Girder Worker Documentation

Release 0.1.0

Kitware, Inc.

Jul 17, 2017

Contents

1 What is Girder Worker? 1

1.1 Installation . . . 1
1.2 Types and formats . . . 3
1.3 API documentation . . . 6
1.4 Examples . . . 17
1.5 Developer documentation . . . 28
1.6 Task Plugins . . . 30
1.7 Application Plugins . . . 32

2 Indices and tables 39

Python Module Index 41

CHAPTER 1

What is Girder Worker?

Girder Worker is a Python application for generic task execution. It can be run within a Celery worker to provide a distributed batch job execution platform.

The application can run tasks in a variety of languages and environments, including Python, R, and Docker, all via a single Python or Celery broker interface. Tasks can be chained together into workflows, and these workflows can actually span multiple languages and environments seamlessly. Data flowing between tasks can be automatically converted into a format understandable in the target environment. For example, a Python object from a Python task can be automatically converted into an R object for an R task at the next stage of a pipeline.

Girder Worker defines a specification that prescribes a loose coupling between a task and its runtime inputs and outputs. That specification is described in the *API documentation* section. This specification is language-independent and instances of the spec are best represented by a hierarchical data format such as JSON or YAML, or an equivalent serializable type such as a `dict` in Python. Several *Examples* of using these specifications to generate tasks and workflows are provided.

Girder Worker is designed to be easily extended to new languages and environments, or to support new data types and formats, or modes of data transfer. This is accomplished via its plugin system, which is described in *Application Plugins*.

Installation

To install the Girder Worker on your system, we recommend using `pip` to install the package. (If you wish to install from source, see the *Installing from source* section of the developer documentation.)

First, install required system packages:

```
# Command for Ubuntu
sudo apt-get install libjpeg-dev zlib1g-dev libssl-dev
```

Next, the following command will install the core dependencies:

```
pip install girder-worker
```

That will install the core girder-worker library, but not the third-party dependencies for any of its plugins. If you want to enable a set of plugins, their IDs should be included as extras to the pip install command. For instance, if you are planning to use the R plugin and the girder_io plugin, you would run:

```
pip install girder-worker[r,girder_io]
```

You can run this command at any time to install dependencies of other plugins, even if the girder worker is already installed.

Remote Execution

Want to run things remotely? Girder worker relies on celery as its distributed task queue. Celery requires a message broker, which can be Mongo, though Celery recommends using RabbitMQ as your message broker.

If you have followed the standard or development installation process, celery will have already been installed.

Run the girder_worker, which will run a celery worker process:

```
girder-worker
```

On the client, run a script akin to the following example:

```
python examples/example_client.py
```

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.

Types and formats

In Girder Worker, analysis inputs and outputs may contain type and format annotations. These annotations only have an effect if the `types` plugin is enabled on the worker, and the specific behaviors of validation and conversion of data formats are controlled by flags to the `run` task called `validate` and `auto_convert` respectively, which default to being enabled if they are not explicitly passed.

A *type* in Girder Worker is a high-level description of a data structure useful for intuitive workflows. It is not tied to a particular representation. For example, the *table* type may be defined as a list of rows with ordered, named column fields. This description does not specify any central representation since the information may be stored in a variety of ways. A type is specified by a string unique to your Girder Worker environment, such as `"table"` for the table type.

An explicit representation of data is called a *format* in Girder Worker. A format is a low-level description of data layout. For example, the table type may have formats for CSV, database table, R data frame, or JSON. The format may be text, serialized binary, or even in-memory data layouts. Just like types, a format is specified by a string unique to your Girder Worker environment, such as `"csv"` for the CSV format. Formats under the same type should be convertible between each other.

Notice that the above uses the phrases such as "may be defined" and "may have formats". This is because at its core Girder Worker does not contain types or formats. The `girder_worker.run()` function will attempt to match given input bindings to analysis inputs, validating data and performing conversions as needed. To make Girder Worker aware of certain types and formats, you must define validation and conversion routines. These routines are themselves Girder Worker algorithms of a particular form, loaded with *girder_worker.plugins.types.format.import_converters()*. See that function's documentation for how to define validators and converters.

The following are the types available in Girder Worker core. Application plugins may add their own types and formats using the `girder_worker.plugins.types.format.import_converters` function. See the *Application Plugins* section for details on plugin-specific types and formats.

`"boolean"` type

A true or false value. Formats:

`"boolean"` An in-memory Python `bool`.

`"json"` A JSON string representing a single boolean (`"true"` or `"false"`).

`"integer"` type

An integer. Formats:

"integer" An in-memory Python `int`.

"json" A JSON string representing a single integer.

"number" type

A numeric value (integer or real). Formats:

"number" An in-memory Python `int` or `float`.

"json" A JSON string representing a single number.

"string" type

A sequence of characters.

"text" A raw string of characters (`str` in Python).

"json" A JSON string representing a single string. This is a quoted string with certain characters escaped.

"integer_list" type

A list of integers. Formats:

"integer_list" An in-memory list of Python `int`.

"json" A JSON string representing a list of integers.

"number_list" type

A list of numbers (integer or real). Formats:

"number_list" An in-memory list of Python `int` or `float`.

"json" A JSON string representing a list of numbers.

"string_list" type

A list of strings. Formats:

"string_list" An in-memory list of Python `str`.

"json" A JSON string representing a list of strings.

"table" type

A list of rows with ordered, named column attributes. Formats:

"rows" A Python dictionary containing keys `"fields"` and `"rows"`. `"fields"` is a list of column names that specifies column order. `"rows"` is a list of dictionaries of the form `field: value` where `field` is the field name and `value` is the value of the field for that row. For example:

```
{
    "fields": ["one", "two"],
    "rows": [{"one": 1, "two": 2}, {"one": 3, "two": 4}]
}
```

"rows.json" The equivalent JSON representation of the "rows" format.

"objectlist" A Python list of dictionaries of the form field: value where field is the field name and value is the value of the field for that row. For example:

```
[{"one": 1, "two": 2}, {"one": 3, "two": 4}]
```

This is identical to the "rows" field of the "rows" format. Note that this format does not preserve column ordering.

"objectlist.json" The equivalent JSON representation of the "objectlist" format.

"objectlist.bson" The equivalent BSON representation of the "objectlist" format. This is the format of MongoDB collections.

"csv" A string containing the contents of a comma-separated CSV file. The first line of the file is assumed to contain column headers.

"tsv" A string containing the contents of a tab-separated TSV file. Column headers are detected the same as for the "csv" format.

"tree" type

A hierarchy of nodes with node and/or link attributes. Formats:

"nested" A nested Python dictionary representing the tree. All nodes may contain a "children" key containing a list of child nodes. Nodes may also contain "node_data" and "edge_data" which are name: value dictionaries of node and edge attributes. The top-level (root node) dictionary contains "node_fields" and "edge_fields" which are lists of node and edge attribute names to preserve ordering. The root should not contain "edge_data" since it does not have a parent edge. For example:

```
{
    "edge_fields": ["weight"],
    "node_fields": ["node name", "node weight"],
    "node_data": {"node name": "", "node weight": 0.0},
    "children": [
        {
            "node_data": {"node name": "", "node weight": 2.0},
            "edge_data": {"weight": 2.0},
            "children": [
                {
                    "node_data": {"node name": "ahli", "node weight": 2.
↪0},
                    "edge_data": {"weight": 0.0}
                },
                {
                    "node_data": {"node name": "allogus", "node weight":␣
↪3.0},
                    "edge_data": {"weight": 1.0}
                }
            ]
        },
```

```
        {
            "node_data": {"node name": "rubribarbus", "node weight": 3.0}
↪,
            "edge_data": {"weight": 3.0}
        }
    ]
}
```

"nested.json" The equivalent JSON representation of the "nested" format.

"newick" A tree in Newick format.

"nexus" A tree in Nexus format.

"phyloxml" A phylogenetic tree in PhyloXML format.

"graph" type

A collection of nodes and edges with optional attributes. Formats:

"networkx" An in-memory representation of a graph using an object of type nx.Graph (or any of its subclasses).

"networkx.json" A JSON representation of a NetworkX graph.

"clique.json" A JSON representation of a Clique graph.

"graphml" An XML String representing a valid GraphML representation.

"adjacencylist" A string representing a very simple adjacency list which does not preserve node or edge attributes.

"image" type

A 2D matrix of uniformly-typed numbers. Formats:

"png" An image in PNG format.

"png.base64" A Base-64 encoded PNG image.

"pil" An image as a PIL.Image from the Python Imaging Library.

