 Sample Configuration

```
## Default job manager is queued and runs 1 concurrent job.
[manager:_default_]
type = queued_python
max_concurrent_jobs=1

## Create a named queued (example) and run as many concurrent jobs as
## server has cores. The Galaxy LWR url should have /managers/example
## appended to it to use a named manager such as this.
#[manager:example]
#type=queued_python
#max_concurrent_jobs=*

## DRMAA backed manager (vanilla).
## Be sure drmaa Python module install and DRMAA_LIBRARY_PATH points
## to a valid DRMAA shared library file. You may also need to adjust
## LD_LIBRARY_PATH.
#[manager:_default_]
#type=queued_drmaa
#native_specification=-P bignodes -R y -pe threads 8

## Condor backed manager.
#[manager:_default_]
#type=queued_condor
## Optionally, additional condor submission parameters can be
## set as follows:
#submit_universe=vanilla
```

```
#submit_request_memory=32
#submit_requirements=OpSys == "LINUX" && Arch =="INTEL"
#submit_rank=Memory >= 64
## These would set universe, request_memory, requirements, and rank
## in the condor submission file to the specified values. For
## more information on condor submission files see the following link:
## http://research.cs.wisc.edu/htcondor/quick-start.html.

## CLI Manager Locally
## Manage jobs via command-line execution of qsub, qdel, stat.
#[manager:_default_]
#type=queued_cli
#job_plugin=Torque

## CLI Manager via Remote Shell
## Manage jobs via qsub, qdel, qstat on remote host `queuemanager` as
## Unix user `queueuser`.
#[manager:_default_]
#type=queued_cli
#job_plugin=Torque
#shell_plugin=SecureShell
#shell_hostname=queuemanager
#shell_username=queueuser

## DRMAA (via external users) manager.
## This variant of the DRMAA manager will run jobs as the supplied user.
#[manager:_default_]
#type=queued_external_drmaa
#production=true
#chown_working_directory_script=scripts/chown_working_directory.bash
#drmaa_kill_script=scripts/drmaa_kill.bash
#drmaa_launch_script=scripts/drmaa_launch.bash


## NOT YET IMPLEMENTED. PBS backed manager.
#[manager:_default_]
#type=queued_pbs

## Disable server-side LWR queuing (suitable for older style LWR use
## when queues were maintained in Galaxy.) Deprecated, will be removed
## at some point soon.
#[manager:_default_]
#type=unqueued


## MQ-Options:
## If using a message queue the LWR will actively monitor status of jobs
## in order to issue status update messages. The following options are
## then available to any managers.

## Minimum seconds between polling intervals (increase to reduce resources
## consumed by the LWR).
#min_polling_interval = 0.5
```

8.2 Running Jobs As External User

TODO: Fill out this section with information from *this thread <http://dev.list.galaxyproject.org/Managing-Data-Locality-tp4662438.html>*.

CHAPTER 9

Galaxy Configuration

9.1 Examples

The most complete and updated documentation for configuring Galaxy job destinations is Galaxy's `job_conf.xml.sample_advanced` file (check it out on Bitbucket). These examples just provide a different LWR-centric perspective on some of the documentation in that file.

9.1.1 Simple Windows LWR Web Server

The following Galaxy `job_conf.xml` assumes you have deployed a simple LWR web server to the Windows host `windowshost.examle.com` on the default port (`8913`) with a `private_key` (defined in `server.ini`) of `123456789changeme`. Most Galaxy jobs will just route use Galaxy's local job runner but `msconvert` and `proteinpilot` will be sent to the LWR server on `windowshost.examle.com`. Sophisticated tool dependency resolution is not available for Windows-based LWR servers so ensure the underlying application are on the LWR's path.

```xml
<?xml version="1.0"?>
<job_conf>
    <plugins>
        <plugin id="local" type="runner" load="galaxy.jobs.runners.local:LocalJobRunner"/>
        <plugin id="lwr" type="runner" load="galaxy.jobs.runners.lwr:LwrJobRunner"/>
    </plugins>
    <handlers>
        <handler id="main"/>
    </handlers>
    <destinations default="local">
        <destination id="local" runner="local"/>
        <destination id="win_lwr" runner="lwr">
            <param id="url">https://windowshost.examle.com:8913/</param>
            <param id="private_token">123456789changeme</param>
        </destination>
    </destinations>
    <tools>
        <tool id="msconvert" destination="win_lwr" />
        <tool id="proteinpilot" destination="win_lwr" />
        </tools>
</job_conf>
```

