Girder Worker Documentation

Release 0.1.0

Kitware, Inc.

Sep 11, 2018

Contents

Girder Worker is a remote task execution engine designed to work with Girder. Girder Worker provides a thin wrapper around Celery which is an asynchronous task queue/job queue based on distributed message passing. Girder Worker relies heavily on Celery for its API and implementation, adding two critical features:

- **Task Discovery** Girder Worker implements a custom mechanism for discovering installed tasks at run time. These "pluggable" tasks are defined as python packages and installed in the environment where Girder Worker is run.
- **Task Tracking** If called from Girder, Girder Worker generates a Girder Job for tracking task status and getting real-time output of job progress. If *not* called from Girder, Girder Worker reverts to traditional Celery behavior, making it amenable to running tasks in a python interpreter, scripts, or Jupyter Notebooks.

CHAPTER 1

Getting Started

1.1 Choosing a Broker

The first step in getting Girder Worker up and running is installing a broker. The broker is a message queue such as RabbitMQ which receives messages and passes them to workers to execute a task. If you are running on an Ubuntu or Debian server you can install RabbitMQ with the following command

```
$ sudo apt-get install rabbitmq-server
```

Alternately, if you have docker installed, you can run the rabbitmq inside a container

```
$ docker run --net=host -d rabbitmq:latest
```

1.2 Installing Girder Worker

Girder Worker is a python package and may be installed with pip

```
$ pip install girder-worker
```

We recommend installing in a virtual environment to prevent package collision with your system Python.

1.3 Creating a Task Plugin

Task plugins are python packages. Multiple tasks may be placed in the same package but they must be installed in your environment to be discovered. Python packages require a certain amount of boilerplate to get started. The easiest way to create a package with a task plugin is to use the cookiecutter tool along with the Girder Worker plugin cookiecutter template.

First install cookiecutter

```
$ pip install cookiecutter
```

Next generate a task plugin Python package

```
$ cookiecutter gh:girder/cookiecutter-gw-plugin
```

This will prompt you with a number of questions about the package. For now you can simply select the defaults by hitting `Enter`. This should create a `gw_task_plugin` folder in your current working directory.

1.3.1 Adding Task Code

Open the `gw_task_plugin/gw_task_plugin/tasks.py` file. You will find the following code.

```
from girder_worker.app import app
from girder_worker.utils import girder_job

# TODO: Fill in the function with the correct argument signature
# and code that performs the task.
@girder_job(title='Example Task')
@app.task(bind=True)
def example_task(self):
    pass
```

Edit example_task function to return the value "Hello World!".

1.4 Installing the Task Plugin

The cookiecutter template has created a barebones Python package which can now be installed with pip. Return to the folder with the outermost `gw_task_plugin` folder and install the package

```
$ pip install gw_task_plugin/
```

1.5 Running the Worker

Now run the worker from a command line

```
$ celery worker -A girder_worker.app -l info
```

If all is well, you should see a message similar to the following

```
-------------- celery@isengard v4.1.0 (latentcall)
---- **** -----
--- * ***  * -- Linux-4.15.5-1-ARCH-x86_64-with-glibc2.2.5 2018-02-27 19:28:07
-- * - **** ---
- ** ---------- [config]
- ** ---------- .> app:         girder_worker:0x7f72fd800ed0
- ** ---------- .> transport:   amqp://guest:**@localhost:5672//
- ** ---------- .> results:     amqp://
- *** --- * --- .> concurrency: 4 (prefork)
-- ******* ---- .> task events: OFF (enable -E to monitor tasks in this worker)
--- ***** -----
```

```
 -------------- [queues]
                .> celery           exchange=celery(direct) key=celery


[tasks]
  . girder_worker.docker.tasks.docker_run
  . gw_task_plugin.tasks.example_task

[2018-02-27 19:28:07,205: INFO/MainProcess] Connected to amqp://guest:**@127.0.0.
↪1:5672//
[2018-02-27 19:28:07,226: INFO/MainProcess] mingle: searching for neighbors
[2018-02-27 19:28:08,266: INFO/MainProcess] mingle: all alone
[2018-02-27 19:28:08,321: INFO/MainProcess] celery@isengard ready.
```

As long as `gw_task_plugin.tasks.example_task` is listed under the `[tasks]` section then you are ready to move on to the next section.

