gwf Documentation

Release 1.5.1

Thomas Mailund

Jun 03, 2019

CONTENTS

gwf is a flexible, pragmatic workflow tool for building and running large, scientific workflows. It runs on Python 3.5+ and is developed at the Bioinformatics Research Centre (BiRC), Aarhus University.

Examples To get a feeling for what a *gwf* workflow looks like, have a look at a few examples.

Getting started To quickly get started writing workflows in *gwf* you can read the *Tutorial*.

Extending We don't have the backend you need to run your workflow on your cluster? See the *Writing Backends* section to roll your own.

Contributing We aim to make *gwf* a community developed project. Learn how to *contribute*.

CHAPTER

ONE

INSTALLATION

To install *gwf* via pip:

```
pip install gwf
```

To install *gwf* via conda:

```
conda config --add channels gwforg
conda install gwf
```

We recommend that you install *gwf* in a project-specific environment:

```
conda config --add channels gwforg
conda create -n myproject python=3.5 gwf dep1 dep2 ...
source activate myproject
```

You can find the code for *gwf* here. You are encouraged to report any issues through the issue tracker, which is also a good place to ask questions.

CHAPTER

TWO

GETTING STARTED

2.1 Tutorial

In this tutorial we will explore various concepts in *gwf*. We will define workflows and see how *gwf* can help us keep track of the progress of workflow execution, the output of targets and dependencies between targets. Have fun!

We'll assume that you have the Anaconda distribution installed and that you are familiar with how to install and manage packages and environments through the *conda* package manager.

First, let's install *gwf* in its own conda environment. Create a new environment for your project, we'll call it *myproject*.

```
$ conda config --add channels gwforg
$ conda create -n myproject python=3.5 gwf
$ source activate myproject
```

You should now be able to run the following command.

```
$ gwf --help
```

This should show you the commands and options available through *gwf*.

Caution: You may see an error similar to this when you try running *gwf*:

```
UnicodeEncodeError: 'charmap' codec can't encode character '\u2020' in
position 477: character maps to <undefined>
```

This error occurs because your isn't configured to use UTF-8 as the default encoding. To fix the error insert the following lines in your `.bashrc` file:

```
export LANG=en_US.utf8
export LC_ALL=en_US.utf8
```

If you're not in the US you may want to set it to something else. For example, if you're in Denmark you may want to use the following configuration:

```
export LANG=da_DK.utf8
export LC_ALL=da_DK.utf8
```

2.1.1 A Minimal Workflow

To get started we must define a *workflow file* containing a workflow to which we can add targets. Unless *gwf* is told otherwise it assumes that the workflow file is called `workflow.py` and that the workflow is called *gwf*:

```
from gwf import Workflow

gwf = Workflow()

gwf.target('MyTarget', inputs=[], outputs=[]) << """
echo hello world
"""
```

In the example above we define a workflow and then add a target called `MyTarget`. A target is a single unit of computation that uses zero or more files (inputs) and produces zero or more files (outputs).

The target defined above does not use any files and doesn't produce any files either. However, it does run a single command (`echo hello world`), but the output of the command is thrown away. Let's fix that! Change the target definition to this:

```
gwf.target('MyTarget', inputs=[], outputs=['greeting.txt']) << """
echo hello world
"""
```

This tells *gwf* that the target will create a file called `greeting.txt` when it is run. However, the target does not actually create the file yet. Let's fix that too:

```
gwf.target('MyTarget', inputs=[], outputs=['greeting.txt']) << """
echo hello world > greeting.txt
"""
```

There you go! We have now declared a workflow with one target and that target creates the file `greeting.txt` with the line `hello world` in it. Now let's try to run our workflow...

2.1.2 Running Your First Workflow

First, let's make a directory for our project. We'll call the directory `myproject`. Now create an empty file called `workflow.py` in the project directory and paste the workflow specification into it:

```
from gwf import Workflow

gwf = Workflow()

gwf.target('MyTarget', inputs=[], outputs=['greeting.txt']) << """
echo hello world > greeting.txt
"""
```

We're now ready to run our workflow. However, *gwf* does not actually execute the targets in a workflow, it only schedules the target using a backend. This may sound cumbersome, but it enables *gwf* to run workflows in very different environments: anything from your laptop to a cluster with thousands of cores available.

For this tutorial we just want to run our workflows locally. To do this we can use the built-in `local` backend. Essentially this backend allows you to run workflows utilizing all cores of your computer and thus it can be very useful for small workflows that don't require a lot of resources.

First, open another terminal window and navigate to the `myproject` directory. Then run the command:

```
$ gwf workers
Started 4 workers, listening on port 12345
```

This will start a pool of workers that *gwf* can now submit targets to. Switch back to the other terminal and then run:

```
$ gwf run
Scheduling target MyTarget
Submitting target MyTarget
```

gwf schedules and then submits `MyTarget` to the pool of workers you started in the other terminal window.

This says that *gwf* considered the target for execution and then decided to submit it to the backend (because the output file, `greeting.txt`, does not already exist).

Within a few seconds you should see `greeting.txt` in the project directory. Try to open it in your favorite text editor!

Now try the same command again:

```
$ gwf run
Scheduling target MyTarget
```

This time, *gwf* considers the target for submission, but decides not to submit it since all of the output files (only one in this case) exist.

Note: When you've completed this tutorial, you probably want to close the local workers. To do this simply change to the terminal where you started the workers and press `Control-c`.

2.1.3 Setting the Default Verbosity

Maybe you got tired of seeing this much output from *gwf* all the time, despite the pretty colors. We can change the verbosity (how chatty *gwf* is) using the `-v/--verbose` flag:

```
$ gwf -v warning run
```

Now *gwf* only prints warnings. However, it quickly gets annoying to type this again and again, so let's configure *gwf* to make `warning` the default verbosity level.

```
$ gwf config set verbose warning
$ gwf run
```

As we'd expect, *gwf* outputs the same as before, but this time we didn't have to set the `-v warning` flag!

We can configure other aspects of *gwf* through the *config* command. For more details, refer to the *Configuration* page.

2.1.4 Defining Targets with Dependencies

Targets in *gwf* represent isolated units of work. However, we can declare dependencies between targets to construct complex workflows. A target B that depends on a target A will only run when A has been run successfully (that is, if all of the output files of A exist).

In *gwf*, dependencies are declared through file dependencies. This is best understood through an example:

```
from gwf import Workflow

gwf = Workflow()

gwf.target('TargetA', inputs=[], outputs=['x.txt']) << """
echo "this is x" > x.txt
```

(continues on next page)

(continued from previous page)

```
"""

gwf.target('TargetB', inputs=[], outputs=['y.txt']) << """
echo "this is y" > y.txt
"""

gwf.target('TargetC', inputs=['x.txt', 'y.txt'], outputs=['z.txt']) << """
cat x.txt y.txt > z.txt
"""
```

In this workflow, `TargetA` and `TargetB` each produce a file. `TargetC` declares that it needs two files as inputs. Since the file names match the file names produced by `TargetA` and `TargetB`, `TargetC` depends on these two targets.

Let's try to run this workflow:

```
$ gwf run
Scheduling target TargetC
Scheduling dependency TargetA of TargetC
Submitting target TargetA
Scheduling dependency TargetB of TargetC
Submitting target TargetB
Submitting target TargetC
```

(You can leave out the *-v info* option if you set it as the default in the previous section).

Notice that *gwf* first attempts to submit `TargetC`. However, because of the file dependencies it first schedules each dependency and submits those to the backend. It then submits `TargetC` and makes sure that it will only be run when both `TargetA` and `TargetB` has been run. If we decided that we needed to re-run `TargetC`, but not `TargetA` and `TargetB`, we could just delete `z.txt` and run `gwf run` again. *gwf* will automatically figure out that it only needs to run `TargetC` again and submit it to the backend.

