Luigi Documentation

Release 3.5.1

The Luigi Authors

May 20, 2024

CONTENTS

Luigi is a Python (3.6, 3.7, 3.8, 3.9, 3.10, 3.11, 3.12 tested) package that helps you build complex pipelines of batch jobs. It handles dependency resolution, workflow management, visualization, handling failures, command line integration, and much more.

Run `pip install luigi` to install the latest stable version from PyPI. Documentation for the latest release is hosted on readthedocs.

Run `pip install luigi[toml]` to install Luigi with TOML-based configs support.

For the bleeding edge code, `pip install git+https://github.com/spotify/luigi.git`. Bleeding edge documentation is also available.

CHAPTER

ONE

BACKGROUND

The purpose of Luigi is to address all the plumbing typically associated with long-running batch processes. You want to chain many tasks, automate them, and failures *will* happen. These tasks can be anything, but are typically long running things like Hadoop jobs, dumping data to/from databases, running machine learning algorithms, or anything else.

There are other software packages that focus on lower level aspects of data processing, like Hive, Pig, or Cascading. Luigi is not a framework to replace these. Instead it helps you stitch many tasks together, where each task can be a Hive query, a Hadoop job in Java, a Spark job in Scala or Python, a Python snippet, dumping a table from a database, or anything else. It's easy to build up long-running pipelines that comprise thousands of tasks and take days or weeks to complete. Luigi takes care of a lot of the workflow management so that you can focus on the tasks themselves and their dependencies.

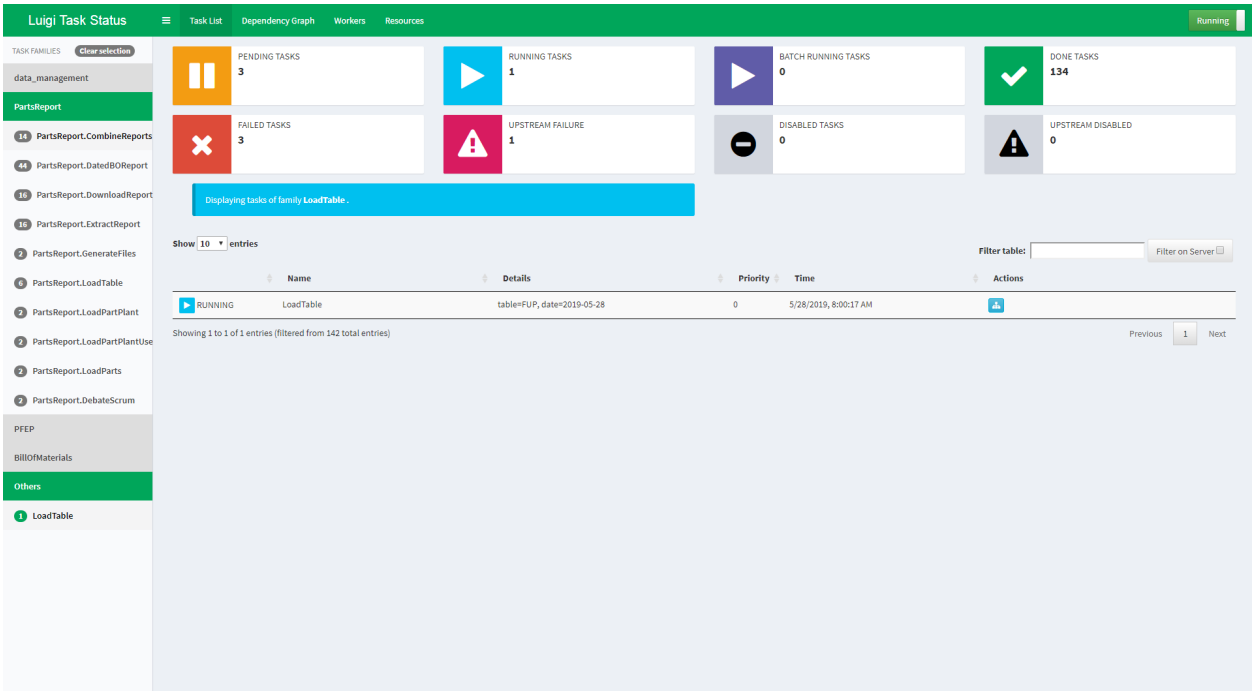
You can build pretty much any task you want, but Luigi also comes with a *toolbox* of several common task templates that you use. It includes support for running Python mapreduce jobs in Hadoop, as well as Hive, and Pig, jobs. It also comes with file system abstractions for HDFS, and local files that ensures all file system operations are atomic. This is important because it means your data pipeline will not crash in a state containing partial data.

CHAPTER

TWO

VISUALISER PAGE

The Luigi server comes with a web interface too, so you can search and filter among all your tasks.

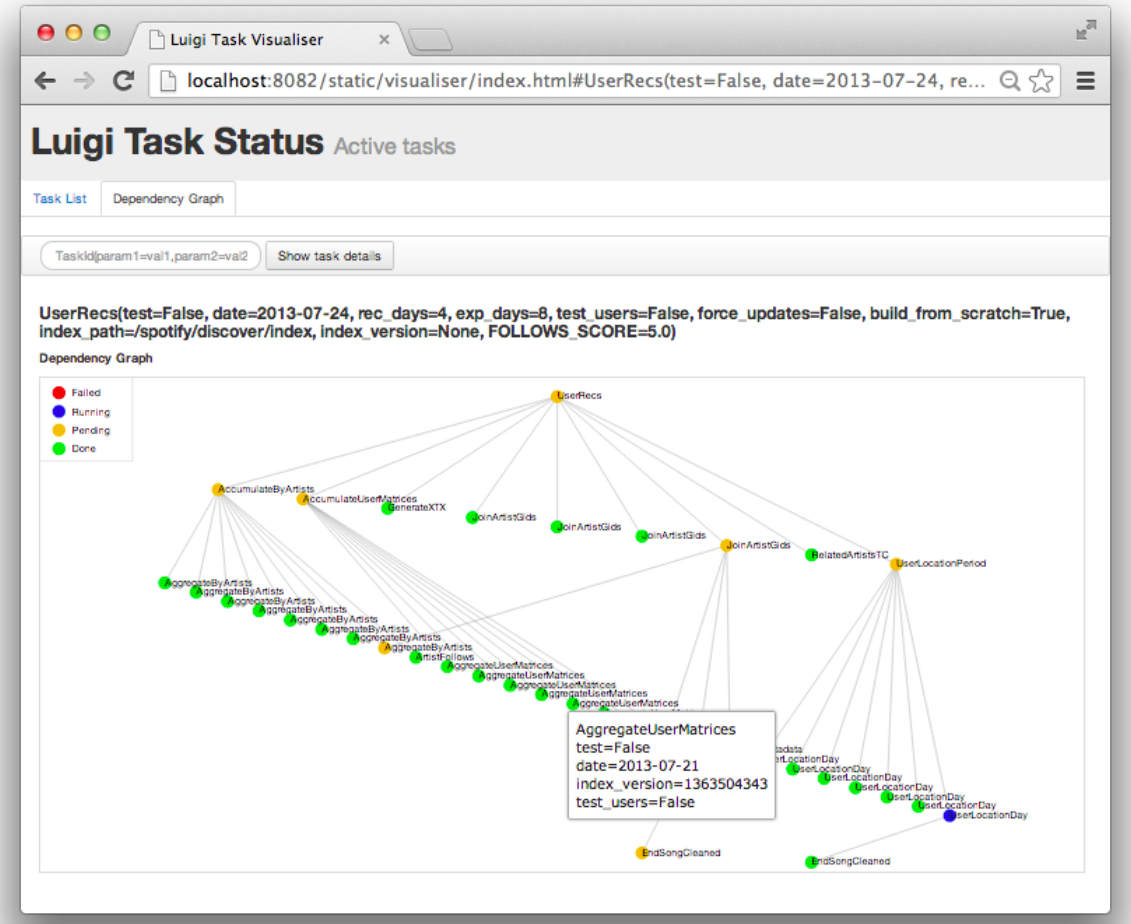


CHAPTER

THREE

DEPENDENCY GRAPH EXAMPLE

Just to give you an idea of what Luigi does, this is a screen shot from something we are running in production. Using Luigi's visualiser, we get a nice visual overview of the dependency graph of the workflow. Each node represents a task which has to be run. Green tasks are already completed whereas yellow tasks are yet to be run. Most of these tasks are Hadoop jobs, but there are also some things that run locally and build up data files.



CHAPTER

FOUR

PHILOSOPHY

Conceptually, Luigi is similar to GNU Make where you have certain tasks and these tasks in turn may have dependencies on other tasks. There are also some similarities to Oozie and Azkaban. One major difference is that Luigi is not just built specifically for Hadoop, and it's easy to extend it with other kinds of tasks.

Everything in Luigi is in Python. Instead of XML configuration or similar external data files, the dependency graph is specified *within Python*. This makes it easy to build up complex dependency graphs of tasks, where the dependencies can involve date algebra or recursive references to other versions of the same task. However, the workflow can trigger things not in Python, such as running Pig scripts or scp'ing files.

CHAPTER

FIVE

WHO USES LUIGI?

We use Luigi internally at Spotify to run thousands of tasks every day, organized in complex dependency graphs. Most of these tasks are Hadoop jobs. Luigi provides an infrastructure that powers all kinds of stuff including recommendations, toplists, A/B test analysis, external reports, internal dashboards, etc.

Since Luigi is open source and without any registration walls, the exact number of Luigi users is unknown. But based on the number of unique contributors, we expect hundreds of enterprises to use it. Some users have written blog posts or held presentations about Luigi:

- Spotify (presentation, 2014)
- Foursquare (presentation, 2013)
- Mortar Data (Datadog) (documentation / tutorial)
- Stripe (presentation, 2014)
- Buffer (blog, 2014)
- SeatGeek (blog, 2015)
- Treasure Data (blog, 2015)
- Growth Intelligence (presentation, 2015)
- AdRoll (blog, 2015)
- 17zuoye (presentation, 2015)
- Custobar (presentation, 2016)
- Blendle (presentation)
- TrustYou (presentation, 2015)
- Groupon / OrderUp (alternative implementation)
- Red Hat - Marketing Operations (blog, 2017)
- GetNinjas (blog, 2017)
- voyages-sncf.com (presentation, 2017)
- Open Targets (blog, 2017)
- Leipzig University Library (presentation, 2016) / (project)
- Synetiq (presentation, 2017)
- Glossier (blog, 2018)
- Data Revenue (blog, 2018)
- Uppsala University (tutorial) / (presentation, 2015) / (slides, 2015) / (poster, 2015) / (paper, 2016) / (project)

- GIPHY (blog, 2019)
- xtream (blog, 2019)
- CIAN (presentation, 2019)

Some more companies are using Luigi but haven't had a chance yet to write about it:

- Schibsted
- enbrite.ly
- Dow Jones / The Wall Street Journal
- Hotels.com
- Newsela
- Squarespace
- OAO
- Grovo
- Weebly
- Deloitte
- Stacktome
- LINX+Neemu+Chaordic
- Foxberry
- Okko
- ISVWorld
- Big Data
- Movio
- Bonnier News
- Starsky Robotics
- BaseTIS
- Hopper
- VOYAGE GROUP/Zucks
- Textpert
- Tracktics
- Whizar
- xtream
- Skyscanner
- Jodel
- Mekar
- M3
- Assist Digital
- Meltwater

- DevSamurai
- Veridas

We're more than happy to have your company added here. Just send a PR on GitHub.

CHAPTER

SIX

EXTERNAL LINKS

- Mailing List for discussions and asking questions. (Google Groups)
- Releases (PyPI)
- Source code (GitHub)
- Hubot Integration plugin for Slack, Hipchat, etc (GitHub)

CHAPTER

SEVEN

AUTHORS

Luigi was built at Spotify, mainly by Erik Bernhardsson and Elias Freider. Many other people have contributed since open sourcing in late 2012. Arash Rouhani was the chief maintainer from 2015 to 2019, and now Spotify's Data Team maintains Luigi.

CHAPTER

EIGHT

TABLE OF CONTENTS

8.1 Example – Top Artists

This is a very simplified case of something we do at Spotify a lot. All user actions are logged to Google Cloud Storage (previously HDFS) where we run a bunch of processing jobs to transform the data. The processing code itself is implemented in a scalable data processing framework, such as Scio, Scalding, or Spark, but the jobs are orchestrated with Luigi. At some point we might end up with a smaller data set that we can bulk ingest into Cassandra, Postgres, or other storage suitable for serving or exploration.

For the purpose of this exercise, we want to aggregate all streams, find the top 10 artists and then put the results into Postgres.

This example is also available in examples/top_artists.py.

8.1.1 Step 1 - Aggregate Artist Streams

```
class AggregateArtists(luigi.Task):
    date_interval = luigi.DateIntervalParameter()

    def output(self):
        return luigi.LocalTarget("data/artist_streams_%s.tsv" % self.date_interval)

    def requires(self):
        return [Streams(date) for date in self.date_interval]

    def run(self):
        artist_count = defaultdict(int)

        for input in self.input():
            with input.open('r') as in_file:
                for line in in_file:
                    timestamp, artist, track = line.strip().split()
                    artist_count[artist] += 1

        with self.output().open('w') as out_file:
            for artist, count in artist_count.iteritems():
                print(artist, count, file=out_file)
```

Note that this is just a portion of the file `examples/top_artists.py`. In particular, `Streams` is defined as a `Task`, acting as a dependency for `AggregateArtists`. In addition, `luigi.run()` is called if the script is executed directly, allowing it to be run from the command line.

There are several pieces of this snippet that deserve more explanation.

- Any `Task` may be customized by instantiating one or more `Parameter` objects on the class level.
- The `output()` method tells Luigi where the result of running the task will end up. The path can be some function of the parameters.
- The `requires()` tasks specifies other tasks that we need to perform this task. In this case it's an external dump named *Streams* which takes the date as the argument.
- For plain Tasks, the `run()` method implements the task. This could be anything, including calling subprocesses, performing long running number crunching, etc. For some subclasses of `Task` you don't have to implement the `run` method. For instance, for the `JobTask` subclass you implement a *mapper* and *reducer* instead.
- `LocalTarget` is a built in class that makes it easy to read/write from/to the local filesystem. It also makes all file operations atomic, which is nice in case your script crashes for any reason.