1.6 Executing the Task

In a separate terminal, open up a python shell and type the following:

```
$ python
```

Import the task:

```
>>> from gw_task_plugin.tasks import example_task
```

Execute the task asynchronously:

```
>>> a = example_task.delay()
>>> a.get()
u'Hello World!'
```

1.7 Wrapping Up

In this tutorial we briefly demonstrated how to:

- Install and run a broker
- Install Girder Worker
- Create and install a task plugin
- Execute the task remotely with a Python interpreter

The goal here was to get up and running as quickly as possible and so each of these topics has been treated lightly.

- Celery supports a few different brokers. For more information see Celery's complete broker documentation.
- Task plugin Python packages do more than just add a `setup.py` and create a `tasks.py` for dumping tasks into. For more information on what the boilerplate the cookiecutter created see *Plugins*.
- Girder Worker aims to provide task execution API that is exactly the same as Celery. For more information on calling tasks see Celery's Calling Tasks documentation. For more information about the knobs and dials available for changing how task execute, see Celery's Task documentation.

Finally, we *highly* recommend reading through the Celery's First Steps with Celery documentation as well as their User Guide. For some important differences between Celery and Girder Worker, we recommend keeping the important-differences page open while working through Celery's documentation.

CHAPTER 2

Installation

To install the Girder Worker on your system, we recommend using `pip` to install the package.

```
pip install girder-worker
```

That will install the core girder-worker library and the built-in *Plugins*.

2.1 Remote Execution

2.2 Configuration

Several aspects of the worker's behavior are controlled via its configuration file. The easiest way to manage configuration is using the `girder-worker-config` command that is installed with the package. After installation, run

```
$ girder-worker-config --help
```

You should see the list of available sub-commands for reading and writing config values. To show all configuration options, run

```
$ girder-worker-config list
```

To set a specific option, use

```
$ girder-worker-config set <section_name> <option_name> <value>
```

For example:

```
$ girder-worker-config set celery broker amqp://me@localhost/
```

To change a setting back to its default value, use the `rm` subcommand

```
$ girder-worker-config rm celery broker
```

The core configuration parameters are outlined below.

- `celery.app_main`: The name of the celery application. Clients will need to use this same name to identify what app to send tasks to. It is recommended to call this "girder_worker" unless you have a reason not to.
- `celery.broker`: This is the broker that celery will connect to in order to listen for new tasks. Celery recommends using RabbitMQ as your message broker.
- `girder_worker.tmp_root`: Each task is given a temporary directory that it can use if it needs filesystem storage. This config setting points to the root directory under which these temporary directories will be created.
- `girder_worker.plugins_enabled`: This is a comma-separated list of plugin IDs that will be enabled at runtime, e.g. `r,docker`.
- `girder_worker.plugin_load_path`: If you have any external plugins that are not inside the **girder_worker/plugins** package directory, set this value to a colon-separated list of directories to search for external plugins that need to be loaded.

Note: After making changes to values in the config file, you will need to restart the worker before the changes will be reflected.

CHAPTER 3

Plugins

3.1 Task Plugin from Cookiecutter

The simplest way to bootstrap your Girder Worker task plugin is to use our cookiecutter plugin to fill in the boilerplate. See *Creating a Task Plugin* for instructions.

3.2 Task Plugin from Scratch

This is an example plugin that demonstrates how to extend girder_worker by allowing it to run additional tasks. Plugins are implemented as separate pip installable packages. To install this example plugin you can checkout this code base, change directories to `examples/plugin_example/` and run `pip install .` This will add the `gwexample` plugin to girder_worker. If you then run girder_worker with a log level of 'info' (e.g. `girder-worker -l info`) you should see the following output:

```
(girder)$ girder-worker -l info

 -------------- celery@minastirith v3.1.23 (Cipater)
---- **** -----
--- * ***  * -- Linux-4.8.6-1-ARCH-x86_64-with-glibc2.2.5
-- * - **** ---
- ** ---------- [config]
- ** ---------- .> app:         girder_worker:0x7f69bfff1050
- ** ---------- .> transport:   amqp://guest:**@localhost:5672//
- ** ---------- .> results:     amqp://
- *** --- * --- .> concurrency: 32 (prefork)
-- ******* ----
--- ***** ----- [queues]
 -------------- .> celery           exchange=celery(direct) key=celery


[tasks]
  . girder_worker.convert
```

```
  . girder_worker.run
  . girder_worker.validators
  . gwexample.analyses.tasks.fibonacci

[2016-11-08 12:22:56,163: INFO/MainProcess] Connected to amqp://guest:**@127.0.0.
↪1:5672//
[2016-11-08 12:22:56,184: INFO/MainProcess] mingle: searching for neighbors
[2016-11-08 12:22:57,198: INFO/MainProcess] mingle: all alone
[2016-11-08 12:22:57,218: WARNING/MainProcess] celery@minastirith ready.
```