9.1.2 Targeting a Linux Cluster (LWR Web Server)

The following Galaxy `job_conf.xml` assumes you have a very typical Galaxy setup - there is a local, smaller cluster that mounts all of Galaxy's data (so no need for the LWR) and a bigger shared resource that cannot mount Galaxy's files requiring the use of the LWR. This variant routes some larger assembly jobs to the remote cluster - namely the *trinity* and *abyss* tools. Be sure the underlying applications required by the `trinity` and `abyss` tools are the LWR path or set `tool_dependency_dir` in `server.ini` and setup Galaxy env.sh-style packages definitions for these applications).

```
<?xml version="1.0"?>
<job_conf>
    <plugins>
        <plugin id="drmaa" type="runner" load="galaxy.jobs.runners.drmaa:DRMAAJobRunner">
        <plugin id="lwr" type="runner" load="galaxy.jobs.runners.lwr:LwrJobRunner"/>
    </plugins>
    <handlers>
        <handler id="main"/>
    </handlers>
    <destinations default="drmaa">
        <destination id="local_cluster" runner="drmaa">
            <param id="native_specification">-P littlenodes -R y -pe threads 4</param>
        </destination>
        <destination id="remote_cluster" runner="lwr">
            <param id="url">http://remotelogin:8913/</param>
            <param id="submit_native_specification">-P bignodes -R y -pe threads 16</param>
            <!-- Look for trinity package at remote location - define tool_dependency_dir
            in the LWR server.ini file.
            -->
            <param id="dependency_resolution">remote</params>
            <!-- Use more correct parameter generation for *nix. Needs testing on Windows
                 servers before this becomes default. -->
            <param id="rewrite_parameters">True</params>
        </destination>
    </destinations>
    <tools>
        <tool id="trinity" destination="remote_cluster" />
        <tool id="abyss" destination="remote_cluster" />
        </tools>
</job_conf>
```

For this configuration, on the LWR side be sure to set a `DRMAA_LIBRARY_PATH` in `local_env.sh`, install the Python `drmaa` module, and configure a DRMAA job manager (example `job_managers.ini` follows).

```
[manager:_default_]
type=queued_drmaa
```

9.1.3 Targeting a Linux Cluster (LWR over Message Queue)

For LWR instances sitting behind a firewall a web server may be impossible. If the same LWR configuration discussed above is additionally configured with a `message_queue_url` of `amqp://rabbituser:rabb8pa8sw0d@mqserver:5672//` in `server.ini` the following Galaxy configuration will cause this message queue to be used for communication. This is also likely better for large file transfers since typically your production Galaxy server will be sitting behind a high-performance proxy but not the LWR.

```
<?xml version="1.0"?>
<job_conf>
    <plugins>
        <plugin id="drmaa" type="runner" load="galaxy.jobs.runners.drmaa:DRMAAJobRunner">
        <plugin id="lwr" type="runner" load="galaxy.jobs.runners.lwr:LwrJobRunner">
            <!-- Must tell LWR where to send files. -->
            <param id="galaxy_url">https://galaxyserver</param>
            <!-- Message Queue Connection (should match message_queue_url in LWR's server.ini)
            -->
            <param id="url">amqp://rabbituser:rabb8pa8sw0d@mqserver:5672//</param>
        </plugin>
    </plugins>
    <handlers>
        <handler id="main"/>
    </handlers>
    <destinations default="drmaa">
        <destination id="local_cluster" runner="drmaa">
            <param id="native_specification">-P littlenodes -R y -pe threads 4</param>
        </destination>
        <destination id="remote_cluster" runner="lwr">
            <!-- Tell Galaxy where files are being store on remote system, no
                 web server it can simply ask for this information.
            -->
            <param id="jobs_directory">/path/to/remote/lwr/lwr_staging/</param>
            <!-- Invert file transfers - have LWR initiate downloads during preprocessing
                 and uploads during postprocessing. -->
            <param id="default_file_action">remote_transfer</param>

            <!-- Remaining parameters same as previous example -->
            <param id="submit_native_specification">-P bignodes -R y -pe threads 16</param>
            <param id="dependency_resolution">remote</params>
            <param id="rewrite_parameters">True</params>
        </destination>
    </destinations>
    <tools>
        <tool id="trinity" destination="remote_cluster" />
        <tool id="abyss" destination="remote_cluster" />
        </tools>
</job_conf>
```

9.1.4 Targeting Apache Mesos (Prototype)

See commit message for initial work on this and this post on galaxy-dev.