8.1.2 Running this Locally

Try running this using eg.

```
$ cd examples
$ luigi --module top_artists AggregateArtists --local-scheduler --date-interval 2012-06
```

Note that *top_artists* needs to be in your PYTHONPATH, or else this can produce an error (*ImportError: No module named top_artists*). Add the current working directory to the command PYTHONPATH with:

```
$ PYTHONPATH='.' luigi --module top_artists AggregateArtists --local-scheduler --date-
↪interval 2012-06
```

You can also try to view the manual using `--help` which will give you an overview of the options.

Running the command again will do nothing because the output file is already created. In that sense, any task in Luigi is *idempotent* because running it many times gives the same outcome as running it once. Note that unlike Makefile, the output will not be recreated when any of the input files is modified. You need to delete the output file manually.

The `--local-scheduler` flag tells Luigi not to connect to a scheduler server. This is not recommended for other purpose than just testing things.

8.1.3 Step 1b - Aggregate artists with Spark

While Luigi can process data inline, it is normally used to orchestrate external programs that perform the actual processing. In this example, we will demonstrate how top artists instead can be read from HDFS and calculated with Spark, orchestrated by Luigi.

```
class AggregateArtistsSpark(luigi.contrib.spark.SparkSubmitTask):
    date_interval = luigi.DateIntervalParameter()

    app = 'top_artists_spark.py'
    master = 'local[*]'

    def output(self):
        return luigi.contrib.hdfs.HdfsTarget("data/artist_streams_%s.tsv" % self.date_
↪interval)

    def requires(self):
```

(continues on next page)

(continued from previous page)

```
        return [StreamsHdfs(date) for date in self.date_interval]

    def app_options(self):
        # :func:`~luigi.task.Task.input` returns the targets produced by the tasks in
        # `~luigi.task.Task.requires`.
        return [','.join([p.path for p in self.input()]),
                self.output().path]
```

`luigi.contrib.hadoop.SparkSubmitTask` doesn't require you to implement a `run()` method. Instead, you specify the command line parameters to send to `spark-submit`, as well as any other configuration specific to Spark.

Python code for the Spark job is found below.

```
import operator
import sys
from pyspark.sql import SparkSession


def main(argv):
    input_paths = argv[1].split(',')
    output_path = argv[2]

    spark = SparkSession.builder.getOrCreate()

    streams = spark.read.option('sep', '\t').csv(input_paths[0])
    for stream_path in input_paths[1:]:
        streams.union(spark.read.option('sep', '\t').csv(stream_path))

    # The second field is the artist
    counts = streams \
        .map(lambda row: (row[1], 1)) \
        .reduceByKey(operator.add)

    counts.write.option('sep', '\t').csv(output_path)


if __name__ == '__main__':
    sys.exit(main(sys.argv))
```

In a typical deployment scenario, the Luigi orchestration definition above as well as the Pyspark processing code would be packaged into a deployment package, such as a container image. The processing code does not have to be implemented in Python, any program can be packaged in the image and run from Luigi.

8.1.4 Step 2 – Find the Top Artists

At this point, we've counted the number of streams for each artists, for the full time period. We are left with a large file that contains mappings of artist -> count data, and we want to find the top 10 artists. Since we only have a few hundred thousand artists, and calculating artists is nontrivial to parallelize, we choose to do this not as a Hadoop job, but just as a plain old for-loop in Python.

```
class Top10Artists(luigi.Task):
    date_interval = luigi.DateIntervalParameter()
    use_hadoop = luigi.BoolParameter()

    def requires(self):
        if self.use_hadoop:
            return AggregateArtistsSpark(self.date_interval)
        else:
            return AggregateArtists(self.date_interval)

    def output(self):
        return luigi.LocalTarget("data/top_artists_%s.tsv" % self.date_interval)

    def run(self):
        top_10 = nlargest(10, self._input_iterator())
        with self.output().open('w') as out_file:
            for streams, artist in top_10:
                print(self.date_interval.date_a, self.date_interval.date_b, artist,␣
↪streams, file=out_file)

    def _input_iterator(self):
        with self.input().open('r') as in_file:
            for line in in_file:
                artist, streams = line.strip().split()
                yield int(streams), int(artist)
```

The most interesting thing here is that this task (*Top10Artists*) defines a dependency on the previous task (*AggregateArtists*). This means that if the output of *AggregateArtists* does not exist, the task will run before *Top10Artists*.

```
$ luigi --module examples.top_artists Top10Artists --local-scheduler --date-interval␣
↪2012-07
```

This will run both tasks.

8.1.5 Step 3 - Insert into Postgres

This mainly serves as an example of a specific subclass *Task* that doesn't require any code to be written. It's also an example of how you can define task templates that you can reuse for a lot of different tasks.

```
class ArtistToplistToDatabase(luigi.contrib.postgres.CopyToTable):
    date_interval = luigi.DateIntervalParameter()
    use_hadoop = luigi.BoolParameter()

    host = "localhost"
    database = "toplists"
    user = "luigi"
```

(continues on next page)

(continued from previous page)

```
    password = "abc123"  # ;)
    table = "top10"

    columns = [("date_from", "DATE"),
               ("date_to", "DATE"),
               ("artist", "TEXT"),
               ("streams", "INT")]

    def requires(self):
        return Top10Artists(self.date_interval, self.use_hadoop)
```

Just like previously, this defines a recursive dependency on the previous task. If you try to build the task, that will also trigger building all its upstream dependencies.

8.1.6 Using the Central Planner

The `--local-scheduler` flag tells Luigi not to connect to a central scheduler. This is recommended in order to get started and or for development purposes. At the point where you start putting things in production we strongly recommend running the central scheduler server. In addition to providing locking so that the same task is not run by multiple processes at the same time, this server also provides a pretty nice visualization of your current work flow.

If you drop the `--local-scheduler` flag, your script will try to connect to the central planner, by default at localhost port 8082. If you run

```
$ luigid
```

in the background and then run your task without the `--local-scheduler` flag, then your script will now schedule through a centralized server. You need Tornado for this to work.

Launching http://localhost:8082 should show something like this:

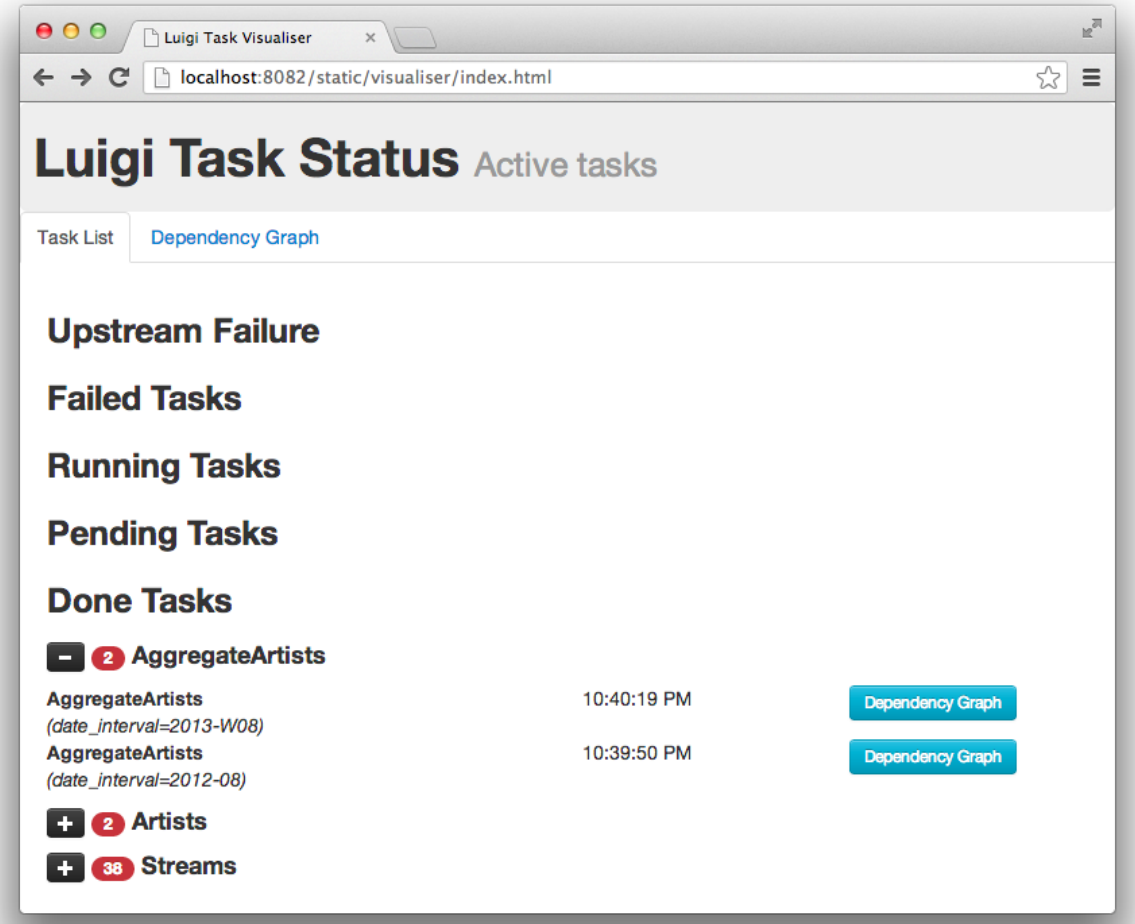
Web server screenshot Looking at the dependency graph for any of the tasks yields something like this:

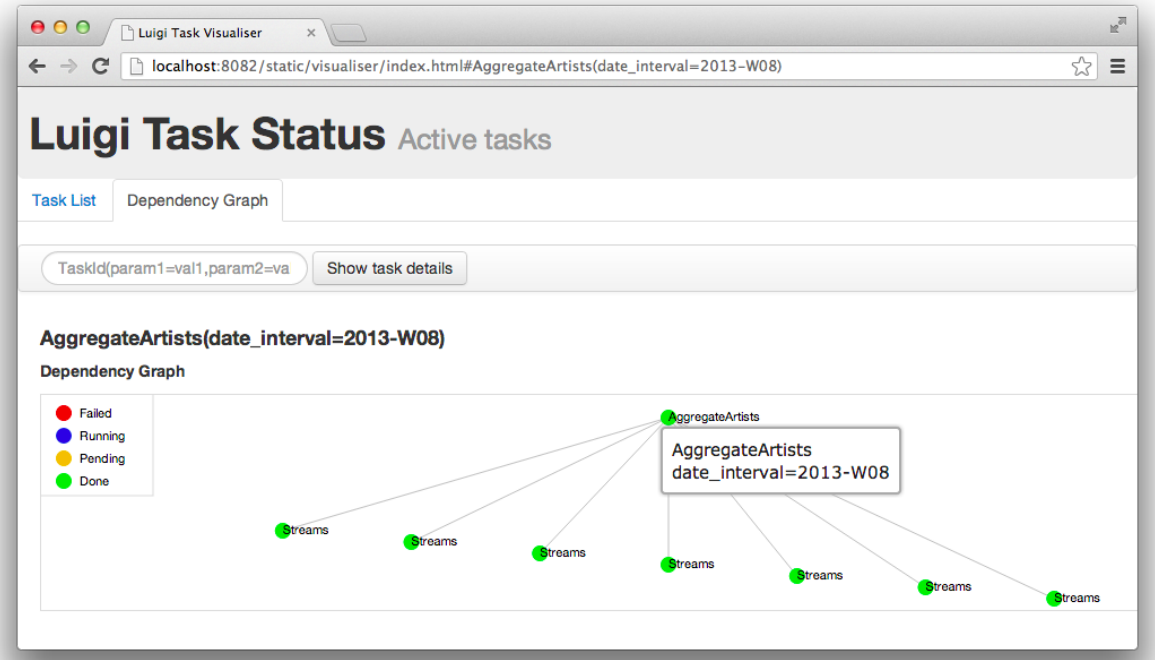
Aggregate artists screenshot

In production, you'll want to run the centralized scheduler. See: *Using the Central Scheduler* for more information.

8.2 Building workflows

There are two fundamental building blocks of Luigi - the `Task` class and the `Target` class. Both are abstract classes and expect a few methods to be implemented. In addition to those two concepts, the `Parameter` class is an important concept that governs how a Task is run.


Luigi Task Visualiser
localhost:8082/static/visualiser/index.html
Luigi Task Status
Active tasks
Task List
Dependency Graph
Upstream Failure
Failed Tasks
Running Tasks
Pending Tasks
Done Tasks
2 AggregateArtists
AggregateArtists
(date_interval=2013-W08)
10:40:19 PM
Dependency Graph
AggregateArtists
(date_interval=2012-08)
10:39:50 PM
Dependency Graph
2 Artists
38 Streams



8.2.1 Target

The `Target` class corresponds to a file on a disk, a file on HDFS or some kind of a checkpoint, like an entry in a database. Actually, the only method that Targets have to implement is the *exists* method which returns True if and only if the Target exists.

In practice, implementing Target subclasses is rarely needed. Luigi comes with a toolbox of several useful Targets. In particular, `LocalTarget` and `HdfsTarget`, but there is also support for other file systems: `luigi.contrib.s3.S3Target`, `luigi.contrib.ssh.RemoteTarget`, `luigi.contrib.ftp.RemoteTarget`, `luigi.contrib.mysqldb.MySqlTarget`, `luigi.contrib.redshift.RedshiftTarget`, and several more.

Most of these targets, are file system-like. For instance, `LocalTarget` and `HdfsTarget` map to a file on the local drive or a file in HDFS. In addition these also wrap the underlying operations to make them atomic. They both implement the `open()` method which returns a stream object that could be read (`mode='r'`) from or written to (`mode='w'`).

