Toil Documentation

Release 5.11.0

UCSC Computational Genomics Lab

Jun 15, 2023

GETTING STARTED

Toil is an open-source pure-Python workflow engine that lets people write better pipelines.

Check out our website for a comprehensive list of Toil's features and read our paper to learn what Toil can do in the real world. Please subscribe to our low-volume announce mailing list and feel free to also join us on GitHub and Gitter.

If using Toil for your research, please cite

> Vivian, J., Rao, A. A., Nothaft, F. A., Ketchum, C., Armstrong, J., Novak, A., … Paten, B. (2017). Toil enables reproducible, open source, big biomedical data analyses. Nature Biotechnology, 35(4), 314–316. http://doi.org/10.1038/nbt.3772

CHAPTER

ONE

INSTALLATION

This document describes how to prepare for and install Toil. Note that Toil requires that the user run all commands inside of a Python virtualenv. Instructions for installing and creating a Python virtual environment are provided below.

1.1 Preparing Your Python Runtime Environment

Toil currently supports Python 3.7, 3.8, 3.9, and 3.10, and requires a virtualenv to be active to install.

If not already present, please install the latest Python `virtualenv` using pip:

```
$ sudo pip install virtualenv
```

And create a virtual environment called `venv` in your home directory:

```
$ virtualenv ~/venv
```

If the user does not have root privileges, there are a few more steps, but one can download a specific virtualenv package directly, untar the file, create, and source the virtualenv (version 15.1.0 as an example) using

```
$ curl -O https://pypi.python.org/packages/d4/0c/
↪9840c08189e030873387a73b90ada981885010dd9aea134d6de30cd24cb8/virtualenv-15.1.0.tar.gz
$ tar xvfz virtualenv-15.1.0.tar.gz
$ cd virtualenv-15.1.0
$ python virtualenv.py ~/venv
```

Now that you've created your virtualenv, activate your virtual environment:

```
$ source ~/venv/bin/activate
```

1.2 Basic Installation

If you need only the basic version of Toil, it can be easily installed using pip:

```
$ pip install toil
```

Now you're ready to run *your first Toil workflow*!

(If you need any of the extra features don't do this yet and instead skip to the next section.)

1.3 Installing Toil with Extra Features

Python headers and static libraries

Needed for the `mesos`, `aws`, `google`, and `encryption` extras.

On Ubuntu:

```
$ sudo apt-get install build-essential python-dev
```

On macOS:

```
$ xcode-select --install
```

Encryption specific headers and library

Needed for the `encryption` extra.

On Ubuntu:

```
$ sudo apt-get install libssl-dev libffi-dev
```

On macOS:

```
$ brew install libssl libffi
```

Or see Cryptography for other systems.

Some optional features, called *extras*, are not included in the basic installation of Toil. To install Toil with all its bells and whistles, first install any necessary headers and libraries (*python-dev*, *libffi-dev*). Then run

```
$ pip install toil[aws,google,mesos,encryption,cwl,wdl,kubernetes,server]
```

or

```
$ pip install toil[all]
```

Here's what each extra provides:

Extra	Description
`all`	Installs all extras (though htcondor is linux-only and will be skipped if not on a linux computer).
`aws`	Provides support for managing a cluster on Amazon Web Service (AWS) using Toil's built in *Cluster Utilities*. Clusters can scale up and down automatically. It also supports storing workflow state.
`google`	Experimental. Stores workflow state in Google Cloud Storage.

`mesos`	Provides support for running Toil on an Apache Mesos cluster. Note that running Toil on other batch systems does not require an extra. The `mesos` extra requires the following native dependencies:
	• Apache Mesos (Tested with Mesos v1.0.0)
	• *Python headers and static libraries*
	--- 	
	Important: If launching toil remotely on a mesos instance, to install Toil with the `mesos` extra in a virtualenv, be sure to create that virtualenv with the `--system-site-packages` flag (only use remotely!):
	`$ virtualenv ~/venv --system-site-packages`
	Otherwise, you'll see something like this:
	`ImportError: No module named mesos.native`
	--- 	

`htcondor`	Support for the htcondor batch system. This currently is a linux only extra.
`encryption`	Provides client-side encryption for files stored in the AWS job store. This extra requires the following native dependencies: • *Python headers and static libraries* • *libffi headers and library*
`cwl`	Provides support for running workflows written using the Common Workflow Language.
`wdl`	Provides support for running workflows written using the Workflow Description Language. This extra has no native dependencies.
`kubernetes`	Provides support for running workflows written using a Kubernetes cluster.
`server`	Provides support for Toil server mode, including support for the GA4GH Workflow Execution Service API.

1.4 Building from Source

If developing with Toil, you will need to build from source. This allows changes you make to Toil to be reflected immediately in your runtime environment.

First, clone the source:

```
$ git clone https://github.com/DataBiosphere/toil.git
$ cd toil
```

Then, create and activate a virtualenv:

```
$ virtualenv venv
$ . venv/bin/activate
```

From there, you can list all available Make targets by running `make`. First and foremost, we want to install Toil's build requirements (these are additional packages that Toil needs to be tested and built but not to be run):

```
$ make prepare
```

Now, we can install Toil in development mode (such that changes to the source code will immediately affect the virtualenv):

```
$ make develop
```

Or, to install with support for all optional *Installing Toil with Extra Features*:

```
$ make develop extras=[aws,mesos,google,encryption,cwl]
```

Or:

```
$ make develop extras=[all]
```

To build the docs, run `make develop` with all extras followed by

```
$ make docs
```

To run a quick batch of tests (this should take less than 30 minutes) run

```
$ export TOIL_TEST_QUICK=True; make test
```

For more information on testing see *Running Tests*.

CHAPTER

TWO

QUICKSTART EXAMPLES

2.1 Running a basic workflow

A Toil workflow can be run with just three steps:

1. Install Toil (see *Installation*)
2. Copy and paste the following code block into a new file called `helloWorld.py`:

```
from toil.common import Toil
from toil.job import Job


def helloWorld(message, memory="1G", cores=1, disk="1G"):
    return f"Hello, world!, here's a message: {message}"


if __name__ == "__main__":
    parser = Job.Runner.getDefaultArgumentParser()
    options = parser.parse_args()
    options.clean = "always"
    with Toil(options) as toil:
        output = toil.start(Job.wrapFn(helloWorld, "You did it!"))
    print(output)
```

3. Specify the name of the *job store* and run the workflow:

```
(venv) $ python helloWorld.py file:my-job-store
```

Note: Don't actually type `(venv) $` in at the beginning of each command. This is intended only to remind the user that they should have their *virtual environment* running.

Congratulations! You've run your first Toil workflow using the default *Batch System*, `singleMachine`, using the `file` job store.

Toil uses batch systems to manage the jobs it creates.

The `singleMachine` batch system is primarily used to prepare and debug workflows on a local machine. Once validated, try running them on a full-fledged batch system (see *Batch System API*). Toil supports many different batch systems such as Apache Mesos and Grid Engine; its versatility makes it easy to run your workflow in all kinds of places.

Toil is totally customizable! Run `python helloWorld.py --help` to see a complete list of available options.

For something beyond a "Hello, world!" example, refer to *A (more) real-world example*.

2.2 Running a basic CWL workflow

The Common Workflow Language (CWL) is an emerging standard for writing workflows that are portable across multiple workflow engines and platforms. Running CWL workflows using Toil is easy.

1. First ensure that Toil is installed with the `cwl` extra (see *Installing Toil with Extra Features*):

   ```
   (venv) $ pip install 'toil[cwl]'
   ```

   This installs the `toil-cwl-runner` executable.

2. Copy and paste the following code block into `example.cwl`:

   ```
   cwlVersion: v1.0
   class: CommandLineTool
   baseCommand: echo
   stdout: output.txt
   inputs:
     message:
       type: string
       inputBinding:
         position: 1
   outputs:
     output:
       type: stdout
   ```

   and this code into `example-job.yaml`:

   ```
   message: Hello world!
   ```

3. To run the workflow simply enter

   ```
   (venv) $ toil-cwl-runner example.cwl example-job.yaml
   ```

   Your output will be in `output.txt`:

   ```
   (venv) $ cat output.txt
   Hello world!
   ```

To learn more about CWL, see the CWL User Guide (from where this example was shamelessly borrowed).

To run this workflow on an AWS cluster have a look at *Running a CWL Workflow on AWS*.

For information on using CWL with Toil see the section *CWL in Toil*

2.3 Running a basic WDL workflow

The Workflow Description Language (WDL) is another emerging language for writing workflows that are portable across multiple workflow engines and platforms. Running WDL workflows using Toil is still in alpha, and currently experimental. Toil currently supports basic workflow syntax (see *WDL in Toil* for more details and examples). Here we go over running a basic WDL helloworld workflow.

1. First ensure that Toil is installed with the `wdl` extra (see *Installing Toil with Extra Features*):

   ```
   (venv) $ pip install 'toil[wdl]'
   ```

   This installs the `toil-wdl-runner` executable.

2. Copy and paste the following code block into `wdl-helloworld.wdl`:

   ```
       workflow write_simple_file {
         call write_file
       }
       task write_file {
         String message
         command { echo ${message} > wdl-helloworld-output.txt }
         output { File test = "wdl-helloworld-output.txt" }
       }

   and this code into ``wdl-helloworld.json``::

       {
         "write_simple_file.write_file.message": "Hello world!"
       }
   ```

3. To run the workflow simply enter

   ```
   (venv) $ toil-wdl-runner wdl-helloworld.wdl wdl-helloworld.json
   ```

   Your output will be in `wdl-helloworld-output.txt`:

   ```
   (venv) $ cat wdl-helloworld-output.txt
   Hello world!
   ```

To learn more about WDL, see the main WDL website .

2.4 A (more) real-world example

For a more detailed example and explanation, we've developed a sample pipeline that merge-sorts a temporary file. This is not supposed to be an efficient sorting program, rather a more fully worked example of what Toil is capable of.

2.4.1 Running the example

1. Download `the example code`
2. Run it with the default settings:

```
(venv) $ python sort.py file:jobStore
```

The workflow created a file called `sortedFile.txt` in your current directory. Have a look at it and notice that it contains a whole lot of sorted lines!

This workflow does a smart merge sort on a file it generates, `fileToSort.txt`. The sort is *smart* because each step of the process—splitting the file into separate chunks, sorting these chunks, and merging them back together—is compartmentalized into a **job**. Each job can specify its own resource requirements and will only be run after the jobs it depends upon have run. Jobs without dependencies will be run in parallel.

Note: Delete `fileToSort.txt` before moving on to #3. This example introduces options that specify dimensions for `fileToSort.txt`, if it does not already exist. If it exists, this workflow will use the existing file and the results will be the same as #2.

---

3. Run with custom options:

```
(venv) $ python sort.py file:jobStore \
             --numLines=5000 \
             --lineLength=10 \
             --overwriteOutput=True \
             --workDir=/tmp/
```

Here we see that we can add our own options to a Toil script. As noted above, the first two options, `--numLines` and `--lineLength`, determine the number of lines and how many characters are in each line. `--overwriteOutput` causes the current contents of `sortedFile.txt` to be overwritten, if it already exists. The last option, `--workDir`, is an option built into Toil to specify where temporary files unique to a job are kept.

2.4.2 Describing the source code

To understand the details of what's going on inside. Let's start with the `main()` function. It looks like a lot of code, but don't worry—we'll break it down piece by piece.

```
def main(options=None):
    if not options:
        # deal with command line arguments
        parser = ArgumentParser()
        Job.Runner.addToilOptions(parser)
        parser.add_argument('--numLines', default=defaultLines, help='Number of lines in␣
↪file to sort.', type=int)
        parser.add_argument('--lineLength', default=defaultLineLen, help='Length of␣
↪lines in file to sort.', type=int)
        parser.add_argument("--fileToSort", help="The file you wish to sort")
        parser.add_argument("--outputFile", help="Where the sorted output will go")
        parser.add_argument("--overwriteOutput", help="Write over the output file if it␣
↪already exists.", default=True)
        parser.add_argument("--N", dest="N",
                            help="The threshold below which a serial sort function is␣
```

(continues on next page)

(continued from previous page)

```
↪used to sort file. "
                                "All lines must of length less than or equal to N or␣
↪program will fail",
                            default=10000)
        parser.add_argument('--downCheckpoints', action='store_true',
                            help='If this option is set, the workflow will make␣
↪checkpoints on its way through'
                                 'the recursive "down" part of the sort')
        parser.add_argument("--sortMemory", dest="sortMemory",
                            help="Memory for jobs that sort chunks of the file.",
                            default=None)

        parser.add_argument("--mergeMemory", dest="mergeMemory",
                            help="Memory for jobs that collate results.",
                            default=None)

        options = parser.parse_args()
    if not hasattr(options, "sortMemory") or not options.sortMemory:
        options.sortMemory = sortMemory
    if not hasattr(options, "mergeMemory") or not options.mergeMemory:
        options.mergeMemory = sortMemory

    # do some input verification
    sortedFileName = options.outputFile or "sortedFile.txt"
    if not options.overwriteOutput and os.path.exists(sortedFileName):
        print(f'Output file {sortedFileName} already exists.  '
              f'Delete it to run the sort example again or use --overwriteOutput=True')
        exit()

    fileName = options.fileToSort
    if options.fileToSort is None:
        # make the file ourselves
        fileName = 'fileToSort.txt'
        if os.path.exists(fileName):
            print(f'Sorting existing file: {fileName}')
        else:
            print(f'No sort file specified. Generating one automatically called:
↪{fileName}.')
            makeFileToSort(fileName=fileName, lines=options.numLines, lineLen=options.
↪lineLength)
    else:
        if not os.path.exists(options.fileToSort):
            raise RuntimeError("File to sort does not exist: %s" % options.fileToSort)

    if int(options.N) <= 0:
        raise RuntimeError("Invalid value of N: %s" % options.N)

    # Now we are ready to run
    with Toil(options) as workflow:
        sortedFileURL = 'file://' + os.path.abspath(sortedFileName)
        if not workflow.options.restart:
            sortFileURL = 'file://' + os.path.abspath(fileName)
```

(continues on next page)

(continued from previous page)

```
            sortFileID = workflow.importFile(sortFileURL)
            sortedFileID = workflow.start(Job.wrapJobFn(setup,
                                                        sortFileID,
                                                        int(options.N),
                                                        options.downCheckpoints,
                                                        options=options,
                                                        memory=sortMemory))
        else:
            sortedFileID = workflow.restart()
        workflow.exportFile(sortedFileID, sortedFileURL)
```

First we make a parser to process command line arguments using the argparse module. It's important that we add the call to `Job.Runner.addToilOptions()` to initialize our parser with all of Toil's default options. Then we add the command line arguments unique to this workflow, and parse the input. The help message listed with the arguments should give you a pretty good idea of what they can do.

Next we do a little bit of verification of the input arguments. The option `--fileToSort` allows you to specify a file that needs to be sorted. If this option isn't given, it's here that we make our own file with the call to `makeFileToSort()`.

Finally we come to the context manager that initializes the workflow. We create a path to the input file prepended with `'file://'` as per the documentation for *toil.common.Toil()* when staging a file that is stored locally. Notice that we have to check whether or not the workflow is restarting so that we don't import the file more than once. Finally we can kick off the workflow by calling *toil.common.Toil.start()* on the job `setup`. When the workflow ends we capture its output (the sorted file's fileID) and use that in *toil.common.Toil.exportFile()* to move the sorted file from the job store back into "userland".

Next let's look at the job that begins the actual workflow, `setup`.

```
def setup(job, inputFile, N, downCheckpoints, options):
    """
    Sets up the sort.
    Returns the FileID of the sorted file
    """
    RealtimeLogger.info("Starting the merge sort")
    return job.addChildJobFn(down,
                             inputFile, N, 'root',
                             downCheckpoints,
                             options = options,
                             preemptible=True,
                             memory=sortMemory).rv()
```

`setup` really only does two things. First it writes to the logs using `Job.log()` and then calls `addChildJobFn()`. Child jobs run directly after the current job. This function turns the 'job function' `down` into an actual job and passes in the inputs including an optional resource requirement, `memory`. The job doesn't actually get run until the call to `Job.rv()`. Once the job `down` finishes, its output is returned here.

Now we can look at what `down` does.

```
def down(job, inputFileStoreID, N, path, downCheckpoints, options, memory=sortMemory):
    """
    Input is a file, a subdivision size N, and a path in the hierarchy of jobs.
    If the range is larger than a threshold N the range is divided recursively and
    a follow on job is then created which merges back the results else
    the file is sorted and placed in the output.
```

(continues on next page)

(continued from previous page)

```
    """

    RealtimeLogger.info("Down job starting: %s" % path)

    # Read the file
    inputFile = job.fileStore.readGlobalFile(inputFileStoreID, cache=False)
    length = os.path.getsize(inputFile)
    if length > N:
        # We will subdivide the file
        RealtimeLogger.critical("Splitting file: %s of size: %s"
                % (inputFileStoreID, length))
        # Split the file into two copies
        midPoint = getMidPoint(inputFile, 0, length)
        t1 = job.fileStore.getLocalTempFile()
        with open(t1, 'w') as fH:
            fH.write(copySubRangeOfFile(inputFile, 0, midPoint+1))
        t2 = job.fileStore.getLocalTempFile()
        with open(t2, 'w') as fH:
            fH.write(copySubRangeOfFile(inputFile, midPoint+1, length))
        # Call down recursively. By giving the rv() of the two jobs as inputs to the␣
↪follow-on job, up,
        # we communicate the dependency without hindering concurrency.
        result = job.addFollowOnJobFn(up,
                                      job.addChildJobFn(down, job.fileStore.
↪writeGlobalFile(t1), N, path + '/0',
                                                        downCheckpoints,␣
↪checkpoint=downCheckpoints, options=options,
                                                        preemptible=True, memory=options.
↪sortMemory).rv(),
                                      job.addChildJobFn(down, job.fileStore.
↪writeGlobalFile(t2), N, path + '/1',
                                                        downCheckpoints,␣
↪checkpoint=downCheckpoints, options=options,
                                                        preemptible=True, memory=options.
↪mergeMemory).rv(),
                                      path + '/up', preemptible=True, options=options,␣
↪memory=options.sortMemory).rv()
    else:
        # We can sort this bit of the file
        RealtimeLogger.critical("Sorting file: %s of size: %s"
                % (inputFileStoreID, length))
        # Sort the copy and write back to the fileStore
        shutil.copyfile(inputFile, inputFile + '.sort')
        sort(inputFile + '.sort')
        result = job.fileStore.writeGlobalFile(inputFile + '.sort')

    RealtimeLogger.info("Down job finished: %s" % path)
    return result
```

Down is the recursive part of the workflow. First we read the file into the local filestore by calling `job.fileStore.readGlobalFile()`. This puts a copy of the file in the temp directory for this particular job. This storage will disappear once this job ends. For a detailed explanation of the filestore, job store, and their interfaces have a look at *Managing files within a workflow*.

Next `down` checks the base case of the recursion: is the length of the input file less than `N` (remember `N` was an option we added to the workflow in `main`)? In the base case, we just sort the file, and return the file ID of this new sorted file.

If the base case fails, then the file is split into two new tempFiles using `job.fileStore.getLocalTempFile()` and the helper function `copySubRangeOfFile`. Finally we add a follow on Job `up` with `job.addFollowOnJobFn()`. We've already seen child jobs. A follow-on Job is a job that runs after the current job and *all* of its children (and their children and follow-ons) have completed. Using a follow-on makes sense because `up` is responsible for merging the files together and we don't want to merge the files together until we *know* they are sorted. Again, the return value of the follow-on job is requested using `Job.rv()`.

Looking at `up`

```
def up(job, inputFileID1, inputFileID2, path, options, memory=sortMemory):
    """
    Merges the two files and places them in the output.
    """

    RealtimeLogger.info("Up job starting: %s" % path)

    with job.fileStore.writeGlobalFileStream() as (fileHandle, outputFileStoreID):
        fileHandle = codecs.getwriter('utf-8')(fileHandle)
        with job.fileStore.readGlobalFileStream(inputFileID1) as inputFileHandle1:
            inputFileHandle1 = codecs.getreader('utf-8')(inputFileHandle1)
            with job.fileStore.readGlobalFileStream(inputFileID2) as inputFileHandle2:
                inputFileHandle2 = codecs.getreader('utf-8')(inputFileHandle2)
                RealtimeLogger.info("Merging %s and %s to %s"
                    % (inputFileID1, inputFileID2, outputFileStoreID))
                merge(inputFileHandle1, inputFileHandle2, fileHandle)
        # Cleanup up the input files - these deletes will occur after the completion is␣
↪successful.
        job.fileStore.deleteGlobalFile(inputFileID1)
        job.fileStore.deleteGlobalFile(inputFileID2)

        RealtimeLogger.info("Up job finished: %s" % path)

        return outputFileStoreID
```

we see that the two input files are merged together and the output is written to a new file using `job.fileStore.writeGlobalFileStream()`. After a little cleanup, the output file is returned.

Once the final `up` finishes and all of the `rv()` promises are fulfilled, `main` receives the sorted file's ID which it uses in `exportFile` to send it to the user.

There are other things in this example that we didn't go over such as *Checkpoints* and the details of much of the *Toil Class API*.

At the end of the script the lines

```
if __name__ == '__main__'
    main()
```

are included to ensure that the main function is only run once in the '__main__' process invoked by you, the user. In Toil terms, by invoking the script you created the *leader process* in which the `main()` function is run. A *worker process* is a separate process whose sole purpose is to host the execution of one or more jobs defined in that script. In any Toil workflow there is always one leader process, and potentially many worker processes.

When using the single-machine batch system (the default), the worker processes will be running on the same machine

as the leader process. With full-fledged batch systems like Mesos the worker processes will typically be started on separate machines. The boilerplate ensures that the pipeline is only started once—on the leader—but not when its job functions are imported and executed on the individual workers.

Typing `python sort.py --help` will show the complete list of arguments for the workflow which includes both Toil's and ones defined inside `sort.py`. A complete explanation of Toil's arguments can be found in *Commandline Options*.

2.4.3 Logging

By default, Toil logs a lot of information related to the current environment in addition to messages from the batch system and jobs. This can be configured with the `--logLevel` flag. For example, to only log `CRITICAL` level messages to the screen:

```
(venv) $ python sort.py file:jobStore \
             --logLevel=critical \
             --overwriteOutput=True
```

This hides most of the information we get from the Toil run. For more detail, we can run the pipeline with `--logLevel=debug` to see a comprehensive output. For more information, see *Commandline Options*.

2.4.4 Error Handling and Resuming Pipelines

With Toil, you can recover gracefully from a bug in your pipeline without losing any progress from successfully completed jobs. To demonstrate this, let's add a bug to our example code to see how Toil handles a failure and how we can resume a pipeline after that happens. Add a bad assertion at line 52 of the example (the first line of `down()`):

```
def down(job, inputFileStoreID, N, downCheckpoints, memory=sortMemory):
    ...
    assert 1 == 2, "Test error!"
```

When we run the pipeline, Toil will show a detailed failure log with a traceback:

```
(venv) $ python sort.py file:jobStore
...
---TOIL WORKER OUTPUT LOG---
...
m/j/jobonrSMP    Traceback (most recent call last):
m/j/jobonrSMP      File "toil/src/toil/worker.py", line 340, in main
m/j/jobonrSMP        job._runner(jobGraph=jobGraph, jobStore=jobStore,␣
↪fileStore=fileStore)
m/j/jobonrSMP      File "toil/src/toil/job.py", line 1270, in _runner
m/j/jobonrSMP        returnValues = self._run(jobGraph, fileStore)
m/j/jobonrSMP      File "toil/src/toil/job.py", line 1217, in _run
m/j/jobonrSMP        return self.run(fileStore)
m/j/jobonrSMP      File "toil/src/toil/job.py", line 1383, in run
m/j/jobonrSMP        rValue = userFunction(*((self,) + tuple(self._args)), **self._
↪kwargs)
m/j/jobonrSMP      File "toil/example.py", line 30, in down
m/j/jobonrSMP        assert 1 == 2, "Test error!"
m/j/jobonrSMP    AssertionError: Test error!
```

If we try and run the pipeline again, Toil will give us an error message saying that a job store of the same name already exists. By default, in the event of a failure, the job store is preserved so that the workflow can be restarted, starting from the previously failed jobs. We can restart the pipeline by running

```
(venv) $ python sort.py file:jobStore \
             --restart \
             --overwriteOutput=True
```

We can also change the number of times Toil will attempt to retry a failed job:

```
(venv) $ python sort.py file:jobStore \
             --retryCount 2 \
             --restart \
             --overwriteOutput=True
```

You'll now see Toil attempt to rerun the failed job until it runs out of tries. `--retryCount` is useful for non-systemic errors, like downloading a file that may experience a sporadic interruption, or some other non-deterministic failure.

To successfully restart our pipeline, we can edit our script to comment out line 30, or remove it, and then run

```
(venv) $ python sort.py file:jobStore \
             --restart \
             --overwriteOutput=True
```

The pipeline will run successfully, and the job store will be removed on the pipeline's completion.

2.4.5 Collecting Statistics

Please see the *Stats Command* section for more on gathering runtime and resource info on jobs.

2.5 Launching a Toil Workflow in AWS

After having installed the `aws` extra for Toil during the *Installation* and set up AWS (see *Preparing your AWS environment*), the user can run the basic `helloWorld.py` script (*Running a basic workflow*) on a VM in AWS just by modifying the run command.

Note that when running in AWS, users can either run the workflow on a single instance or run it on a cluster (which is running across multiple containers on multiple AWS instances). For more information on running Toil workflows on a cluster, see *Running in AWS*.

Also! Remember to use the *Destroy-Cluster Command* command when finished to destroy the cluster! Otherwise things may not be cleaned up properly.

1. Launch a cluster in AWS using the *Launch-Cluster Command* command:

   ```
   (venv) $ toil launch-cluster <cluster-name> \
                --keyPairName <AWS-key-pair-name> \
                --leaderNodeType t2.medium \
                --zone us-west-2a
   ```

   The arguments `keyPairName`, `leaderNodeType`, and `zone` are required to launch a cluster.

2. Copy `helloWorld.py` to the `/tmp` directory on the leader node using the *Rsync-Cluster Command* command:

```
(venv) $ toil rsync-cluster --zone us-west-2a <cluster-name> helloWorld.py :/tmp
```

Note that the command requires defining the file to copy as well as the target location on the cluster leader node.

3. Login to the cluster leader node using the *Ssh-Cluster Command* command:

```
(venv) $ toil ssh-cluster --zone us-west-2a <cluster-name>
```

Note that this command will log you in as the `root` user.

4. Run the Toil script in the cluster:

```
$ python /tmp/helloWorld.py aws:us-west-2:my-S3-bucket
```

In this particular case, we create an S3 bucket called `my-S3-bucket` in the `us-west-2` availability zone to store intermediate job results.

Along with some other `INFO` log messages, you should get the following output in your terminal window: `Hello, world!, here's a message:  You did it!`.

5. Exit from the SSH connection.

```
$ exit
```

6. Use the *Destroy-Cluster Command* command to destroy the cluster:

```
(venv) $ toil destroy-cluster --zone us-west-2a <cluster-name>
```

Note that this command will destroy the cluster leader node and any resources created to run the job, including the S3 bucket.

2.6 Running a CWL Workflow on AWS

After having installed the `aws` and `cwl` extras for Toil during the *Installation* and set up AWS (see *Preparing your AWS environment*), the user can run a CWL workflow with Toil on AWS.

Also! Remember to use the *Destroy-Cluster Command* command when finished to destroy the cluster! Otherwise things may not be cleaned up properly.

1. First launch a node in AWS using the *Launch-Cluster Command* command:

```
(venv) $ toil launch-cluster <cluster-name> \
            --keyPairName <AWS-key-pair-name> \
            --leaderNodeType t2.medium \
            --zone us-west-2a
```

2. Copy `example.cwl` and `example-job.yaml` from the *CWL example* to the node using the *Rsync-Cluster Command* command:

```
(venv) $ toil rsync-cluster --zone us-west-2a <cluster-name> example.cwl :/tmp
(venv) $ toil rsync-cluster --zone us-west-2a <cluster-name> example-job.yaml :/tmp
```

3. SSH into the cluster's leader node using the *Ssh-Cluster Command* utility:

```
(venv) $ toil ssh-cluster --zone us-west-2a <cluster-name>
```

4. Once on the leader node, it's a good idea to update and install the following:

```
sudo apt-get update
sudo apt-get -y upgrade
sudo apt-get -y dist-upgrade
sudo apt-get -y install git
sudo pip install mesos.cli
```

5. Now create a new `virtualenv` with the `--system-site-packages` option and activate:

```
virtualenv --system-site-packages venv
source venv/bin/activate
```

6. Now run the CWL workflow:

```
(venv) $ toil-cwl-runner \
             --provisioner aws \
             --jobStore aws:us-west-2a:any-name \
             /tmp/example.cwl /tmp/example-job.yaml
```

Tip: When running a CWL workflow on AWS, input files can be provided either on the local file system or in S3 buckets using `s3://` URI references. Final output files will be copied to the local file system of the leader node.

---

7. Finally, log out of the leader node and from your local computer, destroy the cluster:

```
(venv) $ toil destroy-cluster --zone us-west-2a <cluster-name>
```

2.7 Running a Workflow with Autoscaling - Cactus

Cactus is a reference-free, whole-genome multiple alignment program that can be run on any of the cloud platforms Toil supports.

Note: Cloud Independence:

This example provides a "cloud agnostic" view of running Cactus with Toil. Most options will not change between cloud providers. However, each provisioner has unique inputs for `--leaderNodeType`, `--nodeType` and `--zone`. We recommend the following:

Option	Used in	AWS	Google
`--leaderNodeType`	launch-cluster	t2.medium	n1-standard-1
`--zone`	launch-cluster	us-west-2a	us-west1-a
`--zone`	cactus	us-west-2	
`--nodeType`	cactus	c3.4xlarge	n1-standard-8

When executing `toil launch-cluster` with `gce` specified for `--provisioner`, the option `--boto` must be specified and given a path to your .boto file. See *Running in Google Compute Engine (GCE)* for more information about the `--boto` option.

Also! Remember to use the *Destroy-Cluster Command* command when finished to destroy the cluster! Otherwise things may not be cleaned up properly.

1. Download `pestis.tar.gz`
2. Launch a leader node using the *Launch-Cluster Command* command:

```
(venv) $ toil launch-cluster <cluster-name> \
            --provisioner <aws, gce> \
            --keyPairName <key-pair-name> \
            --leaderNodeType <type> \
            --zone <zone>
```

Note: A Helpful Tip

When using AWS, setting the environment variable eliminates having to specify the `--zone` option for each command. This will be supported for GCE in the future.

```
(venv) $ export TOIL_AWS_ZONE=us-west-2c
```

---

3. Create appropriate directory for uploading files:

```
(venv) $ toil ssh-cluster --provisioner <aws, gce> <cluster-name>
$ mkdir /root/cact_ex
$ exit
```

4. Copy the required files, i.e., seqFile.txt (a text file containing the locations of the input sequences as well as their phylogenetic tree, see here), organisms' genome sequence files in FASTA format, and configuration files (e.g. blockTrim1.xml, if desired), up to the leader node:

```
(venv) $ toil rsync-cluster --provisioner <aws, gce> <cluster-name> pestis-short-
↪aws-seqFile.txt :/root/cact_ex
(venv) $ toil rsync-cluster --provisioner <aws, gce> <cluster-name> GCF_000169655.1_
↪ASM16965v1_genomic.fna :/root/cact_ex
(venv) $ toil rsync-cluster --provisioner <aws, gce> <cluster-name> GCF_000006645.1_
↪ASM664v1_genomic.fna :/root/cact_ex
(venv) $ toil rsync-cluster --provisioner <aws, gce> <cluster-name> GCF_000182485.1_
↪ASM18248v1_genomic.fna :/root/cact_ex
(venv) $ toil rsync-cluster --provisioner <aws, gce> <cluster-name> GCF_000013805.1_
↪ASM1380v1_genomic.fna :/root/cact_ex
(venv) $ toil rsync-cluster --provisioner <aws, gce> <cluster-name> setup_
↪leaderNode.sh :/root/cact_ex
(venv) $ toil rsync-cluster --provisioner <aws, gce> <cluster-name> blockTrim1.xml␣
↪:/root/cact_ex
(venv) $ toil rsync-cluster --provisioner <aws, gce> <cluster-name> blockTrim3.xml␣
↪:/root/cact_ex
```

5. Log in to the leader node:

```
(venv) $ toil ssh-cluster --provisioner <aws, gce> <cluster-name>
```

6. Set up the environment of the leader node to run Cactus:

```
$ bash /root/cact_ex/setup_leaderNode.sh
$ source cact_venv/bin/activate
(cact_venv) $ cd cactus
(cact_venv) $ pip install --upgrade .
```

7. Run Cactus as an autoscaling workflow:

```
(cact_venv) $ TOIL_APPLIANCE_SELF=quay.io/ucsc_cgl/toil:3.14.0 cactus \
                  --provisioner <aws, gce> \
                  --nodeType <type> \
                  --maxNodes 2 \
                  --minNodes 0 \
                  --retry 10 \
                  --batchSystem mesos \
                  --logDebug \
                  --logFile /logFile_pestis3 \
                  --configFile \
                  /root/cact_ex/blockTrim3.xml <aws, google>:<zone>:cactus-pestis \
                  /root/cact_ex/pestis-short-aws-seqFile.txt \
                  /root/cact_ex/pestis_output3.hal
```

Note: Pieces of the Puzzle:

`TOIL_APPLIANCE_SELF=quay.io/ucsc_cgl/toil:3.14.0` — specifies the version of Toil being used, 3.14.0; if the latest one is desired, please eliminate.

`--nodeType` — determines the instance type used for worker nodes. The instance type specified here must be on the same cloud provider as the one specified with `--leaderNodeType`

`--maxNodes 2` — creates up to two instances of the type specified with `--nodeType` and launches Mesos worker containers inside them.

`--logDebug` — equivalent to `--logLevel DEBUG`.

`--logFile /logFile_pestis3` — writes logs in a file named *logFile_pestis3* under `/` folder.

`--configFile` — this is not required depending on whether a specific configuration file is intended to run the alignment.

`<aws, google>:<zone>:cactus-pestis` — creates a bucket, named `cactus-pestis`, with the specified cloud provider to store intermediate job files and metadata. **NOTE**: If you want to use a GCE-based jobstore, specify `google` here, not `gce`.

The result file, named `pestis_output3.hal`, is stored under `/root/cact_ex` folder of the leader node.

Use `cactus --help` to see all the Cactus and Toil flags available.

---

8. Log out of the leader node:

```
(cact_venv) $ exit
```

9. Download the resulted output to local machine:

```
(venv) $ toil rsync-cluster \
             --provisioner <aws, gce> <cluster-name> \
             :/root/cact_ex/pestis_output3.hal \
             <path-of-folder-on-local-machine>
```

10. Destroy the cluster:

```
(venv) $ toil destroy-cluster --provisioner <aws, gce> <cluster-name>
```

CHAPTER

THREE

INTRODUCTION

Toil runs in various environments, including *locally* and *in the cloud* (Amazon Web Services and Google Compute Engine). Toil also supports two DSLs: *CWL* and (Amazon Web Services and Google Compute Engine). Toil also supports two DSLs: *CWL* and *WDL* (experimental).

Toil is built in a modular way so that it can be used on lots of different systems, and with different configurations. The three configurable pieces are the

- *Job Store API*: A filepath or url that can host and centralize all files for a workflow (e.g. a local folder, or an AWS s3 bucket url).
- *Batch System API*: Specifies either a local single-machine or a currently supported HPC environment (lsf, parasol, mesos, slurm, torque, htcondor, kubernetes, or grid_engine). Mesos is a special case, and is launched for cloud environments.
- *Provisioner*: For running in the cloud only. This specifies which cloud provider provides instances to do the "work" of your workflow.

3.1 Job Store

The job store is a storage abstraction which contains all of the information used in a Toil run. This centralizes all of the files used by jobs in the workflow and also the details of the progress of the run. If a workflow crashes or fails, the job store contains all of the information necessary to resume with minimal repetition of work.

Several different job stores are supported, including the file job store and cloud job stores.

3.1.1 File Job Store

The file job store is for use locally, and keeps the workflow information in a directory on the machine where the workflow is launched. This is the simplest and most convenient job store for testing or for small runs.

For an example that uses the file job store, see *Running a basic workflow*.

3.1.2 Cloud Job Stores

Toil currently supports the following cloud storage systems as job stores:

- *AWS Job Store*: An AWS S3 bucket formatted as "aws:<zone>:<bucketname>" where only numbers, letters, and dashes are allowed in the bucket name. Example: *aws:us-west-2:my-aws-jobstore-name*.
- *Google Job Store*: A Google Cloud Storage bucket formatted as "gce:<zone>:<bucketname>" where only numbers, letters, and dashes are allowed in the bucket name. Example: *gce:us-west2-a:my-google-jobstore-name*.

These use cloud buckets to house all of the files. This is useful if there are several different worker machines all running jobs that need to access the job store.