9.1.5 Etc...

There are many more options for configuring what paths get staging/unstaged how, how Galaxy metadata is generated, running jobs as the real user, defining multiple job managers on the LWR side, etc.... If you ever have any questions please don't hesistate to ask John Chilton (jmchilton@gmail.com).

9.2 File Actions

Most of the parameters settable in Galaxy's job configuration file `job_conf.xml` are straight forward - but specifing how Galaxy and the LWR stage various files may benefit from more explaination.

As demonstrated in the above `default_file_action` describes how inputs, outputs, etc... are staged. The default `transfer` has Galaxy initiate HTTP transfers. This makes little sense in the contxt of message queues so this should be overridden and set to `remote_transfer` which causes the LWR to initiate the file transfers. Additional options are available including `none`, `copy`, and `remote_copy`.

In addition to this default - paths may be overridden based on various patterns to allow optimization of file transfers in real production infrastructures where various systems mount different file stores and file stores with different paths on different systems.

To do this, the LWR destination in `job_conf.xml` may specify a parameter named `file_action_config`. This needs to be some config file path (if relative, relative to Galaxy's root) like `lwr_actions.yaml` (can be YAML or JSON - but older Galaxy's only supported JSON).

```yaml
paths:
  # Use transfer (or remote_transfer) if only Galaxy mounts a directory.
  - path: /galaxy/files/store/1
    action: transfer

  # Use copy (or remote_copy) if remote LWR server also mounts the directory
  # but the actual compute servers do not.
  - path: /galaxy/files/store/2
    action: copy

  # If Galaxy, the LWR, and the compute nodes all mount the same directory
  # staging can be disabled altogether for given paths.
  - path: /galaxy/files/store/3
    action: none

  # Following block demonstrates specifying paths by globs as well as rewriting
  # unstructured data in .loc files.
  - path: /mnt/indices/**/bwa/**/*.fa
    match_type: glob
    path_types: unstructured  # Set to *any* to apply to defaults & unstructured paths.
    action: transfer
    depth: 1  # Stage whole directory with job and just file.

  # Following block demonstrates rewriting paths without staging. Useful for
  # instance if Galaxy's data indices are mounted on both servers but with
  # different paths.
  - path: /galaxy/data
    path_types: unstructured
    action: rewrite
    source_directory: /galaxy/data
    destination_directory: /work/galaxy/data
```

Configuring a Public LWR Server

An LWR server can be pointed at a Galaxy toolbox XML file and opened to the world. By default, an LWR is allowed to run anything Galaxy (or other client) sends it. The toolbox and referenced tool files are used to restrict what what the LWR will run.

This can be sort of thought of as web services defined by Galaxy tool files - with all the advantages (dead simple configuration for clients, ability to hide details related date and computation) and disadvantages (lack of reproducibility if the LWR server goes away, potential lack of transparency).

10.1 Securing a Public LWR

The following options should be set in `server.ini` to configure a public *LWR* server.

- `assign_ids=uuid` - By default the *LWR* will just the ids Galaxy instances. Setting this setting to `uuid` will result in each job being assigned a UUID, ensuring different clients will not and cannot interfer with each other.
- `tool_config_files=/path/to/tools.xml` - As noted above, this is used to restrict what tools clients can run. All tools on public LWR servers should have validators for commands (and optionally for configfiles) defined. The syntax for these elements can be found in the ValidatorTest test case.

10.2 Writing Secure Tools

Validating in this fashion is complicated and potentially error prone, so it is advisable to keep command-lines as simple as possible. configfiles and reorganizing parameter handling in wrappers scripts can assist in this.

Consider the following simple example:

`tool.xml`:

```
<tool>
   <command interpreter="python">wrapper.py --input1 'Text' --input2 'Text2' --input3 4.5</command>
   ...
```

`wrapper.py`:

```
def main():
     parser = OptionParser()
     parser.add_option("--input1")
     parser.add_option("--input2")
     parser.add_option("--input3")
     (options, args) = parser.parse_args()
```

Even this simple example is easier to validate and secure if it is reworked as so:

`tool.xml`:

```
<tool>
  <configfiles>
    <configfile name="args">--input1 'Text' --input2 'Text2' --input3 4.5</configfile>
  </configfiles>
  <command interpreter="python">wrapper.py $args</command>
  ...
```

`wrapper.py`:

```
import sys, shlex

def main():
    args_config = sys.argv[1]
    args_string = open(args_config, "r").read()

    parser = OptionParser()
    parser.add_option("--input1")
    parser.add_option("--input2")
    parser.add_option("--input3")
    (options, args) = parser.parse_args(shlex.split(args_string))
```

CHAPTER 11

Server Code

11.1 `lwr.managers` Module

Job Managers

11.1.1 `lwr.managers.base` Module

Base Classes and Infrastructure Supporting Concret Manager Implementations.