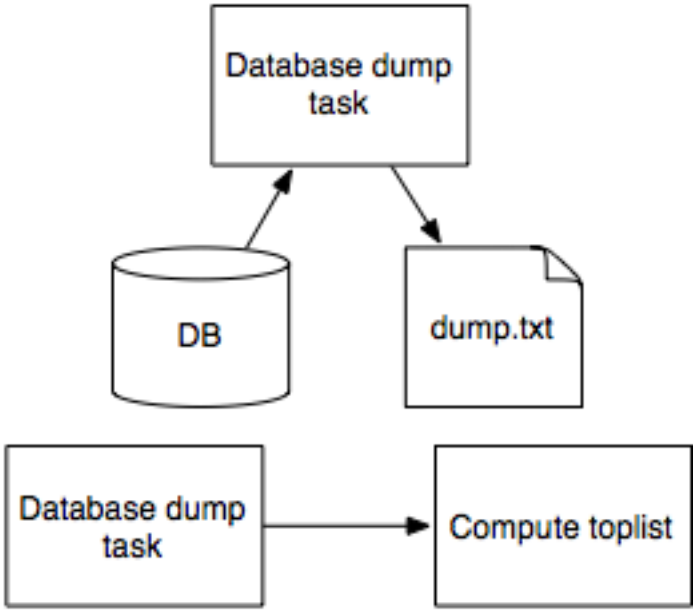
Luigi comes with Gzip support by providing `format=format.Gzip`. Adding support for other formats is pretty simple.

8.2.2 Task

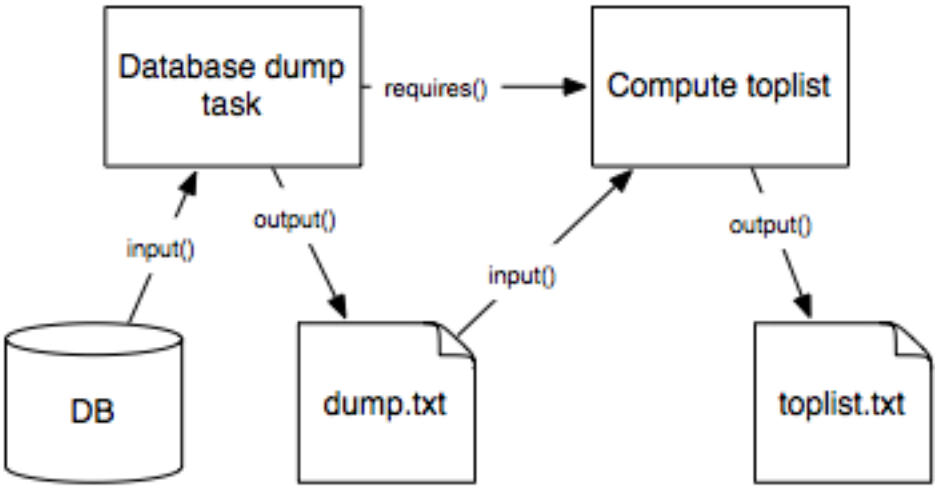
The `Task` class is a bit more conceptually interesting because this is where computation is done. There are a few methods that can be implemented to alter its behavior, most notably `run()`, `output()` and `requires()`.

Tasks consume Targets that were created by some other task. They usually also output targets:

You can define dependencies between *Tasks* using the `requires()` method. See *Tasks* for more info.

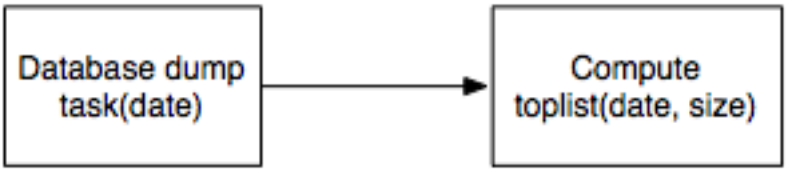


Each task defines its outputs using the `output()` method. Additionally, there is a helper method `input()` that returns the corresponding Target classes for each Task dependency.



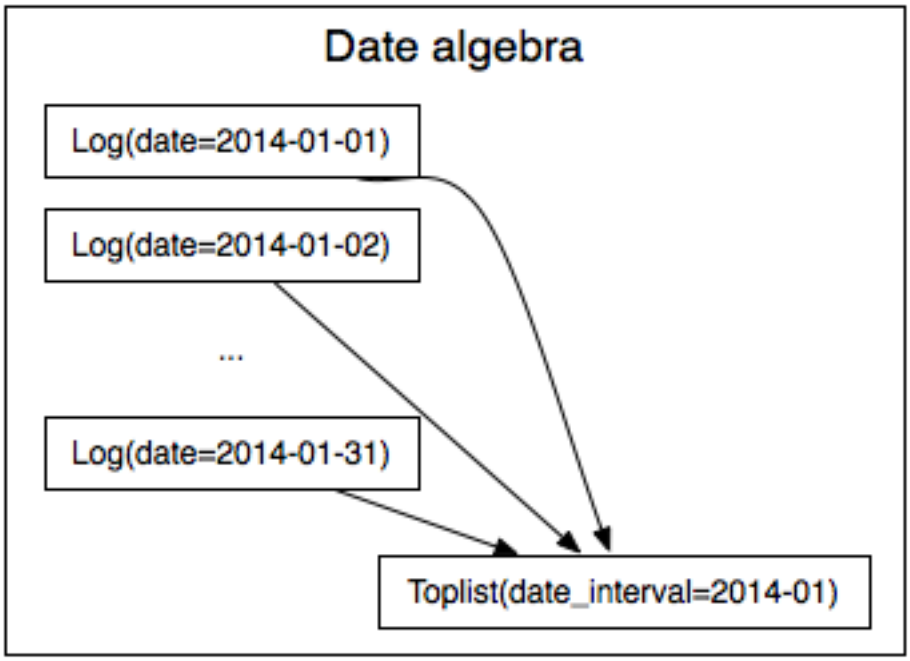
8.2.3 Parameter

The Task class corresponds to some type of job that is run, but in general you want to allow some form of parameterization of it. For instance, if your Task class runs a Hadoop job to create a report every night, you probably want to make the date a parameter of the class. See *Parameters* for more info.



8.2.4 Dependencies

Using tasks, targets, and parameters, Luigi lets you express arbitrary dependencies in *code*, rather than using some kind of awkward config DSL. This is really useful because in the real world, dependencies are often very messy. For instance, some examples of the dependencies you might encounter:

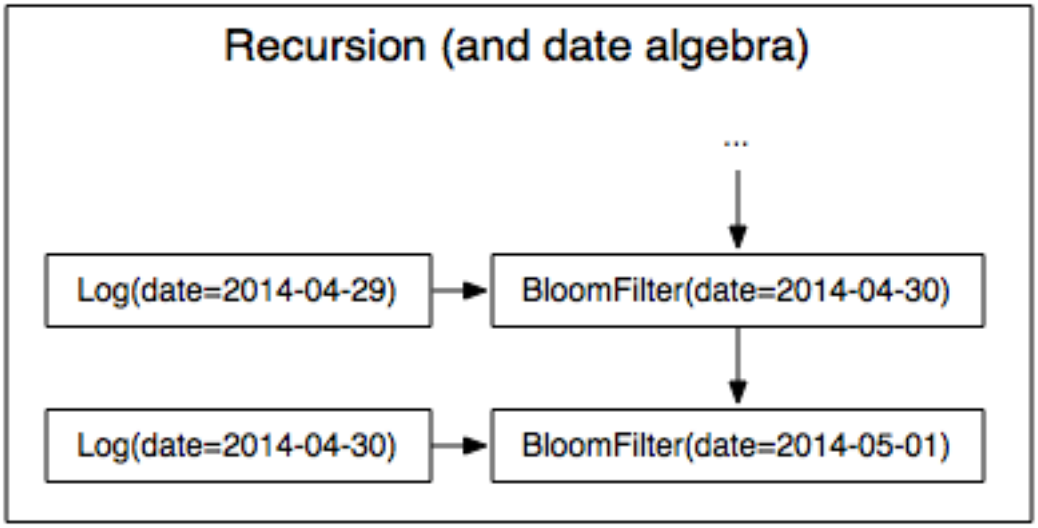


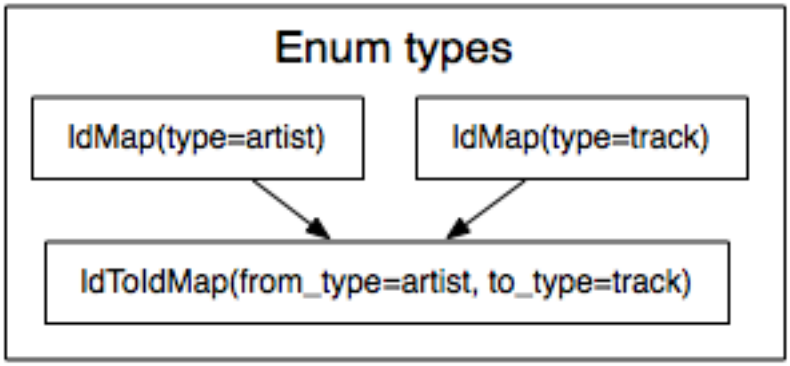
(These diagrams are from a Luigi presentation in late 2014 at NYC Data Science meetup)

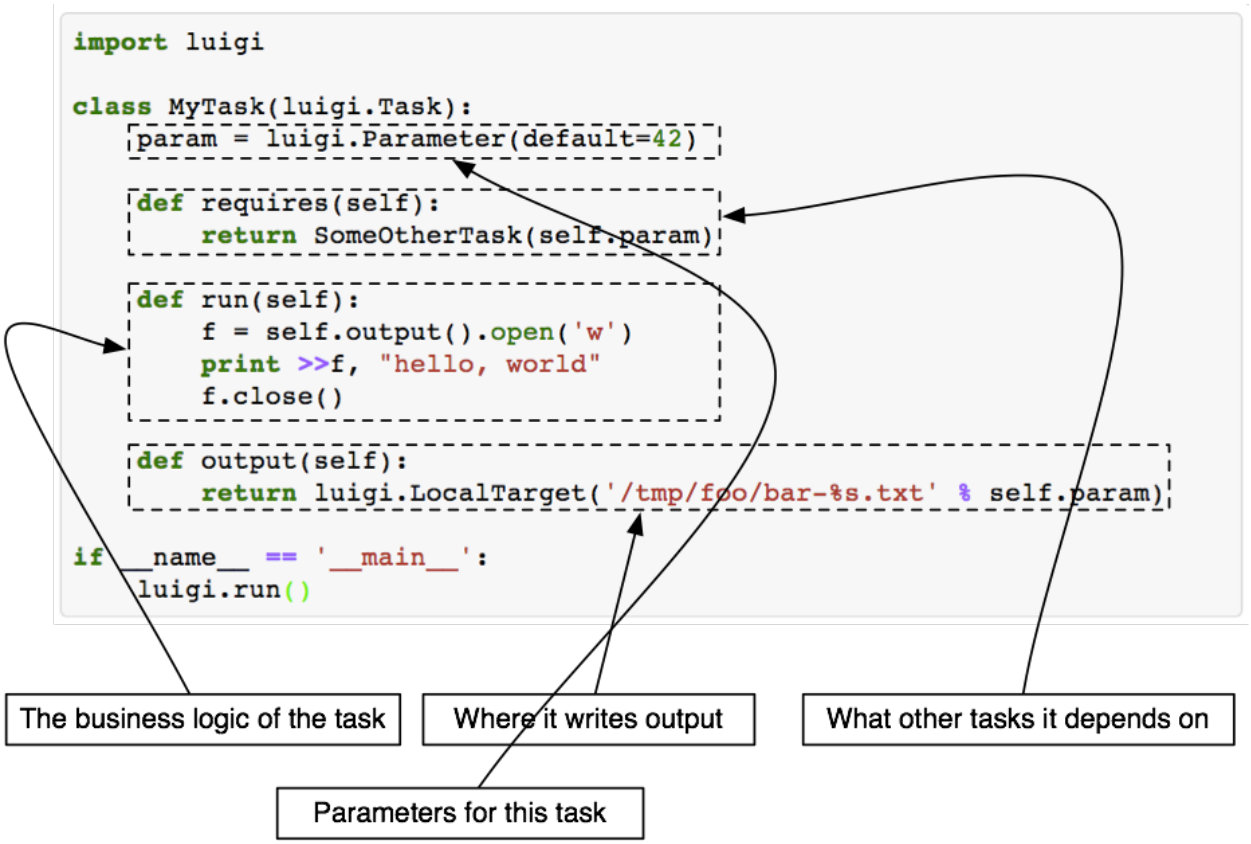
8.3 Tasks

Tasks are where the execution takes place. Tasks depend on each other and output targets.

An outline of how a task can look like:


Recursion (and date algebra)
...
Log(date=2014-04-29)
BloomFilter(date=2014-04-30)
Log(date=2014-04-30)
BloomFilter(date=2014-05-01)


Enum types
IdMap(type=artist)
IdMap(type=track)
IdToIdMap(from_type=artist, to_type=track)



8.3.1 Task.requires

The `requires()` method is used to specify dependencies on other Task object, which might even be of the same class. For instance, an example implementation could be

```
def requires(self):
    return OtherTask(self.date), DailyReport(self.date - datetime.timedelta(1))
```

In this case, the DailyReport task depends on two inputs created earlier, one of which is the same class. requires can return other Tasks in any way wrapped up within dicts/lists/tuples/etc.