What happens if we do something nonsensical like declaring a cyclic dependency? Let's try:

```
from gwf import Workflow

gwf = Workflow()

gwf.target('TargetA', inputs=['x.txt'], outputs=['x.txt']) << """
echo "this is x" > x.txt
"""
```

Run this workflow. You should see the following:

```
Error: Target TargetA depends on itself.
```

2.1.5 Observing Target Execution

As workflows get larger they make take a very long time to run. With *gwf* it's easy to see how many targets have been completed, how many failed and how many are still running using the `gwf status` command. We'll modify the workflow from earlier to fake that each target takes some time to run:

```
from gwf import Workflow

gwf = Workflow()
```

(continues on next page)

(continued from previous page)

```
gwf.target('TargetA', inputs=[], outputs=['x.txt']) << """
sleep 20 && echo "this is x" > x.txt
"""

gwf.target('TargetB', inputs=[], outputs=['y.txt']) << """
sleep 30 && echo "this is y" > y.txt
"""

gwf.target('TargetC', inputs=['x.txt', 'y.txt'], outputs=['z.txt']) << """
sleep 10 && cat x.txt y.txt > z.txt
"""
```

Now run `gwf status` (Remember to remove `x.txt`, `y.txt` and `z.txt`, otherwise *gwf* will not submit the targets again). You should see something like this, but with pretty colors.

```
TargetA    shouldrun       0.00%
TargetB    shouldrun       0.00%
TargetC    shouldrun       0.00%
```

Each target in the workflow is shown on a separate line. We can see the status of the target (*shouldrun*) and percentage completion. The percentage tells us how many dependencies of the target have been completed. If all dependencies of the target, and the target itself, have been completed, the percentage will be 100%.

Let's try to run the workflow and see what happens.

```
$ gwf run
$ gwf status
TargetA    running         0.00%
TargetB    submitted       0.00%
TargetC    submitted       0.00%
```

The `R` shows that one third of the targets are running (since I'm only running with one worker, only one target can run at a time) and the other two thirds have been submitted. Running the status command again after some time should show something like this.

```
TargetA    completed     100.00%
TargetB    running         0.00%
TargetC    submitted      33.33%
```

Now the target that was running before has completed, and another target is now running, while the final target is still just submitted. After some time, run the status command again. The last target should now be running.

```
TargetA    completed     100.00%
TargetB    completed     100.00%
TargetC    running        66.67%
```

After a while, all targets should have completed.

```
TargetA    completed     100.00%
TargetB    completed     100.00%
TargetC    completed     100.00%
```

Here's a few neat things you should know about the status command:

- If you only want to see endpoints (targets that no other targets depend on), you can use the `--endpoints` flag.

- You can use wildcards in target names. For example, `gwf status 'Foo*'` will list all targets beginning with *Foo*. You can specify multiple targets/patterns by separating them with a space. This also works in the cancel and clean commands (but remember the quotes around the pattern)!
- Only want to see which targets are running? You can filter targets by their status using e.g. `gwf status -s running`. You can also combine filters, i.e. `gwf status --endpoints --status running 'Align*'` to show all endpoints that are running and where the name starts with *Align*.

For more details you can always refer to builtin help with `gwf status --help`.

2.1.6 Reusable Targets with Templates

Often you will want to reuse a target definition for a lot of different files. For example, you may have two files with reads that you need to map to a reference genome. The mapping is the same for the two files, so it would be annoying to repeat it in the workflow specification.

Instead, *gwf* allows us to define a template which can be used to generate one or more targets easily. In general, a template is just a function which returns a tuple containing four values:

1. The *inputs* files,
2. The *outputs* files,
3. a dictionary with options for the target that is to be generated, for example how many cores the template needs and which files it depends on,
4. a string which contains the specification of the target that is to be generated.

Templates are great because they allow you to reuse functionality and encapsulate target creation logic. Let's walk through the example above.

Note: Code and data files for this example is available here. To get started, follow these steps:

1. Change your working directory to the `readmapping` directory.
2. Run `conda env create` to create a new environment called *readmapping*. This will install all required packages, including *gwf* itself, *samtools* and *bwa*.
3. Activate the environment with `source activate readmapping`.
4. Open another terminal and navigate to the same directory.
5. Activate the environment in this terminal too, using the same command as above.
6. Start a pool with two workers with `gwf workers -n 2`.
7. Jump back to the first terminal. Configure *gwf* to use the local backend for this project using `gwf config backend local`.
8. You should now be able to run `gwf status` and all of the other *gwf* commands used in this tutorial.

Our reference genome is stored in `ponAbe2.fa.gz`, so we'll need to unzip it first. Let's write a template that unpacks files.

```
def unzip(inputfile, outputfile):
    """A template for unzipping files."""
    inputs = [inputfile]
    outputs = [outputfile]
    options = {
        'cores': 1,
```

(continues on next page)

(continued from previous page)

```
        'memory': '2g',
    }

    spec = '''
    gzcat {} > {}
    '''.format(inputfile, outputfile)

    return inputs, outputs, options, spec
```

This is just a normal Python function that returns a tuple. The function takes two arguments, the name of the input file and the name of the output file. In the function we define the inputs and outputs files, a dictionary that defines the options of the targets created with this template, and a string describing the action of the template.

We can now create a concrete target using this template:

```
gwf.target_from_template('UnzipGenome',
                         unzip(inputfile='ponAbe2.fa.gz',
                               outputfile='ponAbe2.fa'))
```

You could run the workflow now. The `UnzipGenome` target would be scheduled and submitted, and after a few seconds you should have a `ponAbe2.fa` file in the project directory.

Let's now define another template for indexing a genome.

```
def bwa_index(ref_genome):
    """Template for indexing a genome with `bwa index`."""
    inputs = ['{}.fa'.format(ref_genome)]
    outputs = ['{}.amb'.format(ref_genome),
               '{}.ann'.format(ref_genome),
               '{}.pac'.format(ref_genome),
               '{}.bwt'.format(ref_genome),
               '{}.sa'.format(ref_genome),
               ]
    options = {
        'cores': 16,
        'memory': '1g',
    }

    spec = """
    bwa index -p {ref_genome} -a bwtsw {ref_genome}.fa
    """.format(ref_genome=ref_genome)

    return inputs, outputs, options, spec
```

This template looks more complicated, but really it's the same thing as before. We define the inputs and outputs, a dictionary with options and a string with the command that will be executed.

Let's use this template to create a target for indexing the reference genome:

```
gwf.target_from_template('IndexGenome',
                         bwa_index(ref_genome='ponAbe2'))
```

Finally, we'll create a template for actually mapping the reads to the reference.

```
def bwa_map(ref_genome, r1, r2, bamfile):
    """Template for mapping reads to a reference genome with `bwa` and `samtools`."""
    inputs = [r1, r2,
```

(continues on next page)

(continued from previous page)

```
                '{}.amb'.format(ref_genome),
                '{}.ann'.format(ref_genome),
                '{}.pac'.format(ref_genome),
            ]
    outputs = [bamfile]
    options = {
        'cores': 16,
        'memory': '1g',
    }

    spec = '''
    bwa mem -t 16 {ref_genome} {r1} {r2} | \
    samtools sort | \
    samtools rmdup -s - {bamfile}
    '''.format(ref_genome=ref_genome, r1=r1, r2=r2, bamfile=bamfile)

    return inputs, outputs, options, spec
```

This is much the same as the previous template. Here's how we're going to use it:

```
gwf.target_from_template('MapReads',
                         bwa_map(ref_genome='ponAbe2',
                                 r1='Masala_R1.fastq.gz',
                                 r2='Masala_R2.fastq.gz',
                                 bamfile='Masala.bam'))
```

As you can see, templates are just normal Python functions and thus they can be inspected and manipulated in much the same way. Also, templates can be put into modules and imported into your workflow files to facilitate reuse. It's all up to you!