3.2 Batch System

A Toil batch system is either a local single-machine (one computer) or a currently supported HPC cluster of computers (lsf, parasol, mesos, slurm, torque, htcondor, or grid_engine). Mesos is a special case, and is launched for cloud environments. These environments manage individual worker nodes under a leader node to process the work required in a workflow. The leader and its workers all coordinate their tasks and files through a centralized job store location.

See *Batch System API* for a more detailed description of different batch systems.

3.3 Provisioner

The Toil provisioner provides a tool set for running a Toil workflow on a particular cloud platform.

The *Cluster Utilities* are command line tools used to provision nodes in your desired cloud platform. They allows you to launch nodes, ssh to the leader, and rsync files back and forth.

For detailed instructions for using the provisioner see *Running in AWS* or *Running in Google Compute Engine (GCE)*.

CHAPTER

FOUR

COMMANDLINE OPTIONS

A quick way to see all of Toil's commandline options is by executing the following on a toil script:

```
$ toil example.py --help
```

For a basic toil workflow, Toil has one mandatory argument, the job store. All other arguments are optional.

4.1 The Job Store

Running toil scripts requires a filepath or url to a centralizing location for all of the files of the workflow. This is Toil's one required positional argument: the job store. To use the *quickstart* example, if you're on a node that has a large **/scratch** volume, you can specify that the jobstore be created there by executing: `python HelloWorld.py /scratch/my-job-store`, or more explicitly, `python HelloWorld.py file:/scratch/my-job-store`.

Syntax for specifying different job stores:

Local: `file:job-store-name`

AWS: `aws:region-here:job-store-name`

Google: `google:projectID-here:job-store-name`

Different types of job store options can be found below.

4.2 Commandline Options

Core Toil Options Options to specify the location of the Toil workflow and turn on stats collation about the performance of jobs.

--workDir WORKDIR Absolute path to directory where temporary files generated during the Toil run should be placed. Standard output and error from batch system jobs (unless –noStdOutErr) will be placed in this directory. A cache directory may be placed in this directory. Temp files and folders will be placed in a toil-<workflowID> within workDir. The workflowID is generated by Toil and will be reported in the workflow logs. Default is determined by the variables (TMPDIR, TEMP, TMP) via mkdtemp. This directory needs to exist on all machines running jobs; if capturing standard output and error from batch system jobs is desired, it will generally need to be on a shared file system. When sharing a cache between containers on a host, this directory must be shared between the containers.

--coordinationDir COORDINATION_DIR Absolute path to directory where Toil will keep state and lock files. When sharing a cache between containers on a host, this directory must be shared between the containers.

--noStdOutErr Do not capture standard output and error from batch system jobs.

--stats Records statistics about the toil workflow to be used by 'toil stats'.

--clean=STATE Determines the deletion of the jobStore upon completion of the program. Choices: 'always', 'onError','never', or 'onSuccess'. The -\-stats option requires information from the jobStore upon completion so the jobStore will never be deleted with that flag. If you wish to be able to restart the run, choose 'never' or 'onSuccess'. Default is 'never' if stats is enabled, and 'onSuccess' otherwise

--cleanWorkDir STATE Determines deletion of temporary worker directory upon completion of a job. Choices: 'always', 'onError', 'never', or 'onSuccess'. Default = always. WARNING: This option should be changed for debugging only. Running a full pipeline with this option could fill your disk with intermediate data.

--clusterStats FILEPATH If enabled, writes out JSON resource usage statistics to a file. The default location for this file is the current working directory, but an absolute path can also be passed to specify where this file should be written. This option only applies when using scalable batch systems.

--restart If -\-restart is specified then will attempt to restart existing workflow at the location pointed to by the -\-jobStore option. Will raise an exception if the workflow does not exist.

Logging Options Toil hides stdout and stderr by default except in case of job failure. Log levels in toil are based on priority from the logging module:

--logOff Only CRITICAL log levels are shown. Equivalent to `--logLevel=OFF` or `--logLevel=CRITICAL`.

--logCritical Only CRITICAL log levels are shown. Equivalent to `--logLevel=OFF` or `--logLevel=CRITICAL`.

--logError Only ERROR, and CRITICAL log levels are shown. Equivalent to `--logLevel=ERROR`.

--logWarning Only WARN, ERROR, and CRITICAL log levels are shown. Equivalent to `--logLevel=WARNING`.

--logInfo All log statements are shown, except DEBUG. Equivalent to `--logLevel=INFO`.

--logDebug All log statements are shown. Equivalent to `--logLevel=DEBUG`.

--logLevel=LOGLEVEL May be set to: `OFF` (or `CRITICAL`), `ERROR`, `WARN` (or `WARNING`), `INFO`, or `DEBUG`.

--logFile FILEPATH Specifies a file path to write the logging output to.

--rotatingLogging Turn on rotating logging, which prevents log files from getting too big (set using `--maxLogFileSize BYTESIZE`).

--maxLogFileSize BYTESIZE The maximum size of a job log file to keep (in bytes), log files larger than this will be truncated to the last X bytes. Setting this option to zero will prevent any truncation. Setting this option to a negative value will truncate from the beginning. Default=62.5KiB

Sets the maximum log file size in bytes (`--rotatingLogging` must be active).

--log-dir DIRPATH For CWL and local file system only. Log stdout and stderr (if tool requests stdout/stderr) to the DIRPATH.

Batch System Options

--batchSystem BATCHSYSTEM The type of batch system to run the job(s) with, currently can be one of aws_batch, parasol, single_machine, grid_engine, lsf, mesos, slurm, tes, torque, htcondor, kubernetes. (default: single_machine)

--disableAutoDeployment Should auto-deployment of the user script be deactivated? If True, the user script/package should be present at the same location on all workers. Default = False.

--maxJobs MAXJOBS Specifies the maximum number of jobs to submit to the backing scheduler at once. Not supported on Mesos or AWS Batch. Use 0 for unlimited. Defaults to unlimited.

--maxLocalJobs MAXLOCALJOBS Specifies the maximum number of housekeeping jobs to run simultaneously on the local system. Use 0 for unlimited. Defaults to the number of local cores.

--manualMemArgs Do not add the default arguments: 'hv=MEMORY' & 'h_vmem=MEMORY' to the qsub call, and instead rely on TOIL_GRIDGENGINE_ARGS to supply alternative arguments. Requires that TOIL_GRIDGENGINE_ARGS be set.

--runCwlInternalJobsOnWorkers Whether to run CWL internal jobs (e.g. CWLScatter) on the worker nodes instead of the primary node. If false (default), then all such jobs are run on the primary node. Setting this to true can speed up the pipeline for very large workflows with many sub-workflows and/or scatters, provided that the worker pool is large enough.

--statePollingWait STATEPOLLINGWAIT Time, in seconds, to wait before doing a scheduler query for job state. Return cached results if within the waiting period. Only works for grid engine batch systems such as gridengine, htcondor, torque, slurm, and lsf.

--batchLogsDir BATCHLOGSDIR Directory to tell the backing batch system to log into. Should be available on both the leader and the workers, if the backing batch system writes logs to the worker machines' filesystems, as many HPC schedulers do. If unset, the Toil work directory will be used. Only works for grid engine batch systems such as gridengine, htcondor, torque, slurm, and lsf.

--parasolCommand PARASOLCOMMAND The name or path of the parasol program. Will be looked up on PATH unless it starts with a slash. (default: parasol)

--parasolMaxBatches PARASOLMAXBATCHES Maximum number of job batches the Parasol batch is allowed to create. One batch is created for jobs with a unique set of resource requirements. (default: 1000)

--mesosEndpoint MESOSENDPOINT The host and port of the Mesos server separated by a colon. (default: <leader IP>:5050)

--mesosFrameworkId MESOSFRAMEWORKID Use a specific Mesos framework ID.

--mesosRole MESOSROLE Use a Mesos role.

--mesosName MESOSNAME The Mesos name to use. (default: toil)

--kubernetesHostPath KUBERNETES_HOST_PATH Path on Kubernetes hosts to use as shared inter-pod temp directory.

--kubernetesOwner KUBERNETES_OWNER Username to mark Kubernetes jobs with.

--kubernetesServiceAccount KUBERNETES_SERVICE_ACCOUNT Service account to run jobs as.

--kubernetesPodTimeout KUBERNETES_POD_TIMEOUT Seconds to wait for a scheduled Kubernetes pod to start running. (default: 120s)

--tesEndpoint TES_ENDPOINT The http(s) URL of the TES server. (default: http://<leader IP>:8000)

--tesUser TES_USER User name to use for basic authentication to TES server.

--tesPassword TES_PASSWORD Password to use for basic authentication to TES server.

--tesBearerToken TES_BEARER_TOKEN Bearer token to use for authentication to TES server.

--awsBatchRegion AWS_BATCH_REGION The AWS region containing the AWS Batch queue to submit to.

--awsBatchQueue AWS_BATCH_QUEUE The name or ARN of the AWS Batch queue to submit to.

--awsBatchJobRoleArn AWS_BATCH_JOB_ROLE_ARN The ARN of an IAM role to run AWS Batch jobs as, so they can e.g. access a job store. Must be assumable by ecs-tasks.amazonaws.com

--scale SCALE A scaling factor to change the value of all submitted tasks' submitted cores. Used in single_machine batch system. Useful for running workflows on smaller machines than they were designed for, by setting a value less than 1. (default: 1)

Data Storage Options Allows configuring Toil's data storage.

--linkImports When using a filesystem based job store, CWL input files are by default symlinked in. Specifying this option instead copies the files into the job store, which may protect them from being modified externally. When not specified and as long as caching is enabled, Toil will protect the file automatically by changing the permissions to read-only.

--moveExports When using a filesystem based job store, output files are by default moved to the output directory, and a symlink to the moved exported file is created at the initial location. Specifying this option instead copies the files into the output directory. Applies to filesystem-based job stores only.

--disableCaching Disables caching in the file store. This flag must be set to use a batch system that does not support cleanup, such as Parasol.

--caching BOOL Set caching options. This must be set to "false" to use a batch system that does not support cleanup, such as Parasol. Set to "true" if caching is desired.

Autoscaling Options Allows the specification of the minimum and maximum number of nodes in an autoscaled cluster, as well as parameters to control the level of provisioning.

--provisioner CLOUDPROVIDER The provisioner for cluster auto-scaling. This is the main Toil -\-provisioner option, and defaults to None for running on single_machine and non-auto-scaling batch systems. The currently supported choices are 'aws' or 'gce'.

--nodeTypes NODETYPES Specifies a list of comma-separated node types, each of which is composed of slash-separated instance types, and an optional spot bid set off by a colon, making the node type preemptible. Instance types may appear in multiple node types, and the same node type may appear as both preemptible and non-preemptible.

Valid argument specifying two node types:
c5.4xlarge/c5a.4xlarge:0.42,t2.large

Node types:
c5.4xlarge/c5a.4xlarge:0.42 and t2.large

Instance types:
c5.4xlarge, c5a.4xlarge, and t2.large

Semantics:
Bid $0.42/hour for either c5.4xlarge or c5a.4xlarge instances, treated interchangeably, while they are available at that price, and buy t2.large instances at full price

--minNodes MINNODES Minimum number of nodes of each type in the cluster, if using auto-scaling. This should be provided as a comma-separated list of the same length as the list of node types. default=0

--maxNodes MAXNODES Maximum number of nodes of each type in the cluster, if using autoscaling, provided as a comma-separated list. The first value is used as a default if the list length is less than the number of nodeTypes. default=10

--targetTime TARGETTIME Sets how rapidly you aim to complete jobs in seconds. Shorter times mean more aggressive parallelization. The autoscaler attempts to scale up/down so that it expects all queued jobs will complete within targetTime seconds. (Default: 1800)

--betaInertia BETAINERTIA A smoothing parameter to prevent unnecessary oscillations in the number of provisioned nodes. This controls an exponentially weighted moving average of the estimated number of nodes. A value of 0.0 disables any smoothing, and a value of 0.9 will smooth so much that few changes will ever be made. Must be between 0.0 and 0.9. (Default: 0.1)

--scaleInterval SCALEINTERVAL The interval (seconds) between assessing if the scale of the cluster needs to change. (Default: 60)

--preemptibleCompensation PREEMPTIBLECOMPENSATION The preference of the autoscaler to replace preemptible nodes with non-preemptible nodes, when preemptible nodes cannot be started for some reason. Defaults to 0.0. This value must be between 0.0 and 1.0, inclusive. A value of 0.0 disables such compensation, a value of 0.5 compensates two missing preemptible nodes with a non-preemptible one. A value of 1.0 replaces every missing pre-emptable node with a non-preemptible one.

--nodeStorage NODESTORAGE Specify the size of the root volume of worker nodes when they are launched in gigabytes. You may want to set this if your jobs require a lot of disk space. The default value is 50.

--nodeStorageOverrides NODESTORAGEOVERRIDES Comma-separated list of nodeType:nodeStorage that are used to override the default value from -\-nodeStorage for the specified nodeType(s). This is useful for heterogeneous jobs where some tasks require much more disk than others.

--metrics Enable the prometheus/grafana dashboard for monitoring CPU/RAM usage, queue size, and issued jobs.

--assumeZeroOverhead Ignore scheduler and OS overhead and assume jobs can use every last byte of memory and disk on a node when autoscaling.

Service Options Allows the specification of the maximum number of service jobs in a cluster. By keeping this limited we can avoid nodes occupied with services causing deadlocks. (Not for CWL).

--maxServiceJobs MAXSERVICEJOBS The maximum number of service jobs that can be run concurrently, excluding service jobs running on preemptible nodes. default=9223372036854775807

--maxPreemptibleServiceJobs MAXPREEMPTIBLESERVICEJOBS The maximum number of service jobs that can run concurrently on preemptible nodes. default=9223372036854775807

--deadlockWait DEADLOCKWAIT Time, in seconds, to tolerate the workflow running only the same service jobs, with no jobs to use them, before declaring the workflow to be deadlocked and stopping. default=60

--deadlockCheckInterval DEADLOCKCHECKINTERVAL Time, in seconds, to wait between checks to see if the workflow is stuck running only service jobs, with no jobs to use them. Should be shorter than -\-deadlockWait. May need to be increased if the batch system cannot enumerate running jobs quickly enough, or if polling for running jobs is placing an unacceptable load on a shared cluster. default=30

Resource Options The options to specify default cores/memory requirements (if not specified by the jobs themselves), and to limit the total amount of memory/cores requested from the batch system.

--defaultMemory INT The default amount of memory to request for a job. Only applicable to jobs that do not specify an explicit value for this requirement. Standard suffixes like K, Ki, M, Mi, G or Gi are supported. Default is 2.0G

--defaultCores FLOAT The default number of CPU cores to dedicate a job. Only applicable to jobs that do not specify an explicit value for this requirement. Fractions of a core (for example 0.1) are supported on some batch systems, namely Mesos and singleMachine. Default is 1.0

--defaultDisk INT The default amount of disk space to dedicate a job. Only applicable to jobs that do not specify an explicit value for this requirement. Standard suffixes like K, Ki, M, Mi, G or Gi are supported. Default is 2.0G

--defaultAccelerators ACCELERATOR The default amount of accelerators to request for a job. Only applicable to jobs that do not specify an explicit value for this requirement. Each accelerator specification can have a type (gpu [default], nvidia, amd, cuda, rocm, opencl, or a specific model

like nvidia-tesla-k80), and a count [default: 1]. If both a type and a count are used, they must be separated by a colon. If multiple types of accelerators are used, the specifications are separated by commas. Default is [].

--defaultPreemptible BOOL Make all jobs able to run on preemptible (spot) nodes by default.

--maxCores INT The maximum number of CPU cores to request from the batch system at any one time. Standard suffixes like K, Ki, M, Mi, G or Gi are supported.

--maxMemory INT The maximum amount of memory to request from the batch system at any one time. Standard suffixes like K, Ki, M, Mi, G or Gi are supported.

--maxDisk INT The maximum amount of disk space to request from the batch system at any one time. Standard suffixes like K, Ki, M, Mi, G or Gi are supported.

Options for rescuing/killing/restarting jobs. The options for jobs that either run too long/fail or get lost (some batch systems have issues!).

--retryCount RETRYCOUNT Number of times to retry a failing job before giving up and labeling job failed. default=1

--enableUnlimitedPreemptibleRetries If set, preemptible failures (or any failure due to an instance getting unexpectedly terminated) will not count towards job failures and -\-retryCount.

--doubleMem If set, batch jobs which die due to reaching memory limit on batch schedulers will have their memory doubled and they will be retried. The remaining retry count will be reduced by 1. Currently only supported by LSF. default=False.

--maxJobDuration MAXJOBDURATION Maximum runtime of a job (in seconds) before we kill it (this is a lower bound, and the actual time before killing the job may be longer).

--rescueJobsFrequency RESCUEJOBSFREQUENCY Period of time to wait (in seconds) between checking for missing/overlong jobs, that is jobs which get lost by the batch system. Expert parameter.

Log Management Options

--maxLogFileSize MAXLOGFILESIZE The maximum size of a job log file to keep (in bytes), log files larger than this will be truncated to the last X bytes. Setting this option to zero will prevent any truncation. Setting this option to a negative value will truncate from the beginning. Default=62.5 K

--writeLogs FILEPATH Write worker logs received by the leader into their own files at the specified path. Any non-empty standard output and error from failed batch system jobs will also be written into files at this path. The current working directory will be used if a path is not specified explicitly. Note: By default only the logs of failed jobs are returned to leader. Set log level to 'debug' or enable -\-writeLogsFromAllJobs to get logs back from successful jobs, and adjust -\-maxLogFileSize to control the truncation limit for worker logs.

--writeLogsGzip FILEPATH Identical to -\-writeLogs except the logs files are gzipped on the leader.

--writeMessages FILEPATH File to send messages from the leader's message bus to.

--realTimeLogging Enable real-time logging from workers to leader.

Miscellaneous Options

--disableChaining Disables chaining of jobs (chaining uses one job's resource allocation for its successor job if possible).

--disableJobStoreChecksumVerification Disables checksum verification for files transferred to/from the job store. Checksum verification is a safety check to ensure the data is not corrupted during transfer. Currently only supported for non-streaming AWS files

--sseKey SSEKEY Path to file containing 32 character key to be used for server-side encryption on awsJobStore or googleJobStore. SSE will not be used if this flag is not passed.

--setEnv NAME, -e NAME NAME=VALUE or NAME, -e NAME=VALUE or NAME are also valid. Set an environment variable early on in the worker. If VALUE is omitted, it will be looked up in the current environment. Independently of this option, the worker will try to emulate the leader's environment before running a job, except for some variables known to vary across systems. Using this option, a variable can be injected into the worker process itself before it is started.

--servicePollingInterval SERVICEPOLLINGINTERVAL Interval of time service jobs wait between polling for the existence of the keep-alive flag (default=60)

--forceDockerAppliance Disables sanity checking the existence of the docker image specified by TOIL_APPLIANCE_SELF, which Toil uses to provision mesos for autoscaling.

--statusWait INT Seconds to wait between reports of running jobs. (default=3600)

--disableProgress Disables the progress bar shown when standard error is a terminal.

Debug Options Debug options for finding problems or helping with testing.

--debugWorker Experimental no forking mode for local debugging. Specifically, workers are not forked and stderr/stdout are not redirected to the log. (default=False)

--disableWorkerOutputCapture Let worker output go to worker's standard out/error instead of per-job logs.

--badWorker BADWORKER For testing purposes randomly kill -\-badWorker proportion of jobs using SIGKILL. (Default: 0.0)

--badWorkerFailInterval BADWORKERFAILINTERVAL When killing the job pick uniformly within the interval from 0.0 to -\-badWorkerFailInterval seconds after the worker starts. (Default: 0.01)

--kill_polling_interval KILL_POLLING_INTERVAL Interval of time (in seconds) the leader waits between polling for the kill flag inside the job store set by the "toil kill" command. (default=5)

4.3 Restart Option

In the event of failure, Toil can resume the pipeline by adding the argument `--restart` and rerunning the python script. Toil pipelines (but not CWL pipelines) can even be edited and resumed which is useful for development or troubleshooting.

4.4 Running Workflows with Services

Toil supports jobs, or clusters of jobs, that run as *services* to other *accessor* jobs. Example services include server databases or Apache Spark Clusters. As service jobs exist to provide services to accessor jobs their runtime is dependent on the concurrent running of their accessor jobs. The dependencies between services and their accessor jobs can create potential deadlock scenarios, where the running of the workflow hangs because only service jobs are being run and their accessor jobs can not be scheduled because of too limited resources to run both simultaneously. To cope with this situation Toil attempts to schedule services and accessors intelligently, however to avoid a deadlock with workflows running service jobs it is advisable to use the following parameters:

- `--maxServiceJobs`: The maximum number of service jobs that can be run concurrently, excluding service jobs running on preemptible nodes.
- `--maxPreemptibleServiceJobs`: The maximum number of service jobs that can run concurrently on preemptible nodes.

Specifying these parameters so that at a maximum cluster size there will be sufficient resources to run accessors in addition to services will ensure that such a deadlock can not occur.

If too low a limit is specified then a deadlock can occur in which toil can not schedule sufficient service jobs concurrently to complete the workflow. Toil will detect this situation if it occurs and throw a `toil.DeadlockException` exception. Increasing the cluster size and these limits will resolve the issue.

4.5 Setting Options directly with the Toil Script

It's good to remember that commandline options can be overridden in the Toil script itself. For example, *toil.job.Job.Runner.getDefaultOptions()* can be used to run toil with all default options, and in this example, it will override commandline args to run the default options and always run with the "./toilWorkflow" directory specified as the jobstore:

```
options = Job.Runner.getDefaultOptions("./toilWorkflow") # Get the options object

with Toil(options) as toil:
    toil.start(Job())  # Run the script
```

However, each option can be explicitly set within the script by supplying arguments (in this example, we are setting `logLevel = "DEBUG"` (all log statements are shown) and `clean="ALWAYS"` (always delete the jobstore) like so:

```
options = Job.Runner.getDefaultOptions("./toilWorkflow") # Get the options object
options.logLevel = "DEBUG" # Set the log level to the debug level.
options.clean = "ALWAYS" # Always delete the jobStore after a run

with Toil(options) as toil:
    toil.start(Job())  # Run the script
```

However, the usual incantation is to accept commandline args from the user with the following:

```
parser = Job.Runner.getDefaultArgumentParser() # Get the parser
options = parser.parse_args() # Parse user args to create the options object

with Toil(options) as toil:
    toil.start(Job())  # Run the script
```

Which can also, of course, then accept script supplied arguments as before (which will overwrite any user supplied args):

```
parser = Job.Runner.getDefaultArgumentParser() # Get the parser
options = parser.parse_args() # Parse user args to create the options object
options.logLevel = "DEBUG" # Set the log level to the debug level.
options.clean = "ALWAYS" # Always delete the jobStore after a run

with Toil(options) as toil:
    toil.start(Job())  # Run the script
```

CHAPTER

FIVE

TOIL DEBUGGING

Toil has a number of tools to assist in debugging. Here we provide help in working through potential problems that a user might encounter in attempting to run a workflow.

5.1 Introspecting the Jobstore

Note: Currently these features are only implemented for use locally (single machine) with the fileJobStore.

To view what files currently reside in the jobstore, run the following command:

```
$ toil debug-file file:path-to-jobstore-directory \
      --listFilesInJobStore
```

When run from the commandline, this should generate a file containing the contents of the job store (in addition to displaying a series of log messages to the terminal). This file is named "jobstore_files.txt" by default and will be generated in the current working directory.

If one wishes to copy any of these files to a local directory, one can run for example:

```
$ toil debug-file file:path-to-jobstore \
      --fetch overview.txt *.bam *.fastq \
      --localFilePath=/home/user/localpath
```

To fetch `overview.txt`, and all `.bam` and `.fastq` files. This can be used to recover previously used input and output files for debugging or reuse in other workflows, or use in general debugging to ensure that certain outputs were imported into the jobStore.

5.2 Stats and Status

See *Stats Command* for more about gathering statistics about job success, runtime, and resource usage from workflows.

5.3 Using a Python debugger

If you execute a workflow using the `--debugWorker` flag, Toil will not fork in order to run jobs, which means you can either use pdb, or an IDE that supports debugging Python as you would normally. Note that the `--debugWorker` flag will only work with the `singleMachine` batch system (the default), and not any of the custom job schedulers.

CHAPTER

SIX

RUNNING IN THE CLOUD

Toil supports Amazon Web Services (AWS) and Google Compute Engine (GCE) in the cloud and has autoscaling capabilities that can adapt to the size of your workflow, whether your workflow requires 10 instances or 20,000.

Toil does this by creating a virtual cluster with Apache Mesos. Apache Mesos requires a leader node to coordinate the workflow, and worker nodes to execute the various tasks within the workflow. As the workflow runs, Toil will "autoscale", creating and terminating workers as needed to meet the demands of the workflow.

Once a user is familiar with the basics of running toil locally (specifying a *jobStore*, and how to write a toil script), they can move on to the guides below to learn how to translate these workflows into cloud ready workflows.

6.1 Managing a Cluster of Virtual Machines (Provisioning)

Toil can launch and manage a cluster of virtual machines to run using the *provisioner* to run a workflow distributed over several nodes. The provisioner also has the ability to automatically scale up or down the size of the cluster to handle dynamic changes in computational demand (autoscaling). Currently we have working provisioners with AWS and GCE (Azure support has been deprecated).

Toil uses Apache Mesos as the *Batch System*.

See here for instructions for *Running in AWS*.

See here for instructions for *Running in Google Compute Engine (GCE)*.

6.2 Storage (Toil jobStore)

Toil can make use of cloud storage such as AWS or Google buckets to take care of storage needs.

This is useful when running Toil in single machine mode on any cloud platform since it allows you to make use of their integrated storage systems.

For an overview of the job store see *Job Store*.

For instructions configuring a particular job store see:

- *AWS Job Store*
- *Google Job Store*

CHAPTER

SEVEN

CLOUD PLATFORMS

7.1 Running on Kubernetes

Kubernetes is a very popular container orchestration tool that has become a *de facto* cross-cloud-provider API for accessing cloud resources. Major cloud providers like Amazon, Microsoft, Kubernetes owner Google, and DigitalOcean have invested heavily in making Kubernetes work well on their platforms, by writing their own deployment documentation and developing provider-managed Kubernetes-based products. Using minikube, Kubernetes can even be run on a single machine.

Toil supports running Toil workflows against a Kubernetes cluster, either in the cloud or deployed on user-owned hardware.

7.1.1 Preparing your Kubernetes environment

1. **Get a Kubernetes cluster**

   To run Toil workflows on Kubernetes, you need to have a Kubernetes cluster set up. This will not be covered here, but there are many options available, and which one you choose will depend on which cloud ecosystem if any you use already, and on pricing. If you are just following along with the documentation, use `minikube` on your local machine.

   Alternatively, Toil can set up a Kubernetes cluster for you with the *Toil provisioner*. Follow *this* guide to get started with a Toil-managed Kubernetes cluster on AWS.

   Note that currently the only way to run a Toil workflow on Kubernetes is to use the AWS Job Store, so your Kubernetes workflow will currently have to store its data in Amazon's cloud regardless of where you run it. This can result in significant egress charges from Amazon if you run it outside of Amazon.

   Kubernetes Cluster Providers:

   - Your own institution
   - Amazon EKS
   - Microsoft Azure AKS
   - Google GKE
   - DigitalOcean Kubernetes
   - minikube

2. **Get a Kubernetes context on your local machine**

   There are two main ways to run Toil workflows on Kubernetes. You can either run the Toil leader on a machine outside the cluster, with jobs submitted to and run on the cluster, or you can submit the Toil leader itself as a job and have it run inside the cluster. Either way, you will need to configure your own machine to be able to submit

jobs to the Kubernetes cluster. Generally, this involves creating and populating a file named `.kube/config` in your user's home directory, and specifying the cluster to connect to, the certificate and token information needed for mutual authentication, and the Kubernetes namespace within which to work. However, Kubernetes configuration can also be picked up from other files in the `.kube` directory, environment variables, and the enclosing host when running inside a Kubernetes-managed container.

You will have to do different things here depending on where you got your Kubernetes cluster:

- Configuring for Amazon EKS
- Configuring for Microsoft Azure AKS
- Configuring for Google GKE
- Configuring for DigitalOcean Kubernetes Clusters
- Configuring for minikube

Toil's internal Kubernetes configuration logic mirrors that of the `kubectl` command. Toil workflows will use the current `kubectl` context to launch their Kubernetes jobs.

3. **If running the Toil leader in the cluster, get a service account**

   If you are going to run your workflow's leader within the Kubernetes cluster (see *Option 1: Running the Leader Inside Kubernetes*), you will need a service account in your chosen Kubernetes namespace. Most namespaces should have a service account named `default` which should work fine. If your cluster requires you to use a different service account, you will need to obtain its name and use it when launching the Kubernetes job containing the Toil leader.

4. **Set up appropriate permissions**

   Your local Kubernetes context and/or the service account you are using to run the leader in the cluster will need to have certain permissions in order to run the workflow. Toil needs to be able to interact with jobs and pods in the cluster, and to retrieve pod logs. You as a user may need permission to set up an AWS credentials secret, if one is not already available. Additionally, it is very useful for you as a user to have permission to interact with nodes, and to shell into pods.

   The appropriate permissions may already be available to you and your service account by default, especially in managed or ease-of-use-optimized setups such as EKS or minikube.

   However, if the appropriate permissions are not already available, you or your cluster administrator will have to grant them manually. The following `Role` (`toil-user`) and `ClusterRole` (`node-reader`), to be applied with `kubectl apply -f filename.yaml`, should grant sufficient permissions to run Toil workflows when bound to your account and the service account used by Toil workflows. Be sure to replace `YOUR_NAMESPACE_HERE` with the namespace you are running your workflows in

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: YOUR_NAMESPACE_HERE
  name: toil-user
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["explain", "get", "watch", "list", "describe", "logs", "attach", "exec",
↪"port-forward", "proxy", "cp", "auth"]
- apiGroups: ["batch"]
  resources: ["*"]
  verbs: ["get", "watch", "list", "create", "run", "set", "delete"]
- apiGroups: [""]
```

(continues on next page)

(continued from previous page)

```
  resources: ["secrets", "pods", "pods/attach", "podtemplates", "configmaps",
↪"events", "services"]
  verbs: ["patch", "get", "update", "watch", "list", "create", "run", "set", "delete
↪", "exec"]
- apiGroups: [""]
  resources: ["pods", "pods/log"]
  verbs: ["get", "list"]
- apiGroups: [""]
  resources: ["pods/exec"]
  verbs: ["create"]
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: node-reader
rules:
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["get", "list", "describe"]
- apiGroups: [""]
  resources: ["namespaces"]
  verbs: ["get", "list", "describe"]
- apiGroups: ["metrics.k8s.io"]
  resources: ["*"]
  verbs: ["*"]
```

To bind a user or service account to the `Role` or `ClusterRole` and actually grant the permissions, you will need a `RoleBinding` and a `ClusterRoleBinding`, respectively. Make sure to fill in the namespace, username, and service account name, and add more user stanzas if your cluster is to support multiple Toil users.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: toil-developer-member
  namespace: toil
subjects:
- kind: User
  name: YOUR_KUBERNETES_USERNAME_HERE
  apiGroup: rbac.authorization.k8s.io
- kind: ServiceAccount
  name: YOUR_SERVICE_ACCOUNT_NAME_HERE
  namespace: YOUR_NAMESPACE_HERE
roleRef:
  kind: Role
  name: toil-user
  apiGroup: rbac.authorization.k8s.io
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-nodes
subjects:
```

(continues on next page)

(continued from previous page)

```
- kind: User
  name: YOUR_KUBERNETES_USERNAME_HERE
  apiGroup: rbac.authorization.k8s.io
- kind: ServiceAccount
  name: YOUR_SERVICE_ACCOUNT_NAME_HERE
  namespace: YOUR_NAMESPACE_HERE
roleRef:
  kind: ClusterRole
  name: node-reader
  apiGroup: rbac.authorization.k8s.io
```

7.1.2 AWS Job Store for Kubernetes

Currently, the only job store, which is what Toil uses to exchange data between jobs, that works with jobs running on Kubernetes is the AWS Job Store. This requires that the Toil leader and Kubernetes jobs be able to connect to and use Amazon S3 and Amazon SimpleDB. It also requires that you have an Amazon Web Services account.

1. **Get access to AWS S3 and SimpleDB**

   In your AWS account, you need to create an AWS access key. First go to the IAM dashboard; for "us-west1", the link would be:

   ```
   https://console.aws.amazon.com/iam/home?region=us-west-1#/home
   ```

   Then create an access key, and save the Access Key ID and the Secret Key. As documented in the AWS documentation:

   1. On the IAM Dashboard page, choose your account name in the navigation bar, and then choose My Security Credentials.
   2. Expand the Access keys (access key ID and secret access key) section.
   3. Choose Create New Access Key. Then choose Download Key File to save the access key ID and secret access key to a file on your computer. After you close the dialog box, you can't retrieve this secret access key again.

   Make sure that, if your AWS infrastructure requires your user to authenticate with a multi-factor authentication (MFA) token, you obtain a second secret key and access key that don't have this requirement. The secret key and access key used to populate the Kubernetes secret that allows the jobs to contact the job store need to be usable without human intervention.

2. **Configure AWS access from the local machine**

   This only really needs to happen if you run the leader on the local machine. But we need the files in place to fill in the secret in the next step. Run:

   ```
   $ aws configure
   ```

   Then when prompted, enter your secret key and access key. This should create a file `~/.aws/credentials` that looks like this:

   ```
   [default]
   aws_access_key_id =  BLAH
   aws_secret_access_key =  blahblahblah
   ```

3. **Create a Kubernetes secret to give jobs access to AWS**

Go into the directory where the `credentials` file is:

```
$ cd ~/.aws
```

Then, create a Kubernetes secret that contains it. We'll call it `aws-credentials`:

```
$ kubectl create secret generic aws-credentials --from-file credentials
```

7.1.3 Configuring Toil for your Kubernetes environment

To configure your workflow to run on Kubernetes, you will have to configure several environment variables, in addition to passing the `--batchSystem kubernetes` option. Doing the research to figure out what values to give these variables may require talking to your cluster provider.

1. `TOIL_AWS_SECRET_NAME` is the most important, and **must** be set to the secret that contains your AWS `credentials` file, **if** your cluster nodes don't otherwise have access to S3 and SimpleDB (such as through IAM roles). This is required for the AWS job store to work, which is currently the only job store that can be used on Kubernetes. In this example we are using `aws-credentials`.
2. `TOIL_KUBERNETES_HOST_PATH` **can** be set to allow Toil jobs on the same physical host to share a cache. It should be set to a path on the host where the shared cache should be stored. It will be mounted as `/var/lib/toil`, or at `TOIL_WORKDIR` if specified, inside the container. This path must already exist on the host, and must have as much free space as your Kubernetes node offers to jobs. In this example, we are using `/data/scratch`. To actually make use of caching, make sure not to use `--disableCaching`.
3. `TOIL_KUBERNETES_OWNER` **should** be set to the username of the user running the Toil workflow. The jobs that Toil creates will include this username, so they can be more easily recognized, and cleaned up by the user if anything happens to the Toil leader. In this example we are using `demo-user`.

Note that Docker containers cannot be run inside of unprivileged Kubernetes pods (which are themselves containers). The Docker daemon does not (yet) support this. Other tools, such as Singularity in its user-namespace mode, are able to run containers from within containers. If using Singularity to run containerized tools, and you want downloaded container images to persist between Toil jobs, you will also want to set `TOIL_KUBERNETES_HOST_PATH` and make sure that Singularity is downloading its containers under the Toil work directory (`/var/lib/toil` buy default) by setting `SINGULARITY_CACHEDIR`. However, you will need to make sure that no two jobs try to download the same container at the same time; Singularity has no synchronization or locking around its cache, but the cache is also not safe for simultaneous access by multiple Singularity invocations. Some Toil workflows use their own custom workaround logic for this problem; this work is likely to be made part of Toil in a future release.

7.1.4 Running workflows

To run the workflow, you will need to run the Toil leader process somewhere. It can either be run inside Kubernetes as a Kubernetes job, or outside Kubernetes as a normal command.

Option 1: Running the Leader Inside Kubernetes

Once you have determined a set of environment variable values for your workflow run, write a YAML file that defines a Kubernetes job to run your workflow with that configuration. Some configuration items (such as your username, and the name of your AWS credentials secret) need to be written into the YAML so that they can be used from the leader as well.

Note that the leader pod will need your workflow script, its other dependencies, and Toil all installed. An easy way to get Toil installed is to start with the Toil appliance image for the version of Toil you want to use. In this example, we use `quay.io/ucsc_cgl/toil:5.5.0`.