8.3.2 Requiring another Task

Note that `requires()` can *not* return a `Target` object. If you have a simple Target object that is created externally you can wrap it in a Task class like this:

```
class LogFiles(luigi.ExternalTask):
    def output(self):
        return luigi.contrib.hdfs.HdfsTarget('/log')
```

This also makes it easier to add parameters:

```
class LogFiles(luigi.ExternalTask):
    date = luigi.DateParameter()
```

(continues on next page)

(continued from previous page)

```
    def output(self):
        return luigi.contrib.hdfs.HdfsTarget(self.date.strftime('/log/%Y-%m-%d'))
```

8.3.3 Task.output

The `output()` method returns one or more `Target` objects. Similarly to requires, you can return them wrapped up in any way that's convenient for you. However we recommend that any `Task` only return one single `Target` in output. If multiple outputs are returned, atomicity will be lost unless the `Task` itself can ensure that each `Target` is atomically created. (If atomicity is not of concern, then it is safe to return multiple `Target` objects.)

```
class DailyReport(luigi.Task):
    date = luigi.DateParameter()
    def output(self):
        return luigi.contrib.hdfs.HdfsTarget(self.date.strftime('/reports/%Y-%m-%d'))
    # ...
```

8.3.4 Task.run

The `run()` method now contains the actual code that is run. When you are using *Task.requires* and *Task.run* Luigi breaks down everything into two stages. First it figures out all dependencies between tasks, then it runs everything. The `input()` method is an internal helper method that just replaces all Task objects in requires with their corresponding output. An example:

```
class GenerateWords(luigi.Task):

    def output(self):
        return luigi.LocalTarget('words.txt')

    def run(self):

        # write a dummy list of words to output file
        words = [
                'apple',
                'banana',
                'grapefruit'
                ]

        with self.output().open('w') as f:
            for word in words:
                f.write('{word}\n'.format(word=word))


class CountLetters(luigi.Task):

    def requires(self):
        return GenerateWords()

    def output(self):
        return luigi.LocalTarget('letter_counts.txt')
```

(continues on next page)

(continued from previous page)

```
    def run(self):

        # read in file as list
        with self.input().open('r') as infile:
            words = infile.read().splitlines()

        # write each word to output file with its corresponding letter count
        with self.output().open('w') as outfile:
            for word in words:
                outfile.write(
                        '{word} | {letter_count}\n'.format(
                            word=word,
                            letter_count=len(word)
                            )
                        )
```

It's useful to note that if you're writing to a binary file, Luigi automatically strips the `'b'` flag due to how atomic writes/reads work. In order to write a binary file, such as a pickle file, you should instead use `format=Nop` when calling LocalTarget. Following the above example:

```
from luigi.format import Nop

class GenerateWords(luigi.Task):

    def output(self):
        return luigi.LocalTarget('words.pckl', format=Nop)

    def run(self):
        import pickle

        # write a dummy list of words to output file
        words = [
                'apple',
                'banana',
                'grapefruit'
                ]

        with self.output().open('w') as f:
            pickle.dump(words, f)
```

It is your responsibility to ensure that after running `run()`, the task is complete, i.e. `complete()` returns `True`. Unless you have overridden `complete()`, `run()` should generate all the targets defined as outputs. Luigi verifies that you adhere to the contract before running downstream dependencies, and reports `Unfulfilled dependencies at run time` if a violation is detected.

8.3.5 Task.input

As seen in the example above, `input()` is a wrapper around *Task.requires* that returns the corresponding Target objects instead of Task objects. Anything returned by *Task.requires* will be transformed, including lists, nested dicts, etc. This can be useful if you have many dependencies:

```
class TaskWithManyInputs(luigi.Task):
    def requires(self):
        return {'a': TaskA(), 'b': [TaskB(i) for i in xrange(100)]}

    def run(self):
        f = self.input()['a'].open('r')
        g = [y.open('r') for y in self.input()['b']]
```

8.3.6 Dynamic dependencies

Sometimes you might not know exactly what other tasks to depend on until runtime. In that case, Luigi provides a mechanism to specify dynamic dependencies. If you yield another `Task` in the *Task.run* method, the current task will be suspended and the other task will be run. You can also yield a list of tasks.

```
class MyTask(luigi.Task):
    def run(self):
        other_target = yield OtherTask()

        # dynamic dependencies resolve into targets
        f = other_target.open('r')
```

This mechanism is an alternative to *Task.requires* in case you are not able to build up the full dependency graph before running the task. It does come with some constraints: the *Task.run* method will resume from scratch each time a new task is yielded. In other words, you should make sure your *Task.run* method is idempotent. (This is good practice for all Tasks in Luigi, but especially so for tasks with dynamic dependencies). As this might entail redundant calls to tasks' `complete()` methods, you should consider setting the "cache_task_completion" option in the *[worker]*. To further control how dynamic task requirements are handled internally by worker nodes, there is also the option to wrap dependent tasks by `DynamicRequirements`.

For an example of a workflow using dynamic dependencies, see examples/dynamic_requirements.py.

8.3.7 Task status tracking

For long-running or remote tasks it is convenient to see extended status information not only on the command line or in your logs but also in the GUI of the central scheduler. Luigi implements dynamic status messages, progress bar and tracking urls which may point to an external monitoring system. You can set this information using callbacks within *Task.run*:

```
class MyTask(luigi.Task):
    def run(self):
        # set a tracking url
        self.set_tracking_url("http://...")

        # set status messages during the workload
        for i in range(100):
            # do some hard work here
```

(continues on next page)

(continued from previous page)

```
            if i % 10 == 0:
                self.set_status_message("Progress: %d / 100" % i)
                # displays a progress bar in the scheduler UI
                self.set_progress_percentage(i)
```

8.3.8 Events and callbacks

Luigi has a built-in event system that allows you to register callbacks to events and trigger them from your own tasks. You can both hook into some pre-defined events and create your own. Each event handle is tied to a Task class and will be triggered only from that class or a subclass of it. This allows you to effortlessly subscribe to events only from a specific class (e.g. for hadoop jobs).

```
@luigi.Task.event_handler(luigi.Event.SUCCESS)
def celebrate_success(task):
    """Will be called directly after a successful execution
       of `run` on any Task subclass (i.e. all luigi Tasks)
    """
    ...

@luigi.contrib.hadoop.JobTask.event_handler(luigi.Event.FAILURE)
def mourn_failure(task, exception):
    """Will be called directly after a failed execution
       of `run` on any JobTask subclass
    """
    ...

luigi.run()
```

8.3.9 But I just want to run a Hadoop job?

The Hadoop code is integrated in the rest of the Luigi code because we really believe almost all Hadoop jobs benefit from being part of some sort of workflow. However, in theory, nothing stops you from using the `JobTask` class (and also `HdfsTarget`) without using the rest of Luigi. You can simply run it manually using

```
MyJobTask('abc', 123).run()
```

You can use the hdfs.target.HdfsTarget class anywhere by just instantiating it:

```
t = luigi.contrib.hdfs.target.HdfsTarget('/tmp/test.gz', format=format.Gzip)
f = t.open('w')
# ...
f.close() # needed
```

8.3.10 Task priority

The scheduler decides which task to run next from the set of all tasks that have all their dependencies met. By default, this choice is pretty arbitrary, which is fine for most workflows and situations.

If you want to have some control on the order of execution of available tasks, you can set the `priority` property of a task, for example as follows:

```
# A static priority value as a class constant:
class MyTask(luigi.Task):
    priority = 100
    # ...

# A dynamic priority value with a "@property" decorated method:
class OtherTask(luigi.Task):
    @property
    def priority(self):
        if self.date > some_threshold:
            return 80
        else:
            return 40
    # ...
```

Tasks with a higher priority value will be picked before tasks with a lower priority value. There is no predefined range of priorities, you can choose whatever (int or float) values you want to use. The default value is 0.

Warning: task execution order in Luigi is influenced by both dependencies and priorities, but in Luigi dependencies come first. For example: if there is a task A with priority 1000 but still with unmet dependencies and a task B with priority 1 without any pending dependencies, task B will be picked first.

8.3.11 Namespaces, families and ids

In order to avoid name clashes and to be able to have an identifier for tasks, Luigi introduces the concepts *task_namespace*, *task_family* and *task_id*. The namespace and family operate on class level meanwhile the task id only exists on instance level. The concepts are best illustrated using code.

```
import luigi
class MyTask(luigi.Task):
    my_param = luigi.Parameter()
    task_namespace = 'my_namespace'

my_task = MyTask(my_param='hello')
print(my_task)                      # --> my_namespace.MyTask(my_param=hello)

print(my_task.get_task_namespace()) # --> my_namespace
print(my_task.get_task_family())    # --> my_namespace.MyTask
print(my_task.task_id)              # --> my_namespace.MyTask_hello_890907e7ce

print(MyTask.get_task_namespace())  # --> my_namespace
print(MyTask.get_task_family())     # --> my_namespace.MyTask
print(MyTask.task_id)               # --> Error!
```

The full documentation for this machinery exists in the `task` module.

8.3.12 Instance caching

In addition to the stuff mentioned above, Luigi also does some metaclass logic so that if e.g. `DailyReport(datetime.date(2012, 5, 10))` is instantiated twice in the code, it will in fact result in the same object. See *Instance caching* for more info

8.4 Parameters

Parameters is the Luigi equivalent of creating a constructor for each Task. Luigi requires you to declare these parameters by instantiating `Parameter` objects on the class scope:

```
class DailyReport(luigi.contrib.hadoop.JobTask):
    date = luigi.DateParameter(default=datetime.date.today())
    # ...
```

By doing this, Luigi can take care of all the boilerplate code that would normally be needed in the constructor. Internally, the DailyReport object can now be constructed by running `DailyReport(datetime.date(2012, 5, 10))` or just `DailyReport()`. Luigi also creates a command line parser that automatically handles the conversion from strings to Python types. This way you can invoke the job on the command line eg. by passing `--date 2012-05-10`.

The parameters are all set to their values on the Task object instance, i.e.

```
d = DailyReport(datetime.date(2012, 5, 10))
print(d.date)
```

will return the same date that the object was constructed with. Same goes if you invoke Luigi on the command line.

8.4.1 Instance caching

Tasks are uniquely identified by their class name and values of their parameters. In fact, within the same worker, two tasks of the same class with parameters of the same values are not just equal, but the same instance:

```
>>> import luigi
>>> import datetime
>>> class DateTask(luigi.Task):
...   date = luigi.DateParameter()
...
>>> a = datetime.date(2014, 1, 21)
>>> b = datetime.date(2014, 1, 21)
>>> a is b
False
>>> c = DateTask(date=a)
>>> d = DateTask(date=b)
>>> c
DateTask(date=2014-01-21)
>>> d
DateTask(date=2014-01-21)
>>> c is d
True
```

8.4.2 Insignificant parameters

If a parameter is created with `significant=False`, it is ignored as far as the Task signature is concerned. Tasks created with only insignificant parameters differing have the same signature but are not the same instance:

```
>>> class DateTask2(DateTask):
...   other = luigi.Parameter(significant=False)
...
>>> c = DateTask2(date=a, other="foo")
>>> d = DateTask2(date=b, other="bar")
>>> c
DateTask2(date=2014-01-21)
>>> d
DateTask2(date=2014-01-21)
>>> c.other
'foo'
>>> d.other
'bar'
>>> c is d
False
>>> hash(c) == hash(d)
True
```

8.4.3 Parameter visibility

Using `ParameterVisibility` you can configure parameter visibility. By default, all parameters are public, but you can also set them hidden or private.

```
>>> import luigi
>>> from luigi.parameter import ParameterVisibility

>>> luigi.Parameter(visibility=ParameterVisibility.PRIVATE)
```

`ParameterVisibility.PUBLIC` (default) - visible everywhere

`ParameterVisibility.HIDDEN` - ignored in WEB-view, but saved into database if save db_history is true

`ParameterVisibility.PRIVATE` - visible only inside task.

8.4.4 Parameter types

In the examples above, the *type* of the parameter is determined by using different subclasses of `Parameter`. There are a few of them, like `DateParameter`, `DateIntervalParameter`, `IntParameter`, `FloatParameter`, etc.

Python is not a statically typed language and you don't have to specify the types of any of your parameters. You can simply use the base class `Parameter` if you don't care.

The reason you would use a subclass like `DateParameter` is that Luigi needs to know its type for the command line interaction. That's how it knows how to convert a string provided on the command line to the corresponding type (i.e. datetime.date instead of a string).

8.4.5 Setting parameter value for other classes

All parameters are also exposed on a class level on the command line interface. For instance, say you have classes TaskA and TaskB:

```
class TaskA(luigi.Task):
    x = luigi.Parameter()

class TaskB(luigi.Task):
    y = luigi.Parameter()
```

You can run `TaskB` on the command line: `luigi TaskB --y 42`. But you can also set the class value of `TaskA` by running `luigi TaskB --y 42 --TaskA-x 43`. This sets the value of `TaskA.x` to 43 on a *class* level. It is still possible to override it inside Python if you instantiate `TaskA(x=44)`.

All parameters can also be set from the configuration file. For instance, you can put this in the config:

```
[TaskA]
x: 45
```

Just as in the previous case, this will set the value of `TaskA.x` to 45 on the *class* level. And likewise, it is still possible to override it inside Python if you instantiate `TaskA(x=44)`.

8.4.6 Parameter resolution order

Parameters are resolved in the following order of decreasing priority:

1. Any value passed to the constructor, or task level value set on the command line (applies on an instance level)
2. Any value set on the command line (applies on a class level)
3. Any configuration option (applies on a class level)
4. Any default value provided to the parameter (applies on a class level)

See the `Parameter` class for more information.

8.5 Running Luigi

8.5.1 Running from the Command Line

The preferred way to run Luigi tasks is through the `luigi` command line tool that will be installed with the pip package.

```
# my_module.py, available in your sys.path
import luigi

class MyTask(luigi.Task):
    x = luigi.IntParameter()
    y = luigi.IntParameter(default=45)

    def run(self):
        print(self.x + self.y)
```

Should be run like this

```
$ luigi --module my_module MyTask --x 123 --y 456 --local-scheduler
```

Or alternatively like this:

```
$ python -m luigi --module my_module MyTask --x 100 --local-scheduler
```

Note that if a parameter name contains '_', it should be replaced by '-'. For example, if MyTask had a parameter called 'my_parameter':

```
$ luigi --module my_module MyTask --my-parameter 100 --local-scheduler
```

Note: Please make sure to always place task parameters behind the task family!

8.5.2 Running from Python code

Another way to start tasks from Python code is using `luigi.build(tasks, worker_scheduler_factory=None, **env_params)` from `luigi.interface` module.

This way of running luigi tasks is useful if you want to get some dynamic parameters from another source, such as database, or provide additional logic before you start tasks.