2.1.7 Viewing Logs

We may be curious about what the `MapReads` target wrote to the console when the target ran, to see if there were any warnings. If a target failed, it's also valuable to see it's output to diagnose the problem. Luckily, *gwf* makes this very easy.

```
$ gwf logs MapReads
```

When you run this command you'll see nothing. This is because the `gwf logs` command by default only shows things written to stdout by the target, and not stderr, and apparently nothing was written to stdout in this target. Let's try to take a look at stderr instead by applying the `--stderr` flag (or the short version `-e`).

```
$ gwf logs --stderr MapReads
[M::bwa_idx_load_from_disk] read 0 ALT contigs
[M::process] read 15000 sequences (1500000 bp)...
[M::mem_pestat] # candidate unique pairs for (FF, FR, RF, RR): (1, 65, 1, 0)
[M::mem_pestat] skip orientation FF as there are not enough pairs
[M::mem_pestat] analyzing insert size distribution for orientation FR...
[M::mem_pestat] (25, 50, 75) percentile: (313, 369, 429)
[M::mem_pestat] low and high boundaries for computing mean and std.dev: (81, 661)
[M::mem_pestat] mean and std.dev: (372.88, 86.21)
[M::mem_pestat] low and high boundaries for proper pairs: (1, 777)
[M::mem_pestat] skip orientation RF as there are not enough pairs
[M::mem_pestat] skip orientation RR as there are not enough pairs
[M::mem_process_seqs] Processed 15000 reads in 1.945 CPU sec, 0.678 real sec
```

(continues on next page)

(continued from previous page)

```
[main] Version: 0.7.15-r1140
[main] CMD: bwa mem -t 16 ponAbe2 Masala_R1.fastq.gz Masala_R2.fastq.gz
[main] Real time: 0.877 sec; CPU: 2.036 sec
```

We can do this for any target in our workflow. The logs shown are always the most recent ones since *gwf* does not archive logs from old runs of targets.

2.1.8 Cleaning Up

Now that we have run our workflow we may wish to remove intermediate files to save disk space. In *gwf* we can use the `gwf clean` command for this:

```
$ gwf clean
```

This command only removes files produced by an endpoint target (a target which no other target depends on):

```
$ gwf clean
Will delete 1.3MiB of files!
Deleting output files of IndexGenome
Deleting file "/Users/das/Code/gwf/examples/readmapping/ponAbe2.amb" from target
↪"IndexGenome"
Deleting file "/Users/das/Code/gwf/examples/readmapping/ponAbe2.ann" from target
↪"IndexGenome"
Deleting file "/Users/das/Code/gwf/examples/readmapping/ponAbe2.pac" from target
↪"IndexGenome"
Deleting file "/Users/das/Code/gwf/examples/readmapping/ponAbe2.bwt" from target
↪"IndexGenome"
Deleting file "/Users/das/Code/gwf/examples/readmapping/ponAbe2.sa" from target
↪"IndexGenome"
Deleting output files of UnzipGenome
Deleting file "/Users/das/Code/gwf/examples/readmapping/ponAbe2.fa" from target
↪"UnzipGenome"
```

We can tell *gwf* to remove all files by running `gwf clean --all`.

2.1.9 A Note About Reproducibility

Reproducibility is an important part of research and since *gwf* workflows describe every step of your computation, how the steps are connected, and the files produced in each step, it's a valuable tool in making your workflows reproducible. In combination with the `conda` package manager and the concept of environments, you can build completely reproducible workflows in a declarative, flexible fashion.

Consider the read mapping example used above. Since we included a specification of the complete environment through a `environment.yml` file, which even included samtools, bwa and *gwf* itself, we were able to easily create a working environment with exactly the right software versions used for our workflow. The whole workflow could also easily be copied to a cluster and run through e.g. the Slurm backend, since we can exactly reproduce the environment used locally.

2.2 Patterns

This guide takes you through some advanced features and patterns that can be utilized in *gwf*. Remember that *gwf* is just a way of generating workflows using the Python programming language and thus many of these patterns simply

use plain Python code to abstract and automate certain things.

2.2.1 Iterating Over a Parameter Space

Say that you have a workflow that runs a program with many different combinations of parameters, e.g. the parameters *xs*, *ys*, and *zs*. Each parameter can take multiple values:

We now want to run out program *simulate* with all possible combinations of these parameters. To do this, we'll use the Python function `itertools.product()` to create an iterator over all combinations of the parameters:

```
import itertools

parameter_space = itertools.product(xs, ys, zs)
```

We can then iterate over the parameter space:

```
gwf = Workflow()

for x, y, z in parameter_space:
    gwf.target(
        name='sim_{}_{}_{}'.format(x, y, z),
        inputs=['input.txt'],
        outputs=['output_{}_{}_{}.txt'.format(x, y, z)],
    ) << """
    ./simulate {} {} {}
    """.format(x, y, z)
```

2.2.2 Dynamically Generating a Workflow

We can make our workflows more reusable by generating them dynamically. For example, we may wish to make it easy for others to change the inputs to our workflow or let users specify a different output directory. When generating workflows dynamically you can essentially parameterize the workflow in any way you want. In combination with inclusion of workflows into other workflows, this allows for extremely powerful composition.

To dynamically generate a workflow, we simply create a function which builds the workflow and returns it:

```
import os.path.join
from gwf import Workflow

def my_fancy_workflow(output_dir='outputs/'):
    # Create an empty workflow object.
    w = Workflow()

    # Add targets to the workflow object, respecting the value of `output_dir`.
    foo_output = os.path.join(output_dir, 'output1.txt')
    gwf.target(
        name='Foo',
        inputs=['input.txt'],
        outputs=[foo_output],
    ) << """
    ./run_foo > {}
    """.format(foo_output)

    bar_output = os.path.join(output_dir, 'output2.txt')
    gwf.target(
```

(continues on next page)

(continued from previous page)

```
        name='Bar',
        inputs=[foo_output],
        outputs=[bar_output]
    )

    # Now return the workflow.
    return w
```

You can put this function in file next to your workflow, or any other place from which you can import the function. In this case, let's put the file next to `workflow.py` in a file called `fancy.py`.

In `workflow.py` we can then use the workflow as follows:

```
from fancy import my_fancy_workflow

gwf = my_fancy_workflow()
```

We can now run the workflow as usual:

```
$ gwf run
```

However, we can now easily change the output directory:

```
from fancy import my_fancy_workflow

gwf = my_fancy_workflow(output_dir='new_outputs/')
```

Parameterizing the workflow can also let the user choose to deactivate parts of the workflow. For example, imagine that `Bar` generates summary files that may now always be needed. In this case, we can let the user choose to leave it out:

```
import os.path.join
from gwf import Workflow

def my_fancy_workflow(output_dir='outputs/', summarize=True):
    # Create an empty workflow object.
    w = Workflow()

    # Add targets to the workflow object, respecting the value of `output_dir`.
    foo_output = os.path.join(output_dir, 'output1.txt')
    gwf.target(
        name='Foo',
        inputs=['input.txt'],
        outputs=[foo_output],
    ) << """
    ./run_foo > {}
    """.format(foo_output)

    # Only create target `Bar` if we want to summarize the data.
    if summarize:
        bar_output = os.path.join(output_dir, 'output2.txt')
        gwf.target(
            name='Bar',
            inputs=[foo_output],
            outputs=[bar_output]
        )
```

(continues on next page)

(continued from previous page)

```
    # Now return the workflow.
    return w
```

In `workflow.py` we can then use the workflow as follows:

```
from fancy import my_fancy_workflow

gwf = my_fancy_workflow(summarize=False)
```

2.2.3 External Configuration of Workflows

In the previous section we saw how we can parameterize workflows. However, in some cases we may want to let the user of our workflow specify the parameters without touching any Python code at all. That is, we want an external configuration file.

The configuration format could be anything, but in this example we'll use a JSON as the configuration format. First, this is what our configuration file is going to look like:

```
{
    "output_dir": "some_output_directory/",
    "summarize": true
}
```

We put this file next to `workflow.py`, e.g. as `config.json`. We can now read the configuration using the Python `json` module in `workflow.py`:

```
import json
from fancy import my_fancy_workflow

config = json.load(open('config.json'))

gwf = my_fancy_workflow(
    output_dir=config['output_dir'],
    summarize=config['summarize'],
)
```

We can now change the values in `config.json` and run the workflow as usual.

CHAPTER

THREE

TOPIC GUIDES

3.1 Topic Guides

Topic guides cover specific *gwf* features individually, giving an overview of how they're used and how they work.

3.1.1 Configuration

Configuration of *gwf* is project-specific and thus all configuration must be done in the project directory where the workflow file is located.

To see the value of a configuration key, use:

```
$ gwf config get KEY
```

To set the value of a key (or update it, if it already exists):

```
$ gwf config set KEY VALUE
```

Note that a keys are often of the form *this.is.a.key*. For example, the local backend supports the *local.port* setting which sets the port that the workers are running on. To set this settings, just run:

```
$ gwf config set local.port 4321
```

Now, when you run *gwf* with the local backend, it will try to connect workers on port 4321.

