
b2luigi Documentation

Release 0.1.2

Nils Braun

Jul 08, 2018

Contents

1	Content	3
1.1	Installation	3
1.2	Quick Start	3
1.3	Submitting to the Batch System	7
1.4	Advanced Examples	8
1.5	FAQ	8
1.6	TODO items	9
1.7	Development	9

b2luigi is a helper package constructed around luigi, that helps you schedule working packages (so called tasks) locally or on the batch system. Apart from the very powerful dependency management system by luigi, b2luigi extends the user interface and has a build-in support for the queue systems, e.g. LSF.

You can find more information in the [Quick Start](#).

Please note, that most of the core features are handled by luigi, so you might want to have a look into the [luigi documentation](#).

If you find any bugs or want to improve the documentation, please send me a merge request on [github](#).

This project is in beta. Please be extra cautious when using in production mode. You can help me by working with one of the todo items described in [TODO items](#).

1.1 Installation

1. Setup your local environment. If you are using basf2: You can use a local environment (installed on your machine) or a release on cvmfs. For example, run:

```
source /cvmfs/belle.cern.ch/tools/b2setup prerelease-02-00-00c
```

Or you setup the virtual environment of your project:

```
source venv/bin/activate
```

2. Install b2luigi from pipy into your environment.

- (a) If you have a local installation, you can use the normal setup command

```
pip3 install b2luigi
```

- (b) If this fails because you do not have write access to where your virtual environment lives, you can also install b2luigi locally:

```
pip3 install --user b2luigi
```

This will automatically also install *luigi* into your current environment. Please make sure to always setup your environment (e.g. basf2) correctly before using *b2luigi*.

Now you can go on with the [Quick Start](#).

1.2 Quick Start

After having installed b2luigi (see [Installation](#)), we dive directly into the first example. The example shown on this page does not require any knowledge of *luigi*. Nevertheless, it can be a very good idea to have a look into the [luigi](#)

documentation while working with b2luigi¹.

Our first very simple project will consist of two different workloads:

- the first workload will output a file with a random number in it.
- the second one will collect the output of the first workload and calculate the average.

We will repeat the first task 10 times, so the average will be built from 10 random numbers.

```
1 import b2luigi
2 import random
3
4
5 class RandomNumberTask(b2luigi.Task):
6     # The parameter of the task
7     number = b2luigi.IntParameter()
8
9     def output(self):
10         """Define what the output of this task will be"""
11         yield self.add_to_output("number.txt")
12
13     def run(self):
14         """The real work of the task: Generate a random number and store it"""
15         output_file_name = self.get_output_file_names()["number.txt"]
16
17         with open(output_file_name, "w") as f:
18             random_number = random.random()
19             f.write(f"{random_number}\n")
20
21
22 class AveragerTask(b2luigi.Task):
23     def requires(self):
24         """Define the dependencies of this task"""
25         for i in range(10):
26             yield RandomNumberTask(number=i)
27
28     def output(self):
29         """Again, we will have an output file"""
30         yield self.add_to_output("average.txt")
31
32     def run(self):
33         random_numbers = []
34
35         for input_file_name in self.get_input_file_names()["number.txt"]:
36             with open(input_file_name, "r") as f:
37                 random_numbers.append(float(f.read()))
38
39         mean = sum(random_numbers) / len(random_numbers)
40
41         output_file_name = self.get_output_file_names()["average.txt"]
42
43         with open(output_file_name, "w") as f:
44             f.write(f"{mean}\n")
45
46
```

(continues on next page)

¹ b2luigi defines a super set of the luigi features, so if you already worked with luigi, you can feel comfortable and just use the luigi features. But there is more to discover!

(continued from previous page)

```

47 if __name__ == "__main__":
48     b2luigi.process(AveragerTask(), workers=100)

```

After the import in line 1, two Tasks are defined in lines 5-19 and 22-44.

The RandomNumberTask has a single parameter of type `int` defined in line 7. Its job is, to create a new output file called `number.txt` and write the line random number into it. Every Task needs to define the output files it writes (and every task needs to have at least a single output file).

If your task does not write out output files, you need to inherit from `b2luigi.WrapperTask` instead of `b2luigi.Task`.

Please note, how the output files are defined here using the method `self.add_to_output` and the keyword `yield`. If you want to return more than one output, you can `yield` more often:

```

def output(self):
    yield self.add_to_output("first_output.txt")
    yield self.add_to_output("second_output.txt")
    yield self.add_to_output("third_output.txt")

```

You can access the defined output files in your run method by calling `self.get_output_file_names()`. This will return a dict where each name you have given in the output method is one key. The value is the final file name.