API documentation

Overview

The main purpose of Girder Worker is to execute a broad range of tasks. These tasks, along with a set of input bindings and output bindings are passed to the girder_worker.tasks.run() function, which is responsible for fetching the inputs as necessary and executing the task, and finally populating any output variables and sending them to their destination.

The task, its inputs, and its outputs are each passed into the function as Python dictionaries. In this section, we describe the structure of each of those dictionaries.

The task specification

The first argument to `girder_worker.tasks.run()` describes the task to execute, independently of the actual data that it will be executed upon. The most important field of the task is the `mode`, which describes what type of task it is. The structure for the task dictionary is described below. Uppercase names within angle braces represent symbols defined in the specification. Optional parts of the specification are surrounded by parentheses to avoid ambiguity with the square braces, which represent lists in Python or Arrays in JSON. The Python task also accepts a `write_script` parameter that when set to 1 will write task scripts to disk before executing them. This aids in readability for interactive debuggers such as `pdb`.

```
<TASK> ::= <PYTHON_TASK> | <R_TASK> | <DOCKER_TASK> | <WORKFLOW_TASK>

<PYTHON_TASK> ::= {
    "mode": "python",
    "script": <Python code to run as a string>
    (, "inputs": [<TASK_INPUT> (, <TASK_INPUT>, ...)])
    (, "outputs": [<TASK_OUTPUT> (, <TASK_OUTPUT>, ...)])
    (, "write_script": 1)
}

<R_TASK> ::= {
    "mode": "r",
    "script": <R code to run (as a string)>
    (, "inputs": [<TASK_INPUT> (, <TASK_INPUT>, ...)])
    (, "outputs": [<TASK_OUTPUT> (, <TASK_OUTPUT>, ...)])
}

<DOCKER_TASK> ::= {
    "mode": "docker",
    "docker_image": <Docker image name to run>
    (, "container_args": [<container arguments>])
    (, "entrypoint": <custom override for container entry point>)
    (, "inputs": [<TASK_INPUT> (, <TASK_INPUT>, ...)])
    (, "outputs": [<TASK_OUTPUT> (, <TASK_OUTPUT>, ...)])
    (, "progress_pipe": <set to true to create a channel for progress notifications>)
}

<WORKFLOW_TASK> ::= {
    "mode": "workflow",
    "steps": [<WORKFLOW_STEP> (, <WORKFLOW_STEP>, ...)],
    "connections": [<WORKFLOW_CONNECTION> (, <WORKFLOW_CONNECTION>, ...)]
    (, "inputs": [<TASK_INPUT> (, <TASK_INPUT>, ...)])
    (, "outputs": [<TASK_OUTPUT> (, <TASK_OUTPUT>, ...)])
}

<WORKFLOW_STEP> ::= {
    "name": <step name>,
    "task": <TASK>
}

<WORKFLOW_CONNECTION> ::= {
    ("name": <name of top-level input to bind to>)
    (, "input": <input id to bind to for a step>)
    (, "input_step": <input step name to connect>)
    (, "output_step": <output step name to connect>)
}
```

The workflow mode simply allows for a directed acyclic graph of tasks to be specified to `girder_worker.run()`.

See also:

Visualize Facebook data with Girder Worker in *Examples* A full example of how to create workflows in Girder Worker.

```
<TASK_INPUT> ::= {
    "id": <string, the variable name>
    (, "default": <default value if none is bound at runtime>)
    (, "target": <INPUT_TARGET_TYPE>)   ; default is "memory"
    (, "filename": <name of file if target="filepath">)
    (, "stream": <set to true to indicate a streaming input>)
}

<INPUT_TARGET_TYPE> ::= "memory" | "filepath"

<TASK_OUTPUT> ::= {
    "id": <string, the variable name>,
    (, "target": <INPUT_TARGET_TYPE>)   ; default is "memory"
    (, "stream": <set to true to indicate a streaming output>)
}
```

The input specification

The `inputs` argument to `girder_worker.run()` specifies the inputs to the task described by the `task` argument. Specifically, it tells what data should be placed into the task input ports.

```
<INPUTS> ::= {
    <id> : <INPUT_BINDING>
    (, <id> : <INPUT_BINDING>)
    (, ...)
}
```

The input spec is a dictionary mapping each `id` (corresponding to the `id` key of each task input) to its data binding for this execution.

```
<INPUT_BINDING> ::= <INPUT_BINDING_HTTP> | <INPUT_BINDING_LOCAL> |
                    <INPUT_BINDING_MONGODB> | <INPUT_BINDING_INLINE>

<INPUT_BINDING_HTTP> ::= {
    "mode": "http",
    "url": <url of data to download>
    (, "params": <dict of URL parameters to encode>)
    (, "headers": <dict of HTTP headers to send when fetching>)
    (, "method": <http method to use, default is "GET">)
    (, "maxSize": <integer, max size of download in bytes>)
}
```

The http input mode specifies that the data should be fetched over HTTP. Depending on the `target` field of the corresponding task input specifier, the data will either be passed in memory, or streamed to a file on the local filesystem, and the variable will be set to the path of that file.

```
<INPUT_BINDING_LOCAL> ::= {
    "mode": "local",
    "path": <path on local filesystem to the file>
}
```

The local input mode denotes that the data exists on the local filesystem. Its contents will be read into memory and the variable will point to those contents.

```
<INPUT_BINDING_MONGODB> ::= {
    "mode": "mongodb",
    "db": <the database to use>,
    "collection": <the collection to fetch from>
    (, "host": <mongodb host, default is "localhost">)
}
```

The mongodb input mode specifies that the data should be fetched from a mongo collection. This simply binds the entire BSON-encoded collection to the input variable.

```
<INPUT_BINDING_INLINE> ::= {
    "mode": "inline",
    "data": <data to bind to the variable>
}
```

The inline input mode simply passes the data directly in the input binding dictionary as the value of the "data" key. Do not use this for any data that could be large.

Note: The `mode` field is inferred in a few special cases. If there is a `url` field, the `mode` is assumed to be `"http"`, and if there is a `data` field, the `mode` is assumed to be `"inline"`. For example, the following input specifications are equivalent:

```
{
    'url': 'https://upload.wikimedia.org/wikipedia/en/2/24/Lenna.png'
}
```

```
{
    'mode': 'http',
    'url': 'https://upload.wikimedia.org/wikipedia/en/2/24/Lenna.png'
}
```

The following two specifications are also equivalent:

```
{
    'data': 'hello'
}
```

```
{
    'mode': 'inline',
    'data': 'hello'
}
```

The output specification

The optional `outputs` argument to `girder_worker.run()` specifies output variables of the task that should be handled in some way.

```
<OUTPUTS> ::= {
    <id> : <OUTPUT_BINDING>
    (, <id> : <OUTPUT_BINDING>)
    (, ...)
}
```

The output spec is a dictionary mapping each `id` (corresponding to the `id` key of each task output) to some behavior that should be performed with it. Task outputs that do not have bindings in the ouput spec simply get their results set in the return value of `girder_worker.run()`.

```
<OUTPUT_BINDING> ::= <OUTPUT_BINDING_HTTP> | <OUTPUT_BINDING_LOCAL> |
                     <OUTPUT_BINDING_MONGODB>

<OUTPUT_BINDING_HTTP> ::= {
    "mode": "http",
    "url": <url to upload data to>,
    (, "headers": <dict of HTTP headers to send with the request>)
    (, "method": <http method to use, default is "POST">)
    (, "params": <dict of HTTP query parameters to send with the request>)
}

<OUTPUT_BINDING_LOCAL> ::= {
    "mode": "local",
    "path": <path to write data on the local filesystem>
}
```

The local output mode writes the data to the specified path on the local filesystem.

```
<OUTPUT_BINDING_MONGODB> ::= {
    "mode": "mongodb",
    "db": <mongo database to write to>,
    "collection": <mongo collection to write to>
    (, "host": <mongo host to connect to>)
}
```

The mongodb output mode attempts to BSON-decode the bound data, and then overwrites any data in the specified collection with the output data.

Script execution

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

`task_imports`()

Formats

class `girder_worker.plugins.types.format.`**`Validator`**

`type`
The validator type, like `string`.

`format`
The validator format, like `text`.

`is_valid`()
Return whether the type/format combination is valid.

If format is None, checks for the presence of any valid type/format with the specified type.

Returns `True` if `type` and `format` are a valid, loaded type/format pair.

girder_worker.plugins.types.format.**converter_path**(*source*, *target*)

Gives the shortest path that should be taken to go from a source type/format to a target type/format.

Throws a `NetworkXNoPath` exception if it can not find a path.