Notice that the task `gwexample.analyses.tasks.fibonacci` is now available. With the girder-worker processes running, you should be able to execute `python example_client.py` in the current working directory. After a brief delay, this should print out `121393` - the Fibonacci number for 26.

3.2.1 Writing your own plugin

Adding additional tasks to the girder_worker infrastructure is easy and takes three steps. (1) Creating tasks, (2) creating a plugin class and (3) adding a `girder_worker_plugins` entry point to your setup.py.

Creating tasks

Creating tasks follows the standard celery conventions. The only difference is the celery application that decorates the function should be imported from `girder_worker.app`. E.g.:

```
from girder_worker.app import app

@app.task
def fibonacci(n):
    if n == 1 or n == 2:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

Plugin Class

Each plugin must define a plugin class the inherits from *girder_worker.GirderWorkerPluginABC*. GirderWorkerPluginABC's interface is simple. The class must define an `__init__` function and a `task_imports` function. `__init__` takes the girder_worker's celery application as its first argument. This allows the plugin to store a reference to the application, or change configurations of the application as necessary. The `task_imports` function takes no arguments and must return a list of the package paths (e.g. importable strings) that contain the plugin's tasks. As an example:

```
from girder_worker import GirderWorkerPluginABC

class GWExamplePlugin(GirderWorkerPluginABC):
    def __init__(self, app, *args, **kwargs):
        self.app = app

        # Update the celery application's configuration
        # it is not necessary to change the application configuration
        # this is simply included to illustrate that it is possible.
        self.app.config.update({
            'TASK_TIME_LIMIT': 300
        })
```

```
    def task_imports(self):
        return ['gwexample.analyses.tasks']
```

Entry Point

Finally, in order to make the plugin class discoverable, each plugin must define a custom entry point in its `setup.py`. For our example, this entry point looks like this:

```
from setuptools import setup

setup(name='gwexample',
      # ....
      entry_points={
          'girder_worker_plugins': [
              'gwexample = gwexample:GWExamplePlugin',
          ]
      },
      # ....
      )
```

Python Entry Points are a way for python packages to advertise classes and objects to other installed packages. Entry points are defined in the following way:

```
entry_points={
    'entry_point_group_id': [
        'entry_point_name = importable.package.name:class_or_object',
    ]
}
```

The girder_worker package introduces a new entry point group `girder_worker_plugins`. This is followed by a list of strings which are parsed by setuptools. The strings must be in the form `name = module:plugin_class` Where `name` is an arbitrary string (by convention the name of the plugin), `module` is the importable path to the module containing the plugin class, and `plugin_class` is a class that inherits from `GirderWorkerPluginABC`.

3.2.2 Final notes

With these three components (Tasks, Plugin Class, Entry Point) you should be able to add arbitrary tasks to the girder_worker client.

3.3 Writing cancelable tasks

girder_worker provides support for signaling that a task should be canceled using Celery's revoke mechanism. In order for a task to be able to be canceled cleanly it must periodically check if it has been canceled, if it has then is can do any necessary cleanup and return. girder_worker provides a task base class (`girder_worker.utils.Task`) that provides a property that can be used to check if the task has been canceled. An example of its use is shown below:

```
from girder_worker.app import app

@app.task(bind=True)
def my_cancellable_task(task):
```

```
while not task.cancelled:
    # Do work
```

The Girder job model associated with the canceled task will be moved into the `JobStatus.CANCELED` state.

CHAPTER 4

Built-in Plugins

4.1 The docker_run Task

Girder Worker provides a built-in task that can be used to run docker containers. Girder Worker makes it easy to work on data held in girder from within a docker containers.

4.1.1 Container arguments

The `docker_run` task exposes a `container_args` parameter which can be used to pass arguments to the container entrypoint.

4.1.2 BindMountVolumes

The volumes to be bind mounted into a container can be passed to the `docker_run` task in one of two ways.

Using docker-py syntax

In this case the value of the `volumes` parameter is a `dict` conforming to specification defined by docker-py, which is passed directly to docker-py. For example