Your configuration is stored in the current working directory, which will usually be your project directory, in a file called `.gwfconf.json`. This means that all configuration is project-specific, which helps with reproducibility. You can inspect and change the file directly, but this is not recommended unless you really know what you're doing.

Core settings are listed on the *Settings* page. To see which options are available for a specific backend, refer to the *Backends* documentation.

3.1.2 Templates

Templates in *gwf* provide a simple mechanism for constructing a bunch of similar targets. For example, you may have 100 images that you want to transform in some way. We could write a *gwf* workflow to do this:

```
from gwf import Workflow

gwf = Workflow()

```

(continues on next page)

(continued from previous page)

```
photos = gwf.glob('photos/*.jpg')
for index, path in enumerate(photos):
    gwf.target('TransformPhoto.{}'.format(index), inputs=[path], outputs=[path + '.new
↪']) << """
    ./transform_photo {}
    """.format(path)
```

This will generate a target for each photo we've got which is all fine and dandy. However, we can make the code a lot clearer using *templates*. In *gwf* a template is a function that returns either a tuple or an instance of `AnonymousTarget`. Using a tuple is simple, but returning an `AnonymousTarget` is safer and clearer.

If the function returns a tuple, the tuple must contain four things:

1. a list of *input* files, corresponding to the *inputs* argument,
2. a list of *output* files, corresponding to the *outputs* argument,
3. a dictionary with options for the target that is to be generated, for example how many cores the template needs and which files it depends on,
4. a string which contains the specification of the target that is to be generated.

Let's rewrite our workflow from before with a template. First, we'll define the template function:

```
def transform_photo(path):
    inputs = [path]
    outputs = [path + '.new']
    options = {}
    spec = """./transform_photo {}""".format(path)
    return inputs, outputs, options, spec
```

Or if we wanted to return an `AnonymousTarget` (note that you need to import it first):

```
from gwf import AnonymousTarget

def transform_photo(path):
    inputs = [path]
    outputs = [path + '.new']
    options = {}
    spec = """./transform_photo {}""".format(path)
    return AnonymousTarget(inputs=inputs, outputs=outputs, options=options, spec=spec)
```

Next, we tell *gwf* to create a target using the `target_from_template()` method. This works the same regardless of what your template function returns:

```
from gwf import Workflow

gwf = Workflow()

photos = gwf.glob('photos/*.jpg')
for index, path in enumerate(photos):
    gwf.target_from_template('TransformPhoto.{}'.format(index), transform_photo(path))
```

Templates are just Python functions, so you can do pretty much anything in a template function. For example, you can create template functions that work across a wide range of systems. E.g. the template can determine the operating system used and adapt the *spec* according to this.

Templates can also be put in separate files. We can put the `transform_photo()` template function into `templates.py` and then import it as we would import any other Python module:

```
from gwf import Workflow
from templates import transform_photo

gwf = Workflow()

photos = gwf.glob('photos/*.jpg')
for index, path in enumerate(photos):
    gwf.target_from_template('TransformPhoto.{}'.format(index), transform_photo(path))
```

3.1.3 Large Workflows

While *gwf* can handle quite large workflows without any problems, there are some things that may cause significant pain when working with very, very large workflows, especially when the workflows has many (> 50000) targets producing many files. However, the problems depend hugely on your filesystem since most scalability problems are caused by the time it takes *gwf* to access the filesystem when scheduling targets.

In this section we will show a few tricks for handling very large workflows.

I have to run the same pipeline for *a lot* of files and running `gwf status` is very slow.

In this case *gwf* is probably slow because computing the dependency graph for your entire workflow takes a while and because *gwf* needs to access the filesystem for each input and output file in the workflow to check if any targets should be re-run.

One solution to this problem is to dynamically generate individual workflows for each input file, as shown here:

```
from glob import glob
from gwf import Workflow

data_files = ['Sample1', 'Sample2', 'Sample3']
for input_file in data_files:
    workflow_name = 'Analyse.{}'.format(input_file)

    wf = Workflow(name=workflow_name)
    wf.target('{}.Filter'.format(input_file), inputs=[input_file], outputs=[...]) << "
↪"""..."""
    wf.target('{}.ComputeSummaries'.format(input_file), ...) << """..."""

    globals()[workflow_name] = wf
```

You can now run the workflow for a single sample by specifying the name of the workflow:

```
$ gwf run -f workflow.py:Analyse.Sample1 run
```

This will only run the targets associated with *Sample1*. While this means that running *all* workflows in one go involves a bit more work, it also means that *gwf* will only have to compute the dependency graph and check timestamps for the targets associated with the selected sample.

CHAPTER

FOUR

DEVELOPMENT

4.1 Development

You can extend *gwf* by writing backends, which can be distributed as regular Python packages and registered with *gwf* through the entrypoints mechanism.

4.1.1 Writing Backends

Backends in *gwf* are the interface between *gwf* and whatever can be used to execute a target. For example, the Slurm backend included with *gwf* submits targets to the Slurm Workload Manager.

To get started we must declare define a class that inherits from *Backend*:

```
# mybackend/mybackend.py
from gwf.backends import Backend


class MyBackend(Backend):
    pass
```

Registering a Backend

Backends must be registered under the `gwf.backends` entrypoint as shown here:

```
# mybackend/setup.py
from setuptools import setup

setup(
    name="mybackend",
    version="0.0.1",
    py_modules=['mybackend'],
    install_requires=[
      'gwf>=1.0',
    ],
    entry_points={
        'gwf.backends': [
            'mybackend = mybackend:MyBackend',
        ]
    },
)
```

Backends must implement a set of methods that *gwf* uses to submit to the backend and query the backend for the status of targets.

Option Defaults

The backend should define `option_defaults` as an attribute. The value must be a dictionary mapping option names to defaults, e.g.:

```
# mybackend/mybackend.py
from gwf.backends import Backend

class MyBackend(Backend):
    option_defaults = {
        'cores': 1,
    }
```

Internally, *gwf* uses this dictionary to check whether targets contain options not supported by the backend and warn the user if this is the case. Thus, *all* options supported by the backend must be specified in this dictionary.

Targets in a workflow can now declare the number of cores they wish to allocate and we can use this information in `submit()` to allocate the given number of cores for the target to be submitted. If the user doesn't specify the *cores* option it will default to 1.

If want to specify support for an option, but there is no sensible default value (e.g. in the case of a username or e-mail address), use *None* as the default value.

Implementing the Backend Interface

Our backend still doesn't really do anything. We've only told *gwf* that our backend exists (by its entrypoint) and which options are supported. To get a backend to actually work we must implement three methods: `submit()`, `cancel()` and `status()`. If needed, one may also implement the `close()` method, which will be called when the backend is no longer needed (right before *gwf* exits).

All methods must return immediately, that is, calling `submit()` should submit the target for execution in some other process, but not run the target itself. For example, the local backend connects to a set of workers running in a different process, submits jobs to these workers and returns immedicately.

Storing Log Files

Backends can store log files in different ways. For example, the Slurm backend stores log files as files on disk, while other backends may wish to store log files in an S3 bucket or in a database.

To allow for all of these scenarios, *gwf* has the concept of a *log manager*. The log manager interface only assumes that log files can be written and accessed through file-like objects. Log managers should inherit from `LogManager`.

```
from gwf.backends.base import LogManager

class MyLogManager(LogManager):

    def open_stdout(self, target, mode='r'):
        pass

    def open_stderr(self, target, mode='r'):
        pass
```

Each method must return a file-like object providing access to the log data for *target*. Log managers can also provide other methods. For example, the `FileLogManager` provides methods for retrieving the paths of the log files.

Backends should set the `log_manager` attribute on the class to an instance of a the log manager to be used. The log manager must be set as a class attribute to allow access to log files without initializing the backend, which may be slow.