Here's an example YAML file to run a test workflow:

```
apiVersion: batch/v1
kind: Job
metadata:
  # It is good practice to include your username in your job name.
  # Also specify it in TOIL_KUBERNETES_OWNER
  name: demo-user-toil-test
# Do not try and rerun the leader job if it fails

spec:
 backoffLimit: 0
 template:
   spec:
     # Do not restart the pod when the job fails, but keep it around so the
     # log can be retrieved
     restartPolicy: Never
     volumes:
     - name: aws-credentials-vol
       secret:
         # Make sure the AWS credentials are available as a volume.
         # This should match TOIL_AWS_SECRET_NAME
         secretName: aws-credentials
     # You may need to replace this with a different service account name as
     # appropriate for your cluster.
     serviceAccountName: default
     containers:
     - name: main
       image: quay.io/ucsc_cgl/toil:5.5.0
       env:
       # Specify your username for inclusion in job names
       - name: TOIL_KUBERNETES_OWNER
         value: demo-user
       # Specify where to find the AWS credentials to access the job store with
       - name: TOIL_AWS_SECRET_NAME
         value: aws-credentials
       # Specify where per-host caches should be stored, on the Kubernetes hosts.
       # Needs to be set for Toil's caching to be efficient.
```

(continues on next page)

(continued from previous page)

```
      - name: TOIL_KUBERNETES_HOST_PATH
        value: /data/scratch
      volumeMounts:
      # Mount the AWS credentials volume
      - mountPath: /root/.aws
        name: aws-credentials-vol
      resources:
        # Make sure to set these resource limits to values large enough
        # to accommodate the work your workflow does in the leader
        # process, but small enough to fit on your cluster.
        #
        # Since no request values are specified, the limits are also used
        # for the requests.
        limits:
          cpu: 2
          memory: "4Gi"
          ephemeral-storage: "10Gi"
      command:
      - /bin/bash
      - -c
      - |
        # This Bash script will set up Toil and the workflow to run, and run them.
        set -e
        # We make sure to create a work directory; Toil can't hot-deploy a
        # script from the root of the filesystem, which is where we start.
        mkdir /tmp/work
        cd /tmp/work
        # We make a virtual environment to allow workflow dependencies to be
        # hot-deployed.
        #
        # We don't really make use of it in this example, but for workflows
        # that depend on PyPI packages we will need this.
        #
        # We use --system-site-packages so that the Toil installed in the
        # appliance image is still available.
        virtualenv --python python3 --system-site-packages venv
        . venv/bin/activate
        # Now we install the workflow. Here we're using a demo workflow
        # script from Toil itself.
        wget https://raw.githubusercontent.com/DataBiosphere/toil/releases/4.1.0/src/
↪toil/test/docs/scripts/tutorial_helloworld.py
        # Now we run the workflow. We make sure to use the Kubernetes batch
        # system and an AWS job store, and we set some generally useful
        # logging options. We also make sure to enable caching.
        python3 tutorial_helloworld.py \
            aws:us-west-2:demouser-toil-test-jobstore \
            --batchSystem kubernetes \
            --realTimeLogging \
            --logInfo
```

You can save this YAML as `leader.yaml`, and then run it on your Kubernetes installation with:

```
$ kubectl apply -f leader.yaml
```

To monitor the progress of the leader job, you will want to read its logs. If you are using a Kubernetes dashboard such as k9s, you can simply find the pod created for the job in the dashboard, and view its logs there. If not, you will need to locate the pod by hand.

Monitoring and Debugging Kubernetes Jobs and Pods

The following techniques are most useful for looking at the pod which holds the Toil leader, but they can also be applied to individual Toil jobs on Kubernetes, even when the leader is outside the cluster.

Kubernetes names pods for jobs by appending a short random string to the name of the job. You can find the name of the pod for your job by doing:

```
$ kubectl get pods | grep demo-user-toil-test
demo-user-toil-test-g5496                                         1/1     Running     0 ␣
↪         2m
```

Assuming you have set `TOIL_KUBERNETES_OWNER` correctly, you should be able to find all of your workflow's pods by searching for your username:

```
$ kubectl get pods | grep demo-user
```

If the status of a pod is anything other than `Pending`, you will be able to view its logs with:

```
$ kubectl logs demo-user-toil-test-g5496
```

This will dump the pod's logs from the beginning to now and terminate. To follow along with the logs from a running pod, add the `-f` option:

```
$ kubectl logs -f demo-user-toil-test-g5496
```

A status of `ImagePullBackoff` suggests that you have requested to use an image that is not available. Check the `image` section of your YAML if you are looking at a leader, or the value of `TOIL_APPLIANCE_SELF` if you are delaying with a worker job. You also might want to check your Kubernetes node's Internet connectivity and DNS function; in Kubernetes, DNS depends on system-level pods which can be terminated or evicted in cases of resource oversubscription, just like user workloads.

If your pod seems to be stuck `Pending`, `ContainerCreating`, you can get information on what is wrong with it by using `kubectl describe pod`:

```
$ kubectl describe pod demo-user-toil-test-g5496
```

Pay particular attention to the `Events:` section at the end of the output. An indication that a job is too big for the available nodes on your cluster, or that your cluster is too busy for your jobs, is `FailedScheduling` events:

```
Type     Reason            Age                 From               Message
----     ------            ----                ----               -------
Warning  FailedScheduling  13s (x79 over 100m)  default-scheduler  0/4 nodes are␣
↪available: 1 Insufficient cpu, 1 Insufficient ephemeral-storage, 4 Insufficient memory.
```

If a pod is running but seems to be behaving erratically, or seems stuck, you can shell into it and look around:

```
$ kubectl exec -ti demo-user-toil-test-g5496 /bin/bash
```

One common cause of stuck pods is attempting to use more memory than allowed by Kubernetes (or by the Toil job's memory resource requirement), but in a way that does not trigger the Linux OOM killer to terminate the pod's processes. In these cases, the pod can remain stuck at nearly 100% memory usage more or less indefinitely, and attempting to shell into the pod (which needs to start a process within the pod, using some of its memory) will fail. In these cases, the recommended solution is to kill the offending pod and increase its (or its Toil job's) memory requirement, or reduce its memory needs by adapting user code.

When Things Go Wrong

The Toil Kubernetes batch system includes cleanup code to terminate worker jobs when the leader shuts down. However, if the leader pod is removed by Kubernetes, is forcibly killed or otherwise suffers a sudden existence failure, it can go away while its worker jobs live on. It is not recommended to restart a workflow in this state, as jobs from the previous invocation will remain running and will be trying to modify the job store concurrently with jobs from the new invocation.

To clean up dangling jobs, you can use the following snippet:

```
$ kubectl get jobs | grep demo-user | cut -f1 -d' ' | xargs -n10 kubectl delete job
```

This will delete all jobs with `demo-user`'s username in their names, in batches of 10. You can also use the UUID that Toil assigns to a particular workflow invocation in the filter, to clean up only the jobs pertaining to that workflow invocation.

Option 2: Running the Leader Outside Kubernetes

If you don't want to run your Toil leader inside Kubernetes, you can run it locally instead. This can be useful when developing a workflow; files can be hot-deployed from your local machine directly to Kubernetes. However, your local machine will have to have (ideally role-assumption- and MFA-free) access to AWS, and access to Kubernetes. Real time logging will not work unless your local machine is able to listen for incoming UDP packets on arbitrary ports on the address it uses to contact the IPv4 Internet; Toil does no NAT traversal or detection.

Note that if you set `TOIL_WORKDIR` when running your workflow like this, it will need to be a directory that exists both on the host and in the Toil appliance.

Here is an example of running our test workflow leader locally, outside of Kubernetes:

```
$ export TOIL_KUBERNETES_OWNER=demo-user  # This defaults to your local username if not␣
↪set
$ export TOIL_AWS_SECRET_NAME=aws-credentials
$ export TOIL_KUBERNETES_HOST_PATH=/data/scratch
$ virtualenv --python python3 --system-site-packages venv
$ . venv/bin/activate
$ wget https://raw.githubusercontent.com/DataBiosphere/toil/releases/4.1.0/src/toil/test/
↪docs/scripts/tutorial_helloworld.py
$ python3 tutorial_helloworld.py \
      aws:us-west-2:demouser-toil-test-jobstore \
      --batchSystem kubernetes \
      --realTimeLogging \
      --logInfo
```

Running CWL Workflows

Running CWL workflows on Kubernetes can be challenging, because executing CWL can require `toil-cwl-runner` to orchestrate containers of its own, within a Kubernetes job running in the Toil appliance container.

Normally, running a CWL workflow should Just Work, as long as the workflow's Docker containers are able to be executed with Singularity, your Kubernetes cluster does not impose extra capability-based confinement (i.e. SELinux, AppArmor) that interferes with Singularity's use of user-mode namespaces, and you make sure to configure Toil so that its workers know where to store their data within the Kubernetes pods (which would be done for you if using a Toil-managed cluster). For example, you should be able to run a CWL workflow like this:

```
$ export TOIL_KUBERNETES_OWNER=demo-user  # This defaults to your local username if not␣
↪set
$ export TOIL_AWS_SECRET_NAME=aws-credentials
$ export TOIL_KUBERNETES_HOST_PATH=/data/scratch
$ virtualenv --python python3 --system-site-packages venv
$ . venv/bin/activate
$ pip install toil[kubernetes,cwl]==5.8.0
$ toil-cwl-runner  \
    --jobStore  aws:us-west-2:demouser-toil-test-jobstore \
    --batchSystem kubernetes \
    --realTimeLogging \
    --logInfo \
    --disableCaching \
    path/to/cwl/workflow \
    path/to/cwl/input/object
```

Additional `cwltool` options that your workflow might require, such as `--no-match-user`, can be passed to `toil-cwl-runner`, which inherits most `cwltool` options.

AppArmor and Singularity

Kubernetes clusters based on Ubuntu hosts often will have AppArmor enabled on the host. AppArmor is a capability-based security enhancement system that integrates with the Linux kernel to enforce lists of things which programs may or may not do, called **profiles**. For example, an AppArmor profile could be applied to a web server process to stop it from using the `mount()` system call to manipulate the filesystem, because it has no business doing that under normal circumstances but might attempt to do it if compromised by hackers.

Kubernetes clusters also often use Docker as the backing container runtime, to run pod containers. When AppArmor is enabled, Docker will load an AppArmor profile and apply it to all of its containers by default, with the ability for the profile to be overridden on a per-container basis. This profile unfortunately prevents some of the *mount()* system calls that Singularity uses to set up user-mode containers from working inside the pod, even though these calls would be allowed for an unprivileged user under normal circumstances.

On the UCSC Kubernetes cluster, we configure our Ubuntu hosts with an alternative default AppArmor profile for Docker containers which allows these calls. Other solutions include turning off AppArmor on the host, configuring Kubernetes with a container runtime other than Docker, or using Kubernetes's AppArmor integration to apply a more permissive profile or the `unconfined` profile to pods that Toil launches.

Toil does not yet have a way to apply a `container.apparmor.security.beta.kubernetes.io/runner-container: unconfined` annotation to its pods, as described in the Kubernetes AppArmor documentation. This feature is tracked in issue #4331.

7.2 Running in AWS

Toil jobs can be run on a variety of cloud platforms. Of these, Amazon Web Services (AWS) is currently the best-supported solution. Toil provides the *Cluster Utilities* to conveniently create AWS clusters, connect to the leader of the cluster, and then launch a workflow. The leader handles distributing the jobs over the worker nodes and autoscaling to optimize costs.

The *Running a Workflow with Autoscaling* section details how to create a cluster and run a workflow that will dynamically scale depending on the workflow's needs.

The *Static Provisioning* section explains how a static cluster (one that won't automatically change in size) can be created and provisioned (grown, shrunk, destroyed, etc.).

7.2.1 Preparing your AWS environment

To use Amazon Web Services (AWS) to run Toil or to just use S3 to host the files during the computation of a workflow, first set up and configure an account with AWS:

1. If necessary, create and activate an AWS account
2. Next, generate a key pair for AWS with the command (do NOT generate your key pair with the Amazon browser):

```
$ ssh-keygen -t rsa
```

3. This should prompt you to save your key. Please save it in

```
~/.ssh/id_rsa
```

4. Now move this to where your OS can see it as an authorized key:

```
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

5. Next, you'll need to add your key to the *ssh-agent*:

```
$ eval `ssh-agent -s`
$ ssh-add
```

   If your key has a passphrase, you will be prompted to enter it here once.

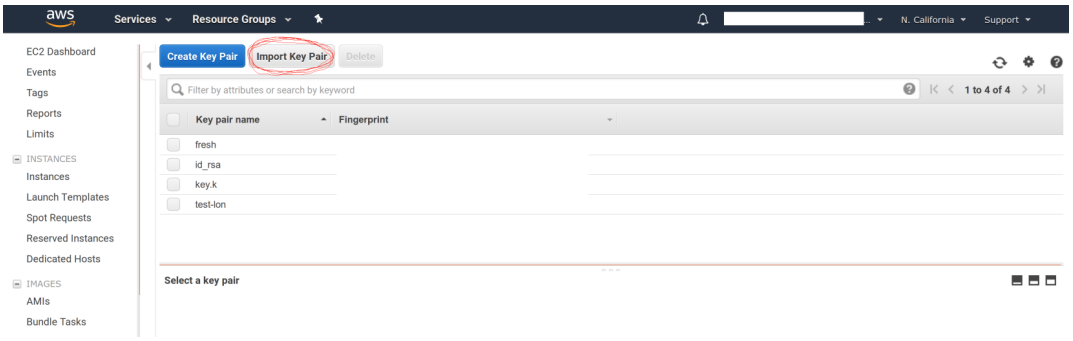
6. You'll also need to chmod your private key (good practice but also enforced by AWS):

```
$ chmod 400 id_rsa
```

7. Now you'll need to add the key to AWS via the browser. For example, on us-west1, this address would accessible at:

```
https://us-west-1.console.aws.amazon.com/ec2/v2/home?region=us-west-1
↪#KeyPairs:sort=keyName
```

8. Now click on the "Import Key Pair" button to add your key:



9. Next, you need to create an AWS access key. First go to the IAM dashboard, again; for "us-west1", the example link would be here:

```
https://console.aws.amazon.com/iam/home?region=us-west-1#/home
```

10. The directions (transcribed from: https://docs.aws.amazon.com/general/latest/gr/managing-aws-access-keys.html) are now:

    1. On the IAM Dashboard page, choose your account name in the navigation bar, and then choose My Security Credentials.
    2. Expand the Access keys (access key ID and secret access key) section.
    3. Choose Create New Access Key. Then choose Download Key File to save the access key ID and secret access key to a file on your computer. After you close the dialog box, you can't retrieve this secret access key again.

11. Now you should have a newly generated "AWS Access Key ID" and "AWS Secret Access Key". We can now install the AWS CLI and make sure that it has the proper credentials:

```
$ pip install awscli --upgrade --user
```

12. Now configure your AWS credentials with:

```
$ aws configure
```

13. Add your "AWS Access Key ID" and "AWS Secret Access Key" from earlier and your region and output format:

```
" AWS Access Key ID [****************Q65Q]: "
" AWS Secret Access Key [****************G0ys]: "
" Default region name [us-west-1]: "
" Default output format [json]: "
```

    This will create the files *~/.aws/config* and *~/.aws/credentials*.

14. If not done already, install toil (example uses version 5.3.0, but we recommend the latest release):

```
$ virtualenv venv
$ source venv/bin/activate
$ pip install toil[all]==5.3.0
```

15. Now that toil is installed and you are running a virtualenv, an example of launching a toil leader node would be the following (again, note that we set TOIL_APPLIANCE_SELF to toil version 5.3.0 in this example, but please set the version to the installed version that you are using if you're using a different version):

```
$ TOIL_APPLIANCE_SELF=quay.io/ucsc_cgl/toil:5.3.0 \
      toil launch-cluster clustername \
      --leaderNodeType t2.medium \
      --zone us-west-1a \
      --keyPairName id_rsa
```

To further break down each of these commands:

TOIL_APPLIANCE_SELF=quay.io/ucsc_cgl/toil:latest — This is optional. It specifies a mesos docker image that we maintain with the latest version of toil installed on it. If you want to use a different version of toil, please specify the image tag you need from https://quay.io/repository/ucsc_cgl/toil?tag=latest&tab=tags.

toil launch-cluster — Base command in toil to launch a cluster.

clustername — Just choose a name for your cluster.

–leaderNodeType t2.medium — Specify the leader node type. Make a t2.medium (2CPU; 4Gb RAM; $0.0464/Hour). List of available AWS instances: https://aws.amazon.com/ec2/pricing/on-demand/

–zone us-west-1a — Specify the AWS zone you want to launch the instance in. Must have the same prefix as the zone in your awscli credentials (which, in the example of this tutorial is: "us-west-1").

–keyPairName id_rsa — The name of your key pair, which should be "id_rsa" if you've followed this tutorial.

Note: You can set the `TOIL_AWS_TAGS` environment variable to a JSON object to specify arbitrary tags for AWS resources. For example, if you `export TOIL_AWS_TAGS='{"project-name": "variant-calling"}'` in your shell before using Toil, AWS resources created by Toil will be tagged with a `project-name` tag with the value `variant-calling`.

7.2.2 AWS Job Store

Using the AWS job store is straightforward after you've finished *Preparing your AWS environment*; all you need to do is specify the prefix for the job store name.

To run the sort example *sort example* with the AWS job store you would type

```
$ python sort.py aws:us-west-2:my-aws-sort-jobstore
```

7.2.3 Toil Provisioner

The Toil provisioner is included in Toil alongside the `[aws]` extra and allows us to spin up a cluster.

Getting started with the provisioner is simple:

1. Make sure you have Toil installed with the AWS extras. For detailed instructions see *Installing Toil with Extra Features*.
2. You will need an AWS account and you will need to save your AWS credentials on your local machine. For help setting up an AWS account see here. For setting up your AWS credentials follow instructions here.

The Toil provisioner is built around the Toil Appliance, a Docker image that bundles Toil and all its requirements (e.g. Mesos). This makes deployment simple across platforms, and you can even simulate a cluster locally (see *Developing with Docker* for details).

Choosing Toil Appliance Image

When using the Toil provisioner, the appliance image will be automatically chosen based on the pip-installed version of Toil on your system. That choice can be overridden by setting the environment variables `TOIL_DOCKER_REGISTRY` and `TOIL_DOCKER_NAME` or `TOIL_APPLIANCE_SELF`. See *Environment Variables* for more information on these variables. If you are developing with autoscaling and want to test and build your own appliance have a look at *Developing with Docker*.

For information on using the Toil Provisioner have a look at *Running a Workflow with Autoscaling*.

7.2.4 Details about Launching a Cluster in AWS

Using the provisioner to launch a Toil leader instance is simple using the `launch-cluster` command. For example, to launch a cluster named "my-cluster" with a t2.medium leader in the us-west-2a zone, run

```
(venv) $ toil launch-cluster my-cluster \
            --leaderNodeType t2.medium \
            --zone us-west-2a \
            --keyPairName <your-AWS-key-pair-name>
```

The cluster name is used to uniquely identify your cluster and will be used to populate the instance's `Name` tag. Also, the Toil provisioner will automatically tag your cluster with an `Owner` tag that corresponds to your keypair name to facilitate cost tracking. In addition, the `ToilNodeType` tag can be used to filter "leader" vs. "worker" nodes in your cluster.

The leaderNodeType is an EC2 instance type. This only affects the leader node.

The `--zone` parameter specifies which EC2 availability zone to launch the cluster in. Alternatively, you can specify this option via the `TOIL_AWS_ZONE` environment variable. Note: the zone is different from an EC2 region. A region corresponds to a geographical area like `us-west-2 (Oregon)`, and availability zones are partitions of this area like `us-west-2a`.

By default, Toil creates an IAM role for each cluster with sufficient permissions to perform cluster operations (e.g. full S3, EC2, and SDB access). If the default permissions are not sufficient for your use case (e.g. if you need access to ECR), you may create a custom IAM role with all necessary permissions and set the `--awsEc2ProfileArn` parameter when launching the cluster. Note that your custom role must at least have *these permissions* in order for the Toil cluster to function properly.

In addition, Toil creates a new security group with the same name as the cluster name with default rules (e.g. opens port 22 for SSH access). If you require additional security groups, you may use the `--awsEc2ExtraSecurityGroupId` parameter when launching the cluster. **Note:** Do not use the same name as the cluster name for the extra security groups as any security group matching the cluster name will be deleted once the cluster is destroyed.

For more information on options try:

```
(venv) $ toil launch-cluster --help
```

Static Provisioning

Toil can be used to manage a cluster in the cloud by using the *Cluster Utilities*. The cluster utilities also make it easy to run a toil workflow directly on this cluster. We call this static provisioning because the size of the cluster does not change. This is in contrast with *Running a Workflow with Autoscaling*.

To launch worker nodes alongside the leader we use the `-w` option:

```
(venv) $ toil launch-cluster my-cluster \
            --leaderNodeType t2.small -z us-west-2a \
            --keyPairName your-AWS-key-pair-name \
            --nodeTypes m3.large,t2.micro -w 1,4
```

This will spin up a leader node of type t2.small with five additional workers — one m3.large instance and four t2.micro.

Currently static provisioning is only possible during the cluster's creation. The ability to add new nodes and remove existing nodes via the native provisioner is in development. Of course the cluster can always be deleted with the *Destroy-Cluster Command* utility.

Uploading Workflows

Now that our cluster is launched, we use the *Rsync-Cluster Command* utility to copy the workflow to the leader. For a simple workflow in a single file this might look like

```
(venv) $ toil rsync-cluster -z us-west-2a my-cluster toil-workflow.py :/
```

Note: If your toil workflow has dependencies have a look at the *Auto-Deployment* section for a detailed explanation on how to include them.

Running a Workflow with Autoscaling

Autoscaling is a feature of running Toil in a cloud whereby additional cloud instances are launched to run the workflow. Autoscaling leverages Mesos containers to provide an execution environment for these workflows.

Note: Make sure you've done the AWS setup in *Preparing your AWS environment*.

---

1. Download `sort.py`
2. Launch the leader node in AWS using the *Launch-Cluster Command* command:

   ```
   (venv) $ toil launch-cluster <cluster-name> \
               --keyPairName <AWS-key-pair-name> \
               --leaderNodeType t2.medium \
               --zone us-west-2a
   ```

3. Copy the `sort.py` script up to the leader node:

   ```
   (venv) $ toil rsync-cluster -z us-west-2a <cluster-name> sort.py :/root
   ```

4. Login to the leader node:

```
(venv) $ toil ssh-cluster -z us-west-2a <cluster-name>
```

5. Run the script as an autoscaling workflow:

```
$ python /root/sort.py aws:us-west-2:<my-jobstore-name> \
      --provisioner aws \
      --nodeTypes c3.large \
      --maxNodes 2 \
      --batchSystem mesos
```

Note: In this example, the autoscaling Toil code creates up to two instances of type *c3.large* and launches Mesos slave containers inside them. The containers are then available to run jobs defined by the *sort.py* script. Toil also creates a bucket in S3 called *aws:us-west-2:autoscaling-sort-jobstore* to store intermediate job results. The Toil autoscaler can also provision multiple different node types, which is useful for workflows that have jobs with varying resource requirements. For example, one could execute the script with `--nodeTypes c3.large,r3.xlarge --maxNodes 5,1`, which would allow the provisioner to create up to five c3.large nodes and one r3.xlarge node for memory-intensive jobs. In this situation, the autoscaler would avoid creating the more expensive r3.xlarge node until needed, running most jobs on the c3.large nodes.

---

1. View the generated file to sort:

```
$ head fileToSort.txt
```

2. View the sorted file:

```
$ head sortedFile.txt
```

For more information on other autoscaling (and other) options have a look at *Commandline Options* and/or run

```
$ python my-toil-script.py --help
```

Important: Some important caveats about starting a toil run through an ssh session are explained in the *Ssh-Cluster Command* section.

Preemptibility

Toil can run on a heterogeneous cluster of both preemptible and non-preemptible nodes. Being a preemptible node simply means that the node may be shut down at any time, while jobs are running. These jobs can then be restarted later somewhere else.

A node type can be specified as preemptible by adding a spot bid to its entry in the list of node types provided with the `--nodeTypes` flag. If spot instance prices rise above your bid, the preemptible node whill be shut down.

Individual jobs can explicitly specify whether they should be run on preemptible nodes via the boolean `preemptible` resource requirement, if this is not specified, the job will not run on preemptible nodes even if preemptible nodes are available unless specified with the `--defaultPreemptible` flag. The `--defaultPreemptible` flag will allow jobs without a `preemptible` requirement to run on preemptible machines. For example:

```
$ python /root/sort.py aws:us-west-2:<my-jobstore-name> \
      --provisioner aws \
```

(continues on next page)

(continued from previous page)

```
    --nodeTypes c3.4xlarge:2.00 \
    --maxNodes 2 \
    --batchSystem mesos \
    --defaultPreemptible
```

Specify Preemptibility Carefully

Ensure that your choices for `--nodeTypes` and `--maxNodes <>` make sense for your workflow and won't cause it to hang. You should make sure the provisioner is able to create nodes large enough to run the largest job in the workflow, and that non-preemptible node types are allowed if there are non-preemptible jobs in the workflow.

Finally, the `--preemptibleCompensation` flag can be used to handle cases where preemptible nodes may not be available but are required for your workflow. With this flag enabled, the autoscaler will attempt to compensate for a shortage of preemptible nodes of a certain type by creating non-preemptible nodes of that type, if non-preemptible nodes of that type were specified in `--nodeTypes`.

Provisioning with a Kubernetes cluster

If you don't have an existing Kubernetes cluster but still want to use Kubernetes to orchestrate jobs, Toil can create a Kubernetes cluster for you using the AWS provisioner.

By default, the `toil launch-cluster` command uses a Mesos cluster as the jobs scheduler. Toil can also create a Kubernetes cluster to schedule Toil jobs. To set up a Kubernetes cluster, simply add the `--clusterType=kubernetes` command line option to `toil launch-cluster`.

For example, to launch a Toil cluster with a Kubernetes scheduler, run:

```
(venv) $ toil launch-cluster <cluster-name> \
        --provisioner=aws \
        --clusterType kubernetes \
        --zone us-west-2a \
        --keyPairName wlgao@ucsc.edu \
        --leaderNodeType t2.medium \
        --leaderStorage 50 \
        --nodeTypes t2.medium -w 1-4 \
        --nodeStorage 20 \
        --logDebug
```

Behind the scenes, Toil installs kubeadm and configures kubelet on the Toil leader and all worker nodes. This Toil cluster can then schedule jobs using Kubernetes.

Note: You should set at least one worker node, otherwise Kubernetes would not be able to schedule any jobs. It is also normal for this step to take a while.

Below is a tutorial on how to launch a Toil job on this newly created cluster. As a demostration, we will use `sort.py` again, but run it on a Toil cluster with Kubernetes. First, download this file and put it to the current working directory.

We then need to copy over the workflow file and SSH into the cluster:

```
(venv) $ toil rsync-cluster -z us-west-2a <cluster-name> sort.py :/root
(venv) $ toil ssh-cluster -z us-west-2a <cluster-name>
```

Remember to replace <cluster-name> with your actual cluster name, and feel free to use your own cluster configuration and/or workflow files. For more information on this step, see the corresponding section of the *Static Provisioning* tutorial.

Now that we are inside the cluster, a Kubernetes environment should already be configured and running. To verify this, simply run:

```
$ kubectl get nodes
```

You should see a leader node with the Ready status. Depending on the number of worker nodes you set to create upfront, you should also see them displayed here.

Additionally, you can also verify that the metrics server is running:

```
$ kubectl get --raw "/apis/metrics.k8s.io/v1beta1/nodes"
```

If there is a JSON response (similar to the output below), and you are not seeing any errors, that means the metrics server is set up and running, and you are good to start running workflows.

```
{"kind":"NodeMetricsList","apiVersion":"metrics.k8s.io/v1beta1", ...}
```

Note: It'll take a while for all nodes to get set up and running, so you might not be able to see all nodes running at first. You can start running workflows already, but Toil might complain until the necessary resources are set up and running.

Now we can run the workflow:

```
$ python sort.py \
        --provisioner aws
        --batchSystem kubernetes \
        aws:<region>:<job-store-name>
```

Make sure to replace <region> and <job-store-name>. It is **required** to use a cloud-accessible job store like AWS or Google when using the Kubernetes batch system.

The sort workflow should start running on the Kubernetes cluster set up by Toil. This workflow would take a while to execute, so you could put the job in the background and monitor the Kubernetes cluster using kubectl. For example, you can check out the pods that are running:

```
$ kubectl get pods
```

You should see an output like:

```
NAME                                                    READY   STATUS              ␣
↪RESTARTS   AGE
root-toil-a864e1b0-2e1f-48db-953c-038e5ad293c7-11-4cwdl   0/1     ContainerCreating   0 ␣
↪          85s
root-toil-a864e1b0-2e1f-48db-953c-038e5ad293c7-14-5dqtk   0/1     Completed           0 ␣
↪          18s
root-toil-a864e1b0-2e1f-48db-953c-038e5ad293c7-7-gkwc9    0/1     ContainerCreating   0 ␣
↪          107s
root-toil-a864e1b0-2e1f-48db-953c-038e5ad293c7-9-t7vsb    1/1     Running             0 ␣
↪          96s
```

If a pod failed for whatever reason or if you want to make sure a pod isn't stuck, you can use `kubectl describe pod <pod-name>` or `kubectl logs <pod-name>` to inspect the pod.

If everything is successful, you should be able to see an output file from the sort workflow:

```
$ head sortedFile.txt
```

You can now run your own workflows!

Using MinIO and S3-Compatible object stores

Toil can be configured to access files stored in an S3-compatible object store such as MinIO. The following environment variables can be used to configure the S3 connection used:

- `TOIL_S3_HOST`: the IP address or hostname to use for connecting to S3
- `TOIL_S3_PORT`: the port number to use for connecting to S3, if needed
- `TOIL_S3_USE_SSL`: enable or disable the usage of SSL for connecting to S3 (`True` by default)

Examples:

```
TOIL_S3_HOST=127.0.0.1
TOIL_S3_PORT=9010
TOIL_S3_USE_SSL=False
```

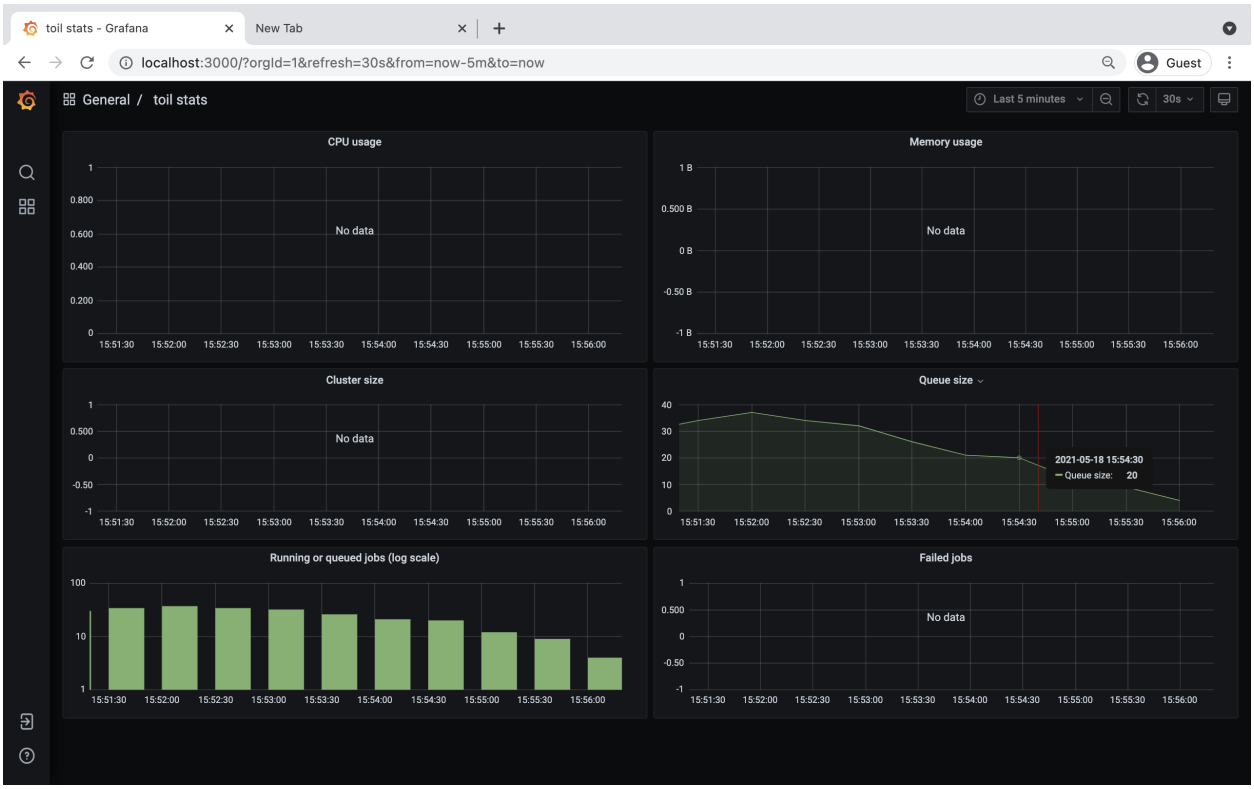
7.2.5 Dashboard

Toil provides a dashboard for viewing the RAM and CPU usage of each node, the number of issued jobs of each type, the number of failed jobs, and the size of the jobs queue. To launch this dashboard for a toil workflow, include the `--metrics` flag in the toil script command. The dashboard can then be viewed in your browser at localhost:3000 while connected to the leader node through `toil ssh-cluster`:

To change the default port number, you can use the `--grafana_port` argument:

```
(venv) $ toil ssh-cluster -z us-west-2a --grafana_port 8000 <cluster-name>
```

On AWS, the dashboard keeps track of every node in the cluster to monitor CPU and RAM usage, but it can also be used while running a workflow on a single machine. The dashboard uses Grafana as the front end for displaying real-time plots, and Prometheus for tracking metrics exported by toil:



In order to use the dashboard for a non-released toil version, you will have to build the containers locally with `make docker`, since the prometheus, grafana, and mtail containers used in the dashboard are tied to a specific toil version.

7.3 Running in Google Compute Engine (GCE)

Toil supports a provisioner with Google, and a *Google Job Store*. To get started, follow instructions for *Preparing your Google environment*.

7.3.1 Preparing your Google environment

Toil supports using the Google Cloud Platform. Setting this up is easy!

1. Make sure that the `google` extra (*Installing Toil with Extra Features*) is installed
2. Follow Google's Instructions to download credentials and set the `GOOGLE_APPLICATION_CREDENTIALS` environment variable
3. Create a new ssh key with the proper format. To create a new ssh key run the command