Parameters

- **source** – Validator tuple indicating the type/format being converted *from*.
- **target** – `Validator` tuple indicating the type/format being converted *to*.

Returns An ordered list of the analyses that need to be run to convert from `source` to `target`.

girder_worker.plugins.types.format.**get_validator_analysis**(*validator*)

Gets a validator's analysis from the conversion graph.

```
>>> analysis = get_validator_analysis(Validator('string', 'text'))
```

Returns an analysis dictionary

```
>>> type(analysis) == dict
True
```

Which contains an inputs key

```
>>> 'inputs' in analysis
True
```

If the validator doesn't exist, an exception will be raised

```
>>> get_validator_analysis(Validator('foo', 'bar'))
Traceback (most recent call last):
...
Exception: No such validator foo/bar
```

Parameters **validator** – A `Validator` namedtuple

Returns A dictionary containing the runnable analysis.

girder_worker.plugins.types.format.**has_converter**(*source*, *target=Validator(type=None, format=None)*)

Determines if any converters exist from a given type, and optionally format.

Underneath, this just traverses the edges until it finds one which matches the arguments.

Parameters

- **source** – `Validator` tuple indicating the type/format being converted *from*.
- **target** – `Validator` tuple indicating the type/format being converted *to*.

Returns `True` if it can converter from `source` to `target`, `False` otherwise.

girder_worker.plugins.types.format.**import_converters**(*search_paths*)

Import converters and validators from the specified search paths. These functions are loaded into `girder_worker.format.conv_graph` with nodes representing validators, and directed edges representing converters.

Any files in a search path matching `validate_*.json` are loaded as validators. Validators should be fast (ideally O(1)) algorithms for determining if data is of the specified format. These are algorithms that have a single input named `"input"` and a single output named `"output"`. The input has the type and format to be

checked. The output must have type and format `"boolean"`. The script performs the validation and sets the output variable to either true or false.

Any `*_to_*.json` files are imported as converters. A converter is simply an analysis with one input named `"input"` and one output named `"output"`. The input and output should have matching type but should be of different formats.

Parameters **search_paths** (*str or list of str*) – A list of search paths relative to the current working directory. Passing a single path as a string also works.

`girder_worker.plugins.types.format.`**import_default_converters**()
Import converters from the default search paths. This is called when the `girder_worker.format` module is first loaded.

`girder_worker.plugins.types.format.`**print_conversion_graph**()
Print a graph of supported conversion paths in DOT format to standard output.

`girder_worker.plugins.types.format.`**print_conversion_table**()
Print a table of supported conversion paths in CSV format with `'from'` and `'to'` columns to standard output.

Pythonic task API

class `girder_worker.core.specs.`**Spec**(**args*, ***kw*)
Defines core utility methods that all spec objects have in common.

Supports dict-like initialization.

```
>>> a = Spec({'a': 1, 'b': {'c': [1, 2, None]}})
>>> b = Spec(a=1, b={'c': [1, 2, None]})
>>> a == b
True
```

Also supports initialization from json.

```
>>> c = Spec('{"a": 1, "b": {"c": [1, 2, null]}}')
>>> a == c
True
```

Multiple initialization method can be used together, which will be inserted in order.

```
>>> Spec('{"a": 0}', {'a': 1}, a=2)
{"a": 2}
```

Updating merging specs is always done recursively.

```
>>> Spec('{"a": {"b": 0}}', a={'c': 1})
{"a": {"b": 0, "c": 1}}
```

Conflicts are resolved by taking the value with highest priority (i.e. the once provided next in the constructor.)

```
>>> Spec('{"a": {"b": 0}}', {"a": []})
{"a": []}
>>> Spec('{"a": {"b": 0}}', {"a": []}, a={"c": 1})
{"a": {"c": 1}}
>>> Spec('{"a": []}', {"a": {"b": 0}}, a={"c": 1})
{"a": {"b": 0, "c": 1}}
```

Serialization is performed as json

```
>>> str(a)
'{"a": 1, "b": {"c": [1, 2, null]}}'
```

Strings are assumed to be utf-8 encoded.

```
>>> str(Spec({u"for\u00eat": u"\ud83c\udf33 \ud83c\udf32 \ud83c\udf34"}))
'{"for\\u00eat": "\\ud83c\\udf33 \\ud83c\\udf32 \\ud83c\\udf34"}'
```

Methods that mutate the state of the Spec will test if the new state is valid, restoring the original state before raising an exception.

```
>>> s = Spec({'a': 0})
>>> try:
...     s['a'] = object
... except Exception:
...     pass
... else:
...     assert False
>>> s
{"a": 0}
```

Spec constructors are idempotent

```
>>> Spec(a='a') == Spec(Spec(a='a'))
True
```

class girder_worker.core.specs.**Port**(**arg*, ***kw*)

A port defines a communication channel between tasks.

Ports enable bidirectional communication between tasks and are responsible for ensuring that the connections are compatible. The primary purpose of ports is to specify what types of data tasks can read and write. This information is used by tasks to determine if they can be connected. Ports also provide documentation for the task by describing its inputs and outputs. Ports also handle fetching data from and pushing data to remote data stores.

```
>>> spec = {'name': 'a', 'type': 'number', 'format': 'number'}
>>> port = Port(spec)
```

The port object is serialized as a json object

```
>>> import json
>>> json.loads(str(port)) == spec
True
```

It has several properties derived from the spec

```
>>> port.name == spec['name']
True
>>> port.type == spec['type']
True
>>> port.format == spec['format']
True
```

It also supports auto converting formats and validation by default

```
>>> port.auto_convert
True
```

```
>>> port.auto_validate
True
```

Spec properties are automatically validated when setting them

```
>>> port = Port()
Traceback (most recent call last):
    ...
ValueError: Port specs require a valid name.
```

```
>>> port = Port(name="my port", type="python", format="object")
>>> port.format = 'invalid'
Traceback (most recent call last):
    ...
ValueError: Unknown format "python.invalid"
```

Checking the `type` is deferred to allow incremental updating

```
>>> port['type'] = 'image'
>>> port.json()
Traceback (most recent call last):
    ...
ValueError: Unknown format "image.object"
```

```
>>> port.format = 'png'
>>> port.json()
'{"type": "image", "name": "my port", "format": "png"}'
>>> port == Port(port)
True
```

auto_convert
If the data format is automatically

auto_validate
If the data is validated by default

convert(*data_spec*, *format*)
Convert to a compatible data format.

Parameters

- **data_spec** (*dict*) – Data specification
- **format** (*str*) – The target data format

Returns dict

```
>>> spec = {'name': 'a', 'type': 'number', 'format': 'number'}
>>> port = Port(spec)
```

```
>>> new_spec = port.convert({'format': 'number', 'data': 1}, 'json')
>>> new_spec['format']
'json'
>>> port.fetch(new_spec)
1
```

fetch(*data_spec*)
Return the data described by the given specification.

Parameters **data_spec** (*dict*) – A data specification object

Returns data

Raises ***ValidationError*** – when the validation check fails

```
>>> port = Port({'name': 'a', 'type': 'number', 'format': 'number'})
>>> port.fetch({'format': 'number', 'data': -1})
-1
```

format
The data format of the port

name
The name of the port

push(*data_spec*)
Write data a to remote destination according the to specification.

Parameters **data_spec** (*dict*) – A data specification object

Returns dict

```
>>> port = Port({'name': 'a', 'type': 'number', 'format': 'number'})
>>> port.push({'format': 'json', 'mode': 'inline', 'data': '2'})['data']
2
```

```
>>> port.push({'format': 'number', 'mode': 'inline', 'data': 3})['data']
3
```

type
The data type of the port

validate(*data_spec*)
Ensure the given data spec is compatible with this port.

Parameters **data_spec** (*dict*) – Data specification

Returns bool

```
>>> spec = {'name': 'a', 'type': 'number', 'format': 'number'}
>>> port = Port(spec)
```

```
>>> port.validate({'format': 'number', 'data': 1.5})
True
>>> port.validate({'format': 'json', 'data': '1.5'})
True
>>> port.validate({'format': 'number', 'data': '1.5'})
False
>>> port.validate({'format': 'unknown format', 'data': '...'})
False
```

class girder_worker.core.specs.**PortList**(*value=None*)
A list that only accepts port specs.