Why is `number.txt` not the final file name? If this would be the case, all RandomNumberTasks would override the outputs of each other, so you would need to include the task parameter into the output file. This cumbersome work is already done for you! The output files will be written into a location defined as

```
./git_hash=<git_hash>/param_1=<param_1>/.../<filename>
```

The AveragerTask defined in lines 22-44 requires the RandomNumberTask to run before it will start (more precise: the ones with the numbers 0 to 9). It has no parameters.

In its run function, all output files of the RandomNumberTask are opened. This is done by using the function `self.get_input_file_names()` which works analogous to the `self.get_output_file_names()` function, except that it will return a dict of lists instead of single file names.

The last line 47 and 48 define what our root task is (AveragerTask) and how many tasks we want to run in parallel at maximum (100 in this case, but we will not need that much).

We can now save this file as `simple-task.py` and start processing it:

```
python3 simple-task.py
```

This will run the tasks locally. It will start the 10 RandomNumberTask first and wait until all of them are finished. It will then start the AveragerTask afterwards. The process will finish with a small summary:

```

===== Luigi Execution Summary =====

Scheduled 11 tasks of which:
* 11 ran successfully:
  - 1 AveragerTask(git_hash=5507aad90c07101f53cb88a2f28dfa74029c499d)
  - 10 RandomNumberTask(git_hash=5507aad90c07101f53cb88a2f28dfa74029c499d, number=0.
↳..9)

This progress looks :) because there were no failed tasks or missing external_
↳dependencies

===== Luigi Execution Summary =====

```

The nice thing of luigi is, that it is idempotent. If you call the script again

```
python3 simple-task.py
```

it will not do anything and just state:

```
===== Luigi Execution Summary =====

Scheduled 1 tasks of which:
* 1 present dependencies were encountered:
  - 1 AveragerTask(git_hash=5507aad90c07101f53cb88a2f28dfa74029c499d)

Did not run any tasks
This progress looks :) because there were no failed tasks or missing external_
↳dependencies

===== Luigi Execution Summary =====
```

Try deleting one of the output files (e.g. the one for number 3) and see what happens. Luigi will only reprocess this file and of course the average building task².

1.2.1 Accessing log files

But what happens if we have an error in one of our tasks? Lets mimic a problem by including the lines

```
if self.number == 3:
    raise ValueError("This is in purpose")
```

to the run function of the RandomNumberTask. Now delete all output files and re-start the processing:

```
===== Luigi Execution Summary =====

Scheduled 11 tasks of which:
* 9 ran successfully:
  - 9 RandomNumberTask(git_hash=5507aad90c07101f53cb88a2f28dfa74029c499d, number=0) _
↳...
* 1 failed:
  - 1 RandomNumberTask(git_hash=5507aad90c07101f53cb88a2f28dfa74029c499d, number=3)
* 1 were left pending, among these:
  * 1 had failed dependencies:
    - 1 AveragerTask(git_hash=5507aad90c07101f53cb88a2f28dfa74029c499d)

This progress looks :( because there were failed tasks

===== Luigi Execution Summary =====
```

As one of the RandomNumberTask has failed, it will also not process the AveragerTask. If you want to have a look into what has gone wrong, you can have a look into the log files. They are stored in a folder called logs in your current working directory. You will find a file called RandomNumberTask_stderr for our erroneous task with number 3 with the correct exception.

After fixing the task, b2luigi will reprocess the failed one and the AveragerTask.

You can go on with scheduling jobs on the queue system in *Submitting to the Batch System*. Or have a look into more advanced examples in *Advanced Examples*.

luigi documentation: <http://luigi.readthedocs.io/en/stable/>

² If you have already worked with luigi you might note, that this behaviour is different from the default luigi one.

1.3 Submitting to the Batch System

We use the task definition file created in [Quick Start](#) and submit it to the batch system.

The only thing you need to do for this is start your file with the option `--batch`, e.g. like so

```
python3 simple-task.py --batch
```

The output file and log files are written to the same folders and the same amount of work is done - the only difference is that the calculation is now running on the batch system.

b2luigi will schedule a single batch job for each requested task. Using the dependency management of luigi, the batch jobs are only scheduled when all dependencies are fulfilled saving you some unneeded CPU time on the batch system. After a job is submitted, b2luigi will check if it is still running or not and handle failed or done tasks correctly.