```
$ ssh-keygen -t rsa -f ~/.ssh/id_rsa -C [USERNAME]
```

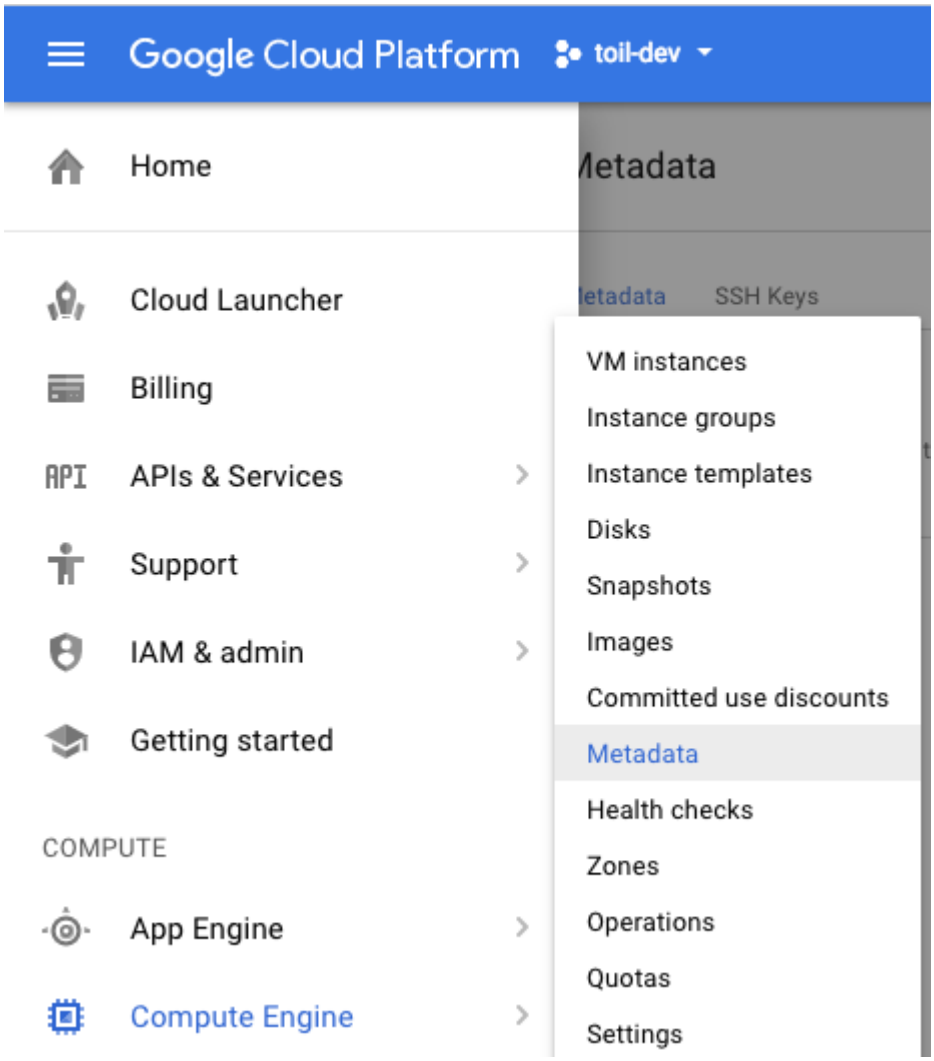
where `[USERNAME]` is something like `jane@example.com`. Make sure to leave your password blank.

> **Warning:** This command could overwrite an old ssh key you may be using. If you have an existing ssh key you would like to use, it will need to be called id_rsa and it needs to have no password set.

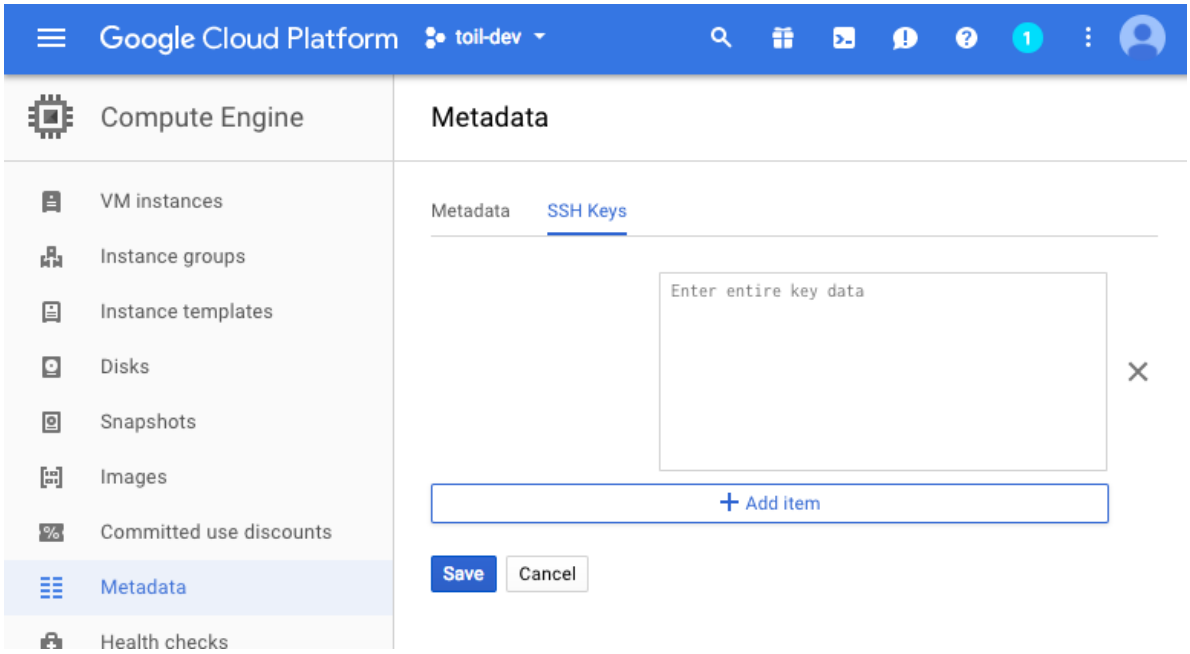
Make sure only you can read the SSH keys:

```
$ chmod 400 ~/.ssh/id_rsa ~/.ssh/id_rsa.pub
```

4. Add your newly formatted public key to Google. To do this, log into your Google Cloud account and go to metadata section under the Compute tab.



Near the top of the screen click on 'SSH Keys', then edit, add item, and paste the key. Then save:



For more details look at Google's instructions for adding SSH keys.

7.3.2 Google Job Store

To use the Google Job Store you will need to set the `GOOGLE_APPLICATION_CREDENTIALS` environment variable by following Google's instructions.

Then to run the sort example with the Google job store you would type

```
$ python sort.py google:my-project-id:my-google-sort-jobstore
```

7.3.3 Running a Workflow with Autoscaling

> **Warning:** Google Autoscaling is in beta!

The steps to run a GCE workflow are similar to those of AWS (*Running a Workflow with Autoscaling*), except you will need to explicitly specify the `--provisioner gce` option which otherwise defaults to `aws`.

1. Download `sort.py`
2. Launch the leader node in GCE using the *Launch-Cluster Command* command:

```
(venv) $ toil launch-cluster <CLUSTER-NAME> \
             --provisioner gce \
             --leaderNodeType n1-standard-1 \
             --keyPairName <SSH-KEYNAME> \
             --zone us-west1-a
```

Where `<SSH-KEYNAME>` is the first part of `[USERNAME]` used when setting up your ssh key. For example if `[USERNAME]` was jane@example.com, `<SSH-KEYNAME>` should be `jane`.

The `--keyPairName` option is for an SSH key that was added to the Google account. If your ssh key `[USERNAME]` was `jane@example.com`, then your key pair name will be just `jane`.

3. Upload the sort example and ssh into the leader:

```
(venv) $ toil rsync-cluster --provisioner gce <CLUSTER-NAME> sort.py :/root
(venv) $ toil ssh-cluster --provisioner gce <CLUSTER-NAME>
```

4. Run the workflow:

```
$ python /root/sort.py  google:<PROJECT-ID>:<JOBSTORE-NAME> \
      --provisioner gce \
      --batchSystem mesos \
      --nodeTypes n1-standard-2 \
      --maxNodes 2
```

5. Clean up:

```
$ exit  # this exits the ssh from the leader node
(venv) $ toil destroy-cluster --provisioner gce <CLUSTER-NAME>
```

7.4 Cluster Utilities

There are several utilities used for starting and managing a Toil cluster using the AWS provisioner. They are installed via the `[aws]` or `[google]` extra. For installation details see *Toil Provisioner*. The cluster utilities are used for *Running in AWS* and are comprised of `toil launch-cluster`, `toil rsync-cluster`, `toil ssh-cluster`, and `toil destroy-cluster` entry points.

Cluster commands specific to `toil` are:

`status` — Reports runtime and resource usage for all jobs in a specified jobstore (workflow must have originally been run using the -\-stats option).

`stats` — Inspects a job store to see which jobs have failed, run successfully, etc.

`destroy-cluster` — For autoscaling. Terminates the specified cluster and associated resources.

`launch-cluster` — For autoscaling. This is used to launch a toil leader instance with the specified provisioner.

`rsync-cluster` — For autoscaling. Used to transfer files to a cluster launched with `toil launch-cluster`.

`ssh-cluster` — SSHs into the toil appliance container running on the leader of the cluster.

`clean` — Delete the job store used by a previous Toil workflow invocation.

`kill` — Kills any running jobs in a rogue toil.

For information on a specific utility run:

```
toil launch-cluster --help
```

for a full list of its options and functionality.

The cluster utilities can be used for *Running in Google Compute Engine (GCE)* and *Running in AWS*.

Tip: By default, all of the cluster utilities expect to be running on AWS. To run with Google you will need to specify the `--provisioner gce` option for each utility.

Note: Boto must be configured with AWS credentials before using cluster utilities.

Running in Google Compute Engine (GCE) contains instructions for

7.5 Stats Command

To use the stats command, a workflow must first be run using the `--stats` option. Using this command makes certain that toil does not delete the job store, no matter what other options are specified (i.e. normally the option `--clean=always` would delete the job, but `--stats` will override this).

An example of this would be running the following:

```
python discoverfiles.py file:my-jobstore --stats
```

Where `discoverfiles.py` is the following:

```
import os
import subprocess

from toil.common import Toil
from toil.job import Job


class discoverFiles(Job):
    """Views files at a specified path using ls."""

    def __init__(self, path, *args, **kwargs):
        self.path = path
        super().__init__(*args, **kwargs)

    def run(self, fileStore):
        if os.path.exists(self.path):
            subprocess.check_call(["ls", self.path])


def main():
    options = Job.Runner.getDefaultArgumentParser().parse_args()
    options.clean = "always"

    job1 = discoverFiles(path="/sys/", displayName='sysFiles')
    job2 = discoverFiles(path=os.path.expanduser("~"), displayName='userFiles')
    job3 = discoverFiles(path="/tmp/")

    job1.addChild(job2)
    job2.addChild(job3)
```

(continues on next page)

(continued from previous page)

```
    with Toil(options) as toil:
        if not toil.options.restart:
            toil.start(job1)
        else:
            toil.restart()


if __name__ == '__main__':
    main()
```

Notice the `displayName` key, which can rename a job, giving it an alias when it is finally displayed in stats. Running this workflow file should record three job names: `sysFiles` (job1), `userFiles` (job2), and `discoverFiles` (job3). To see the runtime and resources used for each job when it was run, type

```
toil stats file:my-jobstore
```

This should output the following:

```
Batch System: singleMachine
Default Cores: 1  Default Memory: 2097152K
Max Cores: 9.22337e+18
Total Clock: 0.56  Total Runtime: 1.01
Worker
    Count |                                     Time* |                                 ␣
↪ Clock |                                     Wait |                                   ␣
↪Memory
        n |      min    med*     ave     max   total |      min     med     ave     max ␣
↪ total |      min     med     ave     max   total |      min     med     ave     max  ␣
↪total
        1 |     0.14    0.14    0.14    0.14    0.14 |     0.13    0.13    0.13    0.13 ␣
↪  0.13 |     0.01    0.01    0.01    0.01    0.01 |      76K     76K     76K     76K  ␣
↪  76K
Job
 Worker Jobs  |     min    med    ave    max
              |       3      3      3      3
    Count |                                     Time* |                                 ␣
↪ Clock |                                     Wait |                                   ␣
↪Memory
        n |      min    med*     ave     max   total |      min     med     ave     max ␣
↪ total |      min     med     ave     max   total |      min     med     ave     max  ␣
↪total
        3 |     0.01    0.06    0.05    0.07    0.14 |     0.00    0.06    0.04    0.07 ␣
↪  0.12 |     0.00    0.01    0.00    0.01    0.01 |      76K     76K     76K     76K  ␣
↪ 229K
 sysFiles
    Count |                                     Time* |                                 ␣
↪ Clock |                                     Wait |                                   ␣
↪Memory
        n |      min    med*     ave     max   total |      min     med     ave     max ␣
↪ total |      min     med     ave     max   total |      min     med     ave     max  ␣
↪total
        1 |     0.01    0.01    0.01    0.01    0.01 |     0.00    0.00    0.00    0.00 ␣
```

(continues on next page)

(continued from previous page)

```
↪  0.00 |     0.01    0.01    0.01    0.01    0.01 |      76K     76K     76K     76K  ␣
↪  76K
 userFiles
    Count |                                  Time* |                                   ␣
↪ Clock |                                  Wait |                                  ␣
↪Memory
        n |      min    med*     ave     max   total |      min     med     ave     max ␣
↪ total |      min     med     ave     max   total |      min     med     ave     max  ␣
↪total
        1 |     0.06    0.06    0.06    0.06    0.06 |     0.06    0.06    0.06    0.06 ␣
↪  0.06 |     0.01    0.01    0.01    0.01    0.01 |      76K     76K     76K     76K  ␣
↪  76K
 discoverFiles
    Count |                                  Time* |                                   ␣
↪ Clock |                                  Wait |                                  ␣
↪Memory
        n |      min    med*     ave     max   total |      min     med     ave     max ␣
↪ total |      min     med     ave     max   total |      min     med     ave     max  ␣
↪total
        1 |     0.07    0.07    0.07    0.07    0.07 |     0.07    0.07    0.07    0.07 ␣
↪  0.07 |     0.00    0.00    0.00    0.00    0.00 |      76K     76K     76K     76K  ␣
↪  76K
```

Once we're done, we can clean up the job store by running

```
toil clean file:my-jobstore
```

7.6 Status Command

Continuing the example from the stats section above, if we ran our workflow with the command

```
python discoverfiles.py file:my-jobstore --stats
```

We could interrogate our jobstore with the status command, for example:

```
toil status file:my-jobstore
```

If the run was successful, this would not return much valuable information, something like

```
2018-01-11 19:31:29,739 - toil.lib.bioio - INFO - Root logger is at level 'INFO', 'toil'␣
↪logger at level 'INFO'.
2018-01-11 19:31:29,740 - toil.utils.toilStatus - INFO - Parsed arguments
2018-01-11 19:31:29,740 - toil.utils.toilStatus - INFO - Checking if we have files for␣
↪Toil
The root job of the job store is absent, the workflow completed successfully.
```

Otherwise, the `status` command should return the following:

> There are `x` unfinished jobs, `y` parent jobs with children, `z` jobs with services, `a` services, and `b` totally failed jobs currently in `c`.

7.7 Clean Command

If a Toil pipeline didn't finish successfully, or was run using `--clean=always` or `--stats`, the job store will exist until it is deleted. `toil clean <jobStore>` ensures that all artifacts associated with a job store are removed. This is particularly useful for deleting AWS job stores, which reserves an SDB domain as well as an S3 bucket.

The deletion of the job store can be modified by the `--clean` argument, and may be set to `always`, `onError`, `never`, or `onSuccess` (default).

Temporary directories where jobs are running can also be saved from deletion using the `--cleanWorkDir`, which has the same options as `--clean`. This option should only be run when debugging, as intermediate jobs will fill up disk space.

7.8 Launch-Cluster Command

Running `toil launch-cluster` starts up a leader for a cluster. Workers can be added to the initial cluster by specifying the `-w` option. An example would be

```
$ toil launch-cluster my-cluster \
      --leaderNodeType t2.small -z us-west-2a \
      --keyPairName your-AWS-key-pair-name \
      --nodeTypes m3.large,t2.micro -w 1,4
```

Options are listed below. These can also be displayed by running

```
$ toil launch-cluster --help
```

launch-cluster's main positional argument is the clusterName. This is simply the name of your cluster. If it does not exist yet, Toil will create it for you.

Launch-Cluster Options

--help -h also accepted. Displays this help menu.

--tempDirRoot TEMPDIRROOT Path to the temporary directory where all temp files are created, by default uses the current working directory as the base.

--version Display version.

--provisioner CLOUDPROVIDER -p CLOUDPROVIDER also accepted. The provisioner for cluster auto-scaling. Both AWS and GCE are currently supported.

--zone ZONE -z ZONE also accepted. The availability zone of the leader. This parameter can also be set via the TOIL_AWS_ZONE or TOIL_GCE_ZONE environment variables, or by the ec2_region_name parameter in your .boto file if using AWS, or derived from the instance metadata if using this utility on an existing EC2 instance.

--leaderNodeType LEADERNODETYPE Non-preemptable node type to use for the cluster leader.

--keyPairName KEYPAIRNAME The name of the AWS or ssh key pair to include on the instance.

--owner OWNER The owner tag for all instances. If not given, the value in TOIL_OWNER_TAG will be used, or else the value of –keyPairName.

--boto BOTOPATH The path to the boto credentials directory. This is transferred to all nodes in order to access the AWS jobStore from non-AWS instances.

--tag KEYVALUE KEYVALUE is specified as KEY=VALUE. -t KEY=VALUE also accepted. Tags are added to the AWS cluster for this node and all of its children. Tags are of the form: -t key1=value1 –tag key2=value2. Multiple tags are allowed and each tag needs its own flag. By default the cluster is tagged with: { “Name”: clusterName, “Owner”: IAM username }.

--vpcSubnet VPCSUBNET VPC subnet ID to launch cluster leader in. Uses default subnet if not specified. This subnet needs to have auto assign IPs turned on.

--nodeTypes NODETYPES Comma-separated list of node types to create while launching the leader. The syntax for each node type depends on the provisioner used. For the AWS provisioner this is the name of an EC2 instance type followed by a colon and the price in dollars to bid for a spot instance, for example ‘c3.8xlarge:0.42’. Must also provide the –workers argument to specify how many workers of each node type to create.

--workers WORKERS -w WORKERS also accepted. Comma-separated list of the number of workers of each node type to launch alongside the leader when the cluster is created. This can be useful if running toil without autoscaling but with need of more hardware support.

--leaderStorage LEADERSTORAGE Specify the size (in gigabytes) of the root volume for the leader instance. This is an EBS volume.

--nodeStorage NODESTORAGE Specify the size (in gigabytes) of the root volume for any worker instances created when using the -w flag. This is an EBS volume.

--nodeStorageOverrides NODESTORAGEOVERRIDES Comma-separated list of nodeType:nodeStorage that are used to override the default value from –nodeStorage for the specified nodeType(s). This is useful for heterogeneous jobs where some tasks require much more disk than others.

Logging Options

--logOff Same as -\-logCritical.

--logCritical Turn on logging at level CRITICAL and above. (default is INFO)

--logError Turn on logging at level ERROR and above. (default is INFO)

--logWarning Turn on logging at level WARNING and above. (default is INFO)

--logInfo Turn on logging at level INFO and above. (default is INFO)

--logDebug Turn on logging at level DEBUG and above. (default is INFO)

--logLevel LOGLEVEL Log at given level (may be either OFF (or CRITICAL), ERROR, WARN (or WARNING), INFO or DEBUG). (default is INFO)

--logFile LOGFILE File to log in.

--rotatingLogging Turn on rotating logging, which prevents log files getting too big.

7.9 Ssh-Cluster Command

Toil provides the ability to ssh into the leader of the cluster. This can be done as follows:

```
$ toil ssh-cluster CLUSTER-NAME-HERE
```

This will open a shell on the Toil leader and is used to start an *Running a Workflow with Autoscaling* run. Issues with docker prevent using `screen` and `tmux` when sshing the cluster (The shell doesn't know that it is a TTY which prevents it from allocating a new screen session). This can be worked around via

```
$ script
$ screen
```

Simply running `screen` within `script` will get things working properly again.

Finally, you can execute remote commands with the following syntax:

```
$ toil ssh-cluster CLUSTER-NAME-HERE remoteCommand
```

It is not advised that you run your Toil workflow using remote execution like this unless a tool like nohup is used to ensure the process does not die if the SSH connection is interrupted.

For an example usage, see *Running a Workflow with Autoscaling*.

7.10 Rsync-Cluster Command

The most frequent use case for the `rsync-cluster` utility is deploying your Toil script to the Toil leader. Note that the syntax is the same as traditional rsync with the exception of the hostname before the colon. This is not needed in `toil rsync-cluster` since the hostname is automatically determined by Toil.

Here is an example of its usage:

```
$ toil rsync-cluster CLUSTER-NAME-HERE \
   ~/localFile :/remoteDestination
```

7.11 Destroy-Cluster Command

The `destroy-cluster` command is the advised way to get rid of any Toil cluster launched using the *Launch-Cluster Command* command. It ensures that all attached nodes, volumes, security groups, etc. are deleted. If a node or cluster is shut down using Amazon's online portal residual resources may still be in use in the background. To delete a cluster run

```
$ toil destroy-cluster CLUSTER-NAME-HERE
```

7.12 Kill Command

To kill all currently running jobs for a given jobstore, use the command

```
toil kill file:my-jobstore
```

CHAPTER

EIGHT

HPC ENVIRONMENTS

Toil is a flexible framework that can be leveraged in a variety of environments, including high-performance computing (HPC) environments. Toil provides support for a number of batch systems, including Grid Engine, Slurm, Torque and LSF, which are popular schedulers used in these environments. Toil also supports HTCondor, which is a popular scheduler for high-throughput computing (HTC). To use one of these batch systems specify the "-\-batchSystem" argument to the toil script.

Due to the cost and complexity of maintaining support for these schedulers we currently consider them to be "community supported", that is the core development team does not regularly test or develop support for these systems. However, there are members of the Toil community currently deploying Toil in HPC environments and we welcome external contributions.

Developing the support of a new or existing batch system involves extending the abstract batch system class *toil.batchSystems.abstractBatchSystem.AbstractBatchSystem*.

8.1 Standard Output/Error from Batch System Jobs

Standard output and error from batch system jobs (except for the Parasol and Mesos batch systems) are redirected to files in the `toil-<workflowID>` directory created within the temporary directory specified by the `--workDir` option; see *Commandline Options*. Each file is named as follows: `toil_job_<Toil job ID>_batch_<name of batch system>_<job ID from batch system>_<file description>.log`, where `<file description>` is `std_output` for standard output, and `std_error` for standard error. HTCondor will also write job event log files with `<file description> = job_events`.

If capturing standard output and error is desired, `--workDir` will generally need to be on a shared file system; otherwise if these are written to local temporary directories on each node (e.g. `/tmp`) Toil will not be able to retrieve them. Alternatively, the `--noStdOutErr` option forces Toil to discard all standard output and error from batch system jobs.

CHAPTER

NINE

CWL IN TOIL

The Common Workflow Language (CWL) is an emerging standard for writing workflows that are portable across multiple workflow engines and platforms. Toil has full support for the CWL v1.0, v1.1, and v1.2 standards.

9.1 Running CWL Locally

The *toil-cwl-runner* command provides cwl-parsing functionality using cwltool, and leverages the job-scheduling and batch system support of Toil.

To run in local batch mode, provide the CWL file and the input object file:

```
$ toil-cwl-runner example.cwl example-job.yml
```

For a simple example of CWL with Toil see *Running a basic CWL workflow*.

9.1.1 Note for macOS + Docker + Toil

When invoking CWL documents that make use of Docker containers if you see errors that look like

```
docker: Error response from daemon: Mounts denied:
The paths /var/...tmp are not shared from OS X and are not known to Docker.
```

you may need to add

```
export TMPDIR=/tmp/docker_tmp
```

either in your startup file (`.bashrc`) or add it manually in your shell before invoking toil.

9.2 Detailed Usage Instructions

Help information can be found by using this toil command:

```
$ toil-cwl-runner -h
```

A more detailed example shows how we can specify both Toil and cwltool arguments for our workflow:

```
$ toil-cwl-runner \
    --singularity \
    --jobStore my_jobStore \
    --batchSystem lsf \
    --workDir `pwd` \
    --outdir `pwd` \
    --logFile cwltoil.log \
    --writeLogs `pwd` \
    --logLevel DEBUG \
    --retryCount 2 \
    --maxLogFileSize 20000000000 \
    --stats \
    standard_bam_processing.cwl \
    inputs.yaml
```

In this example, we set the following options, which are all passed to Toil:

`--singularity`: Specifies that all jobs with Docker format containers specified should be run using the Singularity container engine instead of the Docker container engine.

`--jobStore`: Path to a folder which doesn't exist yet, which will contain the Toil jobstore and all related job-tracking information.

`--batchSystem`: Use the specified HPC or Cloud-based cluster platform.

`--workDir`: The directory where all temporary files will be created for the workflow. A subdirectory of this will be set as the `$TMPDIR` environment variable and this subdirectory can be referenced using the CWL parameter reference `$(runtime.tmpdir)` in CWL tools and workflows.

`--outdir`: Directory where final `File` and `Directory` outputs will be written. References to these and other output types will be in the JSON object printed to the stdout stream after workflow execution.

`--logFile`: Path to the main logfile with logs from all jobs.

`--writeLogs`: Directory where all job logs will be stored.

`--retryCount`: How many times to retry each Toil job.

`--maxLogFileSize`: Logs that get larger than this value will be truncated.

`--stats`: Save resources usages in json files that can be collected with the `toil stats` command after the workflow is done.

`--disable-streaming`: Does not allow streaming of input files. This is enabled by default for files marked with `streamable` flag True and only for remote files when the jobStore is not on local machine.

9.3 Running CWL in the Cloud

To run in cloud and HPC configurations, you may need to provide additional command line parameters to select and configure the batch system to use.

To run a CWL workflow in AWS with toil see *Running a CWL Workflow on AWS*.

9.4 Running CWL within Toil Scripts

A CWL workflow can be run indirectly in a native Toil script. However, this is not the *standard* way to run CWL workflows with Toil and doing so comes at the cost of job efficiency. For some use cases, such as running one process on multiple files, it may be useful. For example, if you want to run a CWL workflow with 3 YML files specifying different samples inputs, it could look something like:

```
import os
import subprocess
import tempfile

from toil.common import Toil
from toil.job import Job


def initialize_jobs(job):
    job.fileStore.logToMaster('initialize_jobs')


def runQC(job, cwl_file, cwl_filename, yml_file, yml_filename, outputs_dir, output_num):
    job.fileStore.logToMaster("runQC")
    tempDir = job.fileStore.getLocalTempDir()

    cwl = job.fileStore.readGlobalFile(cwl_file, userPath=os.path.join(tempDir, cwl_
↪filename))
    yml = job.fileStore.readGlobalFile(yml_file, userPath=os.path.join(tempDir, yml_
↪filename))

    subprocess.check_call(["toil-cwl-runner", cwl, yml])

    output_filename = "output.txt"
    output_file = job.fileStore.writeGlobalFile(output_filename)
    job.fileStore.readGlobalFile(output_file, userPath=os.path.join(outputs_dir, "sample_
↪" + output_num + "_" + output_filename))
    return output_file


if __name__ == "__main__":
    jobstore: str = tempfile.mkdtemp("tutorial_cwlexample")
    os.rmdir(jobstore)
    options = Job.Runner.getDefaultOptions(jobstore)
    options.logLevel = "INFO"
    options.clean = "always"
    with Toil(options) as toil:

        # specify the folder where the cwl and yml files live
        inputs_dir = os.path.join(os.path.dirname(os.path.abspath(__file__)),
↪"cwlExampleFiles")
        # specify where you wish the outputs to be written
        outputs_dir = os.path.join(os.path.dirname(os.path.abspath(__file__)),
↪"cwlExampleFiles")

        job0 = Job.wrapJobFn(initialize_jobs)
```

(continues on next page)

(continued from previous page)

```
        cwl_filename = "hello.cwl"
        cwl_file = toil.importFile("file://" + os.path.abspath(os.path.join(inputs_dir,␣
↪cwl_filename)))

        # add list of yml config inputs here or import and construct from file
        yml_files = ["hello1.yml", "hello2.yml", "hello3.yml"]
        i = 0
        for yml in yml_files:
            i = i + 1
            yml_file = toil.importFile("file://" + os.path.abspath(os.path.join(inputs_
↪dir, yml)))
            yml_filename = yml
            job = Job.wrapJobFn(runQC, cwl_file, cwl_filename, yml_file, yml_filename,␣
↪outputs_dir, output_num=str(i))
            job0.addChild(job)

        toil.start(job0)
```

9.5 Running CWL workflows with InplaceUpdateRequirement

Some CWL workflows use the `InplaceUpdateRequirement` feature, which requires that operations on files have visible side effects that Toil's file store cannot support. If you need to run a workflow like this, you can make sure that all of your worker nodes have a shared filesystem, and use the `--bypass-file-store` option to `toil-cwl-runner`. This will make it leave all CWL intermediate files on disk and share them between jobs using file paths, instead of storing them in the file store and downloading them when jobs need them.

9.6 Toil & CWL Tips

See logs for just one job by using the full log file

This requires knowing the job's toil-generated ID, which can be found in the log files.

```
cat cwltoil.log | grep jobVM1fIs
```

Grep for full tool commands from toil logs

This gives you a more concise view of the commands being run (note that this information is only available from Toil when running with *–logDebug*).

```
pcregrep -M "\[job .*\.cwl.*$\n(.*        .*$\n)*" cwltoil.log
#         ^allows for multiline matching
```

Find Bams that have been generated for specific step while pipeline is running:

```
find . | grep -P '^./out_tmpdir.*_MD\.bam$'
```

See what jobs have been run

```
cat log/cwltoil.log | grep -oP "\[job .*.cwl\]" | sort | uniq
```

or:

```
cat log/cwltoil.log | grep -i "issued job"
```

Get status of a workflow

```
$ toil status /home/johnsoni/TEST_RUNS_3/TEST_run/tmp/jobstore-09ae0acc-c800-11e8-9d09-
↪70106fb1697e
<hostname> 2018-10-04 15:01:44,184 MainThread INFO toil.lib.bioio: Root logger is at␣
↪level 'INFO', 'toil' logger at level 'INFO'.
<hostname> 2018-10-04 15:01:44,185 MainThread INFO toil.utils.toilStatus: Parsed␣
↪arguments
<hostname> 2018-10-04 15:01:47,081 MainThread INFO toil.utils.toilStatus: Traversing the␣
↪job graph gathering jobs. This may take a couple of minutes.

Of the 286 jobs considered, there are 179 jobs with children, 107 jobs ready to run, 0␣
↪zombie jobs, 0 jobs with services, 0 services, and 0 jobs with log files currently in␣
↪file:/home/user/jobstore-09ae0acc-c800-11e8-9d09-70106fb1697e.
```

Toil Stats

You can get run statistics broken down by CWL file. This only works once the workflow is finished:

```
$ toil stats /path/to/jobstore
```

The output will contain CPU, memory, and walltime information for all CWL job types:

```
<hostname> 2018-10-15 12:06:19,003 MainThread INFO toil.lib.bioio: Root logger is at␣
↪level 'INFO', 'toil' logger at level 'INFO'.
<hostname> 2018-10-15 12:06:19,004 MainThread INFO toil.utils.toilStats: Parsed arguments
<hostname> 2018-10-15 12:06:19,004 MainThread INFO toil.utils.toilStats: Checking if we␣
↪have files for toil
<hostname> 2018-10-15 12:06:19,004 MainThread INFO toil.utils.toilStats: Checked␣
↪arguments
Batch System: lsf
Default Cores: 1  Default Memory: 10485760K
Max Cores: 9.22337e+18
Total Clock: 106608.01  Total Runtime: 86634.11
Worker
    Count |                                          Time* |                                 ␣
↪        Clock |                                                Wait |                   ␣
↪                Memory
        n |      min    med*     ave      max     total |      min     med      ave     ␣
↪max     total |        min      med       ave      max      total |      min     med  ␣
↪  ave     max    total
     1659 |     0.00    0.80  264.87 12595.59 439424.40 |     0.00    0.46   449.05␣
↪42240.74 744968.80 |  -35336.69     0.16   -184.17  4230.65 -305544.39 |      48K   ␣
↪223K   1020K  40235K 1692300K
Job
 Worker Jobs  |     min    med    ave    max
              |    1077   1077   1077   1077
    Count |                                          Time* |                                 ␣
↪        Clock |                                                Wait |                   ␣
↪                Memory
        n |      min    med*     ave      max     total |      min     med      ave     ␣
```

(continues on next page)

(continued from previous page)

```
↪max     total |        min      med       ave      max      total |      min     med␣
↪  ave     max    total
    1077 |     0.04    1.18  407.06 12593.43 438404.73 |     0.01    0.28   691.17␣
↪42240.35 744394.14 |  -35336.83     0.27   -284.11  4230.49 -305989.41 |     135K  ␣
↪268K   1633K  40235K 1759734K
ResolveIndirect
   Count |                                        Time* |                          ␣
↪        Clock |                                              Wait |                ␣
↪                Memory
       n |      min    med*     ave      max     total |      min     med      ave     ␣
↪max     total |        min      med       ave      max      total |      min     med  ␣
↪  ave     max    total
     205 |     0.04    0.07    0.16     2.29     31.95 |     0.01    0.02     0.02    ␣
↪0.14      3.60 |       0.02     0.05      0.14     2.28      28.35 |     190K    266K ␣
↪  256K    314K   52487K
CWLGather
   Count |                                        Time* |                          ␣
↪        Clock |                                              Wait |                ␣
↪                Memory
       n |      min    med*     ave      max     total |      min     med      ave     ␣
↪max     total |        min      med       ave      max      total |      min     med  ␣
↪  ave     max    total
      40 |     0.05    0.17    0.29     1.90     11.62 |     0.01    0.02     0.02    ␣
↪0.05      0.80 |       0.03     0.14      0.27     1.88      10.82 |     188K    265K ␣
↪  250K    316K   10039K
CWLWorkflow
   Count |                                        Time* |                          ␣
↪        Clock |                                              Wait |                ␣
↪                Memory
       n |      min    med*     ave      max     total |      min     med      ave     ␣
↪max     total |        min      med       ave      max      total |      min     med  ␣
↪  ave     max    total
     205 |     0.09    0.40    0.98    13.70    200.82 |     0.04    0.15     0.16    ␣
↪1.08     31.78 |       0.04     0.26      0.82    12.62     169.04 |     190K    270K ␣
↪  257K    316K   52826K
file:///home/johnsoni/pipeline_0.0.39/ACCESS-Pipeline/cwl_tools/expression_tools/group_
↪waltz_files.cwl
   Count |                                        Time* |                          ␣
↪        Clock |                                              Wait |                ␣
↪                Memory
       n |      min    med*     ave      max     total |      min     med      ave     ␣
↪max     total |        min      med       ave      max      total |      min     med  ␣
↪  ave     max    total
      99 |     0.29    0.49    0.59     2.50     58.11 |     0.14    0.26     0.29    ␣
↪1.04     28.95 |       0.14     0.22      0.29     1.48      29.16 |     135K    135K ␣
↪  135K    136K   13459K
file:///home/johnsoni/pipeline_0.0.39/ACCESS-Pipeline/cwl_tools/expression_tools/make_
↪sample_output_dirs.cwl
   Count |                                        Time* |                          ␣
↪        Clock |                                              Wait |                ␣
↪                Memory
       n |      min    med*     ave      max     total |      min     med      ave     ␣
```

(continues on next page)

(continued from previous page)

```
↪max     total |        min      med       ave      max      total |      min     med  ␣
↪  ave     max    total
      11 |     0.34    0.52    0.74     2.63      8.18 |     0.20    0.30     0.41    ␣
↪1.17      4.54 |       0.14     0.20      0.33     1.45       3.65 |     136K    136K ␣
↪  136K    136K    1496K
 file:///home/johnsoni/pipeline_0.0.39/ACCESS-Pipeline/cwl_tools/expression_tools/
↪consolidate_files.cwl
    Count |                                        Time* |                            ␣
↪        Clock |                                                Wait |                 ␣
↪                Memory
       n |      min    med*     ave      max     total |      min     med      ave     ␣
↪max     total |        min      med       ave      max      total |      min     med  ␣
↪  ave     max    total
       8 |     0.31    0.59    0.71     1.80      5.69 |     0.18    0.35     0.37    ␣
↪0.63      2.94 |       0.13     0.27      0.34     1.17       2.75 |     136K    136K ␣
↪  136K    136K    1091K
 file:///home/johnsoni/pipeline_0.0.39/ACCESS-Pipeline/cwl_tools/bwa-mem/bwa-mem.cwl
    Count |                                        Time* |                            ␣
↪        Clock |                                                Wait |                 ␣
↪                Memory
       n |      min    med*     ave      max     total |      min     med      ave     ␣
↪max     total |        min      med       ave      max      total |      min     med  ␣
↪  ave     max    total
      22 |   895.76 3098.13 3587.34 12593.43  78921.51 |  2127.02 7910.31  8123.06␣
↪16959.13 178707.34 |  -11049.84 -3827.96  -4535.72    19.49  -99785.83 |    5659K  ␣
↪5950K   5854K   6128K  128807K
```

Understanding toil log files

There is a *worker_log.txt* file for each job, this file is written to while the job is running, and deleted after the job finishes. The contents are printed to the main log file and transferred to a log file in the *–logDir* folder once the job is completed successfully.

The new log file will be named something like:

```
file:<path to cwl tool>.cwl_<job ID>.log

file:---home-johnsoni-pipeline_1.1.14-ACCESS--Pipeline-cwl_tools-marianas-
↪ProcessLoopUMIFastq.cwl_I-O-jobfGsQQw000.log
```

This is the toil job command with spaces replaced by dashes.

CHAPTER

TEN

WDL IN TOIL

Toil has beta support for running WDL workflows, using the `toil-wdl-runner` command.

10.1 Running WDL with Toil

You can run WDL workflows with `toil-wdl-runner`. Currently, `toil-wdl-runner` works by using MiniWDL to parse and interpret the WDL workflow, and has support for workflows in WDL 1.0 or later (which are required to declare a `version` and to use `inputs` and `outputs` sections).

You can write workflows like this by following the official WDL tutorials.

When you reach the point of executing your workflow, instead of running with Cromwell:

```
java -jar Cromwell.jar run myWorkflow.wdl --inputs myWorkflow_inputs.json
```

you can instead run with `toil-wdl-runner`:

```
toil-wdl-runner myWorkflow.wdl --inputs myWorkflow_inputs.json
```

This will default to executing on the current machine, with a job store in an automatically determined temporary location, but you can add a few Toil options to use other Toil-supported batch systems, such as Kubernetes:

```
toil-wdl-runner --jobStore aws:us-west-2:wdl-job-store --batchSystem kubernetes
myWorkflow.wdl --inputs myWorkflow_inputs.json
```

For Toil, the `--inputs` is optional, and inputs can be passed as a positional argument:

```
toil-wdl-runner myWorkflow.wdl myWorkflow_inputs.json
```

You can also run workflows from URLs. For example, to run the MiniWDL self test workflow, you can do:

```
toil-wdl-runner https://raw.githubusercontent.com/DataBiosphere/toil/
36b54c45e8554ded5093bcdd03edb2f6b0d93887/src/toil/test/wdl/miniwdl_self_test/
self_test.wdl https://raw.githubusercontent.com/DataBiosphere/toil/
36b54c45e8554ded5093bcdd03edb2f6b0d93887/src/toil/test/wdl/miniwdl_self_test/inputs.json
```

10.2 Toil WDL Runner Options

'-\-jobStore': Specifies where to keep the Toil state information while running the workflow. Must be accessible from all machines.

'-o' or **'-\-outputDirectory'**: Specifies the output folder to save workflow output files in. Defaults to a new directory in the current directory.

'-m' or **'-\-outputFile'**: Specifies a JSON file to save workflow output values to. Defaults to standard output.

'-i' or **'-\-input'**: Alternative to the positional argument for the input JSON file, for compatibility with other WDL runners.

'-\-outputDialect': Specifies an output format dialect. Can be `cromwell` to just return the workflow's output values as JSON or `miniwdl` to nest that under an `outputs` key and includes a `dir` key.

Any number of other Toil options may also be specified. For defined Toil options, see the documentation: http://toil.readthedocs.io/en/latest/running/cliOptions.html

10.3 WDL Specifications

WDL language specifications can be found here: https://github.com/broadinstitute/wdl/blob/develop/SPEC.md

Toil is not yet fully conformant with the WDL specification, but it inherits most of the functionality of MiniWDL.

10.4 Using the Old WDL Compiler

Up through Toil 5.9.2, `toil-wdl-runner` worked by compiling the WDL code to a Toil Python workflow, and executing that. The old compiler is still available as `toil-wdl-runner-old`.

The compiler implements:

- Scatter
- Many Built-In Functions
- Docker Calls
- Handles Priority, and Output File Wrangling
- Currently Handles Primitives and Arrays

The compiler DOES NOT implement:

- Robust cloud autoscaling
- WDL files that `import` other WDL files (including URI handling for 'http://' and 'https://')

Recommended best practice when running wdl files with `toil-wdl-runner-old` is to first use the Broad's wdltool for syntax validation and generating the needed json input file. Full documentation can be found in the repository, and a precompiled jar binary can be downloaded here: wdltool (this requires java7).

That means two steps. First, make sure your wdl file is valid and devoid of syntax errors by running

```
java -jar wdltool.jar validate example_wdlfile.wdl
```

Second, generate a complementary json file if your wdl file needs one. This json will contain keys for every necessary input that your wdl file needs to run:

```
java -jar wdltool.jar inputs example_wdlfile.wdl
```

When this json template is generated, open the file, and fill in values as necessary by hand. WDL files all require json files to accompany them. If no variable inputs are needed, a json file containing only '{}' may be required.

Once a wdl file is validated and has an appropriate json file, workflows can be compiled and run using:

```
toil-wdl-runner-old example_wdlfile.wdl example_jsonfile.json
```

10.4.1 Toil WDL Compiler Options

'-o' or **'-\-outdir'**: Specifies the output folder, and defaults to the current working directory if not specified by the user.

'-\-dev_mode': Creates "AST.out", which holds a printed AST of the wdl file and "mappings.out", which holds the printed task, workflow, csv, and tsv dictionaries generated by the parser. Also saves the compiled toil python workflow file for debugging.

Any number of arbitrary options may also be specified. These options will not be parsed immediately, but passed down as toil options once the wdl/json files are processed. For valid toil options, see the documentation: http://toil.readthedocs.io/en/latest/running/cliOptions.html

10.4.2 Compiler Example: ENCODE Example from ENCODE-DCC

For this example, we will run a WDL draft-2 workflow. This version is too old to be supported by `toil-wdl-runner`, so we will need to use `toil-wdl-runner-old`.

To follow this example, you will need docker installed. The original workflow can be found here: https://github.com/ENCODE-DCC/pipeline-container

We've included the wdl file and data files in the toil repository needed to run this example. First, download the example code and unzip. The file needed is "testENCODE/encode_mapping_workflow.wdl".

Next, use wdltool (this requires java7) to validate this file:

```
java -jar wdltool.jar validate encode_mapping_workflow.wdl
```

Next, use wdltool to generate a json file for this wdl file:

```
java -jar wdltool.jar inputs encode_mapping_workflow.wdl
```

This json file once opened should look like this:

```
{
"encode_mapping_workflow.fastqs": "Array[File]",
"encode_mapping_workflow.trimming_parameter": "String",
"encode_mapping_workflow.reference": "File"
}
```

You will need to edit this file to replace the types (like `Array[File]`) with values of those types.

The trimming_parameter should be set to 'native'.

For the file parameters, download the example data and unzip. Inside are two data files required for the run

`ENCODE_data/reference/GRCh38_chr21_bwa.tar.gz` `ENCODE_data/ENCFF000VOL_chr21.fq.gz`

Editing the json to include these as inputs, the json should now look something like this:

```
{
"encode_mapping_workflow.fastqs": ["/path/to/unzipped/ENCODE_data/ENCFF000VOL_chr21.fq.gz
↪"],
```

(continues on next page)

(continued from previous page)

```
"encode_mapping_workflow.trimming_parameter": "native",
"encode_mapping_workflow.reference": "/path/to/unzipped/ENCODE_data/reference/GRCh38_
↪chr21_bwa.tar.gz"
}
```

The wdl and json files can now be run using the command:

`toil-wdl-runner-old encode_mapping_workflow.wdl encode_mapping_workflow.json`

This should deposit the output files in the user's current working directory (to change this, specify a new directory with the '-o' option).

10.4.3 Compiler Example: GATK Examples from the Broad

Terra hosts some example documentation for using early, pre-1.0 versions of WDL, originally authored by the Broad: https://support.terra.bio/hc/en-us/sections/360007347652?name=wdl-tutorials

One can follow along with these tutorials, write their own old-style WDL files following the directions and run them using either Cromwell or Toil's old WDL compiler. For example, in tutorial 1, if you've followed along and named your wdl file 'helloHaplotypeCall.wdl', then once you've validated your wdl file using wdltool (this requires java7) using

`java -jar wdltool.jar validate helloHaplotypeCaller.wdl`

and generated a `json` file (and subsequently typed in appropriate file paths and variables) using

`java -jar wdltool.jar inputs helloHaplotypeCaller.wdl`

Note: Absolute filepath inputs are recommended for local testing with the Toil WDL compiler.

---

then the WDL script can be compiled and run using

`toil-wdl-runner-old helloHaplotypeCaller.wdl helloHaplotypeCaller_inputs.json`

CHAPTER

ELEVEN

WORKFLOW EXECUTION SERVICE (WES)

The GA4GH Workflow Execution Service (WES) is a standardized API for submitting and monitoring workflows. Toil has experimental support for setting up a WES server and executing CWL, WDL, and Toil workflows using the WES API. More information about the WES API specification can be found here.

To get started with the Toil WES server, make sure that the `server` extra (*Installing Toil with Extra Features*) is installed.

11.1 Preparing your WES environment

The WES server requires Celery to distribute and execute workflows. To set up Celery:

1. Start RabbitMQ, which is the broker between the WES server and Celery workers:

```
docker run -d --name wes-rabbitmq -p 5672:5672 rabbitmq:3.9.5
```

2. Start Celery workers:

```
celery -A toil.server.celery_app worker --loglevel=INFO
```

11.2 Starting a WES server

To start a WES server on the default port 8080, run the Toil command:

```
$ toil server
```

The WES API will be hosted on the following URL:

```
http://localhost:8080/ga4gh/wes/v1
```

To use another port, e.g.: 3000, you can specify the `--port` argument:

```
$ toil server --port 3000
```

There are many other command line options. Help information can be found by using this command:

```
$ toil server --help
```

Below is a detailed summary of all server-specific options:

--debug Enable debug mode.

--bypass_celery Skip sending workflows to Celery and just run them under the server. For testing.

--host HOST The host interface that the Toil server binds on. (default: "127.0.0.1").

--port PORT The port that the Toil server listens on. (default: 8080).

--swagger_ui If True, the swagger UI will be enabled and hosted on the *{api_base_path}/ui* endpoint. (default: False)

--cors Enable Cross Origin Resource Sharing (CORS). This should only be turned on if the server is intended to be used by a website or domain. (default: False).

--cors_origins CORS_ORIGIN Ignored if -//-cors is False. This sets the allowed origins for CORS. For details about CORS and its security risks, see the GA4GH docs on CORS. (default: "*").

--workers WORKERS, -w WORKERS Ignored if -\-debug is True. The number of worker processes launched by the WSGI server. (default: 2).

--work_dir WORK_DIR The directory where workflows should be stored. This directory should be empty or only contain previous workflows. (default: './workflows').

--state_store STATE_STORE The local path or S3 URL where workflow state metadata should be stored. (default: in -\-work_dir)

--opt OPT, -o OPT Specify the default parameters to be sent to the workflow engine for each run. Options taking arguments must use = syntax. Accepts multiple values. Example: -\-opt=-\-logLevel=CRITICAL -\-opt=-\-workDir=/tmp.

--dest_bucket_base DEST_BUCKET_BASE Direct CWL workflows to save output files to dynamically generated unique paths under the given URL. Supports AWS S3.

--wes_dialect DIALECT Restrict WES responses to a dialect compatible with clients that do not fully implement the WES standard. (default: 'standard')

11.3 Running the Server with *docker-compose*

Instead of manually setting up the server components (`toil server`, RabbitMQ, and Celery), you can use the following `docker-compose.yml` file to orchestrate and link them together.

Make sure to change the credentials for basic authentication by updating the `traefik.http.middlewares.auth.basicauth.users` label. The passwords can be generated with tools like `htpasswd` like this. (Note that single `$` signs need to be replaced with `$$` in the yaml file).

When running on a different host other than `localhost`, make sure to change the `Host` to your tartget host in the `traefik.http.routers.wes.rule` and `traefik.http.routers.wespublic.rule` labels.

You can also change `/tmp/toil-workflows` if you want Toil workflows to live somewhere else, and create the directory before starting the server.

In order to run workflows that require Docker, the `docker.sock` socket must be mounted as volume for Celery. Additionally, the `TOIL_WORKDIR` directory (defaults to: `/var/lib/toil`) and `/var/lib/cwl` (if running CWL workflows with `DockerRequirement`) should exist on the host and also be mounted as volumes.

Also make sure to run it behind a firewall; it opens up the Toil server on port 8080 to anyone who connects.

```
# docker-compose.yml
version: "3.8"

services:
```

(continues on next page)

(continued from previous page)

```
  rabbitmq:
    image: rabbitmq:3.9.5
    hostname: rabbitmq
  celery:
    image: ${TOIL_APPLIANCE_SELF}
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - /var/lib/docker:/var/lib/docker
      - /var/lib/toil:/var/lib/toil
      - /var/lib/cwl:/var/lib/cwl
      - /tmp/toil-workflows:/tmp/toil-workflows
    command: celery --broker=amqp://guest:guest@rabbitmq:5672// -A toil.server.celery_
↪app worker --loglevel=INFO
    depends_on:
      - rabbitmq
  wes-server:
    image: ${TOIL_APPLIANCE_SELF}
    volumes:
      - /tmp/toil-workflows:/tmp/toil-workflows
    environment:
      - TOIL_WES_BROKER_URL=amqp://guest:guest@rabbitmq:5672//
    command: toil server --host 0.0.0.0 --port 8000 --work_dir /tmp/toil-workflows
    expose:
      - 8000
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.wes.rule=Host(`localhost`)"
      - "traefik.http.routers.wes.entrypoints=web"
      - "traefik.http.routers.wes.middlewares=auth"
      - "traefik.http.middlewares.auth.basicauth.users=test:$$2y$$12$$ci.
↪4U63YX83CwkyUrjqxAucnmi2xXOIlEF6T/KdP9824f1Rf1iyNG"
      - "traefik.http.routers.wespublic.rule=Host(`localhost`) && Path(`/ga4gh/wes/v1/
↪service-info`)"
    depends_on:
      - rabbitmq
      - celery
  traefik:
    image: traefik:v2.2
    command:
      - "--providers.docker"
      - "--providers.docker.exposedbydefault=false"
      - "--entrypoints.web.address=:8080"
    ports:
      - "8080:8080"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

Further customization can also be made as needed. For example, if you have a domain, you can set up HTTPS with Let's Encrypt.

Once everything is configured, simply run `docker-compose up` to start the containers. Run `docker-compose down` to stop and remove all containers.

Note: `docker-compose` is not installed on the Toil appliance by default. See the following section to set up the WES server on a Toil cluster.

11.4 Running on a Toil cluster

To run the server on a Toil leader instance on EC2:

1. Launch a Toil cluster with the `toil launch-cluster` command with the AWS provisioner
2. SSH into your cluster with the `--sshOption=-L8080:localhost:8080` option to forward port `8080`
3. Install Docker Compose by running the following commands from the Docker docs:

```
curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-
↪$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
chmod +x /usr/local/bin/docker-compose

# check installation
docker-compose --version
```

   or, install a different version of Docker Compose by changing `"1.29.2"` to another version.

4. Copy the `docker-compose.yml` file from (*Running the Server with docker-compose*) to an empty directory, and modify the configuration as needed.
5. Now, run `docker-compose up -d` to start the WES server in detach mode on the Toil appliance.
6. To stop the server, run `docker-compose down`.

11.5 WES API Endpoints

As defined by the GA4GH WES API specification, the following endpoints with base path `ga4gh/wes/v1/` are supported by Toil:

GET /service-info	Get information about the Workflow Execution Service.
GET /runs	List the workflow runs.
POST /runs	Run a workflow. This endpoint creates a new workflow run and returns a `run_id` to monitor its progress.
GET /runs/{run_id}	Get detailed info about a workflow run.
POST /runs/{run_id}/cancel	Cancel a running workflow.
GET /runs/{run_id}/status	Get the status (overall state) of a workflow run.

When running the WES server with the `docker-compose` setup above, most endpoints (except `GET /service-info`) will be protected with basic authentication. Make sure to set the **Authorization** header with the correct credentials when submitting or retrieving a workflow.

11.6 Submitting a Workflow

Now that the WES API is up and running, we can submit and monitor workflows remotely using the WES API endpoints. A workflow can be submitted for execution using the `POST /runs` endpoint.

As a quick example, we can submit the example CWL workflow from *Running a basic CWL workflow* to our WES API:

```
# example.cwl
cwlVersion: v1.0
class: CommandLineTool
baseCommand: echo
stdout: output.txt
inputs:
 message:
   type: string
   inputBinding:
     position: 1
outputs:
 output:
   type: stdout
```

using cURL:

```
$ curl --location --request POST 'http://localhost:8080/ga4gh/wes/v1/runs' \
    --user test:test \
    --form 'workflow_url="example.cwl"' \
    --form 'workflow_type="cwl"' \
    --form 'workflow_type_version="v1.0"' \
    --form 'workflow_params="{\"message\": \"Hello world!\"}"' \
    --form 'workflow_attachment=@"./toil_test_files/example.cwl"'
{
  "run_id": "4deb8beb24894e9eb7c74b0f010305d1"
}
```

Note that the `--user` argument is used to attach the basic authentication credentials along with the request. Make sure to change `test:test` to the username and password you configured for your WES server. Alternatively, you can also set the **Authorization** header manually as `"Authorization: Basic base64_encoded_auth"`.

If the workflow is submitted successfully, a JSON object containing a `run_id` will be returned. The `run_id` is a unique identifier of your requested workflow, which can be used to monitor or cancel the run.

There are a few required parameters that have to be set for all workflow submissions, which are the following:

workflow_url	The URL of the workflow to run. This can refer to a file from `workflow_attachment`.
workflow_type	The type of workflow language. Toil currently supports one of the following: `"CWL"`, `"WDL"`, or `"py"`. To run a Toil native python script, set this to `"py"`.
workflow_type_version	The version of the workflow language. Supported versions can be found by accessing the `GET /service-info` endpoint of your WES server.
workflow_params	A JSON object that specifies the inputs of the workflow.

Additionally, the following optional parameters are also available:

workflow_attachment	A list of files associated with the workflow run.
workflow_engine_parameters	A JSON key-value map of workflow engine parameters to send to the runner. Example: `{"--logLevel": "INFO", "--workDir": "/tmp/"}`
tags	A JSON key-value map of metadata associated with the workflow.

For more details about these parameters, refer to the Run Workflow section in the WES API spec.

11.6.1 Upload multiple files

Looking at the body of the request of the previous example, note that the `workflow_url` is a relative URL that refers to the `example.cwl` file uploaded from the local path `./toil_test_files/example.cwl`.

To specify the file name (or subdirectory) of the remote destination file, set the `filename` field in the `Content-Disposition` header. You could also upload more than one file by providing the `workflow_attachment` parameter multiple times with different files.

This can be shown by the following example:

```
$ curl --location --request POST 'http://localhost:8080/ga4gh/wes/v1/runs' \
    --user test:test \
    --form 'workflow_url="example.cwl"' \
    --form 'workflow_type="cwl"' \
    --form 'workflow_type_version="v1.0"' \
    --form 'workflow_params="{\"message\": \"Hello world!\"}"' \
    --form 'workflow_attachment=@"./toil_test_files/example.cwl"' \
    --form 'workflow_attachment=@"./toil_test_files/2.fasta";filename=inputs/test.fasta'␣
↪\
    --form 'workflow_attachment=@"./toil_test_files/2.fastq";filename=inputs/test.fastq'
```

On the server, the execution directory would have the following structure from the above request:

```
execution/
├── example.cwl
├── inputs
│   ├── test.fasta
|   └── test.fastq
└── wes_inputs.json
```

11.6.2 Specify Toil options

To pass Toil-specific parameters to the workflow, you can include the `workflow_engine_parameters` parameter along with your request.

For example, to set the logging level to `INFO`, and change the working directory of the workflow, simply include the following as `workflow_engine_parameters`:

```
{"--logLevel": "INFO", "--workDir": "/tmp/"}
```

These options would be appended at the end of existing parameters during command construction, which would override the default parameters if provided. (Default parameters that can be passed multiple times would not be overridden).

11.7 Monitoring a Workflow

With the `run_id` returned when submitting the workflow, we can check the status or get the full logs of the workflow run.

11.7.1 Checking the state

The `GET /runs/{run_id}/status` endpoint can be used to get a simple result with the overall state of your run:

```
$ curl --user test:test http://localhost:8080/ga4gh/wes/v1/runs/
↪4deb8beb24894e9eb7c74b0f010305d1/status
{
  "run_id": "4deb8beb24894e9eb7c74b0f010305d1",
  "state": "RUNNING"
}
```

The possible states here are: `QUEUED`, `INITIALIZING`, `RUNNING`, `COMPLETE`, `EXECUTOR_ERROR`, `SYSTEM_ERROR`, `CANCELING`, and `CANCELED`.

11.7.2 Getting the full logs

To get the detailed information about a workflow run, use the `GET /runs/{run_id}` endpoint:

```
$ curl --user test:test http://localhost:8080/ga4gh/wes/v1/runs/
↪4deb8beb24894e9eb7c74b0f010305d1
{
  "run_id": "4deb8beb24894e9eb7c74b0f010305d1",
  "request": {
    "workflow_attachment": [
      "example.cwl"
    ],
    "workflow_url": "example.cwl",
    "workflow_type": "cwl",
    "workflow_type_version": "v1.0",
    "workflow_params": {
      "message": "Hello world!"
    }
  },
  "state": "RUNNING",
  "run_log": {
    "cmd": [
      "toil-cwl-runner --outdir=/home/toil/workflows/4deb8beb24894e9eb7c74b0f010305d1/
↪outputs --jobStore=file:/home/toil/workflows/4deb8beb24894e9eb7c74b0f010305d1/toil_job_
↪store /home/toil/workflows/4deb8beb24894e9eb7c74b0f010305d1/execution/example.cwl /
↪home/workflows/4deb8beb24894e9eb7c74b0f010305d1/execution/wes_inputs.json"
    ],
    "start_time": "2021-08-30T17:35:50Z",
    "end_time": null,
    "stdout": null,
    "stderr": null,
    "exit_code": null
```

(continues on next page)

(continued from previous page)

```
  },
  "task_logs": [],
  "outputs": {}
}
```

11.7.3 Canceling a run

To cancel a workflow run, use the `POST /runs/{run_id}/cancel` endpoint:

```
$ curl --location --request POST 'http://localhost:8080/ga4gh/wes/v1/runs/
↪4deb8beb24894e9eb7c74b0f010305d1/cancel' \
      --user test:test
{
  "run_id": "4deb8beb24894e9eb7c74b0f010305d1"
}
```

CHAPTER

TWELVE

DEVELOPING A WORKFLOW

This tutorial walks through the features of Toil necessary for developing a workflow using the Toil Python API.

Note: "script" and "workflow" will be used interchangeably

12.1 Scripting Quick Start

To begin, consider this short toil script which illustrates defining a workflow:

```
import os
import tempfile

from toil.common import Toil
from toil.job import Job


def helloWorld(message, memory="2G", cores=2, disk="3G"):
    return f"Hello, world!, here's a message: {message}"


if __name__ == "__main__":
    jobstore: str = tempfile.mkdtemp("tutorial_quickstart")
    os.rmdir(jobstore)
    options = Job.Runner.getDefaultOptions(jobstore)
    options.logLevel = "OFF"
    options.clean = "always"

    hello_job = Job.wrapFn(helloWorld, "Woot")

    with Toil(options) as toil:
        print(toil.start(hello_job))  # prints "Hello, world!, ..."
```

The workflow consists of a single job. The resource requirements for that job are (optionally) specified by keyword arguments (memory, cores, disk). The script is run using *toil.job.Job.Runner.getDefaultOptions()*. Below we explain the components of this code in detail.

12.2 Job Basics

The atomic unit of work in a Toil workflow is a *Job*. User scripts inherit from this base class to define units of work. For example, here is a more long-winded class-based version of the job in the quick start example:

```
from toil.job import Job

class HelloWorld(Job):
    def __init__(self, message):
        Job.__init__(self,  memory="2G", cores=2, disk="3G")
        self.message = message

    def run(self, fileStore):
        return f"Hello, world! Here's a message: {self.message}"
```

In the example a class, HelloWorld, is defined. The constructor requests 2 gigabytes of memory, 2 cores and 3 gigabytes of local disk to complete the work.

The *toil.job.Job.run()* method is the function the user overrides to get work done. Here it just returns a message.

It is also possible to log a message using *toil.job.Job.log()*, which will be registered in the log output of the leader process of the workflow:

```
...
    def run(self, fileStore):
        self.log(f"Hello, world! Here's a message: {self.message}")
```

12.3 Invoking a Workflow

We can add to the previous example to turn it into a complete workflow by adding the necessary function calls to create an instance of HelloWorld and to run this as a workflow containing a single job. For example:

```
import os
import tempfile

from toil.common import Toil
from toil.job import Job


class HelloWorld(Job):
    def __init__(self, message):
        Job.__init__(self, memory="2G", cores=2, disk="3G")
        self.message = message

    def run(self, fileStore):
        return f"Hello, world!, here's a message: {self.message}"


if __name__ == "__main__":
    jobstore: str = tempfile.mkdtemp("tutorial_invokeworkflow")
    os.rmdir(jobstore)
    options = Job.Runner.getDefaultOptions(jobstore)
```

(continues on next page)

(continued from previous page)

```
    options.logLevel = "OFF"
    options.clean = "always"

    hello_job = HelloWorld("Woot")

    with Toil(options) as toil:
        print(toil.start(hello_job))
```

Note: Do not include a . in the name of your python script (besides *.py* at the end). This is to allow toil to import the types and functions defined in your file while starting a new process.

This uses the *toil.common.Toil* class, which is used to run and resume Toil workflows. It is used as a context manager and allows for preliminary setup, such as staging of files into the job store on the leader node. An instance of the class is initialized by specifying an options object. The actual workflow is then invoked by calling the *toil.common.Toil.start()* method, passing the root job of the workflow, or, if a workflow is being restarted, *toil.common.Toil.restart()* should be used. Note that the context manager should have explicit if else branches addressing restart and non restart cases. The boolean value for these if else blocks is toil.options.restart.

For example:

```
import os
import tempfile

from toil.common import Toil
from toil.job import Job

class HelloWorld(Job):
    def __init__(self, message):
        Job.__init__(self,  memory="2G", cores=2, disk="3G")
        self.message = message

    def run(self, fileStore):
        return f"Hello, world!, I have a message: {self.message}"


if __name__ == "__main__":
    jobstore: str = tempfile.mkdtemp("tutorial_invokeworkflow2")
    os.rmdir(jobstore)
    options = Job.Runner.getDefaultOptions(jobstore)
    options.logLevel = "INFO"
    options.clean = "always"

    with Toil(options) as toil:
        if not toil.options.restart:
            job = HelloWorld("Woot!")
            output = toil.start(job)
        else:
            output = toil.restart()
    print(output)
```

The call to *toil.job.Job.Runner.getDefaultOptions()* creates a set of default options for the workflow. The only argument is a description of how to store the workflow's state in what we call a *job-store*. Here the job-store is

contained in a directory within the current working directory called "toilWorkflowRun". Alternatively this string can encode other ways to store the necessary state, e.g. an S3 bucket object store location. By default the job-store is deleted if the workflow completes successfully.

The workflow is executed in the final line, which creates an instance of HelloWorld and runs it as a workflow. Note all Toil workflows start from a single starting job, referred to as the *root* job. The return value of the root job is returned as the result of the completed workflow (see promises below to see how this is a useful feature!).

12.4 Specifying Commandline Arguments

To allow command line control of the options we can use the *toil.job.Job.Runner.getDefaultArgumentParser()* method to create a argparse.ArgumentParser object which can be used to parse command line options for a Toil script. For example:

```
from toil.common import Toil
from toil.job import Job


class HelloWorld(Job):
    def __init__(self, message):
        Job.__init__(self,  memory="2G", cores=2, disk="3G")
        self.message = message

    def run(self, fileStore):
        return "Hello, world!, here's a message: %s" % self.message


if __name__ == "__main__":
    parser = Job.Runner.getDefaultArgumentParser()
    options = parser.parse_args()
    options.logLevel = "OFF"
    options.clean = "always"

    hello_job = HelloWorld("Woot")

    with Toil(options) as toil:
        print(toil.start(hello_job))
```

Creates a fully fledged script with all the options Toil exposed as command line arguments. Running this script with "–help" will print the full list of options.

Alternatively an existing argparse.ArgumentParser or optparse.OptionParser object can have Toil script command line options added to it with the *toil.job.Job.Runner.addToilOptions()* method.

12.5 Resuming a Workflow

In the event that a workflow fails, either because of programmatic error within the jobs being run, or because of node failure, the workflow can be resumed. Workflows can only not be reliably resumed if the job-store itself becomes corrupt.

Critical to resumption is that jobs can be rerun, even if they have apparently completed successfully. Put succinctly, a user defined job should not corrupt its input arguments. That way, regardless of node, network or leader failure the job can be restarted and the workflow resumed.

To resume a workflow specify the "restart" option in the options object passed to *toil.common.Toil.start()*. If node failures are expected it can also be useful to use the integer "retryCount" option, which will attempt to rerun a job retryCount number of times before marking it fully failed.

In the common scenario that a small subset of jobs fail (including retry attempts) within a workflow Toil will continue to run other jobs until it can do no more, at which point *toil.common.Toil.start()* will raise a *toil.exceptions.FailedJobsException* exception. Typically at this point the user can decide to fix the script and resume the workflow or delete the job-store manually and rerun the complete workflow.

12.6 Functions and Job Functions

Defining jobs by creating class definitions generally involves the boilerplate of creating a constructor. To avoid this the classes *toil.job.FunctionWrappingJob* and `toil.job.JobFunctionWrappingTarget` allow functions to be directly converted to jobs. For example, the quick start example (repeated here):

```
import os
import tempfile

from toil.common import Toil
from toil.job import Job


def helloWorld(message, memory="2G", cores=2, disk="3G"):
    return f"Hello, world!, here's a message: {message}"


if __name__ == "__main__":
    jobstore: str = tempfile.mkdtemp("tutorial_quickstart")
    os.rmdir(jobstore)
    options = Job.Runner.getDefaultOptions(jobstore)
    options.logLevel = "OFF"
    options.clean = "always"

    hello_job = Job.wrapFn(helloWorld, "Woot")

    with Toil(options) as toil:
        print(toil.start(hello_job))  # prints "Hello, world!, ..."
```

Is equivalent to the previous example, but using a function to define the job.

The function call:

```
Job.wrapFn(helloWorld, "Woot")
```

Creates the instance of the `toil.job.FunctionWrappingTarget` that wraps the function.

The keyword arguments *memory*, *cores* and *disk* allow resource requirements to be specified as before. Even if they are not included as keyword arguments within a function header they can be passed as arguments when wrapping a function as a job and will be used to specify resource requirements.

We can also use the function wrapping syntax to a *job function*, a function whose first argument is a reference to the wrapping job. Just like a *self* argument in a class, this allows access to the methods of the wrapping job, see `toil.job.JobFunctionWrappingTarget`. For example:

```
import os
import tempfile

from toil.common import Toil
from toil.job import Job


def helloWorld(job, message):
    job.log(f"Hello world, I have a message: {message}")


if __name__ == "__main__":
    jobstore: str = tempfile.mkdtemp("tutorial_jobfunctions")
    os.rmdir(jobstore)
    options = Job.Runner.getDefaultOptions(jobstore)
    options.logLevel = "INFO"
    options.clean = "always"

    hello_job = Job.wrapJobFn(helloWorld, "Woot!")

    with Toil(options) as toil:
        toil.start(hello_job)
```

Here `helloWorld()` is a job function. It uses the *toil.job.Job.log()* to log a message that will be printed to the output console. Here the only subtle difference to note is the line:

```
hello_job = Job.wrapJobFn(helloWorld, "Woot")
```

Which uses the function *toil.job.Job.wrapJobFn()* to wrap the job function instead of *toil.job.Job.wrapFn()* which wraps a vanilla function.

12.7 Workflows with Multiple Jobs

A *parent* job can have *child* jobs and *follow-on* jobs. These relationships are specified by methods of the job class, e.g. *toil.job.Job.addChild()* and *toil.job.Job.addFollowOn()*.

Considering a set of jobs the nodes in a job graph and the child and follow-on relationships the directed edges of the graph, we say that a job B that is on a directed path of child/follow-on edges from a job `A` in the job graph is a *successor* of `A`, similarly `A` is a *predecessor* of `B`.

A parent job's child jobs are run directly after the parent job has completed, and in parallel. The follow-on jobs of a job are run after its child jobs and their successors have completed. They are also run in parallel. Follow-ons allow the easy specification of cleanup tasks that happen after a set of parallel child tasks. The following shows a simple example that uses the earlier `helloWorld()` job function:

```
from toil.common import Toil
from toil.job import Job


def helloWorld(job, message, memory="2G", cores=2, disk="3G"):
    job.log(f"Hello world, I have a message: {message}")


if __name__ == "__main__":
    parser = Job.Runner.getDefaultArgumentParser()
    options = parser.parse_args()
    options.logLevel = "INFO"
    options.clean = "always"

    j1 = Job.wrapJobFn(helloWorld, "first")
    j2 = Job.wrapJobFn(helloWorld, "second or third")
    j3 = Job.wrapJobFn(helloWorld, "second or third")
    j4 = Job.wrapJobFn(helloWorld, "last")

    j1.addChild(j2)
    j1.addChild(j3)
    j1.addFollowOn(j4)

    with Toil(options) as toil:
        toil.start(j1)
```

In the example four jobs are created, first `j1` is run, then `j2` and `j3` are run in parallel as children of `j1`, finally `j4` is run as a follow-on of `j1`.

There are multiple short hand functions to achieve the same workflow, for example:

```
from toil.common import Toil
from toil.job import Job


def helloWorld(job, message, memory="2G", cores=2, disk="3G"):
    job.log(f"Hello world, I have a message: {message}")


if __name__ == "__main__":
    parser = Job.Runner.getDefaultArgumentParser()
    options = parser.parse_args()
    options.logLevel = "INFO"
    options.clean = "always"

    j1 = Job.wrapJobFn(helloWorld, "first")
    j2 = j1.addChildJobFn(helloWorld, "second or third")
    j3 = j1.addChildJobFn(helloWorld, "second or third")
    j4 = j1.addFollowOnJobFn(helloWorld, "last")

    with Toil(options) as toil:
        toil.start(j1)
```

Equivalently defines the workflow, where the functions *toil.job.Job.addChildJobFn()* and *toil.job.Job.*

addFollowOnJobFn() are used to create job functions as children or follow-ons of an earlier job.

Jobs graphs are not limited to trees, and can express arbitrary directed acyclic graphs. For a precise definition of legal graphs see *toil.job.Job.checkJobGraphForDeadlocks()*. The previous example could be specified as a DAG as follows:

```
from toil.common import Toil
from toil.job import Job


def helloWorld(job, message, memory="2G", cores=2, disk="3G"):
    job.log(f"Hello world, I have a message: {message}")


if __name__ == "__main__":
    parser = Job.Runner.getDefaultArgumentParser()
    options = parser.parse_args()
    options.logLevel = "INFO"
    options.clean = "always"

    j1 = Job.wrapJobFn(helloWorld, "first")
    j2 = j1.addChildJobFn(helloWorld, "second or third")
    j3 = j1.addChildJobFn(helloWorld, "second or third")
    j4 = j2.addChildJobFn(helloWorld, "last")
    j3.addChild(j4)

    with Toil(options) as toil:
        toil.start(j1)
```

Note the use of an extra child edge to make `j4` a child of both `j2` and `j3`.

12.8 Dynamic Job Creation

The previous examples show a workflow being defined outside of a job. However, Toil also allows jobs to be created dynamically within jobs. For example:

```
import os
import tempfile

from toil.common import Toil
from toil.job import Job


def binaryStringFn(job, depth, message=""):
    if depth > 0:
        job.addChildJobFn(binaryStringFn, depth-1, message + "0")
        job.addChildJobFn(binaryStringFn, depth-1, message + "1")
    else:
        job.log(f"Binary string: {message}")


if __name__ == "__main__":
```

(continues on next page)

(continued from previous page)

```
    jobstore: str = tempfile.mkdtemp("tutorial_dynamic")
    os.rmdir(jobstore)
    options = Job.Runner.getDefaultOptions(jobstore)
    options.logLevel = "INFO"
    options.clean = "always"

    with Toil(options) as toil:
        toil.start(Job.wrapJobFn(binaryStringFn, depth=5))
```

The job function `binaryStringFn` logs all possible binary strings of length `n` (here `n=5`), creating a total of `2^(n+2) - 1` jobs dynamically and recursively. Static and dynamic creation of jobs can be mixed in a Toil workflow, with jobs defined within a job or job function being created at run time.

12.9 Promises

The previous example of dynamic job creation shows variables from a parent job being passed to a child job. Such forward variable passing is naturally specified by recursive invocation of successor jobs within parent jobs. This can also be achieved statically by passing around references to the return variables of jobs. In Toil this is achieved with promises, as illustrated in the following example:

```
import os
import tempfile

from toil.common import Toil
from toil.job import Job


def fn(job, i):
    job.log("i is: %s" % i, level=100)
    return i + 1


if __name__ == "__main__":
    jobstore: str = tempfile.mkdtemp("tutorial_promises")
    os.rmdir(jobstore)
    options = Job.Runner.getDefaultOptions(jobstore)
    options.logLevel = "INFO"
    options.clean = "always"

    j1 = Job.wrapJobFn(fn, 1)
    j2 = j1.addChildJobFn(fn, j1.rv())
    j3 = j1.addFollowOnJobFn(fn, j2.rv())

    with Toil(options) as toil:
        toil.start(j1)
```

Running this workflow results in three log messages from the jobs: `i is 1` from `j1`, `i is 2` from `j2` and `i is 3` from `j3`.

The return value from the first job is *promised* to the second job by the call to *`toil.job.Job.rv()`* in the following line:

```
j2 = j1.addChildFn(fn, j1.rv())
```

The value of `j1.rv()` is a *promise*, rather than the actual return value of the function, because `j1` for the given input has at that point not been evaluated. A promise (*toil.job.Promise*) is essentially a pointer to for the return value that is replaced by the actual return value once it has been evaluated. Therefore, when `j2` is run the promise becomes 2.

Promises also support indexing of return values:

```
def parent(job):
    indexable = Job.wrapJobFn(fn)
    job.addChild(indexable)
    job.addFollowOnFn(raiseWrap, indexable.rv(2))

def raiseWrap(arg):
    raise RuntimeError(arg) # raises "2"

def fn(job):
    return (0, 1, 2, 3)
```

Promises can be quite useful. For example, we can combine dynamic job creation with promises to achieve a job creation process that mimics the functional patterns possible in many programming languages:

```
import os
import tempfile

from toil.common import Toil
from toil.job import Job


def binaryStrings(job, depth, message=""):
    if depth > 0:
        s = [job.addChildJobFn(binaryStrings, depth - 1, message + "0").rv(),
             job.addChildJobFn(binaryStrings, depth - 1, message + "1").rv()]
        return job.addFollowOnFn(merge, s).rv()
    return [message]


def merge(strings):
    return strings[0] + strings[1]


if __name__ == "__main__":
    jobstore: str = tempfile.mkdtemp("tutorial_promises2")
    os.rmdir(jobstore)
    options = Job.Runner.getDefaultOptions(jobstore)
    options.loglevel = "OFF"
    options.clean = "always"

    with Toil(options) as toil:
        print(toil.start(Job.wrapJobFn(binaryStrings, depth=5)))
```

The return value `l` of the workflow is a list of all binary strings of length 10, computed recursively. Although a toy example, it demonstrates how closely Toil workflows can mimic typical programming patterns.