This class is extended to behave like a read-only dictionary, where the keys are the port names.

```
>>> l = PortList()
>>> l.json()
'[]'
```

Ports can be added as instances or dictionaries

```
>>> l.append(Port(name='z'))
>>> l.append({'name': 'a', 'type': 'image', 'format': 'png'})
```

Normal list methods are supported

```
>>> l[1] = '{"name": "b"}'
>>> l.insert(1, {"name": "c"})
>>> del l[1]
>>> str(l)
'[{"name": "z"}, {"name": "b"}]'
```

Port lists have keys and values methods like dicts

```
>>> l.keys()[0]
'z'
```

Ports can be referenced by either their index in the list or by name

```
>>> l[0] is l['z']
True
>>> 'z' in l
True
```

Ports can be modified after they are added

```
>>> l[0].name = 'y'
>>> l.json()
'[{"name": "y"}, {"name": "b"}]'
```

Several validation checks are performed

```
>>> l[0].name = 'b'
Traceback (most recent call last):
    ...
ValueError: Duplicate keys detected
>>> l[0].name = 'a'
>>> l[0].format = 'png'
Traceback (most recent call last):
    ...
ValueError: Unknown format "python.png"
>>> str(l)
'[{"name": "a"}, {"name": "b"}]'
```

append(*value*)
Append an item if possible.

check()
Check that the port list is valid.

insert(*index*, *value*)
Add an item before the given index if possible.

keys()
Return a list of port names.

values()
Return a list of ports.

exception girder_worker.core.specs.**ValidationError**(*port*, *data_spec*)
An exception type raised when encountering invalid data types.

class girder_worker.core.specs.**TaskSpec**(**args*, ***kw*)
Defines a pipeline element.

A task is an element of the pipeline responsible for completing an atomic task given one or more inputs and pushing the result to the next task or tasks.

inputs
A list of inputs accepted by the task

mode
The execution mode of the task

outputs
A list of outputs returned by the task

script
A script or function to execute

update(*other*, ***kw*)
Extend update to call PortList for input/output properties.

exception girder_worker.core.specs.**ReadOnlyAttributeException**
Exception thrown when attempting to set a read only attribute

exception girder_worker.core.specs.**WorkflowException**
Exception thrown for issues with Workflows

exception girder_worker.core.specs.**DuplicateTaskException**
Exception thrown when adding a duplicate task

Examples

Before getting started, make sure you've followed the necessary steps when it comes to *Installation* of Girder Worker.

Image processing

This example will introduce how to use the Girder Worker to build some simple image processing tasks using Pillow. We will learn how to chain several tasks together in a workflow and finally how to run these workflows both locally and through a remote worker.

Download and view an image

For our first task, we will download a png image from a public website and display it on the screen. We begin by defining a new task that will take a single image object and call its `show` method.

A task is a special kind of dictionary with keys `inputs` and `outputs` as well as other metadata describing how these objects will be used. In the case of simple Python scripts, they can be provided inline as we have done in this example. Each input and output spec in a task is a dict with the following keys:

name The name designated to the datum. This is used both for connecting tasks together in a workflow and, in the case of Python tasks, the name of the variable injected into/extracted from the tasks scope.

type The general data type expected by the task. See *Types and formats* for a list of types provided by the worker's core library as well as *Application Plugins* for additional data types provided by optional plugins.

format The specific representation or encoding of the data type. The worker will automatically convert between different data formats provided that they are of the same base type.

```
show_image = {
    'inputs': [{'name': 'the_image', 'type': 'image', 'format': 'pil'}],
    'outputs': [],
    'script': 'the_image.show()'
}
```

In order to run the task, we will need to provide an *input binding* that tells the worker where it can get the data to be injected into the port. Several I/O modes are supported; in this case, we provide a public URL to an image that the worker will download and open using Pillow. Notice that the worker downloads and reads the file as part of the automatic data format conversion.

```
lenna = {
    'type': 'image',
    'format': 'png',
    'url': 'https://upload.wikimedia.org/wikipedia/en/2/24/Lenna.png'
}
```

Finally to run this task, we only need to provide the task object and the input binding to `girder_worker.tasks.run()`. The object returned by this function contains data extracted and converted through the task's output ports.

```
output = girder_worker.tasks.run(show_image, {'the_image': lenna})
```

Perform an image blur inside a workflow

Now that we know how to generate a simple task using the worker, we want to learn how to connect multiple tasks together in a workflow. The worker's pythonic API allows us to do this easily. Let's create a new task that performs a blur operation on an image. This might look like the following:

```
blur_image = {
   'inputs': [
      {'name': 'blur_input', 'type': 'image', 'format': 'pil'},
      {'name': 'blur_radius', 'type': 'number', 'format': 'number'}
   ],
   'outputs': [{'name': 'blur_output', 'type': 'image', 'format': 'pil'}],
   'script': """
from PIL import ImageFilter
blur_output = blur_input.filter(ImageFilter.GaussianBlur(blur_radius))
"""
}
```

Notice that this task takes an additional numeric input that acts as a parameter for the blurring filter. Connecting our `show_image` task, we can view the result of our image filter. First, we create a new workflow object from the *`girder_worker.core.specs`* module.

```
from girder_worker.core.specs import Workflow
wf = Workflow()
```

Next, we add all the tasks to the workflow. The order in which the tasks are added is insignificant because the worker will automatically sort them according to their position in the workflow.

```
wf.add_task(blur_image, 'blur')
wf.add_task(show_image, 'show')
```

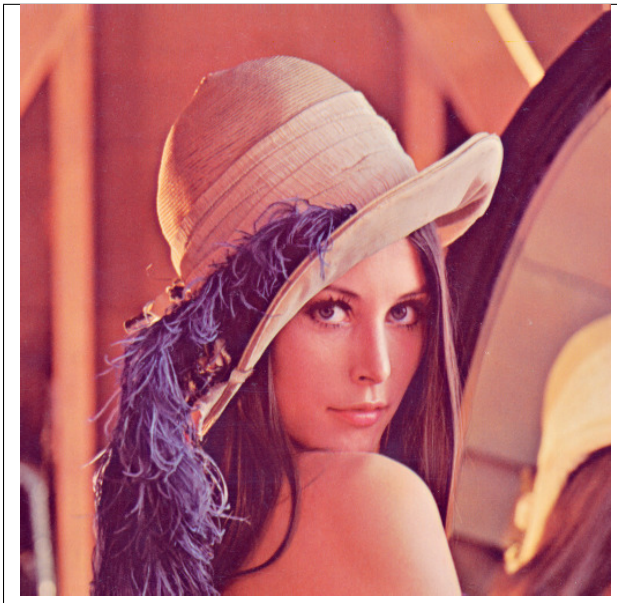
Finally, we connect the two tasks together.

```
wf.connect_tasks('blur', 'show', {'blur_output': 'the_image'})
```

Running a workflow has the same syntax as running a single task.

```
output = girder_worker.tasks.run(
    wf,
    inputs={
        'blur_input': lenna,
        'blur_radius': {'format': 'number', 'data': 5}
    }
)
```

Table 1.1: Blur image workflow



Using a workflow to compute image metrics

Finally, we will create a few more tasks to generate a more complicated workflow that returns some number of interest about an image. First, let's create a task to subtract two images from each other.

```
subtract_image = {
    'inputs': [
        {'name': 'sub_input1', 'type': 'image', 'format': 'pil'},
        {'name': 'sub_input2', 'type': 'image', 'format': 'pil'}
    ],
    'outputs': [
        {'name': 'diff', 'type': 'image', 'format': 'pil'},
    ],
    'script': """
from PIL import ImageMath
diff = ImageMath.eval('abs(int(a) - int(b))', a=sub_input1, b=sub_input2)
"""
}
```

Now another task will compute the average pixel value of the input image.

```
mean_image = {
    'inputs': [
        {'name': 'mean_input', 'type': 'image', 'format': 'pil'},
    ],
    'outputs': [
        {'name': 'mean_value', 'type': 'number', 'format': 'number'},
    ],
    'script': """
from PIL import ImageStat
mean_value = ImageStat.Stat(mean_input).mean[0]
"""
}
```

Finally, let's add all of the tasks to a new workflow and make the appropriate connections.

```
wf = Workflow()
wf.add_task(blur_image, 'blur1')
wf.add_task(blur_image, 'blur2')
wf.add_task(subtract_image, 'subtract')
wf.add_task(mean_image, 'mean')

wf.connect_tasks('blur1', 'subtract', {'blur_output': 'sub_input1'})
wf.connect_tasks('blur2', 'subtract', {'blur_output': 'sub_input2'})
wf.connect_tasks('subtract', 'mean', {'diff': 'mean_input'})
```

This workflow performs blurring operations on a pair of input images, computes the difference between them, and returns the average value of the difference. Let's see how this works with our sample image. Notice that in this case, there is a conflict between the input port names of the two `blur` tasks. We must specify which port we are referring to by prefixing the port name with the task name.

```
output = girder_worker.tasks.run(
   wf,
   inputs={
      'blur1.blur_input': lenna,
      'blur1.blur_radius': {'format': 'number', 'data': 1},
      'blur2.blur_input': lenna,
      'blur2.blur_radius': {'format': 'number', 'data': 8},
   }
)
print output['mean_value']['data']
```

Facebook network analysis

This example demonstrates how to use the worker as a workflow system to load graph data, perform analyses and transformations of the data using NetworkX, and then visualize the result using d3.js.