```
volumes = {
     '/home/docker/data': {
         'bind': '/mnt/docker/',
         'mode': 'rw'
     }

}
docker_run.delay('my/image', pull_image=True, volumes=volumes)
```

Using the BindMountVolume class

Girder Worker provides a utility class *girder_worker.docker.transforms.BindMountVolume* that can be used to define volumes that should be mounted into a container. These classes can also be used in conjunction with other parts of the girder_work docker infrastructure, for example providing a location where a file should be downloaded to. See *Downloading files from Girder*. When using the *girder_worker.docker.transforms.BindMountVolume* class a list of instances is provided as the value for the `volumes` parameter, Girder Worker will take care of ensuring that these volumes are mounted. In the example below we are creating a *girder_worker.docker.transforms.BindMountVolume* instance and passing it as a container argument to provide the mounted location to the container. Girder Worker will take care of transforming the instance into the approriate path inside the container.

```
vol = BindMountVolume('/home/docker/data', '/mnt/docker/')
docker_run.delay('my/image', pull_image=True, volumes=[vol], container_args=[vol])
```

4.1.3 Temporary Volume

A *girder_worker.docker.transforms.TemporaryVolume* class is provided representing a temporary directory on the host machine that is mounted into the container. `girder_worker.docker.transforms.TemporaryVolume.default` holds a default instance that is used as the default location for many other parts of the Girder Worker docker infrastructure, for example when downloading a file. See *Downloading files from Girder*. However, it can also be used explicitly, for example, here it is being passed as a container argument for use within a container. Again, Girder Worker will take care of transforming the *girder_worker.docker.transforms.TemporaryVolume* instance into the appropriate path inside the container, so the container entrypoint will simply received a path.

```
vol = BindMountVolume('/home/docker/data', '/mnt/docker/')
docker_run.delay('my/image', pull_image=True, container_args=[TemporaryVolume.
↪default])
```

Note that because we are using the default path, we don't have to add the instance to the `volumes` parameter as it is automatically added to the list of volumes to mount.

4.1.4 Downloading files from Girder

Accessing files held in girder from within a container is straightforward using the *girder_worker.docker.transforms.girder.GirderFileIdToVolume* utility class. One simply provides the file id as an argument to the constructor and passes the instance as a container argument.

```
docker_run.delay('my/image', pull_image=True,
    container_args=[GirderFileIdToVolume(file_id)])
```

The *girder_worker.docker.transforms.girder.GirderFileIdToVolume* instance will take care of downloading the file from Girder and passing the path it was downloaded to into the docker container's entrypoint as an argument.

If no `volume` parameter is specified then the file will be downloading to the task temporary volume. The file can also be downloaded to a specific *girder_worker.docker.transforms.BindMountVolume* by specifying a volume parameter, as follows:

```
vol = BindMountVolume(host_path, container_path)
docker_run.delay('my/image', pull_image=True,
    container_args=[GirderFileIdToVolume(file_id,volume=vol)])
```

If the file being downloaded is particularly large you may want to consider streaming it into the container using a named pipe. See *Streaming Girder files into a container* for more details.

4.1.5 Uploading files to Girder items

Utility classes are also provided to simplify uploading files generated by a docker container. The *girder_worker.docker.transforms.girder.GirderUploadVolumePathToItem* provides the functionality to upload a file to an item. In the example below, we use the *girder_worker.docker.transforms.VolumePath* utility class to define a file path that we then pass to the docker container. The docker container can write data to this file path. As well as passing the *girder_worker.docker.transforms.VolumePath* instance as a container argument we also pass it to *girder_worker.docker.transforms.girder.GirderUploadVolumePathToItem* , the *girder_worker.docker.transforms.girder.GirderUploadVolumePathToItem* instance is added to `girder_result_hooks`. This tells Girder Worker to upload the file path to the item id provided once the docker container has finished running.