`lwr.managers.base.base_drmaa` Module

class `lwr.managers.base.base_drmaa.`**`BaseDrmaaManager`**(*name*, *app*, ***kwds*)

Bases: *`lwr.managers.base.external.ExternalBaseManager`*

`shutdown`()

`lwr.managers.base.directory` Module

class `lwr.managers.base.directory.`**`DirectoryBaseManager`**(*name*, *app*, ***kwds*)

Bases: *`lwr.managers.base.BaseManager`*

`return_code`(*job_id*)

`stderr_contents`(*job_id*)

`stdout_contents`(*job_id*)

`lwr.managers.base.external` Module

class `lwr.managers.base.external.`**`ExternalBaseManager`**(*name*, *app*, ***kwds*)

Bases: *`lwr.managers.base.directory.DirectoryBaseManager`*

Base class for managers that interact with external distributed resource managers.

`clean`(*job_id*)

`get_status`(*job_id*)

`kill`(*job_id*)

`setup_job`(*input_job_id*, *tool_id*, *tool_version*)

class `lwr.managers.base.`**`BaseManager`**(*name*, *app*, ***kwds*)
Bases: *`lwr.managers.ManagerInterface`*

`clean`(*job_id*)

`job_directory`(*job_id*)

`system_properties`()

class `lwr.managers.base.`**`JobDirectory`**(*staging_directory*, *job_id*, *lock_manager=None*)
Bases: `lwr.lwr_client.job_directory.RemoteJobDirectory`

`calculate_path`(*remote_path*, *input_type*)
Verify remote_path is in directory for input_type inputs and create directory if needed.

`contains_file`(*name*)

`delete`()

`exists`()

`has_metadata`(*metadata_name*)

`load_metadata`(*metadata_name*, *default=None*)

`lock`(*name='.state'*)

`make_directory`(*name*)

`open_file`(*name*, *mode='wb'*)

`outputs_directory_contents`()

`read_file`(*name*, *default=None*)

`remove_file`(*name*)
Quietly remove a job file.

`remove_metadata`(*metadata_name*)

`setup`()

`store_metadata`(*metadata_name*, *metadata_value*)

`working_directory_contents`()

`write_file`(*name*, *contents*)

`lwr.managers.base.`**`get_id_assigner`**(*assign_ids*)

`lwr.managers.base.`**`get_mapped_file`**(*directory*, *remote_path*, *allow_nested_files=False*, *local_path_module=<module 'posixpath' from '/home/docs/checkouts/readthedocs.org/user_builds/lwr/envs/latest/lib/python2.7/pos*
mkdir=True)

```
>>> import ntpath
>>> get_mapped_file(r'C:\lwr\staging\101', 'dataset_1_files/moo/cow', allow_nested_files=True, l
'C:\\lwr\\staging\\101\\dataset_1_files\\moo\\cow'
>>> get_mapped_file(r'C:\lwr\staging\101', 'dataset_1_files/moo/cow', allow_nested_files=False,
'C:\\lwr\\staging\\101\\cow'
>>> get_mapped_file(r'C:\lwr\staging\101', '../cow', allow_nested_files=True, local_path_module=
Traceback (most recent call last):
Exception: Attempt to read or write file outside an authorized directory.
```

11.1.2 `lwr.managers.queued` Module

class `lwr.managers.queued.`**`QueueManager`**(*name*, *app*, ***kwds*)

Bases: *`lwr.managers.unqueued.Manager`*

A job manager that queues up jobs directly (i.e. does not use an external queuing software such PBS, SGE, etc...).

`launch`(*job_id*, *command_line*, *submit_params={}*, *dependencies_description=None*, *env=[]*)

`manager_type` = **'queued_python'**

`run_next`()

Run the next item in the queue (a job waiting to run).

`shutdown`()

11.1.3 `lwr.managers.queued_drmaa` Module

class `lwr.managers.queued_drmaa.`**`DrmaaQueueManager`**(*name*, *app*, ***kwds*)

Bases: *`lwr.managers.base.base_drmaa.BaseDrmaaManager`*

DRMAA backed queue manager.