In this example we will:

1. Obtain a set of Facebook data
2. Find the most "popular" person in our data
3. Find the subgraph of the most popular person's neighborhood
4. Visualize this neighborhood using d3

Obtain the dataset

The dataset is a small sample of Facebook links representing friendships, which can be obtained here[1].

The data we'll be using is in a format commonly used when dealing with graphs, referred to as an adjacency list. The worker supports using adjacency lists with graphs out of the box.

Note: A full list of the supported types and formats is documented in *Types and formats*.

Here is a sample of what the data looks like:

```
86      127
303     325
356     367
373     404
475     484
```

Each integer represents an anonymized Facebook user. Users belonging to the same line in the adjacency list indicates a symmetric relationship in our undirected graph.

Build a workflow

Create a file named `workflow.py`, this is the file we'll be using to create our workflow.

Find the most popular person

One way of measuring who the most "popular" person in our graph is, is by taking the node with the largest degree.

The script below finds the most popular person in the graph.

Note: This script assumes a variable `G` exists, that's because we define it as an input in the `Task` we define in the next step.

```
from networkx import degree

degrees = degree(G)
most_popular_person = max(degrees, key=degrees.get)
```

Defining our task, we can embed this script:

```
most_popular_task = {
    'inputs': [
        {'name': 'G',
         'type': 'graph',
         'format': 'networkx'}
    ],
    'outputs': [
        {'name': 'most_popular_person',
         'type': 'string',
         'format': 'text'},
        {'name': 'G',
```

[1] For attribution refer here.

```
            'type': 'graph',
            'format': 'networkx'}
    ],
    'script':
    """
from networkx import degree

degrees = degree(G)
most_popular_person = max(degrees, key=degrees.get)
    """
}
```

Note: As we saw with our last script assuming `G` would in be scope, this task explicitly states that both `most_popular_person` and `G` will be in scope (as its outputs) when it's done.

Find the neighborhood

Now that we have the most popular node in the graph, we can take the subgraph including only this person and all of their neighbors. These are sometimes referred to as Ego Networks.

```
from networkx import ego_graph

subgraph = ego_graph(G, most_popular_person)
```

Again, we can create a task using our new script, like so:

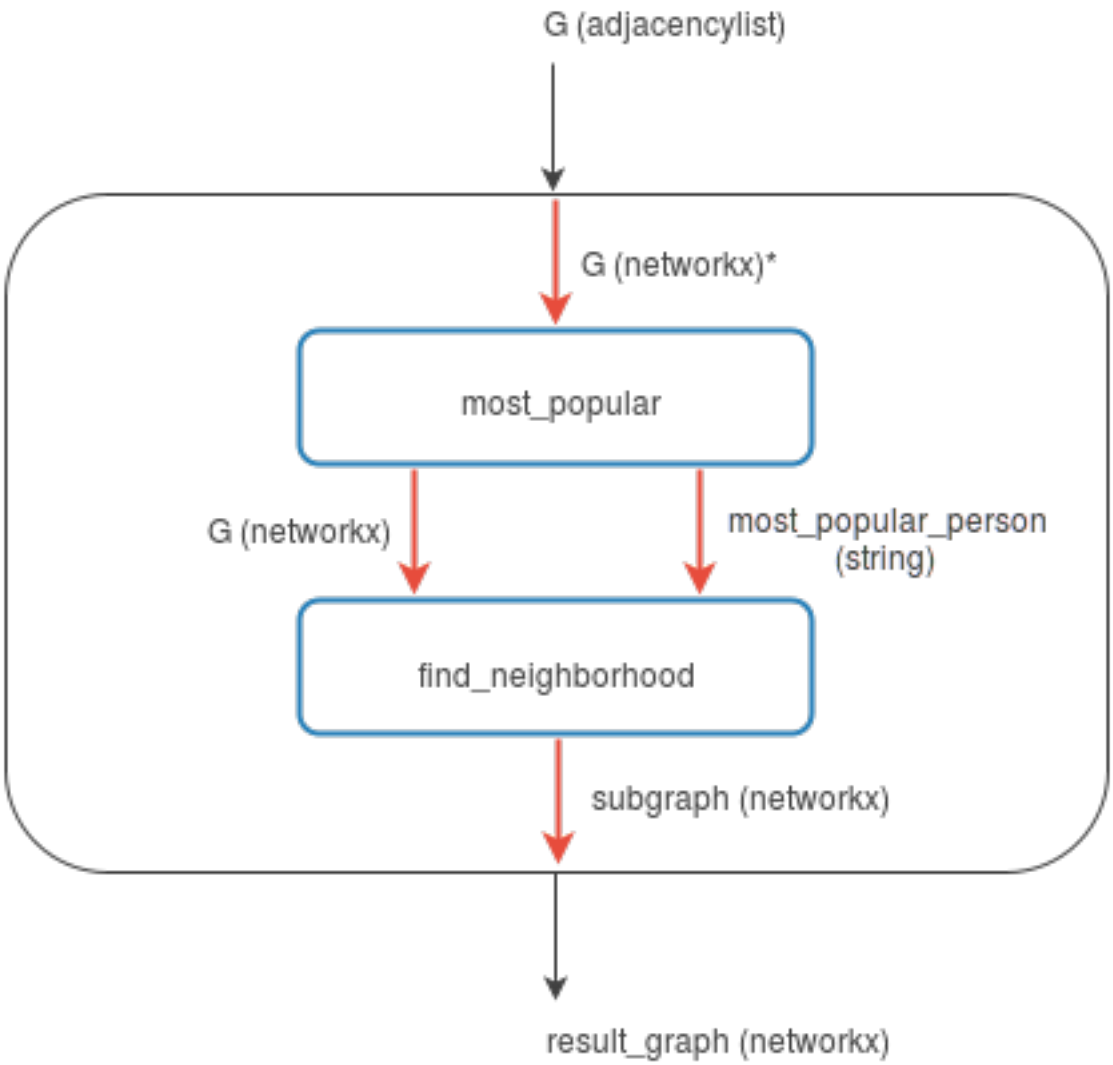
Note: Since these steps are going to be connected, our inputs are going to be the same as the last steps outputs.

```
find_neighborhood_task = {
    'inputs': [
        {'name': 'G',
         'type': 'graph',
         'format': 'networkx'},
        {'name': 'most_popular_person',
         'type': 'string',
         'format': 'text'}
    ],
    'outputs': [
        {'name': 'subgraph',
         'type': 'graph',
         'format': 'networkx'}
    ],
    'script':
    """
from networkx import ego_graph

subgraph = ego_graph(G, most_popular_person)
    """
 }
```

Put it together

Conceptually, this is what our workflow will look like:



* The format changes because of Girder Worker's auto-conversion functionality.

The entire rectangle is our workflow, and the blue rectangles are our tasks. Black arrows represent inputs and outputs and the red arrows represent connections which we'll see shortly.

To make this happen, since we've written the tasks already, we just need to format this in a way the worker understands.

To start, let's create our workflow from a high level, starting with just its inputs and outputs (the black arrows):

```
workflow = {
    'mode': 'workflow',
    'inputs': [
        {'name': 'G',
         'type': 'graph',
         'format': 'adjacencylist'}
    ],
    'outputs': [
```

```
        {'name': 'result_graph',
         'type': 'graph',
         'format': 'networkx'}
    ]
}
```

Now we need to add our tasks to the workflow, which is pretty straightforward since we've defined them in the previous steps.

```
workflow['steps'] = [{'name': 'most_popular',
                      'task': most_popular_task},
                     {'name': 'find_neighborhood',
                      'task': find_neighborhood_task}]
```

Finally, we need to add the red arrows within the workflow, telling the worker how the inputs and outputs are going to flow from each task. These are called *connections* in Girder Worker parlance.

```
workflow['connections'] = [
    {'name': 'G',
     'input_step': 'most_popular',
     'input': 'G'},
    {'output_step': 'most_popular',
     'output': 'G',
     'input_step': 'find_neighborhood',
     'input': 'G'},
    {'output_step': 'most_popular',
     'output': 'most_popular_person',
     'input_step': 'find_neighborhood',
     'input': 'most_popular_person'},
    {'name': 'result_graph',
     'output': 'subgraph',
     'output_step': 'find_neighborhood'}
]
```

We now have a complete workflow! Let's run this, and write the final data to a file.

```
with open('docs/static/facebook-sample-data.txt') as infile:
    output = girder_worker.tasks.run(workflow,
                                     inputs={'G': {'format': 'adjacencylist',
                                                   'data': infile.read()}},
                                     outputs={'result_graph': {'format': 'networkx.json'}})

with open('data.json', 'wb') as outfile:
    outfile.write(output['result_graph']['data'])
```

Running `workflow.py` will produce the JSON in a file called `data.json`, which we'll pass to d3.js in the next step.