At the moment we provide two log managers:

- `FileLogManager` (default)
- `MemoryLogManager`

Handling Configuration

We can allow the user to configure aspects of the backend by using the central configuration object.

```
from gwf.conf import config

key1 = config.get('yourbackend.key1', 'default1')
key2 = config.get('yourbackend.key2', 'default2')
```

Backends should provide reasonable defaults, as shown above. The user can set configuration keys using the builtin `config` command:

```
$ gwf config set yourbackend.key1 value1
$ gwf config set yourbackend.key2 value2
```

If you want to contribute code, documentation or anything else to *gwf*, or you're a maintainer, this is where you should look.

4.1.2 For Contributors

We appreciate all contributions to *gwf*, not just contributions to the code! Think something is missing from the documentation? Defined useful snippets for your text editor? Add it and submit a pull request!

Set Up a Development Environment

We strongly recommend that you use the Anaconda Python distribution and the conda package manager to set up a development environment (actually, we recommend it for all of your Python work). However, feel free to use virtualenvs instead.

1. Download and install the Anaconda Python distribution following the instructions here.
2. Create an environment for *gwf* development:

   ```
   conda create -n gwfdev python=3.5
   ```

3. Activate the environment:

   ```
   source activate gwfdev
   ```

Make Your Changes

1. Fork the repository, clone it and create a branch for your changes:

```
git checkout -b my-change
```

2. Make the necessary changes and add unit tests if necessary.
3. Add a description of the changes to `CHANGELOG.rst` and add yourself to `CONTRIBUTORS.rst` (if you're not already there).
4. Test your changes and check for style violations:

```
make init      # to install gwf for development
gwf ...        # test your changes by running gwf
make lint      # to check for style issues
make test      # to run tests
make coverage  # to check test coverage
```

5. If everything is alright, commit your changes:

```
git add .
git commit -m "Added some-feature"
```

Show Us Your Contribution!

1. Push your committed changes back to your fork on GitHub:

```
git push origin HEAD
```

2. Follow these steps to create a pull request.
3. Check for comments and suggestions on your pull request and keep an eye on the CI output.

4.1.3 For Maintainers

The *gwf* build, testing and deployment process is automated through Travis.

Merging Changes

1. Make sure that the changes have proper test coverage, e.g. by checking the branch on Coveralls.
2. Check that the PR includes necessary updates of `CHANGELOG.rst` and `CONTRIBUTORS.rst`.
3. Always make a merge commit (don't rebase/fast-forward). The merge commit will be referenced in the change log.
4. Add the change to the change log for the coming (draft) release on GitHub. Make sure to follow the formatting used in previous change logs. Also, read about how to keep a change log.

Rolling a New Release

1. Make sure that all changes for the new release have been merged into `master` and that tests pass. Check Travis.
2. Make any other release-related changes such as adding new contributors to `CONTRIBUTORS.rst` or adding missing items to `CHANGELOG.rst`.

3. Increase the version number in `gwf/__init__.py`
4. Commit the changes and push the branch. Wait for tests to run.
5. Make a new release by tagging the merge commit with the version number, e.g. `vX.X.X`. Push the tag and wait for Travis to catch up.
6. Run `make package`, then `make publish` to publish the source distribution and wheel to PyPI.
7. Run `make package-conda`, then ‘‘make publish-conda

The documentation will be automatically be built by *ReadTheDocs*.

CHAPTER

FIVE

REFERENCE

5.1 Settings

This page lists settings that are used by *gwf*. Backends and plugins may define their own settings, but these are documented for each backend/plugin individually. See *Configuration* if in doubt about how to configure *gwf*.

- **backend (str):** Set the backend. Corresponds to the `--backend` flag (default: *local*).
- **verbose (str):** Set the verbosity. Corresponds to the `--verbose` flag (default: *info*).
- **no_color (bool):** If *true*, colors will not be used (default: *false*).

5.2 Backends

gwf supports multiple backends for running workflows. If you don't find a backend that suits your needs here, it's easy to *write your own backend*.

By default, *gwf* comes with the *local*, *slurm*, and *sge* backends.

5.2.1 Local

class gwf.backends.local.**LocalBackend**

Backend that runs targets on a local cluster.

To use this backend you must activate the *local* backend and start a local cluster (with one or more workers) that the backend can submit targets to. To start a cluster with two workers run the command:

```
gwf -b local workers -n 2
```

in the working directory of your project. The workflow file must be accessible to *gwf*. Thus, if your workflow file is not called *workflow.py* or the workflow object is not called *gwf*, you must specify this so that *gwf* can locate the workflow:

```
gwf -f myworkflow.py:wf1 -b local workers -n 2
```

If the local backend is your default backend you can of course omit the `-b local` option.

If the `-n` option is omitted, *gwf* will detect the number of cores available and use all of them.

To run your workflow, open another terminal and then type:

```
gwf -b local run
```

To stop the pool of workers press `Control-c`.

Backend options:

- **local.host (str):** Set the host that the workers are running on (default: localhost).
- **local.port (int):** Set the port used to connect to the workers (default: 12345).

Target options:

None available.

5.2.2 Slurm

class gwf.backends.slurm.**SlurmBackend**
Backend for the Slurm workload manager.

To use this backend you must activate the *slurm* backend.

Backend options:

- **backend.slurm.log_mode (str):** Must be either *full*, *merged* or *none*. If *full*, two log files will be stored for each target, one for standard output and one for standard error. If *merged*, only one log file will be written containing the combined streams. If *none*, no logs will be stored. (default: *full*).

Target options:

- **cores (int):** Number of cores allocated to this target (default: 1).
- **memory (str):** Memory allocated to this target (default: 1).
- **walltime (str):** Time limit for this target (default: 01:00:00).
- **queue (str):** Queue to submit the target to. To specify multiple queues, specify a comma-separated list of queue names.
- **account (str):** Account to be used when running the target.
- **constraint (str):** Constraint string. Equivalent to setting the *–constraint* flag on *sbatch*.
- **qos (str):** Quality-of-service strring. Equivalent to setting the *–qos* flog on *sbatch*.

5.2.3 Sun Grid Engine (SGE)

class gwf.backends.sge.**SGEBackend**
Backend for Sun Grid Engine (SGE).

To use this backend you must activate the *sge* backend. The backend currently assumes that a SGE parallel environment called "smp" is available. You can check which parallel environments are available on your system by running **`qconf -spl`**.

Backend options:

None.

Target options:

- **cores (int):** Number of cores allocated to this target (default: 1).
- **memory (str):** Memory allocated to this target (default: 1).
- **walltime (str):** Time limit for this target (default: 01:00:00).

- **queue (str):** Queue to submit the target to. To specify multiple queues, specify a comma-separated list of queue names.
- **account (str):** Account to be used when running the target. Corresponds to the SGE project.

5.3 API

The implementation of *gwf* consists of a few main abstractions. Units of work are defined by creating `Target` instances which also define the files used and produced by the target. A `Workflow` ties together and allows for easy creation of targets.

When all targets have been defined on a workflow, the workflow is turned into a `Graph` which will compute the entire dependency graph of the workflow, checking the workflow for inconsistencies and circular dependencies.

A target in a `Graph` can be scheduled on a *Backend* using the `Scheduler`.

5.3.1 Core

`gwf.core.`**`graph_from_config`**(*config*)
Return graph for the workflow specified by *config*.

See *graph_from_path()* for further information.

`gwf.core.`**`graph_from_path`**(*path*)
Return graph for the workflow given by *path*.

Returns a `Graph` object containing the workflow graph of the workflow given by *path*. Note that calling this function computes the complete dependency graph which may take some time for large workflows.

Parameters **`path`** (*str*) – Path to a workflow file, optionally specifying a workflow object in that file.

`class` `gwf.core.`**`AnonymousTarget`**(*inputs*, *outputs*, *options*, *working_dir=None*, *spec=''*, *protect=None*)
Represents an unnamed target.

An anonymous target is an unnamed, abstract target much like the tuple returned by function templates. Thus, *AnonymousTarget* can also be used as the return value of a template function.

Variables

- **`inputs`** (*list*) – A list of input paths for this target.
- **`outputs`** (*list*) – A list of output paths for this target.
- **`options`** (*dict*) – Options such as number of cores, memory requirements etc. Options are backend-dependent. Backends will ignore unsupported options.
- **`working_dir`** (*str*) – Working directory of this target.
- **`spec`** (*str*) – The specification of the target.
- **`protect`** (*set*) – An iterable of protected files which will not be removed during cleaning, even if this target is not an endpoint.