```
volumepath = VolumePath('write_data_to_be_upoaded.txt')
docker_run.delay('my/image', pull_image=True, container_args=[volumepath],
    girder_result_hooks=[GirderUploadVolumePathToItem(volumepath, item_id)])
```

4.1.6 Using named pipes to stream data in and out of containers

Girder Worker uses named pipes as a language agnostic way of streaming data in and out of docker containers. Basically a named pipe is created at a path that is mounted into the container. This allows the container to open that pipe for read or write and similarly the Girder Worker infrastructure can open the pipe on the host, thus allowing data write and read from the container.

The are two utility classes used to represent a named pipe, *girder_worker.docker.transforms.NamedOutputPipe* and *girder_worker.docker.transforms.NamedInputPipe*.

NamedOuputPipe

This represents a named pipe that can be opened in a docker container for write, allowing data to be streamed out of a container.

NamedInputPipe

This represents a named pipe that can be opened in a docker container for read, allowing data to be streamed into a container.

These pipes can be connected together using the *girder_worker.docker.transforms.Connect* utility class.

Streaming Girder files into a container

One common example of using a named pipe is to stream a potentially large file into a container. This approach allows the task to start processing immediately rather than having to wait for the entire file to download, it also removes the requirement that the file is held on the local filesystem. In the example below we are creating an instance of *girder_worker.docker.transforms.girder.GirderFileIdToStream* that provides the ability to download a file in chunks. We are also creating a named pipe called `read_in_container`, as no `volume` argument is provided this pipe will be created on the temporary volume automatically mounted by Girder Worker. Finally, we are using the *girder_worker.docker.transforms.Connect* class to "connect" the stream to the pipe

and we pass the instance as a container argument. Girder Worker will take care of the select logic to stream the file into the pipe.

```
stream = GirderFileIdToStream(file_id)
pipe = NamedInputPipe('read_in_container')
docker_run('my/image', pull_image=True, container_args=[Connect(stream, pipe)])
```

All the container has to do is open the path passed into the container entry point and start reading. Below is an example python entry point:

```
# Simply open the path passed into the container.
with open(sys.argv[1]) as fp:
    fp.read() # This will be reading the files contents
```

4.1.7 Streaming progress reporting from Docker tasks to Girder jobs

The *girder_worker.docker.transforms.girder.ProgressPipe* class can be used to facilitate streaming real-time progress reporting from a docker task to its associated Girder job. It uses a named pipe to provide a simple interface within the container that is usable from any runtime environment.

The following example code shows the Girder side task invocation for using `ProgressPipe`:

```
from girder_worker.docker.tasks import docker_run
from girder_worker.docker.transforms.girder import ProgressPipe

docker_run.delay('my_docker_image:latest', container_args=[ProgressPipe()])
```

The corresponding example code running in the container entrypoint uploads progress events at regular intervals, which will automatically reflect in the job progress on the Girder server. This code is shown in python, but the idea is the same regardless of language.

```
import json
import sys
import time

with open(sys.argv[1], 'w') as pipe:
    for i in range(10):
        pipe.write(json.dumps({
            'message': 'Step %d of 10' % i,
            'total': 10,
            'current': i + 1
        }))
        pipe.flush()
        time.sleep(1)
```

The messages written to the pipe must be one per line, and each message must be a JSON Object containing optional `message`, `current`, and `total` values. You **must** call `flush()` on the file handle explicitly for your message to be flushed, since it is a named pipe.

4.1.8 Attaching intermediate / optional artifacts to Girder jobs

It's often useful for debugging/tracing or algorithm analysis to be able to inspect intermediate outputs or other artifacts produced during execution of a task, even (perhaps especially) if the task fails. These artifacts differ from normal output transforms that upload files to Girder in two ways. Firstly, they are optional; if the specified file or directory does not exist, it does not cause any errors. This allows docker image authors to choose either at build time or runtime

whether or not to create and upload artifacts. Secondly, the artifact files are attached to the job document itself, rather than placed within the Girder data hierarchy. This facilitates inspection of job artifacts inline with things like the log and status fields.

The following example code shows an example Girder-side usage of the *girder_worker.docker.transforms.girder.GirderUploadVolumePathJobArtifact* transform to upload job artifacts from your docker task.