For completeness, here is the complete workflow specification as pure JSON:

```
{
  "mode": "workflow",
  "inputs": [
    {
      "type": "graph",
      "name": "G",
      "format": "adjacencylist"
    }
```

```
  ],
  "outputs": [
    {
      "type": "graph",
      "name": "result_graph",
      "format": "networkx"
    }
  ],
  "connections": [
    {
      "input": "G",
      "input_step": "most_popular",
      "name": "G"
    },
    {
      "output": "G",
      "input_step": "find_neighborhood",
      "input": "G",
      "output_step": "most_popular"
    },
    {
      "output": "most_popular_person",
      "input_step": "find_neighborhood",
      "input": "most_popular_person",
      "output_step": "most_popular"
    },
    {
      "output": "subgraph",
      "name": "result_graph",
      "output_step": "find_neighborhood"
    }
  ],
  "steps": [
    {
      "name": "most_popular",
      "task": {
        "inputs": [
          {
            "type": "graph",
            "name": "G",
            "format": "networkx"
          }
        ],
        "script": "\nfrom networkx import degree\n\ndegrees = degree(G)\nmost_popular_
↪person = max(degrees, key=degrees.get)\n",
        "outputs": [
          {
            "type": "string",
            "name": "most_popular_person",
            "format": "text"
          },
          {
            "type": "graph",
            "name": "G",
            "format": "networkx"
          }
        ]
      }
```

```
    },
    {
      "name": "find_neighborhood",
      "task": {
        "inputs": [
          {
            "type": "graph",
            "name": "G",
            "format": "networkx"
          },
          {
            "type": "string",
            "name": "most_popular_person",
            "format": "text"
          }
        ],
        "script": "\nfrom networkx import ego_graph\n\nsubgraph = ego_graph(G, most_
↪popular_person)\n",
        "outputs": [
          {
            "type": "graph",
            "name": "subgraph",
            "format": "networkx"
          }
        ]
      }
    }
  ]
}
```

This file can be loaded with Python's `json` package and directly sent to `girder_worker.tasks.run()`:

```
import json

with open('docs/static/facebook-example-spec.json') as spec:
    workflow = json.load(spec)

with open('docs/static/facebook-sample-data.txt') as infile:
    output = girder_worker.tasks.run(workflow,
                                     inputs={'G': {'format': 'adjacencylist',
                                                   'data': infile.read()}},
                                     outputs={'result_graph': {'format': 'networkx.json'}})

with open('data.json', 'wb') as outfile:
    outfile.write(output['result_graph']['data'])
```

Note: More information on Girder Worker tasks and workflows can be found in *API documentation*.

Visualize the results

Using JavaScript similar to this d3.js example we're going to add the following to our `index.html` file:

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>
```

```
 .node {
     stroke: #fff;
     stroke-width: 1.5px;
 }

 .link {
     stroke: #999;
     stroke-opacity: .6;
 }
</style>
<body>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/d3/3.5.5/d3.min.js"></script>
    <script>
    var width = 700,
    height = 400;

    var force = d3.layout.force()
            .charge(-120)
            .linkDistance(30)
            .size([width, height]);

    var svg = d3.select("#popularity-graph").append("svg")
            .attr("width", width)
            .attr("height", height);

    d3.json("/data.json", function(error, graph) {
        if (error) throw error;

        force
            .nodes(graph.nodes)
            .links(graph.links)
            .start();

        var link = svg.selectAll(".link")
                .data(graph.links)
                .enter().append("line")
                .attr("class", "link")
                .style("stroke-width", function(d) { return 1; });

        var node = svg.selectAll(".node")
                .data(graph.nodes)
                .enter().append("circle")
                .attr("class", "node")
                .attr("r", 5)
                .style("fill", "#1f77b4")
                .call(force.drag);

        node.append("title")
            .text(function(d) { return d.id; });

        force.on("tick", function() {
            link.attr("x1", function(d) { return d.source.x; })
                .attr("y1", function(d) { return d.source.y; })
                .attr("x2", function(d) { return d.target.x; })
                .attr("y2", function(d) { return d.target.y; });

            node.attr("cx", function(d) { return d.x; })
                .attr("cy", function(d) { return d.y; });
```

```
        });
    });
```

Which should leave us with a visualization similar to the following:

This is of course a more verbose than necessary workflow for the purposes of demonstration. This could have easily been done with one task, however by following this you should have learned how to do the following with the Girder Worker:

- Create tasks which consume and produce multiple inputs and outputs
- Run tasks as part of a multi-step workflow
- Use the worker's converter system to serialize it in a format JavaScript can read
- Visualize the data using d3.js

Developer documentation

This section of the documentation is meant for those who wish to contribute to the Girder Worker platform.

Installing from source

Clone from git:

```
git clone https://github.com/girder/girder_worker.git
cd girder_worker
```

Install requirements:

```
pip install -r requirements.txt

# If you plan on developing the worker, you will also want to install system packages␣
↪and Python requirements:

# Command for Ubuntu
sudo apt-get install libxml2-dev libxslt1-dev

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

If you want to run the girder_worker as a remote worker from the source install directory, you'll need to install it with pip in editable mode. If you don't want to include any girder worker plugins:

```
pip install -U -e .
```

Test it:

To test whether the setup without complex dependencies is working

    python -m unittest tests.format_test

To test the setup is working with complex dependencies including R and Mongo

    python -m unittest -v tests.table_test python -m unittest -v tests.tree_test

Some things not working? You can install a few things so they do. For example, install MongoDB and R, in addition to their Python bindings:

```
pip install pymongo rpy2  # may need sudo
```

You'll need to get a MongoDB server listening on localhost by running `mongod`.

In R, you'll need to install some stuff too, currently just the `ape` package:

```
install.packages("ape")
```

Installing worker plugins from pip

An individual plugin can be installed through pip, in editable mode, like so:

```
pip install -U -e .[r]
```

You can run this command at any time to install dependencies of other plugins, even if the worker is already installed.

See also:

For more information on the worker plugin system, see *Application Plugins*.

Creating a new release

Girder Worker releases are uploaded to PyPI for easy installation via `pip`. The recommended process for generating a new release is described here.

1. From the target commit, set the desired version number in the top level `__init__.py`. Create a new commit and note the SHA; this will become the release tag.
2. Ensure that all tests pass.
3. Clone the repository in a new directory and checkout the release SHA. (Packaging in an old directory could cause extraneous files to be mistakenly included in the source distribution.)
4. Run `python setup.py sdist --dist-dir .` to generate the distribution tarball in the project directory, which looks like `girder-worker-x.y.z.tar.gz`.
5. Create a new virtual environment and install the Python package into it. This should not be done in the repository directory because the wrong package will be imported.

   ```
   mkdir test && cd test
   virtualenv release
   source release/bin/activate
   pip install ../girder-worker-<version>.tar.gz
   ```

6. Once that finishes, you should be able to start the worker by simply running `girder-worker`.
7. When you are confident everything is working correctly, generate a new release on GitHub. You must be sure to use a tag version of `v<version>`, where `<version>` is the version number as it exists in `setup.py`. For example, `v0.2.4`. Attach the tarball you generated to the release.
8. Add the tagged version to readthedocs and make sure it builds correctly.
9. Finally, upload the release to PyPI with the following command:

   ```
   python setup.py sdist upload
   ```

Note: The first time you create a release, you will need to register to PyPI before you can run the upload step. To do so, simply run `python setup.py sdist register`.

Task Plugins

This is an example plugin that demonstrates how to extend girder_worker by allowing it to run additional tasks. Plugin's 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.

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 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)
```

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']
```

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`.