12.10 Promised Requirements

Promised requirements are a special case of *Promises* that allow a job's return value to be used as another job's resource requirements.

This is useful when, for example, a job's storage requirement is determined by a file staged to the job store by an earlier job:

```
import os
import tempfile

from toil.common import Toil
from toil.job import Job, PromisedRequirement


def parentJob(job):
    downloadJob = Job.wrapJobFn(stageFn, "file://" + os.path.realpath(__file__), cores=0.
↪1, memory='32M', disk='1M')
    job.addChild(downloadJob)

    analysis = Job.wrapJobFn(analysisJob,
                             fileStoreID=downloadJob.rv(0),
                             disk=PromisedRequirement(downloadJob.rv(1)))
    job.addFollowOn(analysis)


def stageFn(job, url, cores=1):
    importedFile = job.fileStore.import_file(url)
    return importedFile, importedFile.size


def analysisJob(job, fileStoreID, cores=2):
    # now do some analysis on the file
    pass


if __name__ == "__main__":
    jobstore: str = tempfile.mkdtemp("tutorial_requirements")
    os.rmdir(jobstore)
    options = Job.Runner.getDefaultOptions(jobstore)
    options.logLevel = "INFO"
    options.clean = "always"

    with Toil(options) as toil:
        toil.start(Job.wrapJobFn(parentJob))
```

Note that this also makes use of the `size` attribute of the *FileID* object. This promised requirements mechanism can also be used in combination with an aggregator for multiple jobs' output values:

```
def parentJob(job):
    aggregator = []
    for fileNum in range(0, 10):
        downloadJob = Job.wrapJobFn(stageFn, "file://" + os.path.realpath(__file__),␣
```

(continues on next page)

(continued from previous page)

```
↪cores=0.1, memory='32M', disk='1M')
        job.addChild(downloadJob)
        aggregator.append(downloadJob)

    analysis = Job.wrapJobFn(analysisJob,
                             fileStoreID=downloadJob.rv(0),
                             disk=PromisedRequirement(lambda xs: sum(xs), [j.rv(1) for j␣
↪in aggregator]))
    job.addFollowOn(analysis)
```

Limitations

Just like regular promises, the return value must be determined prior to scheduling any job that depends on the return value. In our example above, notice how the dependent jobs were follow ons to the parent while promising jobs are children of the parent. This ordering ensures that all promises are properly fulfilled.

12.11 FileID

The `toil.fileStore.FileID` class is a small wrapper around Python's builtin string class. It is used to represent a file's ID in the file store, and has a `size` attribute that is the file's size in bytes. This object is returned by `importFile` and `writeGlobalFile`.

12.12 Managing files within a workflow

It is frequently the case that a workflow will want to create files, both persistent and temporary, during its run. The *toil.fileStores.abstractFileStore.AbstractFileStore* class is used by jobs to manage these files in a manner that guarantees cleanup and resumption on failure.

The *toil.job.Job.run()* method has a file store instance as an argument. The following example shows how this can be used to create temporary files that persist for the length of the job, be placed in a specified local disk of the node and that will be cleaned up, regardless of failure, when the job finishes:

```
import os
import tempfile

from toil.common import Toil
from toil.job import Job


class LocalFileStoreJob(Job):
    def run(self, fileStore):
        # self.tempDir will always contain the name of a directory within the allocated␣
↪disk space reserved for the job
        scratchDir = self.tempDir

        # Similarly create a temporary file.
        scratchFile = fileStore.getLocalTempFile()
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    jobstore: str = tempfile.mkdtemp("tutorial_managing")
    os.rmdir(jobstore)
    options = Job.Runner.getDefaultOptions(jobstore)
    options.logLevel = "INFO"
    options.clean = "always"

    # Create an instance of FooJob which will have at least 2 gigabytes of storage space.
    j = LocalFileStoreJob(disk="2G")

    # Run the workflow
    with Toil(options) as toil:
        toil.start(j)
```

Job functions can also access the file store for the job. The equivalent of the `LocalFileStoreJob` class is

```
def localFileStoreJobFn(job):
    scratchDir = job.tempDir
    scratchFile = job.fileStore.getLocalTempFile()
```

Note that the `fileStore` attribute is accessed as an attribute of the `job` argument.

In addition to temporary files that exist for the duration of a job, the file store allows the creation of files in a *global* store, which persists during the workflow and are globally accessible (hence the name) between jobs. For example:

```
import os
import tempfile

from toil.common import Toil
from toil.job import Job


def globalFileStoreJobFn(job):
    job.log("The following example exercises all the methods provided "
            "by the toil.fileStores.abstractFileStore.AbstractFileStore class")

    # Create a local temporary file.
    scratchFile = job.fileStore.getLocalTempFile()

    # Write something in the scratch file.
    with open(scratchFile, 'w') as fH:
        fH.write("What a tangled web we weave")

    # Write a copy of the file into the file-store; fileID is the key that can be used␣
↪to retrieve the file.
    # This write is asynchronous by default
    fileID = job.fileStore.writeGlobalFile(scratchFile)

    # Write another file using a stream; fileID2 is the
    # key for this second file.
    with job.fileStore.writeGlobalFileStream(cleanup=True) as (fH, fileID2):
        fH.write(b"Out brief candle")
```

(continues on next page)

(continued from previous page)

```
    # Now read the first file; scratchFile2 is a local copy of the file that is read-
↪only by default.
    scratchFile2 = job.fileStore.readGlobalFile(fileID)

    # Read the second file to a desired location: scratchFile3.
    scratchFile3 = os.path.join(job.tempDir, "foo.txt")
    job.fileStore.readGlobalFile(fileID2, userPath=scratchFile3)

    # Read the second file again using a stream.
    with job.fileStore.readGlobalFileStream(fileID2) as fH:
        print(fH.read())  # This prints "Out brief candle"

    # Delete the first file from the global file-store.
    job.fileStore.deleteGlobalFile(fileID)

    # It is unnecessary to delete the file keyed by fileID2 because we used the cleanup␣
↪flag,
    # which removes the file after this job and all its successors have run (if the file␣
↪still exists)


if __name__ == "__main__":
    jobstore: str = tempfile.mkdtemp("tutorial_managing2")
    os.rmdir(jobstore)
    options = Job.Runner.getDefaultOptions(jobstore)
    options.logLevel = "INFO"
    options.clean = "always"

    with Toil(options) as toil:
        toil.start(Job.wrapJobFn(globalFileStoreJobFn))
```

The example demonstrates the global read, write and delete functionality of the file-store, using both local copies of the files and streams to read and write the files. It covers all the methods provided by the file store interface.

What is obvious is that the file-store provides no functionality to update an existing "global" file, meaning that files are, barring deletion, immutable. Also worth noting is that there is no file system hierarchy for files in the global file store. These limitations allow us to fairly easily support different object stores and to use caching to limit the amount of network file transfer between jobs.

12.12.1 Staging of Files into the Job Store

External files can be imported into or exported out of the job store prior to running a workflow when the *toil.common.Toil* context manager is used on the leader. The context manager provides methods *toil.common.Toil.importFile()*, and *toil.common.Toil.exportFile()* for this purpose. The destination and source locations of such files are described with URLs passed to the two methods. Local files can be imported and exported as relative paths, and should be relative to the directory where the toil workflow is initially run from.

Using absolute paths and appropriate schema where possible (prefixing with "file://" or "s3:/" for example), make imports and exports less ambiguous and is recommended.

A list of the currently supported URLs can be found at *toil.jobStores.abstractJobStore.AbstractJobStore.importFile()*. To import an external file into the job store as a shared file, pass the optional `sharedFileName`

parameter to that method.

If a workflow fails for any reason an imported file acts as any other file in the job store. If the workflow was configured such that it not be cleaned up on a failed run, the file will persist in the job store and needs not be staged again when the workflow is resumed.

Example:

```
import os
import tempfile

from toil.common import Toil
from toil.job import Job


class HelloWorld(Job):
    def __init__(self, id):
        Job.__init__(self, memory="2G", cores=2, disk="3G")
        self.inputFileID = id

    def run(self, fileStore):
        with fileStore.readGlobalFileStream(self.inputFileID, encoding='utf-8') as fi:
            with fileStore.writeGlobalFileStream(encoding='utf-8') as (fo, outputFileID):
                fo.write(fi.read() + 'World!')
        return outputFileID


if __name__ == "__main__":
    jobstore: str = tempfile.mkdtemp("tutorial_staging")
    os.rmdir(jobstore)
    options = Job.Runner.getDefaultOptions(jobstore)
    options.logLevel = "INFO"
    options.clean = "always"

    with Toil(options) as toil:
        if not toil.options.restart:
            ioFileDirectory = os.path.join(os.path.dirname(os.path.abspath(__file__)),
↪"stagingExampleFiles")
            inputFileID = toil.importFile("file://" + os.path.abspath(os.path.
↪join(ioFileDirectory, "in.txt")))
            outputFileID = toil.start(HelloWorld(inputFileID))
        else:
            outputFileID = toil.restart()

        toil.exportFile(outputFileID, "file://" + os.path.abspath(os.path.
↪join(ioFileDirectory, "out.txt")))
```

12.13 Using Docker Containers in Toil

Docker containers are commonly used with Toil. The combination of Toil and Docker allows for pipelines to be fully portable between any platform that has both Toil and Docker installed. Docker eliminates the need for the user to do any other tool installation or environment setup.

In order to use Docker containers with Toil, Docker must be installed on all workers of the cluster. Instructions for installing Docker can be found on the Docker website.

When using Toil-based autoscaling, Docker will be automatically set up on the cluster's worker nodes, so no additional installation steps are necessary. Further information on using Toil-based autoscaling can be found in the *Running a Workflow with Autoscaling* documentation.

In order to use docker containers in a Toil workflow, the container can be built locally or downloaded in real time from an online docker repository like Quay. If the container is not in a repository, the container's layers must be accessible on each node of the cluster.

When invoking docker containers from within a Toil workflow, it is strongly recommended that you use `dockerCall()`, a toil job function provided in `toil.lib.docker`. `dockerCall` leverages docker's own python API, and provides container cleanup on job failure. When docker containers are run without this feature, failed jobs can result in resource leaks. Docker's API can be found at docker-py.

In order to use `dockerCall`, your installation of Docker must be set up to run without `sudo`. Instructions for setting this up can be found here.

An example of a basic `dockerCall` is below:

```
dockerCall(job=job,
           tool='quay.io/ucsc_cgl/bwa',
           workDir=job.tempDir,
           parameters=['index', '/data/reference.fa'])
```

Note the assumption that *reference.fa* file is located in */data*. This is Toil's standard convention as a mount location to reduce boilerplate when calling *dockerCall*. Users can choose their own mount locations by supplying a *volumes* kwarg to *dockerCall*, such as: *volumes={working_dir: {'bind': '/data', 'mode': 'rw'}}*, where *working_dir* is an absolute path on the user's filesystem.

`dockerCall` can also be added to workflows like any other job function:

```
import os
import tempfile

from toil.common import Toil
from toil.job import Job
from toil.lib.docker import apiDockerCall

align = Job.wrapJobFn(apiDockerCall,
                      image='ubuntu',
                      working_dir=os.getcwd(),
                      parameters=['ls', '-lha'])

if __name__ == "__main__":
    jobstore: str = tempfile.mkdtemp("tutorial_docker")
    os.rmdir(jobstore)
    options = Job.Runner.getDefaultOptions(jobstore)
    options.logLevel = "INFO"
```

(continues on next page)

(continued from previous page)

```
    options.clean = "always"

    with Toil(options) as toil:
        toil.start(align)
```

cgl-docker-lib contains `dockerCall`-compatible Dockerized tools that are commonly used in bioinformatics analysis.

The documentation provides guidelines for developing your own Docker containers that can be used with Toil and `dockerCall`. In order for a container to be compatible with `dockerCall`, it must have an `ENTRYPOINT` set to a wrapper script, as described in cgl-docker-lib containerization standards. This can be set by passing in the optional keyword argument, 'entrypoint'. Example:

entrypoint=["/bin/bash","-c"]

dockerCall supports currently the 75 keyword arguments found in the python Docker API, under the 'run' command.

12.14 Services

It is sometimes desirable to run *services*, such as a database or server, concurrently with a workflow. The *toil.job.Job.Service* class provides a simple mechanism for spawning such a service within a Toil workflow, allowing precise specification of the start and end time of the service, and providing start and end methods to use for initialization and cleanup. The following simple, conceptual example illustrates how services work:

```
import os
import tempfile

from toil.common import Toil
from toil.job import Job


class DemoService(Job.Service):
    def start(self, fileStore):
        # Start up a database/service here
        # Return a value that enables another process to connect to the database
        return "loginCredentials"

    def check(self):
        # A function that if it returns False causes the service to quit
        # If it raises an exception the service is killed and an error is reported
        return True

    def stop(self, fileStore):
        # Cleanup the database here
        pass


j = Job()
s = DemoService()
loginCredentialsPromise = j.addService(s)


def dbFn(loginCredentials):
```

(continues on next page)

(continued from previous page)

```
    # Use the login credentials returned from the service's start method to connect to␣
↪the service
    pass


j.addChildFn(dbFn, loginCredentialsPromise)


if __name__ == "__main__":
    jobstore: str = tempfile.mkdtemp("tutorial_services")
    os.rmdir(jobstore)
    options = Job.Runner.getDefaultOptions(jobstore)
    options.logLevel = "INFO"
    options.clean = "always"

    with Toil(options) as toil:
        toil.start(j)
```

In this example the DemoService starts a database in the start method, returning an object from the start method indicating how a client job would access the database. The service's stop method cleans up the database, while the service's check method is polled periodically to check the service is alive.

A DemoService instance is added as a service of the root job `j`, with resource requirements specified. The return value from *toil.job.Job.addService()* is a promise to the return value of the service's start method. When the promised is fulfilled it will represent how to connect to the database. The promise is passed to a child job of `j`, which uses it to make a database connection. The services of a job are started before any of its successors have been run and stopped after all the successors of the job have completed successfully.

Multiple services can be created per job, all run in parallel. Additionally, services can define sub-services using `toil.job.Job.Service.addChild()`. This allows complex networks of services to be created, e.g. Apache Spark clusters, within a workflow.

12.15 Checkpoints

Services complicate resuming a workflow after failure, because they can create complex dependencies between jobs. For example, consider a service that provides a database that multiple jobs update. If the database service fails and loses state, it is not clear that just restarting the service will allow the workflow to be resumed, because jobs that created that state may have already finished. To get around this problem Toil supports *checkpoint* jobs, specified as the boolean keyword argument `checkpoint` to a job or wrapped function, e.g.:

```
j = Job(checkpoint=True)
```

A checkpoint job is rerun if one or more of its successors fails its retry attempts, until it itself has exhausted its retry attempts. Upon restarting a checkpoint job all its existing successors are first deleted, and then the job is rerun to define new successors. By checkpointing a job that defines a service, upon failure of the service the database and the jobs that access the service can be redefined and rerun.

To make the implementation of checkpoint jobs simple, a job can only be a checkpoint if when first defined it has no successors, i.e. it can only define successors within its run method.

12.16 Encapsulation

Let `A` be a root job potentially with children and follow-ons. Without an encapsulated job the simplest way to specify a job `B` which runs after `A` and all its successors is to create a parent of `A`, call it `Ap`, and then make `B` a follow-on of `Ap`. e.g.:

```
import os
import tempfile

from toil.common import Toil
from toil.job import Job

if __name__ == "__main__":
    # A is a job with children and follow-ons, for example:
    A = Job()
    A.addChild(Job())
    A.addFollowOn(Job())

    # B is a job which needs to run after A and its successors
    B = Job()

    # The way to do this without encapsulation is to make a parent of A, Ap, and make B␣
↪a follow-on of Ap.
    Ap = Job()
    Ap.addChild(A)
    Ap.addFollowOn(B)

    jobstore: str = tempfile.mkdtemp("tutorial_encapsulations")
    os.rmdir(jobstore)
    options = Job.Runner.getDefaultOptions(jobstore)
    options.logLevel = "INFO"
    options.clean = "always"

    with Toil(options) as toil:
        print(toil.start(Ap))
```

An *encapsulated job* `E(A)` of `A` saves making `Ap`, instead we can write:

```
import os
import tempfile

from toil.common import Toil
from toil.job import Job

if __name__ == "__main__":
    # A
    A = Job()
    A.addChild(Job())
    A.addFollowOn(Job())

    # Encapsulate A
    A = A.encapsulate()
```

(continues on next page)

(continued from previous page)

```
    # B is a job which needs to run after A and its successors
    B = Job()

    # With encapsulation A and its successor subgraph appear to be a single job, hence:
    A.addChild(B)

    jobstore: str = tempfile.mkdtemp("tutorial_encapsulations2")
    os.rmdir(jobstore)
    options = Job.Runner.getDefaultOptions(jobstore)
    options.logLevel = "INFO"
    options.clean = "always"

    with Toil(options) as toil:
        print(toil.start(A))
```

Note the call to *toil.job.Job.encapsulate()* creates the `toil.job.Job.EncapsulatedJob`.

12.17 Depending on Toil

If you are packing your workflow(s) as a pip-installable distribution on PyPI, you might be tempted to declare Toil as a dependency in your `setup.py`, via the `install_requires` keyword argument to `setup()`. Unfortunately, this does not work, for two reasons: For one, Toil uses Setuptools' *extra* mechanism to manage its own optional dependencies. If you explicitly declared a dependency on Toil, you would have to hard-code a particular combination of extras (or no extras at all), robbing the user of the choice what Toil extras to install. Secondly, and more importantly, declaring a dependency on Toil would only lead to Toil being installed on the leader node of a cluster, but not the worker nodes. Auto-deployment does not work here because Toil cannot auto-deploy itself, the classic "Which came first, chicken or egg?" problem.

In other words, you shouldn't explicitly depend on Toil. Document the dependency instead (as in "This workflow needs Toil version X.Y.Z to be installed") and optionally add a version check to your `setup.py`. Refer to the `check_version()` function in the `toil-lib` project's setup.py for an example. Alternatively, you can also just depend on `toil-lib` and you'll get that check for free.

If your workflow depends on a dependency of Toil, consider not making that dependency explicit either. If you do, you risk a version conflict between your project and Toil. The `pip` utility may silently ignore that conflict, breaking either Toil or your workflow. It is safest to simply assume that Toil installs that dependency for you. The only downside is that you are locked into the exact version of that dependency that Toil declares. But such is life with Python, which, unlike Java, has no means of dependencies belonging to different software components within the same process, and whose favored software distribution utility is incapable of properly resolving overlapping dependencies and detecting conflicts.

12.18 Best Practices for Dockerizing Toil Workflows

Computational Genomics Lab's Dockstore based production system provides workflow authors a way to run Dockerized versions of their pipeline in an automated, scalable fashion. To be compatible with this system of a workflow should meet the following requirements. In addition to the Docker container, a common workflow language descriptor file is needed. For inputs:

- Only command line arguments should be used for configuring the workflow. If the workflow relies on a configuration file, like Toil-RNAseq or ProTECT, a wrapper script inside the Docker container can be used to parse the CLI and generate the necessary configuration file.

- All inputs to the pipeline should be explicitly enumerated rather than implicit. For example, don't rely on one FASTQ read's path to discover the location of its pair. This is necessary since all inputs are mapped to their own isolated directories when the Docker is called via Dockstore.
- All inputs must be documented in the CWL descriptor file. Examples of this file can be seen in both Toil-RNAseq and ProTECT.

For outputs:

- All outputs should be written to a local path rather than S3.
- Take care to package outputs in a local and user-friendly way. For example, don't tar up all output if there are specific files that will care to see individually.
- All output file names should be deterministic and predictable. For example, don't prepend the name of an output file with PASS/FAIL depending on the outcome of the pipeline.
- All outputs must be documented in the CWL descriptor file. Examples of this file can be seen in both Toil-RNAseq and ProTECT.

CHAPTER

THIRTEEN

TOIL CLASS API

The Toil class configures and starts a Toil run.

class toil.common.**Toil**(*options*)

A context manager that represents a Toil workflow.

Specifically the batch system, job store, and its configuration.

Parameters
options (Namespace) –

__init__(*options*)

Initialize a Toil object from the given options.

Note that this is very light-weight and that the bulk of the work is done when the context is entered.

Parameters
options (Namespace) – command line options specified by the user

Return type
None

start(*rootJob*)

Invoke a Toil workflow with the given job as the root for an initial run.

This method must be called in the body of a `with Toil(...) as toil:` statement. This method should not be called more than once for a workflow that has not finished.

Parameters
rootJob (*Job*) – The root job of the workflow

Return type
Any

Returns
The root job's return value

restart()

Restarts a workflow that has been interrupted.

Return type
Any

Returns
The root job's return value

classmethod getJobStore(*locator*)

Create an instance of the concrete job store implementation that matches the given locator.

Parameters

- **locator** (str) – The location of the job store to be represent by the instance
- **locator** –

Return type
AbstractJobStore

Returns
an instance of a concrete subclass of AbstractJobStore

static createBatchSystem(*config*)

Create an instance of the batch system specified in the given config.

Parameters
config (*Config*) – the current configuration

Return type
AbstractBatchSystem

Returns
an instance of a concrete subclass of AbstractBatchSystem

import_file(*src_uri*, *shared_file_name=None*, *symlink=True*)

Import the file at the given URL into the job store.

See *toil.jobStores.abstractJobStore.AbstractJobStore.importFile()* for a full description

Parameters

- **src_uri** (str) –
- **shared_file_name** (Optional[str]) –
- **symlink** (bool) –

Return type
Optional[*FileID*]

export_file(*file_id*, *dst_uri*)

Export file to destination pointed at by the destination URL.

See *toil.jobStores.abstractJobStore.AbstractJobStore.exportFile()* for a full description

Parameters

- **file_id** (*FileID*) –
- **dst_uri** (str) –

Return type
None

static normalize_uri(*uri*, *check_existence=False*)

Given a URI, if it has no scheme, prepend "file:".

Parameters

- **check_existence** (bool) – If set, raise an error if a URI points to a local file that does not exist.
- **uri** (str) –

Return type
str

static getToilWorkDir(*configWorkDir=None*)

Return a path to a writable directory under which per-workflow directories exist.

This directory is always required to exist on a machine, even if the Toil worker has not run yet. If your workers and leader have different temp directories, you may need to set TOIL_WORKDIR.

Parameters
configWorkDir (Optional[str]) – Value passed to the program using the –workDir flag

Return type
str

Returns
Path to the Toil work directory, constant across all machines

classmethod get_toil_coordination_dir(*config_work_dir*, *config_coordination_dir*)

Return a path to a writable directory, which will be in memory if convenient. Ought to be used for file locking and coordination.

Parameters

- **config_work_dir** (Optional[str]) – Value passed to the program using the –workDir flag
- **config_coordination_dir** (Optional[str]) – Value passed to the program using the –coordinationDir flag

Return type
str

Returns
Path to the Toil coordination directory. Ought to be on a POSIX filesystem that allows directories containing open files to be deleted.

classmethod getLocalWorkflowDir(*workflowID*, *configWorkDir=None*)

Return the directory where worker directories and the cache will be located for this workflow on this machine.

Parameters

- **configWorkDir** (Optional[str]) – Value passed to the program using the –workDir flag
- **workflowID** (str) –

Return type
str

Returns
Path to the local workflow directory on this machine

classmethod get_local_workflow_coordination_dir(*workflow_id*, *config_work_dir*, *config_coordination_dir*)

Return the directory where coordination files should be located for this workflow on this machine. These include internal Toil databases and lock files for the machine.

If an in-memory filesystem is available, it is used. Otherwise, the local workflow directory, which may be on a shared network filesystem, is used.

Parameters

- **workflow_id** (str) – Unique ID of the current workflow.

- **config_work_dir** (Optional[str]) – Value used for the work directory in the current Toil Config.
- **config_coordination_dir** (Optional[str]) – Value used for the coordination directory in the current Toil Config.

Return type
str

Returns
Path to the local workflow coordination directory on this machine.

CHAPTER

FOURTEEN

JOB STORE API

The job store interface is an abstraction layer that that hides the specific details of file storage, for example standard file systems, S3, etc. The *AbstractJobStore* API is implemented to support a give file store, e.g. S3. Implement this API to support a new file store.

class toil.jobStores.abstractJobStore.**AbstractJobStore**(*locator*)

Represents the physical storage for the jobs and files in a Toil workflow.

JobStores are responsible for storing *toil.job.JobDescription* (which relate jobs to each other) and files.

Actual *toil.job.Job* objects are stored in files, referenced by JobDescriptions. All the non-file CRUD methods the JobStore provides deal in JobDescriptions and not full, executable Jobs.

To actually get ahold of a *toil.job.Job*, use *toil.job.Job.loadJob()* with a JobStore and the relevant JobDescription.

Parameters
locator (str) –

__init__(*locator*)

Create an instance of the job store.

The instance will not be fully functional until either *initialize()* or *resume()* is invoked. Note that the *destroy()* method may be invoked on the object with or without prior invocation of either of these two methods.

Takes and stores the locator string for the job store, which will be accessible via self.locator.

Parameters
locator (str) –

Return type
None

initialize(*config*)

Initialize this job store.

Create the physical storage for this job store, allocate a workflow ID and persist the given Toil configuration to the store.

Parameters
config (*Config*) – the Toil configuration to initialize this job store with. The given configuration will be updated with the newly allocated workflow ID.

Raises
JobStoreExistsException – if the physical storage for this job store already exists

Return type
None

write_config()

Persists the value of the *AbstractJobStore.config* attribute to the job store, so that it can be retrieved later by other instances of this class.

Return type
None

resume()

Connect this instance to the physical storage it represents and load the Toil configuration into the *AbstractJobStore.config* attribute.

Raises
NoSuchJobStoreException – if the physical storage for this job store doesn't exist

Return type
None

property config: ***Config***

Return the Toil configuration associated with this job store.

Return type
toil.common.Config

property locator: **str**

Get the locator that defines the job store, which can be used to connect to it.

Return type
str

setRootJob(*rootJobStoreID*)

Set the root job of the workflow backed by this job store.

Parameters
rootJobStoreID (*FileID*) –

Return type
None

set_root_job(*job_id*)

Set the root job of the workflow backed by this job store.

Parameters
job_id (*FileID*) – The ID of the job to set as root

Return type
None

load_root_job()

Loads the JobDescription for the root job in the current job store.

Raises
toil.job.JobException – If no root job is set or if the root job doesn't exist in this job store

Return type
JobDescription

Returns
The root job.

create_root_job(*job_description*)

Create the given JobDescription and set it as the root job in this job store.

Parameters
job_description (*JobDescription*) – JobDescription to save and make the root job.

Return type
JobDescription

get_root_job_return_value()

Parse the return value from the root job.

Raises an exception if the root job hasn't fulfilled its promise yet.

Return type
Any

import_file(*src_uri*, *shared_file_name=None*, *hardlink=False*, *symlink=True*)

Imports the file at the given URL into job store. The ID of the newly imported file is returned. If the name of a shared file name is provided, the file will be imported as such and None is returned. If an executable file on the local filesystem is uploaded, its executability will be preserved when it is downloaded.

Currently supported schemes are:

- **'s3' for objects in Amazon S3**
  e.g. s3://bucket/key
- **'file' for local files**
  e.g. file:///local/file/path
- **'http'**
  e.g. http://someurl.com/path
- **'gs'**
  e.g. gs://bucket/file

Parameters

- **src_uri** (str) – URL that points to a file or object in the storage mechanism of a supported URL scheme e.g. a blob in an AWS s3 bucket. It must be a file, not a directory or prefix.
- **shared_file_name** (Optional[str]) – Optional name to assign to the imported file within the job store
- **src_uri** –
- **shared_file_name** –
- **hardlink** (bool) –
- **symlink** (bool) –

Returns
The jobStoreFileID of the imported file or None if shared_file_name was given

Return type
toil.fileStores.FileID or None

export_file(*file_id*, *dst_uri*)

Exports file to destination pointed at by the destination URL. The exported file will be executable if and only if it was originally uploaded from an executable file on the local filesystem.

Refer to *AbstractJobStore.import_file()* documentation for currently supported URL schemes.

Note that the helper method _exportFile is used to read from the source and write to destination. To implement any optimizations that circumvent this, the _exportFile method should be overridden by subclasses of AbstractJobStore.

Parameters

- **file_id** (*FileID*) – The id of the file in the job store that should be exported.
- **dst_uri** (str) – URL that points to a file or object in the storage mechanism of a supported URL scheme e.g. a blob in an AWS s3 bucket.
- **file_id** –
- **dst_uri** –

Return type
None

classmethod list_url(*src_uri*)

List the directory at the given URL. Returned path components can be joined with '/' onto the passed URL to form new URLs. Those that end in '/' correspond to directories. The provided URL may or may not end with '/'.

Currently supported schemes are:

- **'s3' for objects in Amazon S3**
  e.g. s3://bucket/prefix/
- **'file' for local files**
  e.g. file:///local/dir/path/

Parameters

- **src_uri** (str) – URL that points to a directory or prefix in the storage mechanism of a supported URL scheme e.g. a prefix in an AWS s3 bucket.
- **src_uri** –

Return type
List[str]

Returns
A list of URL components in the given directory, already URL-encoded.

classmethod get_is_directory(*src_uri*)

Return True if the thing at the given URL is a directory, and False if it is a file. The URL may or may not end in '/'.

Parameters
src_uri (str) –

Return type
bool

classmethod read_from_url(*src_uri*, *writable*)

Read the given URL and write its content into the given writable stream.

Returns
The size of the file in bytes and whether the executable permission bit is set

Return type
Tuple[int, bool]

Parameters

- **src_uri** (str) –
- **writable** (IO[bytes]) –

abstract classmethod get_size(*src_uri*)

Get the size in bytes of the file at the given URL, or None if it cannot be obtained.

Parameters
src_uri (ParseResult) – URL that points to a file or object in the storage mechanism of a supported URL scheme e.g. a blob in an AWS s3 bucket.

Return type
None

abstract destroy()

The inverse of *initialize()*, this method deletes the physical storage represented by this instance. While not being atomic, this method *is* at least idempotent, as a means to counteract potential issues with eventual consistency exhibited by the underlying storage mechanisms. This means that if the method fails (raises an exception), it may (and should be) invoked again. If the underlying storage mechanism is eventually consistent, even a successful invocation is not an ironclad guarantee that the physical storage vanished completely and immediately. A successful invocation only guarantees that the deletion will eventually happen. It is therefore recommended to not immediately reuse the same job store location for a new Toil workflow.

Return type
None

get_env()

Returns a dictionary of environment variables that this job store requires to be set in order to function properly on a worker.

Return type
dict[str,str]

clean(*jobCache=None*)

Function to cleanup the state of a job store after a restart.

Fixes jobs that might have been partially updated. Resets the try counts and removes jobs that are not successors of the current root job.

Parameters
jobCache (Optional[Dict[Union[str, *TemporaryID*], *JobDescription*]]) – if a value it must be a dict from job ID keys to JobDescription object values. Jobs will be loaded from the cache (which can be downloaded from the job store in a batch) instead of piecemeal when recursed into.

Return type
JobDescription

abstract assign_job_id(*job_description*)

Get a new jobStoreID to be used by the described job, and assigns it to the JobDescription.

Files associated with the assigned ID will be accepted even if the JobDescription has never been created or updated.

Parameters

- **job_description** (*JobDescription*) – The JobDescription to give an ID to
- **job_description** –

Return type
None

batch()

If supported by the batch system, calls to create() with this context manager active will be performed in a batch after the context manager is released.

Return type
Iterator[None]

abstract create_job(*job_description*)

Writes the given JobDescription to the job store. The job must have an ID assigned already.

Must call jobDescription.pre_update_hook()

Returns
The JobDescription passed.

Return type
toil.job.JobDescription

Parameters
job_description (*JobDescription*) –

abstract job_exists(*job_id*)

Indicates whether a description of the job with the specified jobStoreID exists in the job store

Return type
bool

Parameters
job_id (str) –

abstract get_public_url(*file_name*)

Returns a publicly accessible URL to the given file in the job store. The returned URL may expire as early as 1h after its been returned. Throw an exception if the file does not exist.

Parameters

- **file_name** (str) – the jobStoreFileID of the file to generate a URL for
- **file_name** –

Raises
NoSuchFileException – if the specified file does not exist in this job store

Return type
str

abstract get_shared_public_url(*shared_file_name*)

Differs from *getPublicUrl()* in that this method is for generating URLs for shared files written by *writeSharedFileStream()*.

Returns a publicly accessible URL to the given file in the job store. The returned URL starts with 'http:', 'https:' or 'file:'. The returned URL may expire as early as 1h after its been returned. Throw an exception if the file does not exist.

Parameters

- **shared_file_name** (str) – The name of the shared file to generate a publically accessible url for.
- **shared_file_name** –

Raises
NoSuchFileException – raised if the specified file does not exist in the store

Return type
str

abstract load_job(*job_id*)

Loads the description of the job referenced by the given ID, assigns it the job store's config, and returns it.

May declare the job to have failed (see *toil.job.JobDescription.setupJobAfterFailure()*) if there is evidence of a failed update attempt.

Parameters
job_id (str) – the ID of the job to load

Raises
NoSuchJobException – if there is no job with the given ID

Return type
JobDescription

abstract update_job(*job_description*)

Persists changes to the state of the given JobDescription in this store atomically.

Must call jobDescription.pre_update_hook()

Parameters

- **job** (toil.job.JobDescription) – the job to write to this job store
- **job_description** (*JobDescription*) –

Return type
None

abstract delete_job(*job_id*)

Removes the JobDescription from the store atomically. You may not then subsequently call load(), write(), update(), etc. with the same jobStoreID or any JobDescription bearing it.

This operation is idempotent, i.e. deleting a job twice or deleting a non-existent job will succeed silently.

Parameters

- **job_id** (str) – the ID of the job to delete from this job store
- **job_id** –

Return type
None

jobs()

Best effort attempt to return iterator on JobDescriptions for all jobs in the store. The iterator may not return all jobs and may also contain orphaned jobs that have already finished successfully and should not be rerun. To guarantee you get any and all jobs that can be run instead construct a more expensive ToilState object

Returns
Returns iterator on jobs in the store. The iterator may or may not contain all jobs and may contain invalid jobs

Return type
Iterator[toil.job.jobDescription]

abstract write_file(*local_path*, *job_id=None*, *cleanup=False*)

Takes a file (as a path) and places it in this job store. Returns an ID that can be used to retrieve the file at a later time. The file is written in a atomic manner. It will not appear in the jobStore until the write has successfully completed.

Parameters

- **local_path** (*str*) – the path to the local file that will be uploaded to the job store. The last path component (basename of the file) will remain associated with the file in the file store, if supported, so that the file can be searched for by name or name glob.
- **job_id** (*str*) – the id of a job, or None. If specified, the may be associated with that job in a job-store-specific way. This may influence the returned ID.
- **cleanup** (*bool*) – Whether to attempt to delete the file when the job whose jobStoreID was given as jobStoreID is deleted with jobStore.delete(job). If jobStoreID was not given, does nothing.

Raises

- *ConcurrentFileModificationException* – if the file was modified concurrently during an invocation of this method
- *NoSuchJobException* – if the job specified via jobStoreID does not exist

Return type
str

FIXME: some implementations may not raise this

Returns
an ID referencing the newly created file and can be used to read the file in the future.

Return type
str

Parameters

- **local_path** (str) –
- **job_id** (Optional[str]) –
- **cleanup** (bool) –

abstract write_file_stream(*job_id=None*, *cleanup=False*, *basename=None*, *encoding=None*, *errors=None*)

Similar to writeFile, but returns a context manager yielding a tuple of 1) a file handle which can be written to and 2) the ID of the resulting file in the job store. The yielded file handle does not need to and should not be closed explicitly. The file is written in a atomic manner. It will not appear in the jobStore until the write has successfully completed.

Parameters

- **job_id** (*str*) – the id of a job, or None. If specified, the may be associated with that job in a job-store-specific way. This may influence the returned ID.
- **cleanup** (*bool*) – Whether to attempt to delete the file when the job whose jobStoreID was given as jobStoreID is deleted with jobStore.delete(job). If jobStoreID was not given, does nothing.
- **basename** (*str*) – If supported by the implementation, use the given file basename so that when searching the job store with a query matching that basename, the file will be detected.

- **encoding** (*str*) – the name of the encoding used to encode the file. Encodings are the same as for encode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Raises

- ***ConcurrentFileModificationException*** – if the file was modified concurrently during an invocation of this method
- ***NoSuchJobException*** – if the job specified via jobStoreID does not exist

Return type
Iterator[*Tuple*[*IO*[bytes], str]]

FIXME: some implementations may not raise this

Returns
a context manager yielding a file handle which can be written to and an ID that references the newly created file and can be used to read the file in the future.

Return type
Iterator[Tuple[IO[bytes], str]]

Parameters

- **job_id** (Optional[str]) –
- **cleanup** (bool) –
- **basename** (Optional[str]) –
- **encoding** (Optional[str]) –
- **errors** (Optional[str]) –

abstract get_empty_file_store_id(*job_id=None*, *cleanup=False*, *basename=None*)

Creates an empty file in the job store and returns its ID. Call to fileExists(getEmptyFileStoreID(jobStoreID)) will return True.