`property is_sink`
Return whether this target is a sink.

A target is a sink if it does not output any files.

property is_source
Return whether this target is a source.

A target is a source if it does not depend on any files.

class gwf.core.**Target**(*name=None*, ***kwargs*)
Represents a target.

This class inherits from *AnonymousTarget*.

A target is a named unit of work that declare their file *inputs* and *outputs*. Target names must be valid Python identifiers.

A script (or spec) is associated with the target. The script must be a valid Bash script and should produce the files declared as *outputs* and consume the files declared as *inputs*. Both parameters must be provided explicitly, even if no inputs or outputs are needed. In that case, provide the empty list:

```
Target('Foo', inputs=[], outputs=[], options={}, working_dir='/tmp')
```

The target can also specify an *options* dictionary specifying the resources needed to run the target. The options are consumed by the backend and may be ignored if the backend doesn't support a given option. For example, we can set the *cores* option to set the number of cores that the target uses:

```
Target('Foo', inputs=[], outputs=[], options={'cores': 16}, working_dir='/tmp')
```

To see which options are supported by your backend of choice, see the documentation for the backend.

Variables **name** (*str*) – Name of the target.

classmethod empty(*name*)
Return a target with no inputs, outputs and options.

This is mostly useful for testing.

class gwf.core.**Workflow**(*name=None*, *working_dir=None*, *defaults=None*)
Represents a workflow.

This is the most central user-facing abstraction in *gwf*.

A workflow consists of a collection of targets and has methods for adding targets to the workflow in two different ways. A workflow can be initialized with the following arguments:

Variables

- **name** (*str*) – initial value: None The name is used for namespacing when including workflows. See *include()* for more details on namespacing.
- **working_dir** (*str*) – The directory containing the file where the workflow was initialized. All file paths used in targets added to this workflow are relative to the working directory.
- **defaults** (*dict*) – A dictionary with defaults for target options.

By default, *working_dir* is set to the directory of the workflow file which initialized the workflow. However, advanced users may wish to set it manually. Targets added to the workflow will inherit the workflow working directory.

The *defaults* argument is a dictionary of option defaults for targets and overrides defaults provided by the backend. Targets can override the defaults individually. For example:

```
gwf = Workflow(defaults={
    'cores': 12,
    'memory': '16g',
```

(continues on next page)

(continued from previous page)

```
})

gwf.target('Foo', inputs=[], outputs=[]) << """echo hello"""
gwf.target('Bar', inputs=[], outputs=[], cores=2) << """echo world"""
```

In this case *Foo* and *Bar* inherit the *cores* and *memory* options set in *defaults*, but *Bar* overrides the *cores* option.

See *include()* for a description of the use of the *name* argument.

glob(*pathname*, **args*, ***kwargs*)
Return a list of paths matching *pathname*.

This method is equivalent to `glob.glob()`, but searches with relative paths will be performed relative to the working directory of the workflow.

iglob(*pathname*, **args*, ***kwargs*)
Return an iterator which yields paths matching *pathname*.

This method is equivalent to `glob.iglob()`, but searches with relative paths will be performed relative to the working directory of the workflow.

include(*other_workflow*, *namespace=None*)
Include targets from another `gwf.Workflow` into this workflow.

This method can be given either an `gwf.Workflow` instance, a module or a path to a workflow file.

If a module or path the workflow object to include will be determined according to the following rules:

1. If a module object is given, the module must define an attribute named *gwf* containing a `gwf.Workflow` object.
2. If a path is given it must point to a file defining a module with an attribute named *gwf* containing a `gwf.Workflow` object. If you want to include a workflow with another name you can specify the attribute name with a colon, e.g.:

   ```
   /some/path/workflow.py:myworkflow
   ```

   This will include all targets from the workflow *myworkflow* declared in the file */some/path/workflow.py*.

When a `gwf.Workflow` instance has been obtained, all targets will be included directly into this workflow. To avoid name clashes the *namespace* argument must be provided. For example:

```
workflow1 = Workflow()
workflow1.target('TestTarget')

workflow2 = Workflow()
workflow2.target('TestTarget')

workflow1.include(workflow2, namespace='wf1')
```

The workflow now contains two targets named *TestTarget* (defined in *workflow2*) and *wf1.TestTarget* (defined in *workflow1*). The *namespace* parameter can be left out if the workflow to be included has been named:

```
workflow1 = Workflow(name='wf1')
workflow1.target('TestTarget')

workflow2 = Workflow()
```

(continues on next page)

(continued from previous page)

```
workflow2.target('TestTarget')

workflow1.include(workflow2)
```

This yields the same result as before. The *namespace* argument can be used to override the specified name:

```
workflow1 = Workflow(name='wf1')
workflow1.target('TestTarget')

workflow2 = Workflow()
workflow2.target('TestTarget')

workflow1.include(workflow2, namespace='foo')
```

The workflow will now contain targets named *TestTarget* and *foo.TestTarget*.

include_path (*path*, *namespace=None*)
Include targets from another `gwf.Workflow` into this workflow.

See `include()`.

include_workflow (*other_workflow*, *namespace=None*)
Include targets from another `gwf.Workflow` into this workflow.

See `include()`.

shell (**args*, ***kwargs*)
Return the output of a shell command.

This method is equivalent to `subprocess.check_output()`, but automatically runs the command in a shell with the current working directory set to the working directory of the workflow.

Changed in version 1.0: This function no longer return a list of lines in the output, but a byte array with the output, exactly like `subprocess.check_output()`. You may specifically set *universal_newlines* to *True* to get a string with the output instead.

target (*name*, *inputs*, *outputs*, ***options*)
Create a target and add it to the `gwf.Workflow`.

This is syntactic sugar for creating a new `Target` and adding it to the workflow. The target is also returned from the method so that the user can directly manipulate it, if necessary. For example, this allows assigning a spec to a target directly after defining it:

```
workflow = Workflow()
workflow.target('NewTarget', inputs=['test.txt', 'out.txt']) <<< '''
cat test.txt > out.txt
echo hello world >> out.txt
'''
```

This will create a new target named *NewTarget*, add it to the workflow and assign a spec to the target.

Parameters

- **name** (*str*) – Name of the target.
- **inputs** (*iterable*) – List of files that this target depends on.
- **outputs** (*iterable*) – List of files that this target produces.

Any further keyword arguments are passed to the backend.

target_from_template(*name*, *template*, ***options*)
Create a target from a template and add it to the `gwf.Workflow`.

This is syntactic sugar for creating a new `Target` and adding it to the workflow. The target is also returned from the method so that the user can directly manipulate it, if necessary.

```
workflow = Workflow()
workflow.target_from_template('NewTarget', my_template())
```

This will create a new target named *NewTarget*, configure it based on the specification in the template *my_template*, and add it to the workflow.

Parameters

- **name** (*str*) – Name of the target.
- **template** (*tuple*) – Target specification of the form (inputs, outputs, options, spec).

Any further keyword arguments are passed to the backend and will override any options provided by the template.

class `gwf.core.`**Graph**(*targets*, *provides*, *dependencies*, *dependents*, *unresolved*)
Represents a dependency graph for a set of targets.

The graph represents the targets present in a workflow, but also their dependencies and the files they provide.

During construction of the graph the dependencies between targets are determined by looking at target inputs and outputs. If a target specifies a file as input, the file must either be provided by another target or already exist on disk. In case that the file is provided by another target, a dependency to that target will be added:

Variables **dependencies** (*dict*) – A dictionary mapping a target to a set of its dependencies.

If the file is not provided by another target, the file is *unresolved*:

Variables **unresolved** (*set*) – A set containing file paths of all unresolved files.

If the graph is constructed successfully, the following instance variables will be available:

Variables

- **targets** (*dict*) – A dictionary mapping target names to instances of `gwf.Target`.
- **provides** (*dict*) – A dictionary mapping a file path to the target that provides that path.
- **dependents** (*dict*) – A dictionary mapping a target to a set of all targets which depend on the target.

The graph can be manipulated in arbitrary, diabolic ways after it has been constructed. Checks are only performed at construction-time, thus introducing e.g. a circular dependency by manipulating *dependencies* will not raise an exception.