`launch`(*job_id*, *command_line*, *submit_params={}*, *dependencies_description=None*, *env=[]*)

`manager_type` = **'queued_drmaa'**

11.1.4 `lwr.managers.queued_external_drmaa` Module

class `lwr.managers.queued_external_drmaa.`**`ExternalDrmaaQueueManager`**(*name*, *app*, ***kwds*)

Bases: *`lwr.managers.base.base_drmaa.BaseDrmaaManager`*

DRMAA backed queue manager.

`get_status`(*job_id*)

`launch`(*job_id*, *command_line*, *submit_params={}*, *dependencies_description=None*, *env=[]*)

`manager_type` = **'queued_external_drmaa'**

11.1.5 `lwr.managers.queued_condor` Module

class `lwr.managers.queued_condor.`**`CondorQueueManager`**(*name*, *app*, ***kwds*)

Bases: *`lwr.managers.base.external.ExternalBaseManager`*

Job manager backend that plugs into Condor.

`get_status`(*job_id*)

`launch`(*job_id*, *command_line*, *submit_params={}*, *dependencies_description=None*, *env=[]*)

`manager_type` = **'queued_condor'**

11.1.6 `lwr.managers.queued_pbs` Module

class `lwr.managers.queued_pbs.`**`PbsQueueManager`**(*name*, *app*, ***kwds*)
Bases: *`lwr.managers.base.BaseManager`*

Placeholder for PBS-python backed queue manager. Not yet implemented, for many situations this would be used the DRMAA or CLI+Torque managers may be better choices or at least stop gaps.

`manager_type` = **'queued_pbs'**

11.1.7 `lwr.managers.unqueued` Module

class `lwr.managers.unqueued.`**`Manager`**(*name*, *app*, ***kwds*)
Bases: *`lwr.managers.base.directory.DirectoryBaseManager`*

A simple job manager that just directly runs jobs as given (no queueing). Preserved for compatibilty with older versions of LWR client code where Galaxy is used to maintain queue (like Galaxy's local job runner).

`get_status`(*job_id*)

`kill`(*job_id*)

`launch`(*job_id*, *command_line*, *submit_params={}*, *dependencies_description=None*, *env=[]*)

`manager_type` = **'unqueued'**

`setup_job`(*input_job_id*, *tool_id*, *tool_version*)

11.1.8 `lwr.managers.stateful` Module

class `lwr.managers.stateful.`**`ActiveJobs`**(*manager*)
Bases: `object`

Keeps track of active jobs (those that are not yet "complete"). Current implementation is file based, but could easily be made database-based instead.

TODO: Keep active jobs in memory after initial load so don't need to repeatedly hit disk to recover this information.

`activate_job`(*job_id*)

`active_job_ids`()

`deactivate_job`(*job_id*)

class `lwr.managers.stateful.`**`ManagerMonitor`**(*stateful_manager*)
Bases: `object`

Monitors active jobs of a StatefulManagerProxy.

`shutdown`()

class `lwr.managers.stateful.`**`StatefulManagerProxy`**(*manager*, ***manager_options*)
Bases: *`lwr.managers.ManagerProxy`*

`get_status`(*job_id*)
Compute status used proxied manager and handle state transitions and track additional state information needed.

`handle_remote_staging`(*job_id*, *staging_config*)

`launch`(*job_id*, **args*, ***kwargs*)

name

set_state_change_callback(*state_change_callback*)

setup_job(**args*, ***kwargs*)

shutdown()

lwr.managers.stateful.**new_thread_for_manager**(*manager*, *name*, *target*, *daemon*)

11.1.9 lwr.managers.status Module

lwr.managers.status.**is_job_done**(*status*)
Does the supplied status correspond to a finished job (done processing).

11.1.10 lwr.managers.util Module

This module and its submodules contains utilities for running external processes and interfacing with job managers. This module should contain functionality shared between Galaxy and the LWR.