One notable difference is that `build` defaults to not using the identical process lock. If you want to change this behaviour, just pass `no_lock=False`.

```
class MyTask1(luigi.Task):
    x = luigi.IntParameter()
    y = luigi.IntParameter(default=0)

    def run(self):
        print(self.x + self.y)


class MyTask2(luigi.Task):
    x = luigi.IntParameter()
    y = luigi.IntParameter(default=1)
    z = luigi.IntParameter(default=2)

    def run(self):
        print(self.x * self.y * self.z)


if __name__ == '__main__':
    luigi.build([MyTask1(x=10), MyTask2(x=15, z=3)])
```

Also, it is possible to pass additional parameters to `build` such as host, port, workers and local_scheduler:

```
if __name__ == '__main__':
     luigi.build([MyTask1(x=1)], workers=5, local_scheduler=True)
```

To achieve some special requirements you can pass to `build` your `worker_scheduler_factory` which will return your worker and/or scheduler implementations:

```
class MyWorker(Worker):
    # some custom logic


class MyFactory:
  def create_local_scheduler(self):
      return scheduler.Scheduler(prune_on_get_work=True, record_task_history=False)

  def create_remote_scheduler(self, url):
      return rpc.RemoteScheduler(url)

  def create_worker(self, scheduler, worker_processes, assistant=False):
      # return your worker instance
      return MyWorker(
          scheduler=scheduler, worker_processes=worker_processes, assistant=assistant)


if __name__ == '__main__':
    luigi.build([MyTask1(x=1)], worker_scheduler_factory=MyFactory())
```

In some cases (like task queue) it may be useful.

8.5.3 Response of luigi.build()/luigi.run()

- **Default response** By default *luigi.build()/luigi.run()* returns True if there were no scheduling errors. This is the same as the attribute `LuigiRunResult.scheduling_succeeded`.
- **Detailed response** This is a response of type `LuigiRunResult`. This is obtained by passing a keyword argument `detailed_summary=True` to *build/run*. This response contains detailed information about the jobs.

```
if __name__ == '__main__':
    luigi_run_result = luigi.build(..., detailed_summary=True)
    print(luigi_run_result.summary_text)
```

8.5.4 Luigi on Windows

Most Luigi functionality works on Windows. Exceptions:

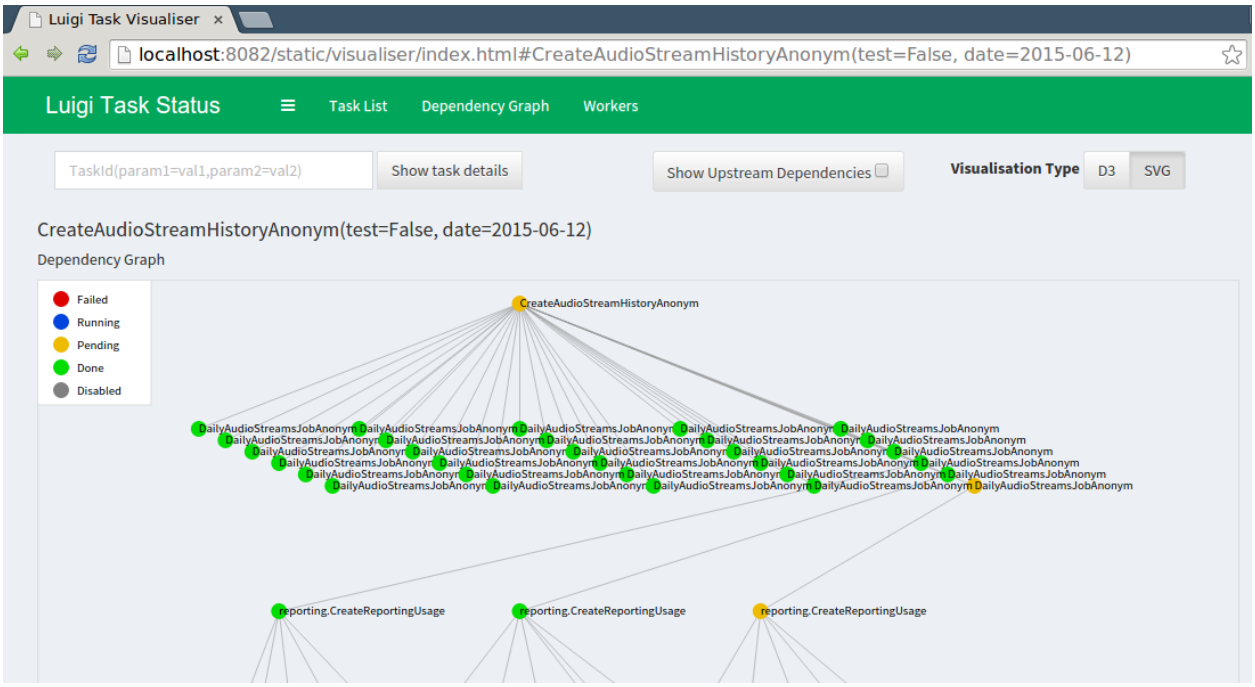
- Specifying multiple worker processes using the `workers` argument for `luigi.build`, or using the `--workers` command line argument. (Similarly, specifying `--worker-force-multiprocessing`). For most programs, this will result in failure (a common sight is `BrokenPipeError`). The reason is that worker processes are assumed to be forked from the main process. Forking is not possible on Windows.
- Running the Luigi central scheduling server as a daemon (i.e. with `--background`). Again, a Unix-only concept.

8.6 Using the Central Scheduler

While the `--local-scheduler` flag is useful for development purposes, it's not recommended for production usage. The centralized scheduler serves two purposes:

- Make sure two instances of the same task are not running simultaneously
- Provide visualization of everything that's going on.

Note that the central scheduler does not execute anything for you or help you with job parallelization. For running tasks periodically, the easiest thing to do is to trigger a Python script from cron or from a continuously running process. There is no central process that automatically triggers jobs. This model may seem limited, but we believe that it makes things far more intuitive and easy to understand.



8.6.1 The luigid server

To run the server as a daemon run:

```
$ luigid --background --pidfile <PATH_TO_PIDFILE> --logdir <PATH_TO_LOGDIR> --state-path
↪<PATH_TO_STATEFILE>
```

Note that this requires `python-daemon`. By default, the server starts on AF_INET and AF_INET6 port `8082` (which can be changed with the `--port` flag) and listens on all IPs. To change the default behavior of listening on all IPs, pass the `--address` flag and the IP address to listen on. To use an AF_UNIX socket use the `--unix-socket` flag.

For a full list of configuration options and defaults, see the *scheduler configuration section*. Note that `luigid` uses the same configuration files as the Luigi client (i.e. `luigi.cfg` or `/etc/luigi/client.cfg` by default).

8.6.2 Enabling Task History

Task History is an experimental feature in which additional information about tasks that have been executed are recorded in a relational database for historical analysis. This information is exposed via the Central Scheduler at `/history`.

To enable the task history, specify `record_task_history = True` in the `[scheduler]` section of `luigi.cfg` and specify `db_connection` under `[task_history]`. The `db_connection` string is used to configure the SQLAlchemy engine. When starting up, `luigid` will create all the necessary tables using create_all.

Example configuration

```
[scheduler]
record_task_history = True
state_path = /usr/local/var/luigi-state.pickle

[task_history]
db_connection = sqlite:////usr/local/var/luigi-task-hist.db
```

The task history has the following pages:

- `/history` a reverse-cronological listing of runs from the past 24 hours. Example screenshot:

Name	Host	Last Action	Status
WordCount	None	2014-12-31 20:16:58.505362	DONE
WordCount	None	2014-12-31 20:16:56.602269	DONE
InputText	None	2014-12-31 20:16:52.233391	PENDING
WordCount	None	2014-12-31 20:16:52.210956	PENDING

- `/history/by_id/{id}` detailed information about a run, including: parameter values, the host on which it ran, and timing information. Example screenshot:

- `/history/by_name/{name}` a listing of all runs of a task with the given task `{name}`. Example screenshot:

- `/history/by_params/{name}?data=params` a listing of all runs of the task `{name}` restricted to runs with `params` matching the given history. The `params` is a json blob describing the parameters, e.g. `data={"foo": "bar"}` looks for a task with `foo=bar`.

Info

Task Id	4
Task Name	WordCount
Host	None
More	All "WordCount" runs

Parameters

Name	Value
date_interval	2014-12-31

Actions

Status	Action Time
DONE	2014-12-31 20:16:58.505362

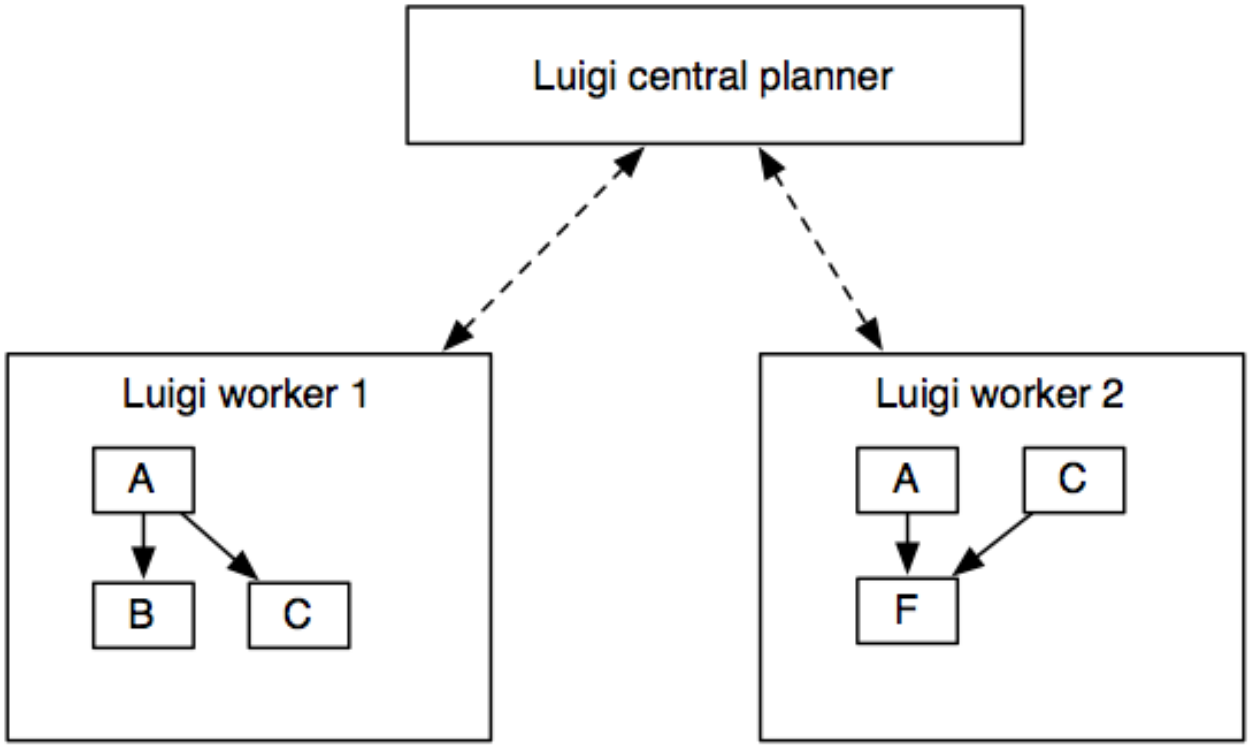
Name	Host	Last Action	Status
WordCount	None	2014-12-31 20:16:52.210956	PENDING
WordCount	None	2014-12-31 20:16:56.602269	DONE
WordCount	None	2014-12-31 20:16:58.505362	DONE

8.7 Execution Model

Luigi has a quite simple model for execution and triggering.

8.7.1 Workers and task execution

The most important aspect is that *no execution is transferred*. When you run a Luigi workflow, the worker schedules all tasks, and also executes the tasks within the process.



The benefit of this scheme is that it's super easy to debug since all execution takes place in the process. It also makes deployment a non-event. During development, you typically run the Luigi workflow from the command line, whereas when you deploy it, you can trigger it using crontab or any other scheduler.

The downside is that Luigi doesn't give you scalability for free. In practice this is not a problem until you start running thousands of tasks.

Isn't the point of Luigi to automate and schedule these workflows? To some extent. Luigi helps you *encode the dependencies* of tasks and build up chains. Furthermore, Luigi's scheduler makes sure that there's a centralized view of the dependency graph and that the same job will not be executed by multiple workers simultaneously.

8.7.2 Scheduler

A client only starts the `run()` method of a task when the single-threaded central scheduler has permitted it. Since the number of tasks is usually very small (in comparison with the petabytes of data one task is processing), we can afford the convenience of a simple centralised server.

The gif is from this presentation, which is about the client and server interaction.

8.7.3 Triggering tasks

Luigi does not include its own triggering, so you have to rely on an external scheduler such as crontab to actually trigger the workflows.

In practice, it's not a big hurdle because Luigi avoids all the mess typically caused by it. Scheduling a complex workflow is fairly trivial using eg. crontab.

In the future, Luigi might implement its own triggering. The dependency on crontab (or any external triggering mechanism) is a bit awkward and it would be nice to avoid.

Trigger example

For instance, if you have an external data dump that arrives every day and that your workflow depends on it, you write a workflow that depends on this data dump. Crontab can then trigger this workflow *every minute* to check if the data has arrived. If it has, it will run the full dependency graph.

```
# my_tasks.py

class DataDump(luigi.ExternalTask):
    date = luigi.DateParameter()
    def output(self): return luigi.contrib.hdfs.HdfsTarget(self.date.strftime('/var/log/
↪dump/%Y-%m-%d.txt'))

class AggregationTask(luigi.Task):
    date = luigi.DateParameter()
    window = luigi.IntParameter()
    def requires(self): return [DataDump(self.date - datetime.timedelta(i)) for i in␣
↪xrange(self.window)]
    def run(self): run_some_cool_stuff(self.input())
    def output(self): return luigi.contrib.hdfs.HdfsTarget('/aggregated-%s-%d' % (self.
↪date, self.window))

class RunAll(luigi.Task):
    ''' Dummy task that triggers execution of a other tasks'''
    def requires(self):
        for window in [3, 7, 14]:
            for d in xrange(10): # guarantee that aggregations were run for the past 10␣
↪days
                yield AggregationTask(datetime.date.today() - datetime.timedelta(d),␣
↪window)
```

In your cronline you would then have something like

```
30 0 * * * my-user luigi RunAll --module my_tasks
```

You can trigger this as much as you want from crontab, and even across multiple machines, because the central scheduler will make sure at most one of each `AggregationTask` task is run simultaneously. Note that this might actually mean multiple tasks can be run because there are instances with different parameters, and this can give you some form of parallelization (eg. `AggregationTask(2013-01-09)` might run in parallel with `AggregationTask(2013-01-08)`).

Of course, some Task types (eg. `HadoopJobTask`) can transfer execution to other places, but this is up to each Task to define.