Raises **gwf.exceptions.WorkflowError** – Raised if the workflow contains a circular dependency.

dfs(*root*)
Return the depth-first traversal path through a graph from *root*.

endpoints()
Return a set of all targets that are not depended on by other targets.

classmethod **from_targets**(*targets*)
Construct a dependency graph from a set of targets.

When a graph is initialized it computes all dependency relations between targets, ensuring that the graph is semantically sane. Therefore, construction of the graph is an expensive operation which may raise a number of exceptions:

Raises **gwf.exceptions.FileProvidedByMultipleTargetsError** – Raised if the same file is provided by multiple targets.

Since this method initializes the graph, it may also raise:

Raises **gwf.exceptions.WorkflowError** – Raised if the workflow contains a circular dependency.

class gwf.core.**Scheduler**(*graph*, *backend*, *dry_run=False*, *file_cache={}*)

Schedule one or more targets and submit to a backend.

Scheduling a target will determine whether the target needs to run based on whether it already has been submitted and whether any of its dependencies have been submitted.

Targets that should run will be submitted to *backend*, unless *dry_run* is set to `True`.

When scheduling a target, the scheduler checks whether any of its inputs are unresolved, meaning that during construction of the graph, no other target providing the file was found. This means that the file should then exist on disk. If it doesn't the following exception is raised:

Raises **gwf.exceptions.FileRequiredButNotProvidedError** – Raised if a target has an input file that does not exist on the file system and that is not provided by another target.

schedule(*target*)

Schedule a target and its dependencies.

Returns `True` if *target* was submitted to the backend (even when *dry_run* is `True`).

Parameters **target** (*gwf.Target*) – Target to be scheduled.

schedule_many(*targets*)

Schedule multiple targets and their dependencies.

This is a convenience method for scheduling multiple targets. See *schedule()* for a detailed description of the arguments and behavior.

Parameters **targets** (*list*) – A list of targets to be scheduled.

should_run(*target*)

Return whether a target should be run or not.

status(*target: gwf.core.Target*) → gwf.core.TargetStatus

Return the status of a target.

Returns the status of a target where it is taken into account whether the target should run or not.

Parameters **target** (`Target`) – The target to return status for.

5.3.2 Backends

gwf.backends.**list_backends**()

Return the names of all registered backends.

gwf.backends.**backend_from_name**(*name*)

Return backend class for the backend given by *name*.

Returns the backend class registered with *name*. Note that the *class* is returned, not the instance, since not all uses requires initialization of the backend (e.g. accessing the backends' log manager), and initialization of the backend may be expensive.

Parameters **name** (*str*) – Path to a workflow file, optionally specifying a workflow object in that file.

gwf.backends.**backend_from_config**(*config*)
Return backend class for the backend specified by *config*.

See *backend_from_name()* for further information.

class gwf.backends.**Backend**
Base class for backends.

cancel(*target*)
Cancel *target*.

Parameters **target** (*gwf.Target*) – The target to cancel.

Raises **gwf.exception.TargetError** – If the target does not exist in the workflow.

close()
Close the backend.

Called when the backend is no longer needed and should close all resources (open files, connections) used by the backend.

log_manager = <gwf.backends.logmanager.FileLogManager object>

classmethod logs(*target*, *stderr=False*)
Return log files for a target.

If the backend cannot return logs a NoLogFoundError is raised.

By default standard output (stdout) is returned. If *stderr=True* standard error will be returned instead.

Parameters

- **target** (*gwf.Target*) – Target to return logs for.
- **stderr** (*bool*) – default: False. If true, return standard error.

Returns A file-like object. The user is responsible for closing the returned file(s) after use.

Raises **gwf.exceptions.NoLogFoundError** – if the backend could not find a log for the given target.

status(*target*)
Return the status of *target*.

Parameters **target** (*gwf.Target*) – The target to return the status of.

Return gwf.backends.Status Status of *target*.

submit(*target*, *dependencies*)
Submit *target* with *dependencies*.

This method must submit the *target* and return immediately. That is, the method must not block while waiting for the target to complete.

Parameters

- **target** (*gwf.Target*) – The target to submit.
- **dependencies** – An iterable of gwf.Target objects that *target* depends on and that have already been submitted to the backend.

class gwf.backends.**Status**
Status of a target.

A target is unknown to the backend if it has not been submitted or the target has completed and thus isn't being tracked anymore by the backend.

A target is submitted if it has been successfully submitted to the backend and is pending execution.

A target is running if it is currently being executed by the backend.

RUNNING = 2
The target is currently running.

SUBMITTED = 1
The target has been submitted, but is not currently running.

UNKNOWN = 0
The backend is not aware of the status of this target (it may be completed or failed).

Log Managers

class gwf.backends.logmanager.**FileLogManager**
A file-based log manager.

This log manager stores logs on disk in the *log_dir* directory (which defaults to *.gwf/logs*).

open_stderr(*target*, *mode='r'*)
Return file handle to standard error log file for target.

Raises **LogError** – If the log could not be found.

open_stdout(*target*, *mode='r'*)
Return file handle to the standard output log file for target.

Raises **LogError** – If the log could not be found.

stderr_path(*target_name*)
Return path of the log file containing standard error for target.

stdout_path(*target_name*)
Return path of the log file containing standard output for target.

class gwf.backends.logmanager.**MemoryLogManager**
A memory-based log manager.

This log manager stores logs in memory.

5.3.3 Filtering

gwf.filtering.**filter_generic**(*targets*, *filters*)
Filter targets given a list of filters.

Return all targets from *targets* passing all *filters*. For example:

```
matched_targets = filter_generic(
    targets=graph.targets.values(),
    filters=[
        NameFilter(patterns=['Foo*'],
        StatusFilter(scheduler=scheduler, status='running'),
    ]
)
```

returns a generator yielding all targets with a name matching `Foo*` which are currently running.

Parameters

- **targets** – A list of targets to be filtered.
- **filters** – A list of `Filter` instances.

`gwf.filtering.`**filter_names**(*targets*, *patterns*)

Filter targets with a list of patterns.

Return all targets in *targets* where the target name matches one or more of the patterns in *pattern*. For example:

```
matched_targets = filter_names(graph.targets.values(), ['Foo*'])
```

returns a generator yielding all targets with a name matching the pattern *Foo**. Multiple patterns can be provided:

```
matched_targets = filter_names(graph.targets.values(), ['Foo*', 'Bar*'])
```

returns all targets with a name matching either `Foo*` or `Bar*`.

This function is a simple wrapper around `NameFilter`.

Helpers for filtering:

class `gwf.filtering.`**ApplyMixin**

A mixin for predicate-based filters providing the *apply* method.

Most filters are predicate-based in the sense that they simply filter targets one by one based on a predicate function that decides whether to include the target or not. Such filters can inherit this mixin and then only need to declare a *`predicate()`* method which returns *True* if the target should be included and *False* otherwise.

For examples of using this mixin, see the `StatusFilter` and `EndpointFilter` filters.

apply(*targets*)

Apply the filter to all *targets*.

This method returns a generator yielding all targets in *targets* for each *`predicate()`* returns *True*.

predicate(*target*)

Return *True* if *target* should be included, *False* otherwise.

This method must be overriden by subclasses.

CHAPTER

SIX

CHANGE LOG

6.1 Version 1.5.1

6.1.1 Fixed

- Crash when Slurm returns unknown job state (#244).

6.2 Version 1.5.0

6.2.1 Added

- Users can now run `gwf init` to bootstrap a new *gwf* project (c78193).
- Add option to protect output files in a target from being removed when `gwf clean` is being run (2f51ed).

6.2.2 Fixed

- Ensure job script end with a newline (#239).
- Ignore missing log files when cleaning on run (#237).

6.3 Version 1.4.0

6.3.1 Added

- Backend for Sun Grid Engine (SGE). The backend does not support all target options supported by the Slurm backend, so workflows can not necessarily run with the SGE backend without changes. See the documentation for a list of supported options.

6.4 Version 1.3.2

6.4.1 Fixed

- Made the `touch` command faster.