```
from girder_worker.docker.tasks import docker_run
from girder_worker.docker.transforms import VolumePath
from girder_worker.docker.transforms.girder import GirderUploadVolumePathJobArtifact

artifacts = VolumePath('job_artifacts')
docker_run.delay(
    'my_docker_image:latest', container_args=[
        artifacts
    ],
    girder_result_hooks=[
        GirderUploadVolumePathJobArtifact(artifacts)
    ])
```

Note that you can write to this path inside your container and make it either a directory or a single file. If it's a directory, all files within the directory will be uploaded and attached to the job as artifacts. This operation is not recursive, i.e. it will not upload anything under subdirectories of the top level directory.

It's often useful to upload any artifact files even if the `docker_run` task failed. For that behavior, simply pass an additional argument to the transform:

```
GirderUploadVolumePathJobArtifact(artifacts, upload_on_exception=True)
```

4.1.9 MacOS Volume mounting issue workaround

Due to some odd symlinking behavior by Docker engine on MacOS, it may be necessary to add a workaround when running the girder_worker. If your `TMPDIR` environment variable is underneath the `/var` directory and you see errors from Docker about `MountsDenied`, try running girder worker with the `TMPDIR` set underneath `/private/var` instead of `/var`. The location should be equivalent since `/var` is a symlink to `/private/var`.

CHAPTER 5

Using Girder Worker with Girder

The most common use case of Girder Worker is running processing tasks on data managed by a Girder server. Typically, either a user action or an automated process running on the Girder server initiates the execution of a task that runs on a Girder Worker.

The task to be run must be installed in both the Girder server environment as well as the worker environment. If you are using a built-in plugin, you can just install girder-worker on the Girder server environment. If you're using a custom task plugin, `pip install` it on both the workers and the Girder server environment.

5.1 Running tasks as Girder jobs

Once installed, starting a job is as simple as importing the task into the python environment and calling *delay()* on it. The following example assumes your task exists in a package called `my_worker_tasks`:

```
from my_worker_tasks import my_task

result = my_task.delay(arg1, arg2, kwarg1='hello', kwarg2='world')
```

Here the `result` variable is a celery result object with Girder-specific properties attached. Most importantly, it contains a `job` attribute that is the created job document associated with this invocation of the task. That job will be owned by the user who initiated the request, and Girder worker will automatically update its status according to the task's execution state. Additionally, any standard output or standard error data will be automatically added to the log of that job. You can also set fields on the job using the *delay* method kwargs `girder_job_title`, `girder_job_type`, `girder_job_public`, and `girder_job_other_fields`. For instance, to set the title and type of the created job:

```
job = my_task.delay(girder_job_title='This is my job', girder_job_type='my_task')
assert job['title'] == 'This is my job'
assert job['type'] == 'my_task'
```

5.2 Downloading files from Girder for use in tasks

Note: This section applies to python tasks, if you are using the built-in `docker_run` task, it has its own set of transforms for dealing with input and output data, which are detailed in the *The docker_run Task* documentation

The following example makes use of a Girder Worker transform for passing a Girder file into a Girder Worker task. The `girder_worker_utils.transforms.girder_io.GirderFileId` transform causes the file with the given ID to be downloaded locally to the worker node, and its local path will then be passed into the function in place of the transform object. For example:

```
from girder_worker_utils.transforms.girder_io import GirderFileId

def process_file(file):
    return my_task.delay(input_file=GirderFileId(file['_id'])).job
```

CHAPTER 6

API documentation

6.1 Core

class girder_worker.**GirderWorkerPluginABC**(*app*, **args*, ***kwargs*)

Abstract base class for Girder Worker plugins. Plugins must descend from this class; see the *Plugins* section for more information.

task_imports()

Plugins must override this method.

class girder_worker.task.**Task**

Girder Worker Task object. Tasks defined by plugins must be subclasses of this class, however you will typically not need to reference it yourself, as it will be automatically instantiated by the girder_worker celery app. See *Creating tasks* for instructions.

canceled

A property to indicate if a task has been canceled.

Returns True is this task has been canceled, False otherwise.

Return type bool

6.1.1 Transforms

class girder_worker_utils.transforms.girder_io.**GirderUploadJobArtifact**(*job_id=None*, *name=None*, ***kwargs*)

This class can be used to upload a directory of files or a single file as artifacts attached to a Girder job. These files are only uploaded if they exist, so this is an optional output.

Currently, only a flat directory of files is supported; the transform does not recurse through nested directories, though that may change in the future.

6.2 Docker

6.2.1 Tasks

6.2.2 Transforms

class girder_worker.docker.transforms.**BindMountVolume**(*host_path*, *container_path*, *mode='rw'*)

A volume that will be bind mounted into a docker container.

Parameters

- **host_path** (*str*) – The path on the host machine.
- **container_path** (*str*) – The path in the container this volume will be mounted at.
- **mode** (*str*) – The mounting mode

class girder_worker.docker.transforms.**ChunkedTransferEncodingStream**(*url*, *headers={}*, ***kwargs*)

A stream transform that allows data to be streamed using HTTP Chunked Transfer Encoding to a server.

Parameters

- **url** (*str*) – Destination URL for the stream.
- **headers** – HTTP headers to send.

class girder_worker.docker.transforms.**Connect**(*input*, *output*)

This utility class represents the connection between a *girder_worker.docker.transforms.NamedOutputPipe* or *girder_worker.docker.transforms.NamedInputPipe* and one of the other streaming transforms. Girder Worker will stream the data to or from the named pipe.

Parameters

- **input** (*girder_worker.docker.transforms.NamedOutputPipe* or *girder_worker.docker.transforms.girder.GirderFileIdToStream*) – The input side of the connection
- **output** (*girder_worker.docker.transforms.NamedInputPipe* or *girder_worker.docker.transforms.ChunkedTransferEncodingStream* or *girder_worker.docker.transforms.HostStdOut* or *girder_worker.docker.transforms.HostStdErr*) – The output side of the connection

class girder_worker.docker.transforms.**ContainerStdErr**

Represents the standard error stream of the container. Can be used with *girder_worker.docker.transforms.Connect* to redirect the containers standard error to another stream.

class girder_worker.docker.transforms.**ContainerStdOut**

Represents the standard output stream of the container. Can be used with *girder_worker.docker.transforms.Connect* to redirect the containers standard output to another stream.

class girder_worker.docker.transforms.**HostStdErr**

Represents the standard error stream on the host machine. Can be used with *girder_worker.docker.transforms.Connect* to write text to stderr.

class girder_worker.docker.transforms.**HostStdOut**
Represents the standard output stream on the host machine. Can be used with *girder_worker.docker.transforms.Connect* to write text to stdout.

class girder_worker.docker.transforms.**NamedInputPipe**(*name*, *container_path=None*, *host_path=None*, *volume=<girder_worker.docker.transforms._DefaultTempo object>*)
A named pipe that can be open for read within a docker container. i.e. To stream data into a container.

Parameters

- **name** (*str*) – The name of the pipe.
- **container_path** (*str*) – The path in the container.
- **host_path** (*str*) – The path on the host machine.
- **volume** – Alternatively a *girder_worker.docker.transforms.BindMountVolume* instance can be provided. In which case the container_path and host_paths from the volume will be used when creating the pipe. The default location is `girder_worker.docker.transforms.TemporaryVolume.default`

class girder_worker.docker.transforms.**NamedOutputPipe**(*name*, *container_path=None*, *host_path=None*, *volume=<girder_worker.docker.transforms._DefaultTemp object>*)
A named pipe that can be opened for write within a docker container. i.e. To stream data out of a container.

Parameters

- **name** (*str*) – The name of the pipe.
- **container_path** (*str*) – The path in the container.
- **host_path** (*str*) – The path on the host machine.
- **volume** – Alternatively a *girder_worker.docker.transforms.BindMountVolume* instance can be provided. In which can the container_path and host_paths from the volume will be use when creating the pipe. The default location is `girder_worker.docker.transforms.TemporaryVolume.default`

class girder_worker.docker.transforms.**TemporaryVolume**(*host_dir=None*, *mode=493*)