8.8 Luigi Patterns

8.8.1 Code Reuse

One nice thing about Luigi is that it's super easy to depend on tasks defined in other repos. It's also trivial to have "forks" in the execution path, where the output of one task may become the input of many other tasks.

Currently, no semantics for "intermediate" output is supported, meaning that all output will be persisted indefinitely. The upside of that is that if you try to run X -> Y, and Y crashes, you can resume with the previously built X. The downside is that you will have a lot of intermediate results on your file system. A useful pattern is to put these files in a special directory and have some kind of periodical garbage collection clean it up.

8.8.2 Triggering Many Tasks

A convenient pattern is to have a dummy Task at the end of several dependency chains, so you can trigger a multitude of pipelines by specifying just one task in command line, similarly to how e.g. make works.

```
class AllReports(luigi.WrapperTask):
    date = luigi.DateParameter(default=datetime.date.today())
    def requires(self):
        yield SomeReport(self.date)
        yield SomeOtherReport(self.date)
        yield CropReport(self.date)
        yield TPSReport(self.date)
        yield FooBarBazReport(self.date)
```

This simple task will not do anything itself, but will invoke a bunch of other tasks. Per each invocation, Luigi will perform as many of the pending jobs as possible (those which have all their dependencies present).

You'll need to use `WrapperTask` for this instead of the usual Task class, because this job will not produce any output of its own, and as such needs a way to indicate when it's complete. This class is used for tasks that only wrap other tasks and that by definition are done if all their requirements exist.

8.8.3 Triggering recurring tasks

A common requirement is to have a daily report (or something else) produced every night. Sometimes for various reasons tasks will keep crashing or lacking their required dependencies for more than a day though, which would lead to a missing deliverable for some date. Oops.

To ensure that the above AllReports task is eventually completed for every day (value of date parameter), one could e.g. add a loop in requires method to yield dependencies on the past few days preceding self.date. Then, so long as Luigi keeps being invoked, the backlog of jobs would catch up nicely after fixing intermittent problems.

Luigi actually comes with a reusable tool for achieving this, called `RangeDailyBase` (resp. `RangeHourlyBase`). Simply putting

```
luigi --module all_reports RangeDailyBase --of AllReports --start 2015-01-01
```

in your crontab will easily keep gaps from occurring from 2015-01-01 onwards. NB - it will not always loop over everything from 2015-01-01 till current time though, but rather a maximum of 3 months ago by default - see `RangeDailyBase` documentation for this and more knobs for tweaking behavior. See also Monitoring below.

8.8.4 Efficiently triggering recurring tasks

RangeDailyBase, described above, is named like that because a more efficient subclass exists, `RangeDaily` (resp. `RangeHourly`), tailored for hundreds of task classes scheduled concurrently with contiguousness requirements spanning years (which would incur redundant completeness checks and scheduler overload using the naive looping approach.) Usage:

```
luigi --module all_reports RangeDaily --of AllReports --start 2015-01-01
```

It has the same knobs as RangeDailyBase, with some added requirements. Namely the task must implement an efficient bulk_complete method, or must be writing output to file system Target with date parameter value consistently represented in the file path.

8.8.5 Backfilling tasks

Also a common use case, sometimes you have tweaked existing recurring task code and you want to schedule recomputation of it over an interval of dates for that or another reason. Most conveniently it is achieved with the above described range tools, just with both start (inclusive) and stop (exclusive) parameters specified:

```
luigi --module all_reports RangeDaily --of AllReportsV2 --start 2014-10-31 --stop 2014-
↪12-25
```

8.8.6 Propagating parameters with Range

Some tasks you want to recur may include additional parameters which need to be configured. The Range classes provide a parameter which accepts a `DictParameter` and passes any parameters onwards for this purpose.

```
luigi RangeDaily --of MyTask --start 2014-10-31 --of-params '{"my_string_param": "123",
↪"my_int_param": 123}'
```

Alternatively, you can specify parameters at the task family level (as described *here*), however these will not appear in the task name for the upstream Range task which can have implications in how the scheduler and visualizer handle task instances.

```
luigi RangeDaily --of MyTask --start 2014-10-31 --MyTask-my-param 123
```

8.8.7 Batching multiple parameter values into a single run

Sometimes it'll be faster to run multiple jobs together as a single batch rather than running them each individually. When this is the case, you can mark some parameters with a batch_method in their constructor to tell the worker how to combine multiple values. One common way to do this is by simply running the maximum value. This is good for tasks that overwrite older data when a newer one runs. You accomplish this by setting the batch_method to max, like so:

```
class A(luigi.Task):
    date = luigi.DateParameter(batch_method=max)
```

What's exciting about this is that if you send multiple As to the scheduler, it can combine them and return one. So if `A(date=2016-07-28)`, `A(date=2016-07-29)` and `A(date=2016-07-30)` are all ready to run, you will start running `A(date=2016-07-30)`. While this is running, the scheduler will show `A(date=2016-07-28)`, `A(date=2016-07-29)` as batch running while `A(date=2016-07-30)` is running. When `A(date=2016-07-30)` is done running and becomes FAILED or DONE, the other two tasks will be updated to the same status.

If you want to limit how big a batch can get, simply set max_batch_size. So if you have

```
class A(luigi.Task):
    date = luigi.DateParameter(batch_method=max)

    max_batch_size = 10
```

then the scheduler will batch at most 10 jobs together. You probably do not want to do this with the max batch method, but it can be helpful if you use other methods. You can use any method that takes a list of parameter values and returns a single parameter value.

If you have two max batch parameters, you'll get the max values for both of them. If you have parameters that don't have a batch method, they'll be aggregated separately. So if you have a class like

```
class A(luigi.Task):
    p1 = luigi.IntParameter(batch_method=max)
    p2 = luigi.IntParameter(batch_method=max)
    p3 = luigi.IntParameter()
```

and you create tasks `A(p1=1, p2=2, p3=0)`, `A(p1=2, p2=3, p3=0)`, `A(p1=3, p2=4, p3=1)`, you'll get them batched as `A(p1=2, p2=3, p3=0)` and `A(p1=3, p2=4, p3=1)`.

Note that batched tasks do not take up *[resources]*, only the task that ends up running will use resources. The scheduler only checks that there are sufficient resources for each task individually before batching them all together.

8.8.8 Tasks that regularly overwrite the same data source

If you are overwriting of the same data source with every run, you'll need to ensure that two batches can't run at the same time. You can do this pretty easily by setting batch_method to max and setting a unique resource:

```
class A(luigi.Task):
    date = luigi.DateParameter(batch_method=max)

    resources = {'overwrite_resource': 1}
```

Now if you have multiple tasks such as `A(date=2016-06-01)`, `A(date=2016-06-02)`, `A(date=2016-06-03)`, the scheduler will just tell you to run the highest available one and mark the lower ones as batch_running. Using a unique resource will prevent multiple tasks from writing to the same location at the same time if a new one becomes available while others are running.

8.8.9 Avoiding concurrent writes to a single file

Updating a single file from several tasks is almost always a bad idea, and you need to be very confident that no other good solution exists before doing this. If, however, you have no other option, then you will probably at least need to ensure that no two tasks try to write to the file _simultaneously_.

By turning 'resources' into a Python property, it can return a value dependent on the task parameters or other dynamic attributes:

```
class A(luigi.Task):
    ...

    @property
    def resources(self):
        return { self.important_file_name: 1 }
```

Since, by default, resources have a usage limit of 1, no two instances of Task A will now run if they have the same *important_file_name* property.

8.8.10 Decreasing resources of running tasks

At scheduling time, the luigi scheduler needs to be aware of the maximum resource consumption a task might have once it runs. For some tasks, however, it can be beneficial to decrease the amount of consumed resources between two steps within their run method (e.g. after some heavy computation). In this case, a different task waiting for that particular resource can already be scheduled.

```
class A(luigi.Task):

    # set maximum resources a priori
    resources = {"some_resource": 3}

    def run(self):
        # do something
        ...

        # decrease consumption of "some_resource" by one
        self.decrease_running_resources({"some_resource": 1})
```

(continues on next page)

(continued from previous page)

```
        # continue with reduced resources
        ...
```

8.8.11 Monitoring task pipelines

Luigi comes with some existing ways in `luigi.notifications` to receive notifications whenever tasks crash. Email is the most common way.

The above mentioned range tools for recurring tasks not only implement reliable scheduling for you, but also emit events which you can use to set up delay monitoring. That way you can implement alerts for when jobs are stuck for prolonged periods lacking input data or otherwise requiring attention.

8.8.12 Atomic Writes Problem

A very common mistake done by luigi plumbers is to write data partially to the final destination, that is, not atomically. The problem arises because completion checks in luigi are exactly as naive as running `luigi.target.Target.exists()`. And in many cases it just means to check if a folder exist on disk. During the time we have partially written data, a task depending on that output would think its input is complete. This can have devestating effects, as in the thanksgiving bug.

The concept can be illustrated by imagining that we deal with data stored on local disk and by running commands:

```
# This the BAD way
$ mkdir /outputs/final_output
$ big-slow-calculation > /outputs/final_output/foo.data
```

As stated earlier, the problem is that only partial data exists for a duration, yet we consider the data to be `complete()` because the output folder already exists. Here is a robust version of this:

```
# This is the good way
$ mkdir /outputs/final_output-tmp-123456
$ big-slow-calculation > /outputs/final_output-tmp-123456/foo.data
$ mv --no-target-directory --no-clobber /outputs/final_output{-tmp-123456,}
$ [[ -d /outputs/final_output-tmp-123456 ]] && rm -r /outputs/final_output-tmp-123456
```

Indeed, the good way is not as trivial. It involves coming up with a unique directory name and a pretty complex `mv` line, the reason `mv` need all those is because we don't want `mv` to move a directory into a potentially existing directory. A directory could already exist in exceptional cases, for example when central locking fails and the same task would somehow run twice at the same time. Lastly, in the exceptional case where the file was never moved, one might want to remove the temporary directory that never got used.

Note that this was an example where the storage was on local disk. But for every storage (hard disk file, hdfs file, database table, etc.) this procedure will look different. But do every luigi user need to implement that complexity? Nope, thankfully luigi developers are aware of these and luigi comes with many built-in solutions. In the case of you're dealing with a file system (`FileSystemTarget`), you should consider using `temporary_path()`. For other targets, you should ensure that the way you're writing your final output directory is atomic.

8.8.13 Sending messages to tasks

The central scheduler is able to send messages to particular tasks. When a running task accepts messages, it can access a multiprocessing.Queue object storing incoming messages. You can implement custom behavior to react and respond to messages:

```
class Example(luigi.Task):

    # common task setup
    ...

    # configure the task to accept all incoming messages
    accepts_messages = True

    def run(self):
        # this example runs some loop and listens for the
        # "terminate" message, and responds to all other messages
        for _ in some_loop():
            # check incomming messages
            if not self.scheduler_messages.empty():
                msg = self.scheduler_messages.get()
                if msg.content == "terminate":
                    break
                else:
                    msg.respond("unknown message")

        # finalize
        ...
```

Messages can be sent right from the scheduler UI which also displays responses (if any). Note that this feature is only available when the scheduler is configured to send messages (see the *[scheduler]* config), and the task is configured to accept them.

8.9 Configuration

All configuration can be done by adding configuration files.

Supported config parsers:

- `cfg` (default), based on Python's standard ConfigParser. Values may refer to environment variables using `${ENVVAR}` syntax.
- `toml`

You can choose right parser via `LUIGI_CONFIG_PARSER` environment variable. For example, `LUIGI_CONFIG_PARSER=toml`.

Default (cfg) parser are looked for in:

- `/etc/luigi/client.cfg` (deprecated)
- `/etc/luigi/luigi.cfg`
- `client.cfg` (deprecated)
- `luigi.cfg`

- `LUIGI_CONFIG_PATH` environment variable

TOML parser are looked for in:

- `/etc/luigi/luigi.toml`
- `luigi.toml`
- `LUIGI_CONFIG_PATH` environment variable

Both config lists increase in priority (from low to high). The order only matters in case of key conflicts (see docs for ConfigParser.read). These files are meant for both the client and `luigid`. If you decide to specify your own configuration you should make sure that both the client and `luigid` load it properly.

The config file is broken into sections, each controlling a different part of the config.

Example cfg config:

```
[hadoop]
version=cdh4
streaming_jar=/usr/lib/hadoop-xyz/hadoop-streaming-xyz-123.jar

[core]
scheduler_host=luigi-host.mycompany.foo
```

Example toml config:

```
[hadoop]
version = "cdh4"
streaming_jar = "/usr/lib/hadoop-xyz/hadoop-streaming-xyz-123.jar"

[core]
scheduler_host = "luigi-host.mycompany.foo"
```

Also see examples/config.toml for more complex example.

8.9.1 Parameters from config Ingestion

All parameters can be overridden from configuration files. For instance if you have a Task definition:

```
class DailyReport(luigi.contrib.hadoop.JobTask):
    date = luigi.DateParameter(default=datetime.date.today())
    # ...
```

Then you can override the default value for `DailyReport().date` by providing it in the configuration:

```
[DailyReport]
date=2012-01-01
```

Configuration classes

Using the *Parameters from config Ingestion* method, we derive the conventional way to do global configuration. Imagine this configuration.

```
[mysection]
option=hello
intoption=123
```

We can create a `Config` class:

```
import luigi

# Config classes should be camel cased
class mysection(luigi.Config):
    option = luigi.Parameter(default='world')
    intoption = luigi.IntParameter(default=555)

mysection().option
mysection().intoption
```

8.9.2 Configurable options

Luigi comes with a lot of configurable options. Below, we describe each section and the parameters available within it.

8.9.3 [core]

These parameters control core Luigi behavior, such as error e-mails and interactions between the worker and scheduler.

autoload_range
Added in version 2.8.11.