6.5 Version 1.3.1

6.5.1 Added

- The `gwf status` command now accepts multiple `-s`/`--status` flags and will show targets matching any of the given states. E.g. `gwf status -s completed -s running` will show all completed and running targets.
- A new command `gwf touch` has been introduced. The command touches all files in the workflow in order, creating missing files and updating timestamps, such that *gwf* thinks that the workflow has been run.
- When specifying the workflow attribute in the workflow path, e.g. `gwf -f workflow.py:foo`, the filename part can now be left out and will default to *workflow.py*. For example, `gwf -f :foo` will access the `foo` workflow object in *workflow.py*.
- Documentation describing advanced patterns for *gwf* workflows.

6.6 Version 1.3.0

This release contains a bunch of new features and plenty of bug fixes. Most noteworthy is the removal of the progress bars in the status command. The status bars were often confusing and didn't communicate much more than a simple "percentage completion". The status command now outputs a table with target name, target status, and percentage completion (see the tutorial for examples). Additionally, the status command now shows all targets by default (not only endpoints). For users who wish to only see endpoints, there's now a `--endpoints` flag.

We aim to make *gwf* a good cluster citizen. Thus, logs from targets that no no longer exist in the workflow will now be removed when running `gwf run`. This ensures that *gwf* doesn't unnecessarily accumulate logs over time.

6.6.1 Fixed

- Add missing import to documentation for function templates (4eddcac).
- Remove reference to `--not-endpoints` flag (d7ed251).
- Remove broken badges in README (e352f09).
- Remove pre-1.0 upgrade documentation (bfa03da6).
- Fixed bug in scheduler that caused an exception when a target's input file did not exist, but the output file did (reported by Jonas Berglund) (92301ef3).

6.6.2 Changed

- Dots have been removed from logging output to make copy-pasting target names easier (f33f7195).
- Now uses pipenv to fix development environment.
- Improved coloring of logging output when running with `-v debug` (ab4ac7e3).
- Remove status bars in `gwf status` command (47cb7b50).

6.6.3 Added

- Added undocumented API which allows core and plugins to register validation functions for configuration keys. This fixes issues like #226 (c8c57d7c7).
- The `gwf clean` command now shows how much data will be removed (d81f143f1).
- Remove log files for targets that are no longer defined in the workflow (beb912bd).
- Note in tutorial on how to terminate the local workers (a long with other updates to the tutorial) (34421498).

6.7 Version 1.2.1

6.7.1 Fixed

- Bug when returning an `AnonymousTarget` from a template function without specifying the *working_dir* in the constructor (#212). Thanks to Steffen Møller-Larsen for reporting this.

6.8 Version 1.2

6.8.1 Fixed

- Bug when using `--format table` and no targets were found (#203).
- Bug when cancelling a target running on the Slurm backend (#199).
- Link to documentation in error message when unable to connect to local workers.
- Fixed bug in the *FileLogManager* where the wrong exception was raised when no log was found.

6.8.2 Changed

- Moved checking of file timestamps to the scheduler. This means that creating a `Graph` object will never touch the file system, and thus won't raise an exception if a target depends on a file that doesn't exist and that's not provided a target. Instead, unresolved paths are added to `Graph.unresolved`. They will then be checked by the scheduler (if necessary). For end users, this means that many commands have become substantially faster.

6.8.3 Added

- Added `AnonymousTarget` which represents an unnamed target. `Target` now inherits from this class and templates may now return an `AnonymousTarget` instead of a tuple.
- Added *backend.slurm.log_mode* option, see the documentation for the Slurm backend for usage (#202).

6.9 Version 1.1

6.9.1 Fixed

- Very slow scheduling when using dry run with unsubmitted targets (#184, 93e71a).

- Fixed cancellation with the Slurm backend (#183, 29445f).
- Fixed wildcard filtering of targets (#185, 036e3d).

6.9.2 Changed

- Move file cache construction out of `Graph` (#186, 93e71a). This change is invisible to end-users, but speeds up the `logs`, `cancel`, `info`, `logs` and `workers` commands.
- Replaced `--not-endpoints` flag in `clean` command with `--all` flag.
- Made filtering more intuitive in all commands.
- The `info` command now outputs JSON instead of invalid YAML.
- The `info` command outputs information for all targets in the workflow by default.
- Backends must now specify a `log_manager` class attribute specifying which log manager to use for accessing target log files.
- Backends should now be used as context managers to make sure that `Backend.close()` is called when the backend is no longer needed, as it is no longer called automatically on exit.

6.9.3 Added

- Added filtering of targets by name in the `info` command.
- Added API documentation for the `gwf.filtering` module.
- Added `gwf.core.graph_from_path()` and `gwf.core.graph_from_config()`.
- Added `gwf.backends.list_backends()`, `gwf.backends.backend_from_name()` and `gwf.backends.backend_from_config()`.
- Added `SlurmBackend.get_job_id()` and `SlurmBackend.forget_job()` to `SlurmBackend` to make it easier for plugins to integrate with Slurm.
- Documentation for log managers.
- Documentation on how to handle large workflows.

6.10 Version 1.0

First stable release of *gwf*! We strongly encourage users of pre-1.0 users to read the tutorial, since quite a lot of things have changed. We also recommend reading the guide for converting pre-1.0 workflows to version 1.0. However, users attempting to do this should be aware that the the template mechanism in 1.0 is slightly different and thus requires rewriting template functions.

6.10.1 Fixed

- Fixed a bug which caused *gwf* to fail when cancelling jobs when using the Slurm backend (8c1717).

6.10.2 Changed

- Documentation in various places, especially the core API.
- Documentation for maintainers.

6.10.3 Added

- Topic guide covering templates (b175fe).
- Added `info` command (6dbdbb).

6.11 Version 1.0b10

6.11.1 Fixed

- Fixed a subtle bug in scheduling which caused problems when resubmitting a workflow where some targets were already running (a5d884).
- Fixed a bug in the `SlurmBackend` which caused *gwf* to crash if the Slurm queue contained a job with many dependencies (eb4446).
- Added back the *-e* flag in the `logs` command.

6.12 Version 1.0b9

6.12.1 Fixed

- Fixed a bug in the `SlurmBackend` which caused running targets as unknown (33a6bd).

6.12.2 Changed

- The Slurm backend's database of tracked jobs is now cleaned on initialization to keep it from growing indefinitely (bd3f95).

6.13 Version 1.0b8

6.13.1 Fixed

- Fixed a bug which caused the *gwf logs* command to always show stderr (01b267).
- Fixed a bug which caused dependencies to be set incorrectly when two targets depended on the same target (4d9e07).

6.13.2 Changed

- Improved error message when trying to create a target from an invalid template (d27d1f).
- Improved error message when assigning a non-string spec to a target (2aca0a).
- *gwf logs* command now outputs logs via a pager when the system supports it, unless *–no-pager* is used (01b267).

6.13.3 Added

- Added more tests to cover scenarios with included workflows when building the workflow graph (86a68d0).
- Added a bunch of documentation (69e136, 51a0e7, 942b05).

6.14 Version 1.0b7

6.14.1 Fixed

- Fixed bug in scheduling which was actually the cause of the incorrect scheduling that was "fixed" in 1.0b6. Also added documentation for `gwf.core.schedule` (7c47cb).

6.14.2 Changed

- Updated documentation in a bunch of places, mostly styling.

6.15 Version 1.0b6

6.15.1 Fixed

- A bug in `SlurmBackend` which caused dependencies between targets to not be set correctly (6b71d2).

6.15.2 Changed

- More improvements to and clean up of build process.
- Updated some examples in the tutorial with current output from *gwf* (42c5da).
- Logging output is now more consistent (b95af04).

6.15.3 Added

- Documentation for maintainers on how to merge in contributions and rolling a new release (fe1ee3).

6.16 Version 1.0b5

6.16.1 Fixed

- Unset option passed to backend causes error (#166, dcff44).
- Set import path to allow import of module in workflow file (64841c).

6.16.2 Changed

- Vastly improved build and deploy process. We're now actually building and testing with conda.

6.17 Contributors

- Thomas Mailund
- Dan Søndergaard
- Anders Halager
- Michael Knudsen
- Tobias Madsen

PYTHON MODULE INDEX

g

INDEX

T

U

W