1.3.1 Choosing the LSF queue

By default, all tasks will be sent to the short queue. This behaviour can be changed on a per task level by giving the task a property called `queue` and setting it to the queue it should run on, e.g.

```
class MyLongTask(b2luigi.Task):
    queue = "l"
```

1.3.2 Start a Central Scheduler

When the number of tasks grows, it is sometimes hard to keep track of all of them (despite the summary in the end). For this, luigi brings a nice visualisation tool called the central scheduler.

To start this you need to call the `luigid` executable. Where to find this depends on your installation type:

1. If you have a installed b2luigi without user flag, you can just call the executable as it is already in your path:

```
luigid --port PORT
```

2. If you have a local installation, luigid is installed into your home directory:

```
~/.local/bin/luigid --port PORT
```

The default port is 8082, but you can choose any non-occupied port.

The central scheduler will register the tasks you want to process and keep track of which tasks are already done.

To use this scheduler, call b2luigi by giving the connection details:

```
python3 simple-task.py [--batch] --scheduler-host HOST --scheduler-port PORT
```

which works for batch as well as non-batch jobs. You can now visit the url <http://HOST:PORT> with your browser and see a nice summary of the current progress of your tasks.

You are now ready to face some more [Advanced Examples](#) or have a look into the [FAQ](#).

1.3.3 Drawbacks of the batch mode

Although the batch mode has many benefits, it would be unfair to not mention its downsides:

- You have to choose the queue depending in your requirements (e.g. wall clock time) by yourself. So you need to make sure that the tasks will actually finish before the batch system kills them because of timeout.
- There is currently now resubmission implemented. This means dying jobs because of batch system failures are just dead. But because of the dependency checking mechanism of `luigi` it is simple to just redo the calculation and re-calculate what is missing.
- The `luigi` feature to request new dependencies while task running (via `yield`) is not implemented for the batch mode.
- We need to check the status of the tasks quite often. If your site has restrictions on this, you might fall into them.

1.4 Advanced Examples

1.4.1 Simple Basf2 Task

```
import b2luigi
import time
import basf2

class MyTask(b2luigi.Basf2Task):
    number = b2luigi.IntParameter()

    def create_path(self):
        path = basf2.create_path()
        path.add_module("EventInfoSetter", evtNumList=[self.number])
        path.add_module("RootOutput", outputFileNames=self.get_output_file_names() [
↪ "output.root"])

        return path

    def output(self):
        yield self.add_to_output("output.root")

class MainTask(b2luigi.Basf2FileMergeTask):
    def requires(self):
        for i in range(1, 10):
            yield MyTask(number=i)

if __name__ == "__main__":
    b2luigi.process(MainTask(), workers=5)
```

1.5 FAQ

TODO

1.6 TODO items

Make sure to check out [Development](#) for an introduction into developing b2luigi.

- Add support for different batch systems, e.g. htcondor
- Integrate gbasf2 as another batch system
- Document the API of the task class
- Add more examples on how to do complex tasks with a full “analysis example”.
- Add a function for writing into temporary files and moving automatically
- Add helper messages on events (e.g. failed)
- Clearly document that there is currently no implementation for dependencies on demand!
- port b2luigi to basf2

1.7 Development

You want to help developing b2luigi? Great! Here are some first steps to help you dive in:

1. Make sure you uninstall b2luigi if you have installed it from pypi

```
pip3 uninstall b2luigi
```

2. Clone the repository from github

```
git clone https://github.com/nils-braun/b2luigi
```

3. b2luigi is not using `setuptools` but the newer (and better) `flit` as a builder. Install it via

```
pip3 [ --user ] install flit
```

You can now install b2luigi from the cloned git repository in development mode:

```
flit install -s
```

4. The documentation is hosted on read the docs and build automatically on every commit to master. You can (and should) also build the documentation locally by installing `sphinx`

```
pip3 [ --user ] install sphinx sphinx-autobuild
```

And starting the automatic build process in the projects root folder

```
sphinx-autobuild docs build
```

The autobuild will rebuild the project whenever you change something. It displays a URL where to find the created docs now (most likely <http://127.0.0.1:8000>). Please make sure the documentation looks fine before creating a pull request.

5. If you are a core developer and want to release a new version:
 - (a) Make sure all changes are committed and merged on master
 - (b) Use the `bumpversion` package to update the version in the python file `b2luigi/__init__.py` as well as the git tag. `flit` will automatically use this.

```
bumpversion patch/minor/major
```

- (c) Push the new commit and the tags

```
git push  
git push --tags
```

- (d) Publish to pipy

```
flit publish
```

At a later stage, I will try to automate this.