If false, prevents range tasks from autoloading. They can still be loaded using `--module luigi.tools.range`. Defaults to true. Setting this to true explicitly disables the deprecation warning.

default_scheduler_host
Hostname of the machine running the scheduler. Defaults to localhost.

default_scheduler_port
Port of the remote scheduler api process. Defaults to 8082.

default_scheduler_url
Full path to remote scheduler. Defaults to `http://localhost:8082/`. For TLS support use the URL scheme: `https`, example: `https://luigi.example.com:443/` (Note: you will have to terminate TLS using an HTTP proxy) You can also use this to connect to a local Unix socket using the non-standard URI scheme: `http+unix` example: `http+unix://%2Fvar%2Frun%2Fluigid%2Fluigid.sock/`

hdfs_tmp_dir
Base directory in which to store temporary files on hdfs. Defaults to tempfile.gettempdir()

history_filename
If set, specifies a filename for Luigi to write stuff (currently just job id) to in mapreduce job's output directory. Useful in a configuration where no history is stored in the output directory by Hadoop.

log_level
The default log level to use when no logging_conf_file is set. Must be a valid name of a Python log level. Default is `DEBUG`.

logging_conf_file
Location of the logging configuration file.

no_configure_logging
If true, logging is not configured. Defaults to false.

parallel_scheduling
If true, the scheduler will compute complete functions of tasks in parallel using multiprocessing. This can significantly speed up scheduling, but requires that all tasks can be pickled. Defaults to false.

parallel_scheduling_processes
The number of processes to use for parallel scheduling. If not specified the default number of processes will be the total number of CPUs available.

rpc_connect_timeout
Number of seconds to wait before timing out when making an API call. Defaults to 10.0

rpc_retry_attempts
The maximum number of retries to connect the central scheduler before giving up. Defaults to 3

rpc_retry_wait
Number of seconds to wait before the next attempt will be started to connect to the central scheduler between two retry attempts. Defaults to 30

8.9.4 [cors]

Added in version 2.8.0.

These parameters control `/api/<method>` `CORS` behaviour (see: W3C Cross-Origin Resource Sharing).

enabled
Enables CORS support. Defaults to false.

allowed_origins
A list of allowed origins. Used only if `allow_any_origin` is false. Configure in JSON array format, e.g. [“foo”, “bar”]. Defaults to empty.

allow_any_origin
Accepts requests from any origin. Defaults to false.

allow_null_origin
Allows the request to set `null` value of the `Origin` header. Defaults to false.

max_age
Content of `Access-Control-Max-Age`. Defaults to 86400 (24 hours).

allowed_methods
Content of `Access-Control-Allow-Methods`. Defaults to `GET, OPTIONS`.

allowed_headers
Content of `Access-Control-Allow-Headers`. Defaults to `Accept, Content-Type, Origin`.

exposed_headers
Content of `Access-Control-Expose-Headers`. Defaults to empty string (will NOT be sent as a response header).

allow_credentials
Indicates that the actual request can include user credentials. Defaults to false.

8.9.5 [worker]

These parameters control Luigi worker behavior.

count_uniques
If true, workers will only count unique pending jobs when deciding whether to stay alive. So if a worker can't get a job to run and other workers are waiting on all of its pending jobs, the worker will die. `worker_keep_alive` must be `true` for this to have any effect. Defaults to false.

keep_alive
If true, workers will stay alive when they run out of jobs to run, as long as they have some pending job waiting to be run. Defaults to false.

ping_interval
Number of seconds to wait between pinging scheduler to let it know that the worker is still alive. Defaults to 1.0.

task_limit
Added in version 1.0.25.

Maximum number of tasks to schedule per invocation. Upon exceeding it, the worker will issue a warning and proceed with the workflow obtained thus far. Prevents incidents due to spamming of the scheduler, usually accidental. Default: no limit.

timeout
Added in version 1.0.20.

Number of seconds after which to kill a task which has been running for too long. This provides a default value for all tasks, which can be overridden by setting the `worker_timeout` property in any task. Default value is 0, meaning no timeout.

wait_interval
Number of seconds for the worker to wait before asking the scheduler for another job after the scheduler has said that it does not have any available jobs.

wait_jitter
Duration of jitter to add to the worker wait interval such that the multiple workers do not ask the scheduler for another job at the same time, in seconds. Default: 5.0

max_keep_alive_idle_duration
Added in version 2.8.4.

Maximum duration in seconds to keep worker alive while in idle state. Default: 0 (Indefinitely)

max_reschedules
The maximum number of times that a job can be automatically rescheduled by a worker before it will stop trying. Workers will reschedule a job if it is found to not be done when attempting to run a dependent job. This defaults to 1.

retry_external_tasks
If true, incomplete external tasks (i.e. tasks where the `run()` method is NotImplemented) will be retested for completion while Luigi is running. This means that if external dependencies are satisfied after a workflow has started, any tasks dependent on that resource will be eligible for running. Note: Every time the task remains incomplete, it will count as FAILED, so normal retry logic applies (see: `retry_count` and `retry_delay`). This setting works best with `worker_keep_alive: true`. If false, external tasks will only be evaluated when Luigi is first invoked. In this case, Luigi will not check whether external dependencies are satisfied while a workflow is in progress, so dependent tasks will remain PENDING until the workflow is reinvoked. Defaults to false for backwards compatibility.

no_install_shutdown_handler
By default, workers will stop requesting new work and finish running pending tasks after receiving a `SIGUSR1` signal. This provides a hook for gracefully shutting down workers that are in the process of running (potentially

expensive) tasks. If set to true, Luigi will NOT install this shutdown hook on workers. Note this hook does not work on Windows operating systems, or when jobs are launched outside the main execution thread. Defaults to false.

send_failure_email
Controls whether the worker will send e-mails on task and scheduling failures. If set to false, workers will only send e-mails on framework errors during scheduling and all other e-mail must be handled by the scheduler. Defaults to true.

check_unfulfilled_deps
If true, the worker checks for completeness of dependencies before running a task. In case unfulfilled dependencies are detected, an exception is raised and the task will not run. This mechanism is useful to detect situations where tasks do not create their outputs properly, or when targets were removed after the dependency tree was built. It is recommended to disable this feature only when the completeness checks are known to be bottlenecks, e.g. when the `exists()` calls of the dependencies' outputs are resource-intensive. Defaults to true.

force_multiprocessing
By default, luigi uses multiprocessing when *more than one* worker process is requested. When set to true, multiprocessing is used independent of the number of workers. Defaults to false.

check_complete_on_run
By default, luigi tasks are marked as 'done' when they finish running without raising an error. When set to true, tasks will also verify that their outputs exist when they finish running, and will fail immediately if the outputs are missing. Defaults to false.

cache_task_completion
By default, luigi task processes might check the completion status multiple times per task which is a safe way to avoid potential inconsistencies. For tasks with many dynamic dependencies, yielded in multiple stages, this might become expensive, e.g. in case the per-task completion check entails remote resources. When set to true, completion checks are cached so that tasks declared as complete once are not checked again. Defaults to false.

8.9.6 [elasticsearch]

These parameters control use of elasticsearch

marker_index
Defaults to "update_log".

marker_doc_type
Defaults to "entry".

8.9.7 [email]

General parameters

force_send
If true, e-mails are sent in all run configurations (even if stdout is connected to a tty device). Defaults to False.

format
Type of e-mail to send. Valid values are "plain", "html" and "none". When set to html, tracebacks are wrapped in <pre> tags to get fixed- width font. When set to none, no e-mails will be sent.

Default value is plain.

method
Valid values are "smtp", "sendgrid", "ses" and "sns". SES and SNS are services of Amazon web services. SendGrid is an email delivery service. The default value is "smtp".

In order to send messages through Amazon SNS or SES set up your AWS config files or run Luigi on an EC2 instance with proper instance profile.

In order to use sendgrid, fill in your sendgrid API key in the *[sendgrid]* section.

In order to use smtp, fill in the appropriate fields in the *[smtp]* section.

prefix
Optional prefix to add to the subject line of all e-mails. For example, setting this to "[LUIGI]" would change the subject line of an e-mail from "Luigi: Framework error" to "[LUIGI] Luigi: Framework error"

receiver
Recipient of all error e-mails. If this is not set, no error e-mails are sent when Luigi crashes unless the crashed job has owners set. If Luigi is run from the command line, no e-mails will be sent unless output is redirected to a file.

Set it to SNS Topic ARN if you want to receive notifications through Amazon SNS. Make sure to set method to sns in this case too.

sender
User name in from field of error e-mails. Default value: luigi-client@<server_name>

traceback_max_length
Maximum length for traceback included in error email. Default is 5000.

8.9.8 [batch_notifier]

Parameters controlling the contents of batch notifications sent from the scheduler

email_interval_minutes
Number of minutes between e-mail sends. Making this larger results in fewer, bigger e-mails. Defaults to 60.

batch_mode
Controls how tasks are grouped together in the e-mail. Suppose we have the following sequence of failures:

1. TaskA(a=1, b=1)
2. TaskA(a=1, b=1)
3. TaskA(a=2, b=1)
4. TaskA(a=1, b=2)
5. TaskB(a=1, b=1)

For any setting of batch_mode, the batch e-mail will record 5 failures and mention them in the subject. The difference is in how they will be displayed in the body. Here are example bodies with error_messages set to 0.

"all" only groups together failures for the exact same task:

- TaskA(a=1, b=1) (2 failures)
- TaskA(a=1, b=2) (1 failure)
- TaskA(a=2, b=1) (1 failure)
- TaskB(a=1, b=1) (1 failure)

"family" groups together failures for tasks of the same family:

- TaskA (4 failures)
- TaskB (1 failure)

"unbatched_params" groups together tasks that look the same after removing batched parameters. So if TaskA has a batch_method set for parameter a, we get the following:

- TaskA(b=1) (3 failures)
- TaskA(b=2) (1 failure)
- TaskB(a=1, b=2) (1 failure)

Defaults to "unbatched_params", which is identical to "all" if you are not using batched parameters.

error_lines
Number of lines to include from each error message in the batch e-mail. This can be used to keep e-mails shorter while preserving the more useful information usually found near the bottom of stack traces. This can be set to 0 to include all lines. If you don't wish to see error messages, instead set `error_messages` to 0. Defaults to 20.

error_messages
Number of messages to preserve for each task group. As most tasks that fail repeatedly do so for similar reasons each time, it's not usually necessary to keep every message. This controls how many messages are kept for each task or task group. The most recent error messages are kept. Set to 0 to not include error messages in the e-mails. Defaults to 1.

group_by_error_messages
Quite often, a system or cluster failure will cause many disparate task types to fail for the same reason. This can cause a lot of noise in the batch e-mails. This cuts down on the noise by listing items with identical error messages together. Error messages are compared after limiting by `error_lines`. Defaults to true.

8.9.9 [hadoop]

Parameters controlling basic hadoop tasks

command
Name of command for running hadoop from the command line. Defaults to "hadoop"

python_executable
Name of command for running python from the command line. Defaults to "python"

scheduler
Type of scheduler to use when scheduling hadoop jobs. Can be "fair" or "capacity". Defaults to "fair".

streaming_jar
Path to your streaming jar. Must be specified to run streaming jobs.

version
Version of hadoop used in your cluster. Can be "cdh3", "chd4", or "apache1". Defaults to "cdh4".

8.9.10 [hdfs]

Parameters controlling the use of snakebite to speed up hdfs queries.

client
Client to use for most hadoop commands. Options are "snakebite", "snakebite_with_hadoopcli_fallback", "webhdfs" and "hadoopcli". Snakebite is much faster, so use of it is encouraged. webhdfs is fast and works with Python 3 as well, but has not been used that much in the wild. Both snakebite and webhdfs requires you to install it separately on the machine. Defaults to "hadoopcli".

client_version
Optionally specifies hadoop client version for snakebite.

effective_user
Optionally specifies the effective user for snakebite.

namenode_host
The hostname of the namenode. Needed for snakebite if snakebite_autoconfig is not set.

namenode_port
The port used by snakebite on the namenode. Needed for snakebite if snakebite_autoconfig is not set.

snakebite_autoconfig
If true, attempts to automatically detect the host and port of the namenode for snakebite queries. Defaults to false.

tmp_dir
Path to where Luigi will put temporary files on hdfs

8.9.11 [hive]

Parameters controlling hive tasks

command
Name of the command used to run hive on the command line. Defaults to "hive".

hiverc_location
Optional path to hive rc file.

metastore_host
Hostname for metastore.

metastore_port
Port for hive to connect to metastore host.

release
If set to "apache", uses a hive client that better handles apache hive output. All other values use the standard client Defaults to "cdh4".

8.9.12 [kubernetes]

Parameters controlling Kubernetes Job Tasks

auth_method
Authorization method to access the cluster. Options are "kubeconfig" or "service-account"

kubeconfig_path
Path to kubeconfig file, for cluster authentication. It defaults to `~/.kube/config`, which is the default location when using minikube. When auth_method is "service-account" this property is ignored.

max_retrials
Maximum number of retrials in case of job failure.

8.9.13 [mysql]

Parameters controlling use of MySQL targets

marker_table
Table in which to store status of table updates. This table will be created if it doesn't already exist. Defaults to "table_updates".

8.9.14 [postgres]

Parameters controlling the use of Postgres targets

local_tmp_dir
Directory in which to temporarily store data before writing to postgres. Uses system default if not specified.

marker_table
Table in which to store status of table updates. This table will be created if it doesn't already exist. Defaults to "table_updates".

8.9.15 [redshift]

Parameters controlling the use of Redshift targets

marker_table
Table in which to store status of table updates. This table will be created if it doesn't already exist. Defaults to "table_updates".

8.9.16 [resources]

This section can contain arbitrary keys. Each of these specifies the amount of a global resource that the scheduler can allow workers to use. The scheduler will prevent running jobs with resources specified from exceeding the counts in this section. Unspecified resources are assumed to have limit 1. Example resources section for a configuration with 2 hive resources and 1 mysql resource:

```
[resources]
hive=2
mysql=1
```

Note that it was not necessary to specify the 1 for mysql here, but it is good practice to do so when you have a fixed set of resources.

8.9.17 [retcode]

Configure return codes for the Luigi binary. In the case of multiple return codes that could apply, for example a failing task and missing data, the *numerically greatest* return code is returned.