Parameters

- **job_id** (Optional[str]) – the id of a job, or None. If specified, the may be associated with that job in a job-store-specific way. This may influence the returned ID.
- **cleanup** (bool) – Whether to attempt to delete the file when the job whose jobStoreID was given as jobStoreID is deleted with jobStore.delete(job). If jobStoreID was not given, does nothing.
- **basename** (Optional[str]) – If supported by the implementation, use the given file basename so that when searching the job store with a query matching that basename, the file will be detected.
- **job_id** –
- **cleanup** –
- **basename** –

Returns
a jobStoreFileID that references the newly created file and can be used to reference the file in the future.

Return type
str

abstract read_file(*file_id*, *local_path*, *symlink=False*)

Copies or hard links the file referenced by jobStoreFileID to the given local file path. The version will be consistent with the last copy of the file written/updated. If the file in the job store is later modified via updateFile or updateFileStream, it is implementation-defined whether those writes will be visible at localFilePath. The file is copied in an atomic manner. It will not appear in the local file system until the copy has completed.

The file at the given local path may not be modified after this method returns!

Note! Implementations of readFile need to respect/provide the executable attribute on FileIDs.

Parameters

- **file_id** (str) – ID of the file to be copied
- **local_path** (str) – the local path indicating where to place the contents of the given file in the job store
- **symlink** (bool) – whether the reader can tolerate a symlink. If set to true, the job store may create a symlink instead of a full copy of the file or a hard link.
- **file_id** –
- **local_path** –
- **symlink** –

Return type
None

abstract read_file_stream(*file_id*, *encoding=None*, *errors=None*)

Similar to readFile, but returns a context manager yielding a file handle which can be read from. The yielded file handle does not need to and should not be closed explicitly.

Parameters

- **file_id** (Union[*FileID*, str]) – ID of the file to get a readable file handle for
- **encoding** (Optional[str]) – the name of the encoding used to decode the file. Encodings are the same as for decode(). Defaults to None which represents binary mode.
- **errors** (Optional[str]) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.
- **file_id** –
- **encoding** –
- **errors** –

Returns
a context manager yielding a file handle which can be read from

Return type
Iterator[Union[IO[bytes], IO[str]]]

abstract delete_file(*file_id*)

Deletes the file with the given ID from this job store. This operation is idempotent, i.e. deleting a file twice or deleting a non-existent file will succeed silently.

Parameters

- **file_id** (str) – ID of the file to delete

- **file_id** –

Return type
None

fileExists(*jobStoreFileID*)

Determine whether a file exists in this job store.

Parameters
jobStoreFileID (str) –

Return type
bool

abstract file_exists(*file_id*)

Determine whether a file exists in this job store.

Parameters
file_id (str) – an ID referencing the file to be checked

Return type
bool

getFileSize(*jobStoreFileID*)

Get the size of the given file in bytes.

Parameters
jobStoreFileID (str) –

Return type
int

abstract get_file_size(*file_id*)

Get the size of the given file in bytes, or 0 if it does not exist when queried.

Note that job stores which encrypt files might return overestimates of file sizes, since the encrypted file may have been padded to the nearest block, augmented with an initialization vector, etc.

Parameters

- **file_id** (str) – an ID referencing the file to be checked
- **file_id** –

Return type
int

updateFile(*jobStoreFileID*, *localFilePath*)

Replaces the existing version of a file in the job store.

Parameters

- **jobStoreFileID** (str) –
- **localFilePath** (str) –

Return type
None

abstract update_file(*file_id*, *local_path*)

Replaces the existing version of a file in the job store.

Throws an exception if the file does not exist.

Parameters

- **file_id** (str) – the ID of the file in the job store to be updated
- **local_path** (str) – the local path to a file that will overwrite the current version in the job store

Raises

- *ConcurrentFileModificationException* – if the file was modified concurrently during an invocation of this method
- *NoSuchFileException* – if the specified file does not exist

Return type
None

abstract update_file_stream(*file_id*, *encoding=None*, *errors=None*)

Replaces the existing version of a file in the job store. Similar to writeFile, but returns a context manager yielding a file handle which can be written to. The yielded file handle does not need to and should not be closed explicitly.

Parameters

- **file_id** (str) – the ID of the file in the job store to be updated
- **encoding** (Optional[str]) – the name of the encoding used to encode the file. Encodings are the same as for encode(). Defaults to None which represents binary mode.
- **errors** (Optional[str]) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.
- **file_id** –
- **encoding** –
- **errors** –

Raises

- *ConcurrentFileModificationException* – if the file was modified concurrently during an invocation of this method
- *NoSuchFileException* – if the specified file does not exist

Return type
Iterator[IO[Any]]

abstract write_shared_file_stream(*shared_file_name*, *encrypted=None*, *encoding=None*, *errors=None*)

Returns a context manager yielding a writable file handle to the global file referenced by the given name. File will be created in an atomic manner.

Parameters

- **shared_file_name** (str) – A file name matching AbstractJobStore.fileNameRegex, unique within this job store
- **encrypted** (Optional[bool]) – True if the file must be encrypted, None if it may be encrypted or False if it must be stored in the clear.
- **encoding** (Optional[str]) – the name of the encoding used to encode the file. Encodings are the same as for encode(). Defaults to None which represents binary mode.

- **errors** (Optional[str]) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.
- **shared_file_name** –
- **encrypted** –
- **encoding** –
- **errors** –

Raises
ConcurrentFileModificationException – if the file was modified concurrently during an invocation of this method

Returns
a context manager yielding a writable file handle

Return type
Iterator[IO[bytes]]

abstract read_shared_file_stream(*shared_file_name*, *encoding=None*, *errors=None*)

Returns a context manager yielding a readable file handle to the global file referenced by the given name.

Parameters

- **shared_file_name** (str) – A file name matching AbstractJobStore.fileNameRegex, unique within this job store
- **encoding** (Optional[str]) – the name of the encoding used to decode the file. Encodings are the same as for decode(). Defaults to None which represents binary mode.
- **errors** (Optional[str]) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.
- **shared_file_name** –
- **encoding** –
- **errors** –

Returns
a context manager yielding a readable file handle

Return type
Iterator[IO[bytes]]

abstract write_logs(*msg*)

Stores a message as a log in the jobstore.

Parameters

- **msg** (str) – the string to be written
- **msg** –

Raises
ConcurrentFileModificationException – if the file was modified concurrently during an invocation of this method

Return type
None

abstract read_logs(*callback*, *read_all=False*)

Reads logs accumulated by the write_logs() method. For each log this method calls the given callback function with the message as an argument (rather than returning logs directly, this method must be supplied with a callback which will process log messages).

Only unread logs will be read unless the read_all parameter is set.

Parameters

- **callback** (Callable[..., Any]) – a function to be applied to each of the stats file handles found
- **read_all** (bool) – a boolean indicating whether to read the already processed stats files in addition to the unread stats files
- **callback** –
- **read_all** –

Raises

ConcurrentFileModificationException – if the file was modified concurrently during an invocation of this method

Returns

the number of stats files processed

Return type

int

write_leader_pid()

Write the pid of this process to a file in the job store.

Overwriting the current contents of pid.log is a feature, not a bug of this method. Other methods will rely on always having the most current pid available. So far there is no reason to store any old pids.

Return type

None

read_leader_pid()

Read the pid of the leader process to a file in the job store.

Raises

NoSuchFileException – If the PID file doesn't exist.

Return type

int

write_leader_node_id()

Write the leader node id to the job store. This should only be called by the leader.

Return type

None

read_leader_node_id()

Read the leader node id stored in the job store.

Raises

NoSuchFileException – If the node ID file doesn't exist.

Return type

str

write_kill_flag(*kill=False*)

Write a file inside the job store that serves as a kill flag.

The initialized file contains the characters "NO". This should only be changed when the user runs the "toil kill" command.

Changing this file to a "YES" triggers a kill of the leader process. The workers are expected to be cleaned up by the leader.

Parameters
kill (bool) –

Return type
None

read_kill_flag()

Read the kill flag from the job store, and return True if the leader has been killed. False otherwise.

Return type
bool

default_caching()

Jobstore's preference as to whether it likes caching or doesn't care about it. Some jobstores benefit from caching, however on some local configurations it can be flaky.

see https://github.com/DataBiosphere/toil/issues/4218

Return type
bool

CHAPTER

FIFTEEN

TOIL JOB API

Functions to wrap jobs and return values (promises).

15.1 FunctionWrappingJob

The subclass of Job for wrapping user functions.

class toil.job.**FunctionWrappingJob**(*userFunction*, **args*, ***kwargs*)

Job used to wrap a function. In its *run* method the wrapped function is called.

__init__(*userFunction*, **args*, ***kwargs*)

Parameters

userFunction (*callable*) – The function to wrap. It will be called with `*args` and `**kwargs` as arguments.

The keywords `memory`, `cores`, `disk`, `accelerators`, ``preemptible` and `checkpoint` are reserved keyword arguments that if specified will be used to determine the resources required for the job, as `toil.job.Job.__init__()`. If they are keyword arguments to the function they will be extracted from the function definition, but may be overridden by the user (as you would expect).

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

15.2 JobFunctionWrappingJob

The subclass of FunctionWrappingJob for wrapping user job functions.

class toil.job.**JobFunctionWrappingJob**(*userFunction*, **args*, ***kwargs*)

A job function is a function whose first argument is a *Job* instance that is the wrapping job for the function. This can be used to add successor jobs for the function and perform all the functions the *Job* class provides.

To enable the job function to get access to the *toil.fileStores.abstractFileStore.AbstractFileStore* instance (see *toil.job.Job.run()*), it is made a variable of the wrapping job called fileStore.

To specify a job's resource requirements the following default keyword arguments can be specified:

- memory
- disk
- cores
- accelerators
- preemptible

For example to wrap a function into a job we would call:

```
Job.wrapJobFn(myJob, memory='100k', disk='1M', cores=0.1)
```

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

15.3 EncapsulatedJob

The subclass of Job for *encapsulating* a job, allowing a subgraph of jobs to be treated as a single job.

class toil.job.**EncapsulatedJob**(*job*, *unitName=None*)

A convenience Job class used to make a job subgraph appear to be a single job.

Let A be the root job of a job subgraph and B be another job we'd like to run after A and all its successors have completed, for this use encapsulate:

```
#  Job A and subgraph, Job B
A, B = A(), B()
Aprime = A.encapsulate()
Aprime.addChild(B)
#  B will run after A and all its successors have completed, A and its subgraph of
# successors in effect appear to be just one job.
```

If the job being encapsulated has predecessors (e.g. is not the root job), then the encapsulated job will inherit these predecessors. If predecessors are added to the job being encapsulated after the encapsulated job is created then the encapsulating job will NOT inherit these predecessors automatically. Care should be exercised to ensure the encapsulated job has the proper set of predecessors.

The return value of an encapsulated job (as accessed by the *toil.job.Job.rv()* function) is the return value of the root job, e.g. A().encapsulate().rv() and A().rv() will resolve to the same value after A or A.encapsulate() has been run.

__init__(*job*, *unitName=None*)

Parameters

- **job** (toil.job.Job) – the job to encapsulate.

- **unitName** (*str*) – human-readable name to identify this job instance.

addChild(*childJob*)

Add a childJob to be run as child of this job.

Child jobs will be run directly after this job's *toil.job.Job.run()* method has completed.

Returns
childJob: for call chaining

addService(*service*, *parentService=None*)

Add a service.

The *toil.job.Job.Service.start()* method of the service will be called after the run method has completed but before any successors are run. The service's *toil.job.Job.Service.stop()* method will be called once the successors of the job have been run.

Services allow things like databases and servers to be started and accessed by jobs in a workflow.

Raises
toil.job.JobException – If service has already been made the child of a job or another service.

Parameters

- **service** – Service to add.
- **parentService** – Service that will be started before 'service' is started. Allows trees of services to be established. parentService must be a service of this job.

Returns
a promise that will be replaced with the return value from *toil.job.Job.Service.start()* of service in any successor of the job.

addFollowOn(*followOnJob*)

Add a follow-on job.

Follow-on jobs will be run after the child jobs and their successors have been run.

Returns
followOnJob for call chaining

rv(**path*)

Create a *promise* (*toil.job.Promise*).

The "promise" representing a return value of the job's run method, or, in case of a function-wrapping job, the wrapped function's return value.

Parameters
path (*(Any)*) – Optional path for selecting a component of the promised return value. If absent or empty, the entire return value will be used. Otherwise, the first element of the path is used to select an individual item of the return value. For that to work, the return value must be a list, dictionary or of any other type implementing the *__getitem__()* magic method. If the selected item is yet another composite value, the second element of the path can be used to select an item from it, and so on. For example, if the return value is *[6,{'a':42}]*, *.rv(0)* would select *6* , *rv(1)* would select *{'a':3}* while *rv(1,'a')* would select *3*. To select a slice from a return value that is slicable, e.g. tuple or list, the path element should be a *slice* object. For example, assuming that the return value is *[6, 7, 8, 9]* then *.rv(slice(1, 3))* would select *[7, 8]*. Note that slicing really only makes sense at the end of path.

Return type
Promise

Returns
A promise representing the return value of this jobs *toil.job.Job.run()* method.

prepareForPromiseRegistration(*jobStore*)

Set up to allow this job's promises to register themselves.

Prepare this job (the promisor) so that its promises can register themselves with it, when the jobs they are promised to (promisees) are serialized.

The promissee holds the reference to the promise (usually as part of the job arguments) and when it is being pickled, so will the promises it refers to. Pickling a promise triggers it to be registered with the promissor.

15.4 Promise

The class used to reference return values of jobs/services not yet run/started.

class toil.job.**Promise**(*job*, *path*)

References a return value from a method as a *promise* before the method itself is run.

References a return value from a *toil.job.Job.run()* or *toil.job.Job.Service.start()* method as a *promise* before the method itself is run.

Let T be a job. Instances of *Promise* (termed a *promise*) are returned by T.rv(), which is used to reference the return value of T's run function. When the promise is passed to the constructor (or as an argument to a wrapped function) of a different, successor job the promise will be replaced by the actual referenced return value. This mechanism allows a return values from one job's run method to be input argument to job before the former job's run function has been executed.

Parameters

- **job** (*Job*) –
- **path** (Any) –

Return type
Promise

filesToDelete = {}

A set of IDs of files containing promised values when we know we won't need them anymore

__init__(*job*, *path*)

Initialize this promise.

Parameters

- **job** (*Job*) – the job whose return value this promise references
- **path** (Any) – see *Job.rv()*
- **job** –

class toil.job.**PromisedRequirement**(*valueOrCallable*, **args*)

Class for dynamically allocating job function resource requirements.

(involving *toil.job.Promise* instances.)

Use when resource requirements depend on the return value of a parent function. PromisedRequirements can be modified by passing a function that takes the *Promise* as input.

For example, let f, g, and h be functions. Then a Toil workflow can be defined as follows:: A = Job.wrapFn(f) B = A.addChildFn(g, cores=PromisedRequirement(A.rv()) C = B.addChildFn(h, cores=PromisedRequirement(lambda x: 2*x, B.rv()))

__init__(*valueOrCallable*, **args*)

Initialize this Promised Requirement.

Parameters

- **valueOrCallable** – A single Promise instance or a function that takes args as input parameters.
- **args** (*int or .Promise*) – variable length argument list

getValue()

Return PromisedRequirement value.

static convertPromises(*kwargs*)

Return True if reserved resource keyword is a Promise or PromisedRequirement instance.

Converts Promise instance to PromisedRequirement.

Parameters
kwargs (Dict[str, Any]) – function keyword arguments

Return type
bool

CHAPTER

SIXTEEN

JOB METHODS API

Jobs are the units of work in Toil which are composed into workflows.

class toil.job.**Job**(*memory=None*, *cores=None*, *disk=None*, *accelerators=None*, *preemptible=None*, *preemptable=None*, *unitName=''*, *checkpoint=False*, *displayName=''*, *descriptionClass=None*, *local=None*)

Class represents a unit of work in toil.

Parameters

- **memory** (Union[str, int, None]) –
- **cores** (Union[str, int, float, None]) –
- **disk** (Union[str, int, None]) –
- **accelerators** (Union[str, int, Mapping[str, Any], *AcceleratorRequirement*, Sequence[Union[str, int, Mapping[str, Any], *AcceleratorRequirement*]], None]) –
- **preemptible** (Union[str, int, bool, None]) –
- **preemptable** (Union[str, int, bool, None]) –
- **unitName** (Optional[str]) –
- **checkpoint** (Optional[bool]) –
- **displayName** (Optional[str]) –
- **descriptionClass** (Optional[type]) –
- **local** (Optional[bool]) –

__init__(*memory=None*, *cores=None*, *disk=None*, *accelerators=None*, *preemptible=None*, *preemptable=None*, *unitName=''*, *checkpoint=False*, *displayName=''*, *descriptionClass=None*, *local=None*)

Job initializer.

This method must be called by any overriding constructor.

Parameters

- **memory** (*int or string convertible by toil.lib.conversions.human2bytes to an int*) – the maximum number of bytes of memory the job will require to run.
- **cores** (*float, int, or string convertible by toil.lib.conversions.human2bytes to an int*) – the number of CPU cores required.
- **disk** (*int or string convertible by toil.lib.conversions.human2bytes to an int*) – the amount of local disk space required by the job, expressed in bytes.

- **accelerators** (*int, string, dict, or list of those. Strings and dicts must be parseable by parse_accelerator.*) – the computational accelerators required by the job. If a string, can be a string of a number, or a string specifying a model, brand, or API (with optional colon-delimited count).
- **preemptible** (*bool, int in {0, 1}, or string in {'false', 'true'} in any case*) – if the job can be run on a preemptible node.
- **preemptable** (Union[str, int, bool, None]) – legacy preemptible parameter, for backwards compatibility with workflows not using the preemptible keyword
- **unitName** (*str*) – Human-readable name for this instance of the job.
- **checkpoint** (*bool*) – if any of this job's successor jobs completely fails, exhausting all their retries, remove any successor jobs and rerun this job to restart the subtree. Job must be a leaf vertex in the job graph when initially defined, see `toil.job.Job.checkNewCheckpointsAreCutVertices()`.
- **displayName** (*str*) – Human-readable job type display name.
- **descriptionClass** (*class*) – Override for the JobDescription class used to describe the job.
- **local** (Optional[bool]) – if the job can be run on the leader.

Return type
None

property jobStoreID: Union[str, *TemporaryID*]

Get the ID of this Job.

Return type
Union[str, *TemporaryID*]

property description: *JobDescription*

Expose the JobDescription that describes this job.

Return type
JobDescription

property disk: int

The maximum number of bytes of disk the job will require to run.

Return type
int

property memory

The maximum number of bytes of memory the job will require to run.

property cores: Union[int, float]

The number of CPU cores required.

Return type
Union[int, float]

property accelerators: List[*AcceleratorRequirement*]

Any accelerators, such as GPUs, that are needed.

Return type
List[*AcceleratorRequirement*]

property preemptible: bool

Whether the job can be run on a preemptible node.

Return type
bool

property checkpoint: bool

Determine if the job is a checkpoint job or not.

Return type
bool

assignConfig(*config*)

Assign the given config object.

It will be used by various actions implemented inside the Job class.

Parameters
config (*Config*) – Config object to query

Return type
None

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters
fileStore (*AbstractFileStore*) – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Return type
Any

Returns
The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

addChild(*childJob*)

Add a childJob to be run as child of this job.

Child jobs will be run directly after this job's *toil.job.Job.run()* method has completed.

Return type
Job

Returns
childJob: for call chaining

Parameters
childJob (*Job*) –

hasChild(*childJob*)

Check if childJob is already a child of this job.

Return type
bool

Returns
True if childJob is a child of the job, else False.

Parameters
childJob (*Job*) –

addFollowOn(*followOnJob*)

Add a follow-on job.

Follow-on jobs will be run after the child jobs and their successors have been run.

Return type
Job

Returns
followOnJob for call chaining

Parameters
followOnJob (*Job*) –

hasPredecessor(*job*)

Check if a given job is already a predecessor of this job.

Parameters
job (*Job*) –

Return type
bool

hasFollowOn(*followOnJob*)

Check if given job is already a follow-on of this job.

Return type
bool

Returns
True if the followOnJob is a follow-on of this job, else False.

Parameters
followOnJob (*Job*) –

addService(*service*, *parentService=None*)

Add a service.

The *toil.job.Job.Service.start()* method of the service will be called after the run method has completed but before any successors are run. The service's *toil.job.Job.Service.stop()* method will be called once the successors of the job have been run.

Services allow things like databases and servers to be started and accessed by jobs in a workflow.

Raises
toil.job.JobException – If service has already been made the child of a job or another service.

Parameters

- **service** (Service) – Service to add.
- **parentService** (Optional[Service]) – Service that will be started before 'service' is started. Allows trees of services to be established. parentService must be a service of this job.

Return type
Promise

Returns
a promise that will be replaced with the return value from *toil.job.Job.Service.start()* of service in any successor of the job.

hasService(*service*)

Return True if the given Service is a service of this job, and False otherwise.

Parameters
service (Service) –

Return type
bool

addChildFn(*fn*, **args*, ***kwargs*)

Add a function as a child job.

Parameters
fn (Callable) – Function to be run as a child job with `*args` and `**kwargs` as arguments to this function. See toil.job.FunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Return type
FunctionWrappingJob

Returns
The new child job that wraps fn.

addFollowOnFn(*fn*, **args*, ***kwargs*)

Add a function as a follow-on job.

Parameters
fn (Callable) – Function to be run as a follow-on job with `*args` and `**kwargs` as arguments to this function. See toil.job.FunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Return type
FunctionWrappingJob

Returns
The new follow-on job that wraps fn.

addChildJobFn(*fn*, **args*, ***kwargs*)

Add a job function as a child job.

See *toil.job.JobFunctionWrappingJob* for a definition of a job function.

Parameters
fn (Callable) – Job function to be run as a child job with `*args` and `**kwargs` as arguments to this function. See toil.job.JobFunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Return type
FunctionWrappingJob

Returns
The new child job that wraps fn.

addFollowOnJobFn(*fn*, **args*, ***kwargs*)

Add a follow-on job function.

See *toil.job.JobFunctionWrappingJob* for a definition of a job function.

Parameters
fn (Callable) – Job function to be run as a follow-on job with `*args` and `**kwargs` as arguments to this function. See toil.job.JobFunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Return type
FunctionWrappingJob

Returns
The new follow-on job that wraps fn.

property tempDir: str

Shortcut to calling `job.fileStore.getLocalTempDir()`.

Temp dir is created on first call and will be returned for first and future calls :return: Path to tempDir. See *job.fileStore.getLocalTempDir*

Return type
str

log(*text*, *level=20*)

Log using `fileStore.logToMaster()`.

Parameters
text (str) –

Return type
None

static wrapFn(*fn*, **args*, ***kwargs*)

Makes a Job out of a function.

Convenience function for constructor of *toil.job.FunctionWrappingJob*.

Parameters
fn – Function to be run with `*args` and `**kwargs` as arguments. See toil.job.JobFunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Return type
FunctionWrappingJob

Returns
The new function that wraps fn.

static wrapJobFn(*fn*, **args*, ***kwargs*)

Makes a Job out of a job function.

Convenience function for constructor of *toil.job.JobFunctionWrappingJob*.

Parameters
fn – Job function to be run with `*args` and `**kwargs` as arguments. See toil.job.JobFunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Return type
JobFunctionWrappingJob

Returns
The new job function that wraps fn.

encapsulate(*name=None*)

Encapsulates the job, see *toil.job.EncapsulatedJob*. Convenience function for constructor of *toil.job.EncapsulatedJob*.

Parameters
name (Optional[str]) – Human-readable name for the encapsulated job.

Return type
EncapsulatedJob

Returns
an encapsulated version of this job.

rv(**path*)

Create a *promise* (*toil.job.Promise*).

The "promise" representing a return value of the job's run method, or, in case of a function-wrapping job, the wrapped function's return value.

Parameters
path (*(Any)*) – Optional path for selecting a component of the promised return value. If absent or empty, the entire return value will be used. Otherwise, the first element of the path is used to select an individual item of the return value. For that to work, the return value must be a list, dictionary or of any other type implementing the *__getitem__()* magic method. If the selected item is yet another composite value, the second element of the path can be used to select an item from it, and so on. For example, if the return value is *[6,{'a':42}]*, *.rv(0)* would select *6* , *rv(1)* would select *{'a':3}* while *rv(1,'a')* would select *3*. To select a slice from a return value that is slicable, e.g. tuple or list, the path element should be a *slice* object. For example, assuming that the return value is *[6, 7, 8, 9]* then *.rv(slice(1, 3))* would select *[7, 8]*. Note that slicing really only makes sense at the end of path.

Return type
Promise

Returns
A promise representing the return value of this jobs *toil.job.Job.run()* method.

prepareForPromiseRegistration(*jobStore*)

Set up to allow this job's promises to register themselves.

Prepare this job (the promisor) so that its promises can register themselves with it, when the jobs they are promised to (promisees) are serialized.

The promissee holds the reference to the promise (usually as part of the job arguments) and when it is being pickled, so will the promises it refers to. Pickling a promise triggers it to be registered with the promissor.

Parameters
jobStore (*AbstractJobStore*) –

Return type
None

checkJobGraphForDeadlocks()

Ensures that a graph of Jobs (that hasn't yet been saved to the JobStore) doesn't contain any pathological relationships between jobs that would result in deadlocks if we tried to run the jobs.

See *toil.job.Job.checkJobGraphConnected()*, toil.job.Job.checkJobGraphAcyclic() and *toil.job.Job.checkNewCheckpointsAreLeafVertices()* for more info.

Raises
toil.job.JobGraphDeadlockException – if the job graph is cyclic, contains multiple roots or contains checkpoint jobs that are not leaf vertices when defined (see toil.job.Job.checkNewCheckpointsAreLeaves()).

getRootJobs()

Return the set of root job objects that contain this job.

A root job is a job with no predecessors (i.e. which are not children, follow-ons, or services).

Only deals with jobs created here, rather than loaded from the job store.

Return type
Set[*Job*]

checkJobGraphConnected()

Raises
toil.job.JobGraphDeadlockException – if *toil.job.Job.getRootJobs()* does not contain exactly one root job.

As execution always starts from one root job, having multiple root jobs will cause a deadlock to occur.

Only deals with jobs created here, rather than loaded from the job store.

checkJobGraphAcylic()

Raises
toil.job.JobGraphDeadlockException – if the connected component of jobs containing this job contains any cycles of child/followOn dependencies in the *augmented job graph* (see below). Such cycles are not allowed in valid job graphs.

A follow-on edge (A, B) between two jobs A and B is equivalent to adding a child edge to B from (1) A, (2) from each child of A, and (3) from the successors of each child of A. We call each such edge an edge an "implied" edge. The augmented job graph is a job graph including all the implied edges.

For a job graph G = (V, E) the algorithm is `O(|V|^2)`. It is `O(|V| + |E|)` for a graph with no follow-ons. The former follow-on case could be improved!

Only deals with jobs created here, rather than loaded from the job store.

checkNewCheckpointsAreLeafVertices()

A checkpoint job is a job that is restarted if either it fails, or if any of its successors completely fails, exhausting their retries.

A job is a leaf it is has no successors.

A checkpoint job must be a leaf when initially added to the job graph. When its run method is invoked it can then create direct successors. This restriction is made to simplify implementation.

Only works on connected components of jobs not yet added to the JobStore.

Raises
toil.job.JobGraphDeadlockException – if there exists a job being added to the graph for which checkpoint=True and which is not a leaf.

Return type
None

defer(*function*, **args*, ***kwargs*)

Register a deferred function, i.e. a callable that will be invoked after the current attempt at running this job concludes. A job attempt is said to conclude when the job function (or the *toil.job.Job.run()* method for class-based jobs) returns, raises an exception or after the process running it terminates abnormally. A deferred function will be called on the node that attempted to run the job, even if a subsequent attempt is made on another node. A deferred function should be idempotent because it may be called multiple times on the same node or even in the same process. More than one deferred function may be registered per job attempt by calling this method repeatedly with different arguments. If the same function is registered twice with the same or different arguments, it will be called twice per job attempt.

Examples for deferred functions are ones that handle cleanup of resources external to Toil, like Docker containers, files outside the work directory, etc.

Parameters

- **function** (*callable*) – The function to be called after this job concludes.
- **args** (*list*) – The arguments to the function
- **kwargs** (*dict*) – The keyword arguments to the function

Return type
None

getTopologicalOrderingOfJobs()

Return type
List[*Job*]

Returns
a list of jobs such that for all pairs of indices i, j for which i < j, the job at index i can be run before the job at index j.

Only considers jobs in this job's subgraph that are newly added, not loaded from the job store.

Ignores service jobs.

saveBody(*jobStore*)

Save the execution data for just this job to the JobStore, and fill in the JobDescription with the information needed to retrieve it.

The Job's JobDescription must have already had a real jobStoreID assigned to it.

Does not save the JobDescription.

Parameters
jobStore (*AbstractJobStore*) – The job store to save the job body into.

Return type
None

saveAsRootJob(*jobStore*)

Save this job to the given jobStore as the root job of the workflow.

Return type
JobDescription

Returns
the JobDescription describing this job.

Parameters
jobStore (*AbstractJobStore*) –

classmethod loadJob(*jobStore*, *jobDescription*)

Retrieves a *toil.job.Job* instance from a JobStore

Parameters

- **jobStore** (*AbstractJobStore*) – The job store.
- **jobDescription** (*JobDescription*) – the JobDescription of the job to retrieve.

Return type
Job

Returns
The job referenced by the JobDescription.

16.1 JobDescription

The class used to store all the information that the Toil Leader ever needs to know about a Job.

class toil.job.**JobDescription**(*requirements*, *jobName*, *unitName=''*, *displayName=''*, *command=None*, *local=None*)

Stores all the information that the Toil Leader ever needs to know about a Job.

(requirements information, dependency information, commands to issue, etc.)

Can be obtained from an actual (i.e. executable) Job object, and can be used to obtain the Job object from the JobStore.

Never contains other Jobs or JobDescriptions: all reference is by ID.

Subclassed into variants for checkpoint jobs and service jobs that have their specific parameters.

Parameters

- **requirements** (Mapping[str, Union[int, str, bool]]) –
- **jobName** (str) –
- **unitName** (Optional[str]) –
- **displayName** (Optional[str]) –
- **command** (Optional[str]) –
- **local** (Optional[bool]) –

__init__(*requirements*, *jobName*, *unitName=''*, *displayName=''*, *command=None*, *local=None*)

Create a new JobDescription.

Parameters

- **requirements** (Mapping[str, Union[int, str, bool]]) – Dict from string to number, string, or bool describing the resource requirements of the job. 'cores', 'memory', 'disk', and 'preemptible' fields, if set, are parsed and broken out into properties. If unset, the relevant property will be unspecified, and will be pulled from the assigned Config object if queried (see *toil.job.Requirer.assignConfig()*).
- **jobName** (str) – Name of the kind of job this is. May be used in job store IDs and logging. Also used to let the cluster scaler learn a model for how long the job will take. Ought to be the job class's name if no real user-defined name is available.
- **unitName** (Optional[str]) – Name of this instance of this kind of job. May appear with jobName in logging.
- **displayName** (Optional[str]) – A human-readable name to identify this particular job instance. Ought to be the job class's name if no real user-defined name is available.
- **local** (Optional[bool]) – If True, the job is meant to use minimal resources but is sensitive to execution latency, and so should be executed by the leader.
- **command** (Optional[str]) –

Return type
None

serviceHostIDsInBatches()

Find all batches of service host job IDs that can be started at the same time.

(in the order they need to start in)

Return type
Iterator[List[str]]

successorsAndServiceHosts()

Get an iterator over all child, follow-on, and service job IDs.

Return type
Iterator[str]

allSuccessors()

Get an iterator over all child, follow-on, and chained, inherited successor job IDs.

Follow-ons will come before children.

Return type
Iterator[str]

successors_by_phase()

Get an iterator over all child/follow-on/chained inherited successor job IDs, along with their phase numbere on the stack.

Phases ececute higher numbers to lower numbers.

Return type
Iterator[Tuple[int, str]]

property services

Get a collection of the IDs of service host jobs for this job, in arbitrary order.

Will be empty if the job has no unfinished services.

nextSuccessors()

Return the collection of job IDs for the successors of this job that are ready to run.

If those jobs have multiple predecessor relationships, they may still be blocked on other jobs.

Returns None when at the final phase (all successors done), and an empty collection if there are more phases but they can't be entered yet (e.g. because we are waiting for the job itself to run).

Return type
Set[str]

filterSuccessors(*predicate*)

Keep only successor jobs for which the given predicate function approves.

The predicate function is called with the job's ID.

Treats all other successors as complete and forgets them.

Parameters
predicate (Callable[[str], bool]) –

Return type
None

filterServiceHosts(*predicate*)

Keep only services for which the given predicate approves.

The predicate function is called with the service host job's ID.

Treats all other services as complete and forgets them.

Parameters
predicate (Callable[[str], bool]) –

Return type
None

clear_nonexistent_dependents(*job_store*)

Remove all references to child, follow-on, and associated service jobs that do not exist.

That is to say, all those that have been completed and removed.

Parameters
job_store (*AbstractJobStore*) –

Return type
None

clear_dependents()

Remove all references to successor and service jobs.

Return type
None

is_subtree_done()

Check if the subtree is done.

Return type
bool

Returns
True if the job appears to be done, and all related child, follow-on, and service jobs appear to be finished and removed.

replace(*other*)

Take on the ID of another JobDescription, retaining our own state and type.

When updated in the JobStore, we will save over the other JobDescription.

Useful for chaining jobs: the chained-to job can replace the parent job.

Merges cleanup state and successors other than this job from the job being replaced into this one.

Parameters
other (*JobDescription*) – Job description to replace.

Return type
None

addChild(*childID*)

Make the job with the given ID a child of the described job.

Parameters
childID (str) –

Return type
None

addFollowOn(*followOnID*)

Make the job with the given ID a follow-on of the described job.

Parameters
followOnID (str) –

Return type
None

addServiceHostJob(*serviceID*, *parentServiceID=None*)

Make the ServiceHostJob with the given ID a service of the described job.

If a parent ServiceHostJob ID is given, that parent service will be started first, and must have already been added.

hasChild(*childID*)

Return True if the job with the given ID is a child of the described job.

Parameters
childID (str) –

Return type
bool

hasFollowOn(*followOnID*)

Test if the job with the given ID is a follow-on of the described job.

Parameters
followOnID (str) –

Return type
bool

hasServiceHostJob(*serviceID*)

Test if the ServiceHostJob is a service of the described job.

Return type
bool

renameReferences(*renames*)

Apply the given dict of ID renames to all references to jobs.

Does not modify our own ID or those of finished predecessors. IDs not present in the renames dict are left as-is.

Parameters
renames (Dict[*TemporaryID*, str]) – Rename operations to apply.

Return type
None

addPredecessor()

Notify the JobDescription that a predecessor has been added to its Job.

Return type
None

onRegistration(*jobStore*)

Perform setup work that requires the JobStore.

Called by the Job saving logic when this JobDescription meets the JobStore and has its ID assigned.

Overridden to perform setup work (like hooking up flag files for service jobs) that requires the JobStore.

Parameters
jobStore (*AbstractJobStore*) – The job store we are being placed into

Return type
None

setupJobAfterFailure(*exit_status=None*, *exit_reason=None*)

Configure job after a failure.

Reduce the remainingTryCount if greater than zero and set the memory to be at least as big as the default memory (in case of exhaustion of memory, which is common).

Requires a configuration to have been assigned (see *toil.job.Requirer.assignConfig()*).

Parameters

- **exit_status** (Optional[int]) – The exit code from the job.
- **exit_reason** (Optional[*BatchJobExitReason*]) – The reason the job stopped, if available from the batch system.

Return type
None

getLogFileHandle(*jobStore*)

Create a context manager that yields a file handle to the log file.

Assumes logJobStoreFileID is set.

property remainingTryCount

Get the number of tries remaining.

The try count set on the JobDescription, or the default based on the retry count from the config if none is set.

clearRemainingTryCount()

Clear remainingTryCount and set it back to its default value.

Return type
bool

Returns
True if a modification to the JobDescription was made, and False otherwise.

pre_update_hook()

Run before pickling and saving a created or updated version of this job.

Called by the job store.

Return type
None

get_job_kind()

Return an identifying string for the job.

The result may contain spaces.

Return type
str

Returns: Either the unit name, job name, or display name, which identifies
the kind of job it is to toil. Otherwise "Unknown Job" in case no identifier is available

CHAPTER

SEVENTEEN

JOB.RUNNER API

The Runner contains the methods needed to configure and start a Toil run.

class Job.**Runner**

Used to setup and run Toil workflow.

static getDefaultArgumentParser()

Get argument parser with added toil workflow options.

Return type
ArgumentParser

Returns
The argument parser used by a toil workflow with added Toil options.

static getDefaultOptions(*jobStore*)

Get default options for a toil workflow.