11.1.11 lwr.managers.staging Module

This module contains the code that allows the LWR to stage file's during preprocessing (currently this means downloading or copying files) and then unstage or send results back to client during postprocessing.

lwr.managers.staging.preprocess Module

lwr.managers.staging.preprocess.**preprocess**(*job_directory*, *setup_actions*)

lwr.managers.staging.postprocess Module

lwr.managers.staging.postprocess.**postprocess**(*job_directory*)

class lwr.managers.**ManagerInterface**
Bases: object

Defines the interface to various job managers.

clean(*job_id*)
Delete job directory and clean up resources associated with job with id *job_id*.

get_status(*job_id*)
Return status of job as string, currently supported statuses include 'cancelled', 'running', 'queued', and 'complete'.

job_directory(*job_id*)
Return a JobDirectory abstraction describing the state of the job working directory.

kill(*job_id*)
End or cancel execution of the specified job.

launch(*job_id*, *command_line*, *submit_params={}*, *dependencies_description=None*, *env=[]*)
Called to indicate that the client is ready for this job with specified job id and command line to be executed (i.e. run or queue this job depending on implementation).

return_code (*job_id*)
: Return integer indicating return code of specified execution or LWR_UNKNOWN_RETURN_CODE.

setup_job (*input_job_id*, *tool_id*, *tool_version*)
: Setup a job directory for specified input (galaxy) job id, tool id, and tool version.

stderr_contents (*job_id*)
: After completion, return contents of stderr associated with specified job.

stdout_contents (*job_id*)
: After completion, return contents of stdout associated with specified job.

class `lwr.managers.`**ManagerProxy** (*manager*)

Bases: `object`

Subclass to build override proxy a manager and override specific functionality.

clean (**args*, ***kwargs*)

get_status (**args*, ***kwargs*)

job_directory (**args*, ***kwargs*)

kill (**args*, ***kwargs*)

launch (**args*, ***kwargs*)

return_code (**args*, ***kwargs*)

setup_job (**args*, ***kwargs*)

shutdown ()
: Optional.

stderr_contents (**args*, ***kwargs*)

stdout_contents (**args*, ***kwargs*)

system_properties ()

11.2 `lwr.core` Module

class `lwr.core.`**LwrApp** (***conf*)

Bases: `object`

shutdown ()

11.3 `lwr.daemon` Module

class `lwr.daemon.`**ArgumentParser** (***kwargs*)

Bases: `optparse.OptionParser`

add_argument (**args*, ***kwargs*)

parse_args ()

class `lwr.daemon.`**LwrConfigBuilder** (*args=None*, ***kwds*)

Bases: `object`

Generate paste-like configuration from supplied command-line arguments.

load ()

classmethod populate_options(*clazz*, *arg_parser*)

setup_logging()

to_dict()

class lwr.daemon.**LwrManagerConfigBuilder**(*args=None*, ***kwds*)
Bases: *lwr.daemon.LwrConfigBuilder*

classmethod populate_options(*clazz*, *arg_parser*)

to_dict()

lwr.daemon.**app_loop**(*args*)

lwr.daemon.**load_lwr_app**(*config_builder*, *config_env=False*, *log=None*, ***kwds*)

lwr.daemon.**main**()

11.4 lwr.scripts Module

This module contains entry points into various LWR scripts. Corresponding shell scripts that setup the deployment specific environments and then delegate to these Python scripts can be found in `LWR_ROOT/scripts`.

11.4.1 lwr.scripts.chown_working_directory Module

lwr.scripts.chown_working_directory.**main**()

11.4.2 lwr.scripts.drmaa_kill Module

lwr.scripts.drmaa_kill.**main**()

11.4.3 lwr.scripts.drmaa_launch Module

lwr.scripts.drmaa_launch.**main**()

11.4.4 lwr.scripts.lwr_submit Module

lwr.scripts.lwr_submit.**main**()

lwr.scripts.lwr_submit.**manager_from_args**(*config_builder*)

lwr.scripts.lwr_submit.**wait_for_job**(*manager*, *job_config*, *poll_time=2*)

11.4.5 lwr.scripts.mesos_executor Module

class lwr.scripts.mesos_executor.**LwrExecutor**
Bases: `object`

frameworkMessage(*driver*, *message*)

launchTask(*driver*, *task*)

lwr.scripts.mesos_executor.**run_executor**()

11.4.6 `lwr.scripts.mesos_framework` Module

`lwr.scripts.mesos_framework.`**`main`**`()`

11.5 `lwr.web` Module

The code explicitly related to the LWR web server can be found in this module and its submodules.

11.5.1 `lwr.web.framework` Module

Tiny framework used to power LWR application, nothing in here is specific to running or staging jobs. Mostly deals with routing web traffic and parsing parameters.

class `lwr.web.framework.`**`Controller`**`(`*response_type='OK'*`)`

Bases: `object`

Wraps python functions into controller methods.

`body``(`*result*`)`

class `lwr.web.framework.`**`FileIterator`**`(`*path*`)`

Bases: `six.Iterator`

class `lwr.web.framework.`**`RoutingApp`**

Bases: `object`

Abstract definition for a python web application.