Final notes

With these three components (Tasks, Plugin Class, Entry Point) you should be able to add arbitrary tasks to the girder_worker client. By default, jobs created in girder using the 'worker' plugin run the `girder_worker.run` task. This can be overridden to call custom plugin tasks by generating jobs with a `celeryTaskName` defined in the job's `otherFields` key word argument. E.g.:

```
# Create a job to be handled by the worker plugin
job = jobModel.createJob(
    title='Some Job', type='some_type', handler='worker_handler',
    user=self.admin, public=False, args=(25), kwargs={},
    otherFields={
        'celeryTaskName': 'gwexample.analyses.tasks.fibonacci'
    })

jobModel.scheduleJob(job)
```

This will schedule a job that runs `gwexample.analyses.tasks.fibonacci(25)` on the girder worker.

Finally, by default the core girder_worker tasks (`run`, `convert`, `validate`) are included along with their plugins etc. If you wish to prevent these tasks from being loaded inside the celery instance, you can configure `core_tasks=false` in `worker.local.cfg` under the `[girder_worker]` section of the configuration.

Application Plugins

The Girder Worker application plugin system is used to extend the core functionality of Girder Worker in a number of ways. Application plugins can execute any Python code when they are loaded at runtime, but the most common augmentations they perform are:

- **Adding new execution modes.** Without any application plugins enabled, the core Girder Worker application can only perform two types of tasks: `python` and `workflow` modes. It's common for application plugins to implement other task execution modes.
- **Adding new data types or formats.** Application plugins can make Girder Worker aware of new data types and formats, and provide implementations for how to validate and convert to and from those formats.
- **Adding new IO modes.** One of the primary functions of Girder Worker is to fetch input data from heterogenous sources and expose it to tasks in a uniform way. Application plugins can implement novel modes of fetching and pushing input and output data for a task.

Below is a list of the application plugins that are shipped with the girder_worker package. They can be enabled via the configuration file (see *Configuration*).

Docker

- **Plugin ID:** `docker`
- **Description:** This plugin exposes a new task execution mode, `docker`. These tasks pull a Docker image and run a container using that image, with optional command line arguments. Docker tasks look like:

```
<DOCKER_TASK> ::= {
    "mode": "docker",
    "docker_image": <Docker image name to run>
    (, "pull_image": <true (the default) or false>)
    (, "container_args": [<container arguments>])
    (, "docker_run_args": [<additional arguments to `docker run`>])
    (, "entrypoint": <custom override for container entry point>)
    (, "inputs": [<TASK_INPUT> (, <TASK_INPUT>, ...)])
    (, "outputs": [<TASK_OUTPUT> (, <TASK_OUTPUT>, ...)])
}
```

The optional `container_args` parameter is a list of arguments to pass to the container. If an `entrypoint` argument is passed, it will override the built-in `ENTRYPOINT` directive of the image. Since it's often the case that task inputs will need to passed to the container as arguments, a special syntax can be used to declare that a command line argument should be expanded at runtime to the value of an input:

```
"container_args": ["$input{my_input_id}"]
```

It is not necessary for the entire argument to be a variable expansion; any part of an argument can also be expanded, e.g.:

```
"container_args": ["--some-parameter=$input{some_parameter_value}"]
```

Some command line arguments represent boolean flag values, and they should either be present or absent depending on a boolean input value. For example, perhaps your container accepts a command line argument `--verbose` to switch to verbose output. To support this as an input, you could use the following task input:

```
{
"id": "verbose",
"name": "Verbose output",
"description": "Prints more information during processing.",
"type": "boolean",
"format": "boolean",
"arg": "--verbose"
}
```

Then, in your `container_args` list, you can use a special `$flag{id}` token to control whether this argument (specified via the `arg` parameter) is included or omitted:

```
"container_args": [..., "$flag{verbose}", ...]
```

The temporary directory for the Girder Worker task is mapped into the running container under the directory `/mnt/girder_worker/data`, so any files that were fetched into that temp directory will be available inside the running container at that path.

By default, the image you specify will be pulled using the `docker pull` command. In some cases, you may not want to perform a pull, and instead want to rely on the image already being present on the worker system. If so, set `pull_image` to false.

To ensure the execution context is the expected one, it is recommended to specify the `docker_image` using the `Image[@digest]` format (e.g. `debian@sha256:cbbf2f9a99b47fc460d422812b6a5adff7dfee951d8fa2e4a98caa0382cfbdbf`). This will prevent `docker pull` from systematically downloading the latest available image. In that case, setting `pull_image` to false is less relevant since the image will be pulled only if it is not already available.

If you want to pass additional command line options to `docker run` that should come before the container name, pass them as a list via the `"docker_run_args"` key.

Note: The Docker plugin currently does not support running `dockerd` with the option `--selinux-enabled`. Running with this option may result in an error like:

```
Exception: Docker tempdir chmod returned code 1.
```

Outputs from Docker tasks

Docker tasks can have two types of outputs: streams (i.e. standard output and standard error) and files written by the container. If you want the contents of standard output or standard error to become a task output, use the special output IDs `_stdout` or `_stderr`, as in the following example:

```
"task": {
    "mode": "docker",
    "outputs": [{
        "id": "_stdout",
        "type": "string",
        "format": "text"
    }],
    ...
```

If you want to have your container write files that will be treated as outputs, write them into the `/mnt/girder_worker/data` directory inside the container, then declare them in the task output specification with `"target": "filepath"`. The following example shows how to specify a file written to `/mnt/girder_worker/data/my_image.png` as a task output:

```
"task": {
    "mode": "docker",
    "outputs": [{
        "id": "my_image.png",
        "target": "filepath",
        "type": "string",
        "format": "text"
    }],
    ...
```

You don't have to use the output ID to specify the path; you can instead pass a `path` field in the output spec:

```
"task": {
    "mode": "docker",
    "outputs": [{
        "id": "some_output",
        "target": "filepath",
        "type": "string",
        "format": "text",
        "path": "/mnt/girder_worker/data/some_subdirectory/my_image.png"
    }],
    ...
```

Paths that are specified as relative paths are assumed to be relative to `/mnt/girder_worker/data`. If you specify an absolute path, it must start with `/mnt/girder_worker/data/`, otherwise an exception will be thrown before the task is run. These conventions apply whether the path is specified in the `id` or `path` field.

Progress reporting from docker tasks

Docker tasks have the option of reporting progress back to Girder via a special named pipe. If you want to do this, specify `"progress_pipe": true` in your docker task specification. This will create a special named pipe at `/mnt/girder_worker/data/.girder_progress`. In your container logic, you may write progress notification messages to this named pipe, one per line. Progress messages should be JSON objects with the following fields, all of which are optional:

- `message` (string): A human-readable message about the current task progress.
- `current` (number): A numeric value representing current progress, which should always be less than or equal to the `total` value.
- `total` (number): A numeric value representing the maximum progress value, i.e. the value of `current` when the task is complete. Only pass this field if the total is changing or being initially set.

An example progress notification string:

```
{"message": "Halfway there!", "total": 100, "current": 50}
```

Note: When writing to the named pipe, you should explicitly call `flush` on the file descriptor afterward, otherwise the messages may sit in a buffer and may not reach the Girder server as you write them.

Note: This feature may not work on Docker on non-Linux platforms, and the call to open the pipe for writing from within the container may block indefinitely.

Management of Docker Containers and Images

Docker images may be pulled when a task is run. By default, these images are never removed. Docker containers are automatically removed when the task is complete.

As an alternative, a 'garbage collection' process can be used instead. It can be enabled by modifying settings in the `[docker]` section of the config file, which can be done using the command:

```
girder-worker-config set docker gc True
```

When the `gc` config value is set to `True`, containers are not removed when the task ends. Instead, periodically, any images not associated with a container will be removed, and then any stopped containers will be removed. This will free disk space associated with the images, but may remove images that are not directly related to Girder Worker.

When garbage collection is turned on, images can be excluded from the process by setting `exclude_images` to a comma-separated list of image names. For instance:

```
girder-worker-config set docker exclude_images dsarchive/histomicstk,rabbitmq
```

Only containers that have been stopped longer than a certain time are removed. This time defaults to an hour, and can be specified as any number of seconds via the `cache_timeout` setting.