We recommend that you copy this set of exit codes to your `luigi.cfg` file:

```
[retcode]
# The following return codes are the recommended exit codes for Luigi
# They are in increasing level of severity (for most applications)
already_running=10
missing_data=20
```

(continues on next page)

(continued from previous page)

```
not_run=25
task_failed=30
scheduling_error=35
unhandled_exception=40
```

already_running
: This can happen in two different cases. Either the local lock file was taken at the time the invocation starts up. Or, the central scheduler have reported that some tasks could not have been run, because other workers are already running the tasks.

missing_data
: For when an `ExternalTask` is not complete, and this caused the worker to give up. As an alternative to fiddling with this, see the [worker] keep_alive option.

not_run
: For when a task is not granted run permission by the scheduler. Typically because of lack of resources, because the task has been already run by another worker or because the attempted task is in DISABLED state. Connectivity issues with the central scheduler might also cause this. This does not include the cases for which a run is not allowed due to missing dependencies (missing_data) or due to the fact that another worker is currently running the task (already_running).

task_failed
: For signaling that there were last known to have failed. Typically because some exception have been raised.

scheduling_error
: For when a task's `complete()` or `requires()` method fails with an exception, or when the limit number of tasks is reached.

unhandled_exception
: For internal Luigi errors. Defaults to 4, since this type of error probably will not recover over time.

If you customize return codes, prefer to set them in range 128 to 255 to avoid conflicts. Return codes in range 0 to 127 are reserved for possible future use by Luigi contributors.

8.9.18 [scalding]

Parameters controlling running of scalding jobs

scala_home
: Home directory for scala on your machine. Defaults to either SCALA_HOME or /usr/share/scala if SCALA_HOME is unset.

scalding_home
: Home directory for scalding on your machine. Defaults to either SCALDING_HOME or /usr/share/scalding if SCALDING_HOME is unset.

scalding_provided
: Provided directory for scalding on your machine. Defaults to either SCALDING_HOME/provided or /usr/share/scalding/provided

scalding_libjars
: Libjars directory for scalding on your machine. Defaults to either SCALDING_HOME/libjars or /usr/share/scalding/libjars

8.9.19 [scheduler]

Parameters controlling scheduler behavior

batch_emails
Whether to send batch e-mails for failures and disables rather than sending immediate disable e-mails and just relying on workers to send immediate batch e-mails. Defaults to false.

disable_hard_timeout
Hard time limit after which tasks will be disabled by the server if they fail again, in seconds. It will disable the task if it fails **again** after this amount of time. E.g. if this was set to 600 (i.e. 10 minutes), and the task first failed at 10:00am, the task would be disabled if it failed again any time after 10:10am. Note: This setting does not consider the values of the `retry_count` or `disable_window` settings.

retry_count
Number of times a task can fail within `disable_window` before the scheduler will automatically disable it. If not set, the scheduler will not automatically disable jobs.

disable_persist
Number of seconds for which an automatic scheduler disable lasts. Defaults to 86400 (1 day).

disable_window
Number of seconds during which `retry_count` failures must occur in order for an automatic disable by the scheduler. The scheduler forgets about disables that have occurred longer ago than this amount of time. Defaults to 3600 (1 hour).

max_shown_tasks
Added in version 1.0.20.

The maximum number of tasks returned in a task_list api call. This will restrict the number of tasks shown in task lists in the visualiser. Small values can alleviate frozen browsers when there are too many done tasks. This defaults to 100000 (one hundred thousand).

max_graph_nodes
Added in version 2.0.0.

The maximum number of nodes returned by a dep_graph or inverse_dep_graph api call. Small values can greatly speed up graph display in the visualiser by limiting the number of nodes shown. Some of the nodes that are not sent to the visualiser will still show up as dependencies of nodes that were sent. These nodes are given TRUNCATED status.

record_task_history
If true, stores task history in a database. Defaults to false.

remove_delay
Number of seconds to wait before removing a task that has no stakeholders. Defaults to 600 (10 minutes).

retry_delay
Number of seconds to wait after a task failure to mark it pending again. Defaults to 900 (15 minutes).

state_path
Path in which to store the Luigi scheduler's state. When the scheduler is shut down, its state is stored in this path. The scheduler must be shut down cleanly for this to work, usually with a kill command. If the kill command includes the -9 flag, the scheduler will not be able to save its state. When the scheduler is started, it will load the state from this path if it exists. This will restore all scheduled jobs and other state from when the scheduler last shut down.

Sometimes this path must be deleted when restarting the scheduler after upgrading Luigi, as old state files can become incompatible with the new scheduler. When this happens, all workers should be restarted after the scheduler both to become compatible with the updated code and to reschedule the jobs that the scheduler has now forgotten about.

This defaults to /var/lib/luigi-server/state.pickle

worker_disconnect_delay
Number of seconds to wait after a worker has stopped pinging the scheduler before removing it and marking all of its running tasks as failed. Defaults to 60.

pause_enabled
If false, disables pause/unpause operations and hides the pause toggle from the visualiser.

send_messages
When true, the scheduler is allowed to send messages to running tasks and the central scheduler provides a simple prompt per task to send messages. Defaults to true.

metrics_collector
Optional setting allowing Luigi to use a contribution to collect metrics about the pipeline to a third-party. By default this uses the default metric collector that acts as a shell and does nothing. The currently available options are "datadog", "prometheus" and "custom". If it's custom the 'metrics_custom_import' needs to be set.

metrics_custom_import
Optional setting allowing Luigi to import a custom subclass of MetricsCollector at runtime. The string should be formatted like "module.sub_module.ClassName".

8.9.20 [sendgrid]

These parameters control sending error e-mails through SendGrid.

apikey
API key of the SendGrid account.

8.9.21 [smtp]

These parameters control the smtp server setup.

host
Hostname for sending mail through smtp. Defaults to localhost.

local_hostname
If specified, overrides the FQDN of localhost in the HELO/EHLO command.

no_tls
If true, connects to smtp without TLS. Defaults to false.

password
Password to log in to your smtp server. Must be specified for username to have an effect.

port
Port number for smtp on smtp_host. Defaults to 0.

ssl
If true, connects to smtp through SSL. Defaults to false.

timeout
Sets the number of seconds after which smtp attempts should time out. Defaults to 10.

username
Username to log in to your smtp server, if necessary.

8.9.22 [spark]

Parameters controlling the default execution of `SparkSubmitTask` and `PySparkTask`:

Deprecated since version 1.1.1: `SparkJob`, `Spark1xJob` and `PySpark1xJob` are deprecated. Please use `SparkSubmitTask` or `PySparkTask`.

spark_submit
Command to run in order to submit spark jobs. Default: `"spark-submit"`

master
Master url to use for `spark_submit`. Example: local[*], spark://masterhost:7077. Default: Spark default (Prior to 1.1.1: yarn-client)

deploy_mode
Whether to launch the driver programs locally ("client") or on one of the worker machines inside the cluster ("cluster"). Default: Spark default

jars
Comma-separated list of local jars to include on the driver and executor classpaths. Default: Spark default

packages
Comma-separated list of packages to link to on the driver and executors

py_files
Comma-separated list of .zip, .egg, or .py files to place on the `PYTHONPATH` for Python apps. Default: Spark default

files
Comma-separated list of files to be placed in the working directory of each executor. Default: Spark default

conf:
Arbitrary Spark configuration property in the form Prop=Value|Prop2=Value2. Default: Spark default

properties_file
Path to a file from which to load extra properties. Default: Spark default

driver_memory
Memory for driver (e.g. 1000M, 2G). Default: Spark default

driver_java_options
Extra Java options to pass to the driver. Default: Spark default

driver_library_path
Extra library path entries to pass to the driver. Default: Spark default

driver_class_path
Extra class path entries to pass to the driver. Default: Spark default

executor_memory
Memory per executor (e.g. 1000M, 2G). Default: Spark default

Configuration for Spark submit jobs on Spark standalone with cluster deploy mode only:

driver_cores
Cores for driver. Default: Spark default

supervise
If given, restarts the driver on failure. Default: Spark default

Configuration for Spark submit jobs on Spark standalone and Mesos only:

total_executor_cores
Total cores for all executors. Default: Spark default

Configuration for Spark submit jobs on YARN only:

executor_cores
Number of cores per executor. Default: Spark default

queue
The YARN queue to submit to. Default: Spark default

num_executors
Number of executors to launch. Default: Spark default

archives
Comma separated list of archives to be extracted into the working directory of each executor. Default: Spark default

hadoop_conf_dir
Location of the hadoop conf dir. Sets HADOOP_CONF_DIR environment variable when running spark. Example: /etc/hadoop/conf

Extra configuration for PySparkTask jobs:

py_packages
Comma-separated list of local packages (in your python path) to be distributed to the cluster.

Parameters controlling the execution of SparkJob jobs (deprecated):

8.9.23 [task_history]

Parameters controlling storage of task history in a database

db_connection
Connection string for connecting to the task history db using sqlalchemy.

8.9.24 [execution_summary]

Parameters controlling execution summary of a worker

summary_length
Maximum number of tasks to show in an execution summary. If the value is 0, then all tasks will be displayed. Default value is 5.

8.9.25 [webhdfs]

port
The port to use for webhdfs. The normal namenode port is probably on a different port from this one.

user
Perform file system operations as the specified user instead of $USER. Since this parameter is not honored by any of the other hdfs clients, you should think twice before setting this parameter.

client_type
The type of client to use. Default is the "insecure" client that requires no authentication. The other option is the "kerberos" client that uses kerberos authentication.

8.9.26 [datadog]

api_key
The api key found in the account settings of Datadog under the API sections.

app_key
The application key found in the account settings of Datadog under the API sections.

default_tags
Optional settings that adds the tag to all the metrics and events sent to Datadog. Default value is "application:luigi".

environment
Allows you to tweak multiple environment to differentiate between production, staging or development metrics within Datadog. Default value is "development".

statsd_host
The host that has the statsd instance to allow Datadog to send statsd metric. Default value is "localhost".

statsd_port
The port on the host that allows connection to the statsd host. Defaults value is 8125.

metric_namespace
Optional prefix to add to the beginning of every metric sent to Datadog. Default value is "luigi".

8.9.27 Per Task Retry-Policy

Luigi also supports defining `retry_policy` per task.

```
class GenerateWordsFromHdfs(luigi.Task):

   retry_count = 2

    ...

class GenerateWordsFromRDBM(luigi.Task):

   retry_count = 5

    ...

class CountLetters(luigi.Task):

    def requires(self):
        return [GenerateWordsFromHdfs()]

    def run():
        yield GenerateWordsFromRDBM()

    ...
```

If none of retry-policy fields is defined per task, the field value will be **default** value which is defined in luigi config file.

To make luigi sticks to the given retry-policy, be sure you run luigi worker with `keep_alive` config. Please check `keep_alive` config in *[worker]* section.

8.9.28 Retry-Policy Fields

The fields below are in retry-policy and they can be defined per task.

- `retry_count`
- `disable_hard_timeout`
- `disable_window`

8.10 Configure logging

8.10.1 Config options:

Some config options for config [core] section

log_level
The default log level to use when no logging_conf_file is set. Must be a valid name of a Python log level. Default is `DEBUG`.

logging_conf_file
Location of the logging configuration file.

no_configure_logging
If true, logging is not configured. Defaults to false.

8.10.2 Config section

If you're use TOML for configuration file, you can configure logging via `logging` section in this file. See example for more details.

8.10.3 Luigid CLI options:

`--background`
Run daemon in background mode. Disable logging setup and set up log level to INFO for root logger.

`--logdir`
set logging with INFO level and output in `$logdir/luigi-server.log` file

8.10.4 Worker CLI options:

`--logging-conf-file`
Configuration file for logging.

`--log-level`
Default log level. Available values: NOTSET, DEBUG, INFO, WARNING, ERROR, CRITICAL. Default DEBUG. See Python documentation For information about levels difference.

8.10.5 Configuration options resolution order:

1. no_configure_logging option
2. `--background`
3. `--logdir`
4. `--logging-conf-file`
5. logging_conf_file option
6. `logging` section
7. `--log-level`
8. log_level option

8.11 Design and limitations

Luigi is the successor to a couple of attempts that we weren't fully happy with. We learned a lot from our mistakes and some design decisions include:

- Straightforward command-line integration.
- As little boilerplate as possible.
- Focus on job scheduling and dependency resolution, not a particular platform. In particular, this means no limitation to Hadoop. Though Hadoop/HDFS support is built-in and is easy to use, this is just one of many types of things you can run.
- A file system abstraction where code doesn't have to care about where files are located.
- Atomic file system operations through this abstraction. If a task crashes it won't lead to a broken state.
- The dependencies are decentralized. No big config file in XML. Each task just specifies which inputs it needs and cross-module dependencies are trivial.
- A web server that renders the dependency graph and does locking, etc for free.
- Trivial to extend with new file systems, file formats, and job types. You can easily write jobs that inserts a Tokyo Cabinet into Cassandra. Adding support for new systems is generally not very hard. (Feel free to send us a patch when you're done!)
- Date algebra included.
- Lots of unit tests of the most basic stuff.

It wouldn't be fair not to mention some limitations with the current design:

- Its focus is on batch processing so it's probably less useful for near real-time pipelines or continuously running processes.
- The assumption is that each task is a sizable chunk of work. While you can probably schedule a few thousand jobs, it's not meant to scale beyond tens of thousands.
- Luigi does not support distribution of execution. When you have workers running thousands of jobs daily, this starts to matter, because the worker nodes get overloaded. There are some ways to mitigate this (trigger from many nodes, use resources), but none of them are ideal.
- Luigi does not come with built-in triggering, and you still need to rely on something like crontab to trigger workflows periodically.

Also, it should be mentioned that Luigi is named after the world's second most famous plumber.

CHAPTER

NINE

API REFERENCE

luigi	Package containing core luigi functionality.
luigi.contrib	Package containing optional and-on functionality.
luigi.tools	Sort of a standard library for doing stuff with Tasks at a somewhat abstract level.
luigi.local_target	`LocalTarget` provides a concrete implementation of a `Target` class that uses files on the local file system

9.1 luigi

Package containing core luigi functionality.

9.2 luigi.contrib

Package containing optional and-on functionality.

9.3 luigi.tools

Sort of a standard library for doing stuff with Tasks at a somewhat abstract level.

Submodule introduced to stop growing util.py unstructured.

9.4 luigi.local_target

`LocalTarget` provides a concrete implementation of a `Target` class that uses files on the local file system

Classes

`LocalFileSystem`()	Wrapper for access to file system operations.
`LocalTarget`([path, format, is_tmp])	Initializes a FileSystemTarget instance.
`atomic_file`(path)	Simple class that writes to a temp file and moves it on close() Also cleans up the temp file if close is not invoked

9.5 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

L

M