`add_route``(`*route*, *controller*, ***args*`)`

`lwr.web.framework.`**`build_func_args`**`(`*func*, **arg_dicts*`)`

`lwr.web.framework.`**`file_response`**`(`*path*`)`

11.5.2 `lwr.web.routes` Module

class `lwr.web.routes.`**`LwrController`**`(`***kwargs*`)`

Bases: *`lwr.web.framework.Controller`*

class `lwr.web.routes.`**`LwrDataset`**`(`*id*`)`

Bases: `object`

Intermediary between lwr and objectstore.

11.5.3 `lwr.web.wsgi` Module

class `lwr.web.wsgi.`**`LwrWebApp`**`(`*lwr_app*`)`

Bases: *`lwr.web.framework.RoutingApp`*

Web application for LWR web server.

`lwr.web.wsgi.`**`app_factory`**`(`*global_conf*, ***local_conf*`)`

Returns the LWR WSGI application.

11.6 `lwr.messaging` Module

This module contains the server-side only code for interfacing with message queues. Code shared between client and server can be found in submodules of `lwr.lwr_client`.

11.6.1 `lwr.messaging.bind_amqp` Module

`lwr.messaging.bind_amqp.`**`bind_manager_to_queue`**(*manager*, *queue_state*, *connection_string*, *conf*)

`lwr.messaging.bind_amqp.`**`get_exchange`**(*connection_string*, *manager_name*, *conf*)

class `lwr.messaging.`**`QueueState`**

Bases: `object`

Passed through to event loops, should be "non-zero" while queues should be active.

`deactivate`()

`lwr.messaging.`**`bind_app`**(*app*, *queue_id*, *connect_ssl=None*)

11.7 `lwr.mesos` Module

This module and submodules contain code for interfacing the Apache Mesos framework.

11.7.1 `lwr.mesos.framework` Module

class `lwr.mesos.framework.`**`LwrScheduler`**(*executor*, *manager_options*, *mesos_url*)

Bases: `object`

`frameworkMessage`(*driver*, *executorId*, *slaveId*, *message*)

`handle_setup_message`(*body*, *message*)

`registered`(*driver*, *frameworkId*, *masterInfo*)

`resourceOffers`(*driver*, *offers*)

`statusUpdate`(*driver*, *update*)

`lwr.mesos.framework.`**`run`**(*master*, *manager_options*, *config*)

`lwr.mesos.`**`ensure_mesos_libs`**()

Raise import error if mesos is not actually available. Original import errors above supressed because mesos is meant as an optional dependency for the LWR.

11.8 `lwr.app` Module

Deprecated module for wsgi app factory. LWR servers should transition to `lwr.web.wsgi:app_factory`.

`lwr.app.`**`app_factory`**(*global_conf*, ***local_conf*)

Returns the LWR WSGI application.

11.9 `lwr.manager_endpoint_util` Module

Composite actions over managers shared between HTTP endpoint (routes.py) and message queue.

`lwr.manager_endpoint_util.`**`full_status`**(*manager*, *job_status*, *job_id*)

`lwr.manager_endpoint_util.`**`setup_job`**(*manager*, *job_id*, *tool_id*, *tool_version*)
Setup new job from these inputs and return dict summarizing state (used to configure command line).

`lwr.manager_endpoint_util.`**`submit_job`**(*manager*, *job_config*)
Launch new job from specified config. May have been previously 'setup' if 'setup_params' in job_config is empty.

11.10 `lwr.manager_factory` Module

`lwr.manager_factory.`**`build_managers`**(*app*, *conf*)
Takes in a config file as outlined in job_managers.ini.sample and builds a dictionary of job manager objects from them.

11.11 `lwr.locks` Module

class `lwr.locks.`**`LockManager`**(*lockfile=None*)

`free_lock`(*path*)

`get_lock`(*path*)
Get a job lock corresponding to the path - assumes parent directory exists but the file itself does not.

11.12 `lwr.tools` Module

Tools

11.12.1 `lwr.tools.toolbox` Module

class `lwr.tools.toolbox.`**`InputsValidator`**(*command_validator*, *config_validators*)
Bases: `object`

`validate_command`(*job_directory*, *command*)

`validate_config`(*job_directory*, *name*, *path*)

class `lwr.tools.toolbox.`**`SimpleToolConfig`**(*tool_el*, *tool_path*)
Bases: *`lwr.tools.toolbox.ToolConfig`*

Abstract description of a Galaxy tool loaded from a toolbox with the *tool* tag not containing a guid, i.e. one not from the toolshed.

class `lwr.tools.toolbox.`**`ToolBox`**(*path_string*)
Bases: `object`

Abstraction over a tool config file largely modelled after Galaxy's shed_tool_conf.xml. Hopefully over time this toolbox schema will be a direct superset of Galaxy's with extensions to support simple, non-toolshed based tool setups.