Girder IO

- **Plugin ID:** `girder_io`

- **Description:** This plugin adds new fetch and push modes called `girder`. The fetch mode for inputs supports downloading folders, items, or files from a Girder server. Inputs can be downloaded anonymously (if they are public) or using an authentication token. The downloaded data is either written to disk and passed as a file, or read into memory, depending on whether the corresponding task input's `target` field is set to `"filepath"` or `"memory"`. Likewise for uploads, the value of the output variable is interpreted as a path to a file to be uploaded if the task output `target` is set to `filepath`. If it's set to `memory`, the value of the output variable becomes the contents of the uploaded file. The URL to access the Girder API must be specified either as a full URL in the `api_url` field, or in parts via the `host`, `port`, `api_root`, and `scheme` fields.

```
<GIRDER_INPUT> ::= {
    "mode": "girder",
    "id": <the _id value of the resource to download>,
    "name": <the name of the resource to download>,
    "format": "text",
    "type": "string"
    (, "api_url": <full URL to the API, can be used instead of scheme/host/port/api_
↪root>)
    (, "host": <the hostname of the girder server. Required if no api_url is passed>)
    (, "port": <the port of the girder server, default is 80 for http: and 443 for␣
↪https:>)
    (, "api_root": <path to the girder REST API, default is "/api/v1")
    (, "scheme": <"http" or "https", default is "http">)
    (, "token": <girder token used for authentication>)
    (, "resource_type": <"file", "item", or "folder", default is "file">)
    (, "fetch_parent": <whether to download the whole parent resource as well,␣
↪default is false>)
}
```

Note: For historical reasons, task inputs that do not specify a `target` field and are bound to a Girder input will default to having the data downloaded to a file (i.e. `target="filepath"` behavior). This is different from the normal default behavior for other IO modes, which is to download the data to an object in memory. For this reason, it is suggested that if your task input is going to support Girder IO mode, that you specify the `target` field explicitly on it rather than using the default.

The output mode also assumes data of format `string/text` that is a path to a file in the filesystem. That file will then be uploaded under an existing folder (under a new item with the same name as the file), or into an existing item.

```
<GIRDER_OUTPUT> ::= {
    "mode": "girder",
    "token": <girder token used for authentication>,
    "parent_id": <the _id value of the folder or item to upload into>,
    "format": "text",
    "type": "string"
    (, "name": <optionally override name of the file to upload>)
    (, "api_url": <full URL to the API, can be used instead of scheme/host/port/api_
↪root>)
    (, "host": <the hostname of the girder server. Required if no api_url is passed>)
    (, "port": <the port of the girder server, default is 80 for http: and 443 for␣
↪https:>)
    (, "api_root": <path to the girder REST API, default is "/api/v1")
    (, "scheme": <"http" or "https", default is "http">)
    (, "parent_type": <"folder" or "item", default is "folder">)
    (, "reference": <arbitrary reference string to pass to the server>)
}
```

Metadata outputs

In addition to outputting blob data into Girder files, you may also output structured metadata that can be attached to an item. You can upload an output *as* metadata onto a new or existing item, or attach a pre-defined set of metadata to an output item that is uploaded as a file.

To push an output as the metadata on an *existing* item, use the `as_metadata` field with the `item_id` field set in your output binding:

```
{
    "mode": "girder",
    "as_metadata": true,
    "item_id": <ID of the target item>,
    "api_url": "...",
    "token": "..."
}
```

To push an output as the metadata on a *new* item, use `as_metadata` with the `parent_id` field set, and a `name` field. The `name` is not required if the corresponding task output has `"target": "filepath"`, in which case the filename will be used as the name for the new item.

```
{
    "mode": "girder",
    "as_metadata": true,
    "parent_id": <ID of the parent folder>,
    "name": "My new metadata item",
    "api_url": "...",
    "token": "..."
}
```

Note: The `as_metadata` behavior will only work if your output data is a JSON Object.

To add additional pre-defined metadata fields to a normal Girder IO output, you can use the `metadata` field on a normal Girder IO output. The `metadata` field must contain a JSON object, and its value will be set as the metadata on the output item.

```
{
    "mode": "girder",
    "metadata": {
        "some": "other",
        "metadata": "values"
    },
    "parent_id": <ID of the parent folder>,
    "parent_type": "folder",
    "name": "My output data",
    "token": "...",
    "api_url": "..."
}
```

Cache Configuration

The Girder Client (used by the Girder IO plugin) supports caching of files downloaded from Girder. These cache settings are exposed in the Girder Worker configuration. The following options are available:

- `diskcache_enabled` (default=0): enable or disable diskcache for files downloaded with the girder client

- `diskcache_directory` (default=girder_file_cache): directory to use for the diskcache
- `diskcache_eviction_policy` (default=least-recently-used): eviction policy used when diskcache size limit is reached
- `diskcache_size_limit` (default=1073741824): maximum size of the disk cache, 1GB default
- `diskcache_cull_limit` (default=10): maximum number of items to cull when evicting items
- `diskcache_large_value_threshold` (default=1024): cached values below this size are stored directly in the cache's sqlite db

R

- **Plugin ID:** `r`
- **Description:** The R plugin enables the execution of R scripts as tasks via the `r` execution mode. It also exposes a new data type, `r`, and several new data formats and converters for existing data types. Just like `python` mode, the R code to run is passed via the `script` field of the task specification. The `r` data type refers to objects compatible with the R runtime environment.
- **Converters added:**
  - `r/object` `r/serialized`
  - `table/csv` `table/r.dataframe`
  - `tree/newick` `tree/r.apetree`
  - `tree/nexus` `tree/r.apetree`
  - `tree/r.apetree` → `tree/treestore`
- **Validators added:**
  - `r/object`: An in-memory R object.
  - `r/serialized`: A serialized version of an R object created using R's `serialize` function.
  - `table/r.dataframe`: An R data frame. If the first column contains unique values, these are set as the row names of the data frame.
  - `tree/r.apetree`: A tree in the R package `ape` format.

Types

- **Plugin ID:** `types`
- **Description:** This plugin allows type and format annotations of inputs and outputs to be added to task specs and IO bindings. It can also perform automatic conversion between different formats for known types, as well as validating the correctness of the data formats. These behaviors are enabled with optional boolean arguments to the `run` task: `validate` and `auto_convert`. Other plugins such as VTK and R add additional types and formats to the typesystem supported by this plugin. The full documentation including a list of supported types and formats can be found in the *Types and formats* section.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

girder_worker, 10
girder_worker.core.specs, 12
girder_worker.plugins.types.format, 10

Index

A

append() (girder_worker.core.specs.PortList method), 16
auto_convert (girder_worker.core.specs.Port attribute), 14
auto_validate (girder_worker.core.specs.Port attribute), 14

C

check() (girder_worker.core.specs.PortList method), 16
convert() (girder_worker.core.specs.Port method), 14
converter_path() (in module girder_worker.plugins.types.format), 10

D

DuplicateTaskException, 17

F

fetch() (girder_worker.core.specs.Port method), 14
format (girder_worker.core.specs.Port attribute), 15
format (girder_worker.plugins.types.format.Validator attribute), 10

G

get_validator_analysis() (in module girder_worker.plugins.types.format), 11
girder_worker (module), 10
girder_worker.core.specs (module), 12
girder_worker.plugins.types.format (module), 10
GirderWorkerPluginABC (class in girder_worker), 10

H

has_converter() (in module girder_worker.plugins.types.format), 11

I

import_converters() (in module girder_worker.plugins.types.format), 11
import_default_converters() (in module girder_worker.plugins.types.format), 12
inputs (girder_worker.core.specs.TaskSpec attribute), 17
insert() (girder_worker.core.specs.PortList method), 16
is_valid() (girder_worker.plugins.types.format.Validator method), 10

K

keys() (girder_worker.core.specs.PortList method), 16

M

mode (girder_worker.core.specs.TaskSpec attribute), 17

N

name (girder_worker.core.specs.Port attribute), 15

O

outputs (girder_worker.core.specs.TaskSpec attribute), 17

P

Port (class in girder_worker.core.specs), 13
PortList (class in girder_worker.core.specs), 15
print_conversion_graph() (in module girder_worker.plugins.types.format), 12
print_conversion_table() (in module girder_worker.plugins.types.format), 12
push() (girder_worker.core.specs.Port method), 15

R

ReadOnlyAttributeException, 17

S

script (girder_worker.core.specs.TaskSpec attribute), 17
Spec (class in girder_worker.core.specs), 12

T

task_imports() (girder_worker.GirderWorkerPluginABC method), 10
TaskSpec (class in girder_worker.core.specs), 17
type (girder_worker.core.specs.Port attribute), 15
type (girder_worker.plugins.types.format.Validator attribute), 10

U

update() (girder_worker.core.specs.TaskSpec method), 17

V

validate() (girder_worker.core.specs.Port method), 15
ValidationError, 16
Validator (class in girder_worker.plugins.types.format), 10
values() (girder_worker.core.specs.PortList method), 16

W

WorkflowException, 17