This is a class used to represent a temporary directory on the host that will be mounted into a docker container. girder_worker will automatically attach a default temporary volume. This can be reference using *TemporaryVolume.default* class attribute. A temporary volume can also be create in a particular host directory by providing the *host_dir* param.

Parameters

- **host_dir** (*str*) – The root directory on the host to use when creating the the temporary host path.
- **mode** (*int*) – The default mode applied to the temporary volume if it does not already exist.

class girder_worker.docker.transforms.**VolumePath**(*filename*, *volume=<girder_worker.docker.transforms._DefaultTemporaryV object>*)
A path on a docker volume. Must be a path relative to the root of the volume.

Parameters

- **filename** – The file name.

- **volume** (*girder_worker.docker.transforms.BindMountVolume*) – The volume this file lived on. If no volume is provided then the file will be on `girder_worker.docker.transforms.TemporaryVolume.default`

class girder_worker.docker.transforms.girder.**GirderFileIdToStream**(*_id*, ***kwargs*)

This can be used to stream a Girder file into a docker container. See *Streaming Girder files into a container* for example usage.

Parameters **_id** (*str or ObjectId*) – The Girder file ID.

class girder_worker.docker.transforms.girder.**GirderFileIdToVolume**(*_id*, *volume=<girder_worker.docker.transfo object>*, *filename=None*, ***kwargs*)

This can be used to pass a Girder file into a docker container. It downloads the file to a bind mounted volume, and returns the container path of the file.

Parameters

- **_id** (*str or ObjectId*) – The Girder file ID.
- **volume** (*girder_worker.docker.transforms.BindMountVolume*) – The bind mount volume where the file will reside.
- **filename** (*str*) – Alternate name for the file. Default is to use the name from Girder.

class girder_worker.docker.transforms.girder.**GirderFolderIdToVolume**(*_id*, *volume=<girder_worker.docker.tran object>*, *folder_name=None*, ***kwargs*)

This can be used to pass a Girder folder into a docker container. It downloads the folder to a bind mounted volume, and returns the container path of the directory.

Parameters

- **_id** (*str or ObjectId*) – The Girder folder ID.
- **volume** (*girder_worker.docker.transforms.BindMountVolume*) – The bind mount volume where the directory will reside.
- **folder_name** (*str*) – Alternate name for the directory. Default is to use the name from Girder.

class girder_worker.docker.transforms.girder.**GirderUploadVolumePathJobArtifact**(*volumepath*, *job_id=None*, *name=None*, *upload_on_excepti **kwargs*)

This transform can be used to upload artifacts produced during a docker task execution and attach them to the corresponding job in Girder. This can be useful for tracing and debugging jobs, or simply collecting intermediate information during job execution. If the passed in path does not exist, this is a no-op.

Parameters

- **volumepath** (*girder_worker.docker.transforms.VolumePath*) – A volume path pointing to a mounted directory or file. If a directory, all files within the directory

will be uploaded as artifacts to the job. If a file, just uploads the single file. If it does not exist, no action is performed.

- **job_id** (*str*) – The job ID to attach the artifacts to. If calling this from Girder via `docker_run.delay`, you will not need to set this, as it will be set automatically.
- **name** (*str*) – A name for the artifact. Only applies for single file paths. If not specified, will use the basename of the file.
- **upload_on_exception** (*bool*) – If True, this transform will occur even if the docker task fails. This can be used to debug failed `docker_run` tasks.

class girder_worker.docker.transforms.girder.**GirderUploadVolumePathToFolder**(*volumepath*, *folder_id*, *delete_file=False*, ***kwargs*)

This transform uploads data in a bind mount volume to a Girder folder. This should be used in `girder_result_hooks` to upload data produced by the task.

Parameters

- **volumepath** (*girder_worker.docker.transforms.VolumePath*) – The location of the file or directory to upload.
- **folder_id** (*str or ObjectId*) – The folder ID in Girder.
- **delete_file** (*bool*) – Whether to delete the data afterward.

class girder_worker.docker.transforms.girder.**GirderUploadVolumePathToItem**(*volumepath*, *item_id*, *delete_file=False*, ***kwargs*)

This transform uploads data in a bind mount volume to a Girder item. This should be used in `girder_result_hooks` to upload files produced by the task.

Parameters

- **volumepath** (*girder_worker.docker.transforms.VolumePath*) – The location of the file to upload.
- **item_id** (*str or ObjectId*) – The item ID in Girder.
- **delete_file** (*bool*) – Whether to delete the file afterward.

class girder_worker.docker.transforms.girder.**ProgressPipe**(*name='.girder_progress'*, *volume=<girder_worker.docker.transforms._Defau*
object>)

This can be used to stream progress information out of a running docker container as part of a docker_run task. For a usage example, see *Streaming progress reporting from Docker tasks to Girder jobs*.

Parameters

- **name** (*str*) – The filename, which will be a named pipe open for reading from the host.
- **volume** (*girder_worker.docker.transforms.BindMountVolume*) – The bind mount volume where the underlying named pipe will reside.

Indices and tables

- genindex
- modindex

- search

Python Module Index

g

Index