`get_tool`(*id*)

class `lwr.tools.toolbox.`**`ToolConfig`**
Bases: `object`

Abstract description of a Galaxy tool.

`get_tool_dir`()

`inputs_validator`

class `lwr.tools.toolbox.`**`ToolShedToolConfig`**(*tool_el*, *tool_path*)
Bases: *`lwr.tools.toolbox.SimpleToolConfig`*

Abstract description of a Galaxy tool loaded from a toolbox with the *tool* tag, i.e. one from the toolshed.

<tool file="../shed_tools/gvk.bx.psu.edu/repos/test/column_maker/f06aa1bf1e8a/column_maker/column_maker.xml" guid=
<tool_shed>gvk.bx.psu.edu:9009</tool_shed> <repository_name>column_maker</repository_name> <repository_owner>test</repository_owner> <installed_changeset_revision>f06aa1bf1e8a</installed_changeset_revision <id>gvk.bx.psu.edu:9009/repos/test/column_maker/Add_a_column1/1.1.0</id> <version>1.1.0</version>

</tool>

11.12.2 `lwr.tools.authorization` Module

class `lwr.tools.authorization.`**`AllowAnyAuthorization`**
Bases: `object`

`authorize_config_file`(*job_directory*, *name*, *path*)

`authorize_execution`(*job_directory*, *command_line*)

`authorize_setup`()

`authorize_tool_file`(*name*, *contents*)

class `lwr.tools.authorization.`**`AllowAnyAuthorizer`**
Bases: `object`

Allow any, by default LWR is assumed to be secured using a firewall or private_token.

`ALLOW_ANY_AUTHORIZATION` = <lwr.tools.authorization.AllowAnyAuthorization object>

`get_authorization`(*tool_id*)

class `lwr.tools.authorization.`**`ToolBasedAuthorization`**(*tool*)
Bases: *`lwr.tools.authorization.AllowAnyAuthorization`*

`authorize_config_file`(*job_directory*, *name*, *path*)

`authorize_execution`(*job_directory*, *command_line*)

`authorize_setup`()

`authorize_tool_file`(*name*, *contents*)

class `lwr.tools.authorization.`**`ToolBasedAuthorizer`**(*toolbox*)
Bases: `object`

Work In Progress: Implement tool based white-listing of what jobs can run and what those jobs can do.

get_authorization(*tool_id*)

lwr.tools.authorization.**get_authorizer**(*toolbox*)

11.12.3 `lwr.tools.validator` Module

class lwr.tools.validator.**ExpressionValidator**(*xml_el*)
Bases: object

validate(*job_directory*, *string*)

class lwr.tools.**ToolBox**(*path_string*)
Bases: object

Abstraction over a tool config file largely modelled after Galaxy's shed_tool_conf.xml. Hopefully over time this toolbox schema will be a direct superset of Galaxy's with extensions to support simple, non-toolshed based tool setups.

get_tool(*id*)

CHAPTER 12

Client Code

12.1 `lwr.lwr_client` Module

12.1.1 lwr_client

This module contains logic for interfacing with an external LWR server.

Configuring Galaxy

Galaxy job runners are configured in Galaxy's `job_conf.xml` file. See `job_conf.xml.sample_advanced` in your Galaxy code base or on Bitbucket for information on how to configure Galaxy to interact with the LWR.

Galaxy also supports an older, less rich configuration of job runners directly in its main `universe_wsgi.ini` file. The following section describes how to configure Galaxy to communicate with the LWR in this legacy mode.

Legacy

A Galaxy tool can be configured to be executed remotely via LWR by adding a line to the `universe_wsgi.ini` file under the `galaxy:tool_runners` section with the format:

```
<tool_id> = lwr://http://<lwr_host>:<lwr_port>
```

As an example, if a host named remotehost is running the LWR server application on port `8913`, then the tool with id `test_tool` can be configured to run remotely on remotehost by adding the following line to `universe.ini`:

```
test_tool = lwr://http://remotehost:8913
```

Remember this must be added after the `[galaxy:tool_runners]` header in the `universe.ini` file.

CHAPTER 13

Indices and tables

- genindex
- modindex
- search