Parameters
jobStore (str) – A string describing the jobStore for the workflow.

Return type
Namespace

Returns
The options used by a toil workflow.

static addToilOptions(*parser*)

Adds the default toil options to an optparse or argparse parser object.

Parameters
parser (Union[OptionParser, ArgumentParser]) – Options object to add toil options to.

Return type
None

static startToil(*job*, *options*)

Run the toil workflow using the given options.

Deprecated by toil.common.Toil.start.

(see Job.Runner.getDefaultOptions and Job.Runner.addToilOptions) starting with this job. :type job: *Job* :param job: root job of the workflow :raises: toil.exceptions.FailedJobsException if at the end of function there remain failed jobs. :rtype: Any :return: The return value of the root job's run function.

Parameters
job (Job) –

Return type
Any

CHAPTER

EIGHTEEN

JOB.FILESTORE API

The AbstractFileStore is an abstraction of a Toil run's shared storage.

class toil.fileStores.abstractFileStore.**AbstractFileStore**(*jobStore*, *jobDesc*, *file_store_dir*, *waitForPreviousCommit*)

Interface used to allow user code run by Toil to read and write files.

Also provides the interface to other Toil facilities used by user code, including:

- normal (non-real-time) logging
- finding the correct temporary directory for scratch work
- importing and exporting files into and out of the workflow

Stores user files in the jobStore, but keeps them separate from actual jobs.

May implement caching.

Passed as argument to the *toil.job.Job.run()* method.

Access to files is only permitted inside the context manager provided by *toil.fileStores.abstractFileStore.AbstractFileStore.open()*.

Also responsible for committing completed jobs back to the job store with an update operation, and allowing that commit operation to be waited for.

Parameters

- **jobStore** (*AbstractJobStore*) –
- **jobDesc** (*JobDescription*) –
- **file_store_dir** (str) –
- **waitForPreviousCommit** (Callable[[], Any]) –

__init__(*jobStore*, *jobDesc*, *file_store_dir*, *waitForPreviousCommit*)

Create a new file store object.

Parameters

- **jobStore** (*AbstractJobStore*) – the job store in use for the current Toil run.
- **jobDesc** (*JobDescription*) – the JobDescription object for the currently running job.
- **file_store_dir** (str) – the per-worker local temporary directory where the file store should store local files. Per-job directories will be created under here by the file store.
- **waitForPreviousCommit** (Callable[[], Any]) – the waitForCommit method of the previous job's file store, when jobs are running in sequence on the same worker. Used to

prevent this file store's startCommit and the previous job's startCommit methods from running at the same time and racing. If they did race, it might be possible for the later job to be fully marked as completed in the job store before the eralier job was.

Return type
None

static createFileStore(*jobStore*, *jobDesc*, *file_store_dir*, *waitForPreviousCommit*, *caching*)

Create a concreate FileStore.

Parameters

- **jobStore** (*AbstractJobStore*) –
- **jobDesc** (*JobDescription*) –
- **file_store_dir** (str) –
- **waitForPreviousCommit** (Callable[[], Any]) –
- **caching** (Optional[bool]) –

Return type
Union[*NonCachingFileStore*, *CachingFileStore*]

static shutdownFileStore(*workflowID*, *config_work_dir*, *config_coordination_dir*)

Carry out any necessary filestore-specific cleanup.

This is a destructive operation and it is important to ensure that there are no other running processes on the system that are modifying or using the file store for this workflow.

This is the intended to be the last call to the file store in a Toil run, called by the batch system cleanup function upon batch system shutdown.

Parameters

- **workflowID** (str) – The workflow ID for this invocation of the workflow
- **config_work_dir** (Optional[str]) – The path to the work directory in the Toil Config.
- **config_coordination_dir** (Optional[str]) – The path to the coordination directory in the Toil Config.

Return type
None

open(*job*)

Create the context manager around tasks prior and after a job has been run.

File operations are only permitted inside the context manager.

Implementations must only yield from within *with super().open(job):*.

Parameters
job (*Job*) – The job instance of the toil job to run.

Return type
Generator[None, None, None]

getLocalTempDir()

Get a new local temporary directory in which to write files.

The directory will only persist for the duration of the job.

Return type
str

Returns
The absolute path to a new local temporary directory. This directory will exist for the duration of the job only, and is guaranteed to be deleted once the job terminates, removing all files it contains recursively.

getLocalTempFile(*suffix=None*, *prefix=None*)

Get a new local temporary file that will persist for the duration of the job.

Parameters

- **suffix** (Optional[str]) – If not None, the file name will end with this string. Otherwise, default value ".tmp" will be used
- **prefix** (Optional[str]) – If not None, the file name will start with this string. Otherwise, default value "tmp" will be used

Return type
str

Returns
The absolute path to a local temporary file. This file will exist for the duration of the job only, and is guaranteed to be deleted once the job terminates.

getLocalTempFileName(*suffix=None*, *prefix=None*)

Get a valid name for a new local file. Don't actually create a file at the path.

Parameters

- **suffix** (Optional[str]) – If not None, the file name will end with this string. Otherwise, default value ".tmp" will be used
- **prefix** (Optional[str]) – If not None, the file name will start with this string. Otherwise, default value "tmp" will be used

Return type
str

Returns
Path to valid file

abstract writeGlobalFile(*localFileName*, *cleanup=False*)

Upload a file (as a path) to the job store.

If the file is in a FileStore-managed temporary directory (i.e. from *toil.fileStores.abstractFileStore.AbstractFileStore.getLocalTempDir()*), it will become a local copy of the file, eligible for deletion by *toil.fileStores.abstractFileStore.AbstractFileStore.deleteLocalFile()*.

If an executable file on the local filesystem is uploaded, its executability will be preserved when it is downloaded again.

Parameters

- **localFileName** (str) – The path to the local file to upload. The last path component (basename of the file) will remain associated with the file in the file store, if supported by the backing JobStore, so that the file can be searched for by name or name glob.
- **cleanup** (bool) – if True then the copy of the global file will be deleted once the job and all its successors have completed running. If not the global file must be deleted manually.

Return type
FileID

Returns
an ID that can be used to retrieve the file.

writeGlobalFileStream(*cleanup=False*, *basename=None*, *encoding=None*, *errors=None*)

Similar to writeGlobalFile, but allows the writing of a stream to the job store. The yielded file handle does not need to and should not be closed explicitly.

Parameters

- **encoding** (Optional[str]) – The name of the encoding used to decode the file. Encodings are the same as for decode(). Defaults to None which represents binary mode.
- **errors** (Optional[str]) – Specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.
- **cleanup** (bool) – is as in *toil.fileStores.abstractFileStore.AbstractFileStore.writeGlobalFile()*.
- **basename** (Optional[str]) – If supported by the backing JobStore, use the given file basename so that when searching the job store with a query matching that basename, the file will be detected.

Return type
Iterator[Tuple[*WriteWatchingStream*, *FileID*]]

Returns
A context manager yielding a tuple of 1) a file handle which can be written to and 2) the toil.fileStores.FileID of the resulting file in the job store.

logAccess(*fileStoreID*, *destination=None*)

Record that the given file was read by the job.

(to be announced if the job fails)

If destination is not None, it gives the path that the file was downloaded to. Otherwise, assumes that the file was streamed.

Must be called by *readGlobalFile()* and *readGlobalFileStream()* implementations.

Parameters

- **fileStoreID** (Union[*FileID*, str]) –
- **destination** (Optional[str]) –

Return type
None

abstract readGlobalFile(*fileStoreID*, *userPath=None*, *cache=True*, *mutable=False*, *symlink=False*)

Make the file associated with fileStoreID available locally.

If mutable is True, then a copy of the file will be created locally so that the original is not modified and does not change the file for other jobs. If mutable is False, then a link can be created to the file, saving disk resources. The file that is downloaded will be executable if and only if it was originally uploaded from an executable file on the local filesystem.

If a user path is specified, it is used as the destination. If a user path isn't specified, the file is stored in the local temp directory with an encoded name.

The destination file must not be deleted by the user; it can only be deleted through deleteLocalFile.

Implementations must call *logAccess()* to report the download.

Parameters

- **fileStoreID** (str) – job store id for the file
- **userPath** (Optional[str]) – a path to the name of file to which the global file will be copied or hard-linked (see below).
- **cache** (bool) – Described in toil.fileStores.CachingFileStore.readGlobalFile()
- **mutable** (bool) – Described in toil.fileStores.CachingFileStore.readGlobalFile()
- **symlink** (bool) – True if caller can accept symlink, False if caller can only accept a normal file or hardlink

Return type
str

Returns
An absolute path to a local, temporary copy of the file keyed by fileStoreID.

abstract readGlobalFileStream(*fileStoreID*, *encoding=None*, *errors=None*)

Read a stream from the job store; similar to readGlobalFile.

The yielded file handle does not need to and should not be closed explicitly.

Parameters

- **encoding** (Optional[str]) – the name of the encoding used to decode the file. Encodings are the same as for decode(). Defaults to None which represents binary mode.
- **errors** (Optional[str]) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.
- **fileStoreID** (*str*) –

Return type
AbstractContextManager[*Union*[*IO*[bytes], *IO*[str]]]

Implementations must call *logAccess()* to report the download.

Return type
AbstractContextManager[Union[IO[bytes], IO[str]]]

Returns
a context manager yielding a file handle which can be read from.

Parameters

- **fileStoreID** (str) –
- **encoding** (*Optional[str]*) –
- **errors** (*Optional[str]*) –

getGlobalFileSize(*fileStoreID*)

Get the size of the file pointed to by the given ID, in bytes.

If a FileID or something else with a non-None 'size' field, gets that.

Otherwise, asks the job store to poll the file's size.

Note that the job store may overestimate the file's size, for example if it is encrypted and had to be augmented with an IV or other encryption framing.

Parameters
fileStoreID (Union[*FileID*, str]) – File ID for the file

Return type
int

Returns
File's size in bytes, as stored in the job store

abstract deleteLocalFile(*fileStoreID*)

Delete local copies of files associated with the provided job store ID.

Raises an OSError with an errno of errno.ENOENT if no such local copies exist. Thus, cannot be called multiple times in succession.

The files deleted are all those previously read from this file ID via readGlobalFile by the current job into the job's file-store-provided temp directory, plus the file that was written to create the given file ID, if it was written by the current job from the job's file-store-provided temp directory.

Parameters
fileStoreID (Union[*FileID*, str]) – File Store ID of the file to be deleted.

Return type
None

abstract deleteGlobalFile(*fileStoreID*)

Delete local files and then permanently deletes them from the job store.

To ensure that the job can be restarted if necessary, the delete will not happen until after the job's run method has completed.

Parameters
fileStoreID (Union[*FileID*, str]) – the File Store ID of the file to be deleted.

Return type
None

logToMaster(*text*, *level=20*)

Send a logging message to the leader. The message will also be logged by the worker at the same level.

Parameters

- **text** (str) – The string to log.
- **level** (int) – The logging level.

Return type
None

abstract startCommit(*jobState=False*)

Update the status of the job on the disk.

May start an asynchronous process. Call waitForCommit() to wait on that process.

Parameters
jobState (bool) – If True, commit the state of the FileStore's job, and file deletes. Otherwise, commit only file creates/updates.

Return type
None

abstract waitForCommit()

Blocks while startCommit is running.

This function is called by this job's successor to ensure that it does not begin modifying the job store until after this job has finished doing so.

Might be called when startCommit is never called on a particular instance, in which case it does not block.

Return type
bool

Returns
Always returns True

abstract classmethod shutdown(*shutdown_info*)

Shutdown the filestore on this node.

This is intended to be called on batch system shutdown.

Parameters
shutdown_info (Any) – The implementation-specific shutdown information, for shutting down the file store and removing all its state and all job local temp directories from the node.

Return type
None

class toil.fileStores.**FileID**(*fileStoreID*, *size*, *executable=False*)

A small wrapper around Python's builtin string class.

It is used to represent a file's ID in the file store, and has a size attribute that is the file's size in bytes. This object is returned by importFile and writeGlobalFile.

Calls into the file store can use bare strings; size will be queried from the job store if unavailable in the ID.

Parameters

- **fileStoreID** (str) –
- **size** (int) –
- **executable** (bool) –
- **args** (*Any*) –

Return type
FileID

__init__(*fileStoreID*, *size*, *executable=False*)

Parameters

- **fileStoreID** (str) –
- **size** (int) –
- **executable** (bool) –

Return type
None

pack()

Pack the FileID into a string so it can be passed through external code.

Return type
str

classmethod unpack(*packedFileStoreID*)

Unpack the result of pack() into a FileID object.

Parameters
packedFileStoreID (str) –

Return type
FileID

CHAPTER

NINETEEN

BATCH SYSTEM API

The batch system interface is used by Toil to abstract over different ways of running batches of jobs, for example Slurm, GridEngine, Mesos, Parasol and a single node. The *toil.batchSystems.abstractBatchSystem.AbstractBatchSystem* API is implemented to run jobs using a given job management system, e.g. Mesos.

19.1 Batch System Enivronmental Variables

Environmental variables allow passing of scheduler specific parameters.

For SLURM there are two environment variables - the first applies to all jobs, while the second defined the partition to use for parallel jobs:

```
export TOIL_SLURM_ARGS="-t 1:00:00 -q fatq"
export TOIL_SLURM_PE='multicore'
```

For TORQUE there are two environment variables - one for everything but the resource requirements, and another - for resources requirements (without the *-l* prefix):

```
export TOIL_TORQUE_ARGS="-q fatq"
export TOIL_TORQUE_REQS="walltime=1:00:00"
```

For GridEngine (SGE, UGE), there is an additional environmental variable to define the parallel environment for running multicore jobs:

```
export TOIL_GRIDENGINE_PE='smp'
export TOIL_GRIDENGINE_ARGS='-q batch.q'
```

For HTCondor, additional parameters can be included in the submit file passed to condor_submit:

```
export TOIL_HTCONDOR_PARAMS='requirements = TARGET.has_sse4_2 == true; accounting_group␣
↪= test'
```

The environment variable is parsed as a semicolon-separated string of `parameter = value` pairs.

19.2 Batch System API

class toil.batchSystems.abstractBatchSystem.**AbstractBatchSystem**

An abstract base class to represent the interface the batch system must provide to Toil.

abstract classmethod supportsAutoDeployment()

Whether this batch system supports auto-deployment of the user script itself.

If it does, the *setUserScript()* can be invoked to set the resource object representing the user script.

Note to implementors: If your implementation returns True here, it should also override

Return type
bool

abstract classmethod supportsWorkerCleanup()

Indicates whether this batch system invokes *BatchSystemSupport.workerCleanup()* after the last job for a particular workflow invocation finishes. Note that the term *worker* refers to an entire node, not just a worker process. A worker process may run more than one job sequentially, and more than one concurrent worker process may exist on a worker node, for the same workflow. The batch system is said to *shut down* after the last worker process terminates.

Return type
bool

setUserScript(*userScript*)

Set the user script for this workflow. This method must be called before the first job is issued to this batch system, and only if *supportsAutoDeployment()* returns True, otherwise it will raise an exception.

Parameters
userScript (*Resource*) – the resource object representing the user script or module and the modules it depends on.

Return type
None

set_message_bus(*message_bus*)

Give the batch system an opportunity to connect directly to the message bus, so that it can send informational messages about the jobs it is running to other Toil components.

Parameters
message_bus (*MessageBus*) –

Return type
None

abstract issueBatchJob(*jobDesc*, *job_environment=None*)

Issues a job with the specified command to the batch system and returns a unique jobID.

Parameters

- **jobDesc** (*JobDescription*) – a toil.job.JobDescription
- **job_environment** (Optional[Dict[str, str]]) – a collection of job-specific environment variables to be set on the worker.

Return type
int

Returns
a unique jobID that can be used to reference the newly issued job

abstract killBatchJobs(*jobIDs*)

Kills the given job IDs. After returning, the killed jobs will not appear in the results of getRunningBatchJobIDs. The killed job will not be returned from getUpdatedBatchJob.

Parameters
jobIDs (List[int]) – list of IDs of jobs to kill

Return type
None

abstract getIssuedBatchJobIDs()

Gets all currently issued jobs

Return type
List[int]

Returns
A list of jobs (as jobIDs) currently issued (may be running, or may be waiting to be run). Despite the result being a list, the ordering should not be depended upon.

abstract getRunningBatchJobIDs()

Gets a map of jobs as jobIDs that are currently running (not just waiting) and how long they have been running, in seconds.

Return type
Dict[int, float]

Returns
dictionary with currently running jobID keys and how many seconds they have been running as the value

abstract getUpdatedBatchJob(*maxWait*)

Returns information about job that has updated its status (i.e. ceased running, either successfully or with an error). Each such job will be returned exactly once.

Does not return info for jobs killed by killBatchJobs, although they may cause None to be returned earlier than maxWait.

Parameters
maxWait (int) – the number of seconds to block, waiting for a result

Return type
Optional[*UpdatedBatchJobInfo*]

Returns
If a result is available, returns UpdatedBatchJobInfo. Otherwise it returns None. wallTime is the number of seconds (a strictly positive float) in wall-clock time the job ran for, or None if this batch system does not support tracking wall time.

getSchedulingStatusMessage()

Get a log message fragment for the user about anything that might be going wrong in the batch system, if available.

If no useful message is available, return None.

This can be used to report what resource is the limiting factor when scheduling jobs, for example. If the leader thinks the workflow is stuck, the message can be displayed to the user to help them diagnose why it might be stuck.

Return type
Optional[str]

Returns
User-directed message about scheduling state.

abstract shutdown()

Called at the completion of a toil invocation. Should cleanly terminate all worker threads.

Return type
None

setEnv(*name*, *value=None*)

Set an environment variable for the worker process before it is launched.

The worker process will typically inherit the environment of the machine it is running on but this method makes it possible to override specific variables in that inherited environment before the worker is launched. Note that this mechanism is different to the one used by the worker internally to set up the environment of a job. A call to this method affects all jobs issued after this method returns. Note to implementors: This means that you would typically need to copy the variables before enqueuing a job.

If no value is provided it will be looked up from the current environment.

Parameters

- **name** (str) –
- **value** (Optional[str]) –

Return type
None

classmethod add_options(*parser*)

If this batch system provides any command line options, add them to the given parser.

Parameters
parser (Union[ArgumentParser, _ArgumentGroup]) –

Return type
None

classmethod setOptions(*setOption*)

Process command line or configuration options relevant to this batch system.

Parameters
setOption (*OptionSetter*) – A function with signature setOption(option_name, parsing_function=None, check_function=None, default=None, env=None) returning nothing, used to update run configuration as a side effect.

Return type
None

getWorkerContexts()

Get a list of picklable context manager objects to wrap worker work in, in order.

Can be used to ask the Toil worker to do things in-process (such as configuring environment variables, hot-deploying user scripts, or cleaning up a node) that would otherwise require a wrapping "executor" process.

Return type
List[AbstractContextManager[Any]]

CHAPTER

TWENTY

JOB.SERVICE API

The Service class allows databases and servers to be spawned within a Toil workflow.

class Job.**Service**(*memory=None*, *cores=None*, *disk=None*, *accelerators=None*, *preemptible=None*, *unitName=None*)

Abstract class used to define the interface to a service.

Should be subclassed by the user to define services.

Is not executed as a job; runs within a ServiceHostJob.

__init__(*memory=None*, *cores=None*, *disk=None*, *accelerators=None*, *preemptible=None*, *unitName=None*)

Memory, core and disk requirements are specified identically to as in `toil.job.Job.__init__()`.

abstract start(*job*)

Start the service.

Parameters

job (*Job*) – The underlying host job that the service is being run in. Can be used to register deferred functions, or to access the fileStore for creating temporary files.

Return type

Any

Returns

An object describing how to access the service. The object must be pickleable and will be used by jobs to access the service (see *toil.job.Job.addService()*).

abstract stop(*job*)

Stops the service. Function can block until complete.

Parameters

job (*Job*) – The underlying host job that the service is being run in. Can be used to register deferred functions, or to access the fileStore for creating temporary files.

Return type

None

check()

Checks the service is still running.

Raises

exceptions.RuntimeError – If the service failed, this will cause the service job to be labeled failed.

Return type

bool

Returns

True if the service is still running, else False. If False then the service job will be terminated, and considered a success. Important point: if the service job exits due to a failure, it should raise a RuntimeError, not return False!

CHAPTER

TWENTYONE

EXCEPTIONS API

Toil specific exceptions.

exception toil.job.**JobException**(*message*)

General job exception.

Parameters
message (str) –

Return type
None

__init__(*message*)

Parameters
message (str) –

Return type
None

exception toil.job.**JobGraphDeadlockException**(*string*)

An exception raised in the event that a workflow contains an unresolvable dependency, such as a cycle. See *toil.job.Job.checkJobGraphForDeadlocks()*.

__init__(*string*)

exception toil.jobStores.abstractJobStore.**ConcurrentFileModificationException**(*jobStoreFileID*)

Indicates that the file was attempted to be modified by multiple processes at once.

Parameters
jobStoreFileID (*FileID*) –

__init__(*jobStoreFileID*)

Parameters
jobStoreFileID (*FileID*) – the ID of the file that was modified by multiple workers or processes concurrently

exception toil.jobStores.abstractJobStore.**JobStoreExistsException**(*locator*)

Indicates that the specified job store already exists.

Parameters
locator (str) –

__init__(*locator*)

Parameters

- **locator** (str) – The location of the job store

- **locator** –

exception toil.jobStores.abstractJobStore.**NoSuchFileException**(*jobStoreFileID*, *customName=None*, **extra*)

Indicates that the specified file does not exist.

Parameters

- **jobStoreFileID** (*FileID*) –
- **customName** (Optional[str]) –
- **extra** (Any) –

__init__(*jobStoreFileID*, *customName=None*, **extra*)

Parameters

- **jobStoreFileID** (*FileID*) – the ID of the file that was mistakenly assumed to exist
- **customName** (Optional[str]) – optionally, an alternate name for the nonexistent file
- **extra** (Any) – optional extra information to add to the error message
- **extra** –

exception toil.jobStores.abstractJobStore.**NoSuchJobException**(*jobStoreID*)

Indicates that the specified job does not exist.

Parameters

jobStoreID (*FileID*) –

__init__(*jobStoreID*)

Parameters

- **jobStoreID** (*FileID*) – the jobStoreID that was mistakenly assumed to exist
- **jobStoreID** –

exception toil.jobStores.abstractJobStore.**NoSuchJobStoreException**(*locator*)

Indicates that the specified job store does not exist.

Parameters

locator (str) –

__init__(*locator*)

Parameters

- **locator** (str) – The location of the job store
- **locator** –

CHAPTER

TWENTYTWO

RUNNING TESTS

Test make targets, invoked as `$ make <target>`, subject to which environment variables are set (see *Running Integration Tests*).

TARGET	DESCRIPTION
test	Invokes all tests.
integration_test	Invokes only the integration tests.
test_offline	Skips building the Docker appliance and only invokes tests that have no docker dependencies.
integration_test_local	Makes integration tests easier to debug locally by running the integration tests serially and doesn't redirect output. This makes it appears on the terminal as expected.

Before running tests for the first time, initialize your virtual environment following the steps in *Building from Source*.

Run all tests (including slow tests):

```
$ make test
```

Run only quick tests (as of Jul 25, 2018, this was ~ 20 minutes):

```
$ export TOIL_TEST_QUICK=True; make test
```

Run an individual test with:

```
$ make test tests=src/toil/test/sort/sortTest.py::SortTest::testSort
```

The default value for `tests` is `"src"` which includes all tests in the `src/` subdirectory of the project root. Tests that require a particular feature will be skipped implicitly. If you want to explicitly skip tests that depend on a currently installed *feature*, use

```
$ make test tests="-m 'not aws' src"
```

This will run only the tests that don't depend on the `aws` extra, even if that extra is currently installed. Note the distinction between the terms *feature* and *extra*. Every extra is a feature but there are features that are not extras, such as the `gridengine` and `parasol` features. To skip tests involving both the `parasol` feature and the `aws` extra, use the following:

```
$ make test tests="-m 'not aws and not parasol' src"
```

22.1 Running Tests with pytest

Often it is simpler to use pytest directly, instead of calling the `make` wrapper. This usually works as expected, but some tests need some manual preparation. To run a specific test with pytest, use the following:

```
python -m pytest src/toil/test/sort/sortTest.py::SortTest::testSort
```

For more information, see the pytest documentation.

22.2 Running Integration Tests

These tests are generally only run using in our CI workflow due to their resource requirements and cost. However, they can be made available for local testing:

- Running tests that make use of Docker (e.g. autoscaling tests and Docker tests) require an appliance image to be hosted. First, make sure you have gone through the set up found in *Using Docker with Quay*. Then to build and host the appliance image run the `make` target `push_docker`.

  ```
  $ make push_docker
  ```

- Running integration tests require activation via an environment variable as well as exporting information relevant to the desired tests. Enable the integration tests:

  ```
  $ export TOIL_TEST_INTEGRATIVE=True
  ```

- Finally, set the environment variables for keyname and desired zone:

  ```
  $ export TOIL_X_KEYNAME=[Your Keyname]
  $ export TOIL_X_ZONE=[Desired Zone]
  ```

  Where `X` is one of our currently supported cloud providers (`GCE`, `AWS`).

- See the above sections for guidance on running tests.

22.3 Test Environment Variables

TOIL_TEST_TEMP	An absolute path to a directory where Toil tests will write their temporary files. Defaults to the system's standard temporary directory.
TOIL_TEST_INTEGRATIVE	If `True`, this allows the integration tests to run. Only valid when running the tests from the source directory via `make test` or `make test_parallel`.
TOIL_AWS_KEYNAME	An AWS keyname (see *Preparing your AWS environment*), which is required to run the AWS tests.
TOIL_GOOGLE_PROJECTID	A Google Cloud account projectID (see *Running in Google Compute Engine (GCE)*), which is required to to run the Google Cloud tests.
TOIL_TEST_QUICK	If `True`, long running tests are skipped.

Partial install and failing tests

Some tests may fail with an ImportError if the required extras are not installed. Install Toil with all of the extras do prevent such errors.

22.4 Using Docker with Quay

Docker is needed for some of the tests. Follow the appropriate installation instructions for your system on their website to get started.

When running `make test` you might still get the following error:

```
$ make test
Please set TOIL_DOCKER_REGISTRY, e.g. to quay.io/USER.
```

To solve, make an account with Quay and specify it like so:

```
$ TOIL_DOCKER_REGISTRY=quay.io/USER make test
```

where `USER` is your Quay username.

For convenience you may want to add this variable to your bashrc by running

```
$ echo 'export TOIL_DOCKER_REGISTRY=quay.io/USER' >> $HOME/.bashrc
```

22.5 Running Mesos Tests

If you're running Toil's Mesos tests, be sure to create the virtualenv with `--system-site-packages` to include the Mesos Python bindings. Verify this by activating the virtualenv and running `pip list | grep mesos`. On macOS, this may come up empty. To fix it, run the following:

```
for i in /usr/local/lib/python2.7/site-packages/*mesos*; do ln -snf $i venv/lib/python2.
↪7/site-packages/; done
```

CHAPTER

TWENTYTHREE

DEVELOPING WITH DOCKER

To develop on features reliant on the Toil Appliance (the docker image toil uses for AWS autoscaling), you should consider setting up a personal registry on Quay or Docker Hub. Because the Toil Appliance images are tagged with the Git commit they are based on and because only commits on our master branch trigger an appliance build on Quay, as soon as a developer makes a commit or dirties the working copy they will no longer be able to rely on Toil to automatically detect the proper Toil Appliance image. Instead, developers wishing to test any appliance changes in autoscaling should build and push their own appliance image to a personal Docker registry. This is described in the next section.

23.1 Making Your Own Toil Docker Image

Note! Toil checks if the docker image specified by TOIL_APPLIANCE_SELF exists prior to launching by using the docker v2 schema. This should be valid for any major docker repository, but there is an option to override this if desired using the option: -\-*forceDockerAppliance*.

Here is a general workflow (similar instructions apply when using Docker Hub):

1. Make some changes to the provisioner of your local version of Toil
2. Go to the location where you installed the Toil source code and run

   ```
   $ make docker
   ```

   to automatically build a docker image that can now be uploaded to your personal Quay account. If you have not installed Toil source code yet see *Building from Source*.
3. If it's not already you will need Docker installed and need to log into Quay. Also you will want to make sure that your Quay account is public.
4. Set the environment variable `TOIL_DOCKER_REGISTRY` to your Quay account. If you find yourself doing this often you may want to add

   ```
   export TOIL_DOCKER_REGISTRY=quay.io/<MY_QUAY_USERNAME>
   ```

   to your `.bashrc` or equivalent.
5. Now you can run

   ```
   $ make push_docker
   ```

   which will upload the docker image to your Quay account. Take note of the image's tag for the next step.
6. Finally you will need to tell Toil from where to pull the Appliance image you've created (it uses the Toil release you have installed by default). To do this set the environment variable `TOIL_APPLIANCE_SELF` to the url of your image. For more info see *Environment Variables*.

7. Now you can launch your cluster! For more information see *Running a Workflow with Autoscaling*.

23.2 Running a Cluster Locally

The Toil Appliance container can also be useful as a test environment since it can simulate a Toil cluster locally. An important caveat for this is autoscaling, since autoscaling will only work on an EC2 instance and cannot (at this time) be run on a local machine.

To spin up a local cluster, start by using the following Docker run command to launch a Toil leader container:

```
docker run \
    --entrypoint=mesos-master \
    --net=host \
    -d \
    --name=leader \
    --volume=/home/jobStoreParentDir:/jobStoreParentDir \
    quay.io/ucsc_cgl/toil:3.6.0 \
    --registry=in_memory \
    --ip=127.0.0.1 \
    --port=5050 \
    --allocation_interval=500ms
```

A couple notes on this command: the `-d` flag tells Docker to run in daemon mode so the container will run in the background. To verify that the container is running you can run `docker ps` to see all containers. If you want to run your own container rather than the official UCSC container you can simply replace the `quay.io/ucsc_cgl/toil:3.6.0` parameter with your own container name.

Also note that we are not mounting the job store directory itself, but rather the location where the job store will be written. Due to complications with running Docker on MacOS, I recommend only mounting directories within your home directory. The next command will launch the Toil worker container with similar parameters:

```
docker run \
    --entrypoint=mesos-slave \
    --net=host \
    -d \
    --name=worker \
    --volume=/home/jobStoreParentDir:/jobStoreParentDir \
    quay.io/ucsc_cgl/toil:3.6.0 \
    --work_dir=/var/lib/mesos \
    --master=127.0.0.1:5050 \
    --ip=127.0.0.1 \
    --attributes=preemptable:False \
    --resources=cpus:2
```

Note here that we are specifying 2 CPUs and a non-preemptable worker. We can easily change either or both of these in a logical way. To change the number of cores we can change the 2 to whatever number you like, and to change the worker to be preemptable we change `preemptable:False` to `preemptable:True`. Also note that the same volume is mounted into the worker. This is needed since both the leader and worker write and read from the job store. Now that your cluster is running, you can run

```
docker exec -it leader bash
```

to get a shell in your leader 'node'. You can also replace the `leader` parameter with `worker` to get shell access in your worker.

Docker-in-Docker issues

If you want to run Docker inside this Docker cluster (Dockerized tools, perhaps), you should also mount in the Docker socket via `-v /var/run/docker.sock:/var/run/docker.sock`. This will give the Docker client inside the Toil Appliance access to the Docker engine on the host. Client/engine version mismatches have been known to cause issues, so we recommend using Docker version 1.12.3 on the host to be compatible with the Docker client installed in the Appliance. Finally, be careful where you write files inside the Toil Appliance - 'child' Docker containers launched in the Appliance will actually be siblings to the Appliance since the Docker engine is located on the host. This means that the 'child' container can only mount in files from the Appliance if the files are located in a directory that was originally mounted into the Appliance from the host - that way the files are accessible to the sibling container. Note: if Docker can't find the file/directory on the host it will silently fail and mount in an empty directory.

CHAPTER

TWENTYFOUR

MAINTAINER'S GUIDELINES

In general, as developers and maintainers of the code, we adhere to the following guidelines:

- We strive to never break the build on master. All development should be done on branches, in either the main Toil repository or in developers' forks.
- Pull requests should be used for any and all changes (except truly trivial ones).
- Pull requests should be in response to issues. If you find yourself making a pull request without an issue, you should create the issue first.

24.1 Naming Conventions

- **Commit messages** *should* be great. Most importantly, they *must*:
  - Have a short subject line. If in need of more space, drop down **two** lines and write a body to explain what is changing and why it has to change.
  - Write the subject line as a command: *Destroy all humans*, not *All humans destroyed*.
  - Reference the issue being fixed in a Github-parseable format, such as *(resolves #1234)* at the end of the subject line, or *This will fix #1234.* somewhere in the body. If no single commit on its own fixes the issue, the cross-reference must appear in the pull request title or body instead.
- **Branches** in the main Toil repository *must* start with `issues/`, followed by the issue number (or numbers, separated by a dash), followed by a short, lowercase, hyphenated description of the change. (There can be many open pull requests with their associated branches at any given point in time and this convention ensures that we can easily identify branches.)

  Say there is an issue numbered #123 titled *Foo does not work*. The branch name would be `issues/123-fix-foo` and the title of the commit would be *Fix foo in case of bar (resolves #123).*

24.2 Pull Requests

- All pull requests must be reviewed by a person other than the request's author. Review the PR by following the *Reviewing Pull Requests* checklist.
- Modified pull requests must be re-reviewed before merging. **Note that Github does not enforce this!**
- Merge pull requests by following the *Merging Pull Requests* checklist.
- When merging a pull request, make sure to update the Draft Changelog on the Github wiki, which we will use to produce the changelog for the next release. The PR template tells you to do this, so don't forget. New entries should go at the bottom.

- Pull requests will not be merged unless CI tests pass. Gitlab tests are only run on code in the main Toil repository on some branch, so it is the responsibility of the approving reviewer to make sure that pull requests from outside repositories are copied to branches in the main repository. This can be accomplished with (from a Toil clone):

```
./contrib/admin/test-pr theirusername their-branch issues/123-fix-description-here
```

  This must be repeated every time the PR submitter updates their PR, after checking to see that the update is not malicious.

  If there is no issue corresponding to the PR, after which the branch can be named, the reviewer of the PR should first create the issue.

  Developers who have push access to the main Toil repository are encouraged to make their pull requests from within the repository, to avoid this step.

- Prefer using "Squash and marge" when merging pull requests to master especially when the PR contains a "single unit" of work (i.e. if one were to rewrite the PR from scratch with all the fixes included, they would have one commit for the entire PR). This makes the commit history on master more readable and easier to debug in case of a breakage.

  When squashing a PR from multiple authors, please add Co-authored-by to give credit to all contributing authors.

  See Issue #2816 for more details.

24.3 Publishing a Release

These are the steps to take to publish a Toil release:

- Determine the release version **X.Y.Z**. This should follow semantic versioning; if user-workflow-breaking changes are made, **X** should be incremented, and **Y** and **Z** should be zero. If non-breaking changes are made but new functionality is added, **X** should remain the same as the last release, **Y** should be incremented, and **Z** should be zero. If only patches are released, **X** and **Y** should be the same as the last release and **Z** should be incremented.
- If it does not exist already, create a release branch in the Toil repo named `X.Y.x`, where **x** is a literal lower-case "x". For patch releases, find the existing branch and make sure it is up to date with the patch commits that are to be released. They may be cherry-picked over from master.
- On the release branch, edit `version_template.py` in the root of the repository. Find the line that looks like this (slightly different for patch releases):

```
baseVersion = 'X.Y.0a1'
```

  Make it look like this instead:

```
baseVersion = 'X.Y.Z'
```

  Commit your change to the branch.

- Tag the current state of the release branch as `releases/X.Y.Z`.
- Make the Github release here, referencing that tag. For a non-patch release, fill in the description with the changelog from the wiki page, which you should clear. For a patch release, just describe the patch.
- For a non-patch release, set up the main branch so that development builds will declare themselves to be alpha versions of what the next release will probably be. Edit `version_template.py` in the root of the repository on the main branch to set `baseVersion` like this:

```
baseVersion = 'X.Y+1.0a1'
```

Make sure to replace `X` and `Y+1` with actual numbers.

24.4 Using Git Hooks

In the `contrib/hooks` directory, there are two scripts, `mypy-after-commit.py` and `mypy-before-push.py`, that can be set up as Git hooks to make sure you don't accidentally push commits that would immediately fail type-checking. These are supposed to eliminate the need to run `make mypy` constantly. You can install them into your Git working copy like this

```
ln -rs ./contrib/hooks/mypy-after-commit.py .git/hooks/post-commit
ln -rs ./contrib/hooks/mypy-before-push.py .git/hooks/pre-push
```

After you make a commit, the post-commit script will start type-checking it, and if it takes too long re-launch the process in the background. When you push, the pre-push script will see if the commit you are pushing type-checked successfully, and if it hasn't been type-checked but is currently checked out, it will be type-checked. If type-checking fails, the push will be aborted.

Type-checking will only be performed if you are in a Toil development virtual environment. If you aren't, the scripts won't do anything.

To bypass or override pre-push hook, if it is wrong or if you need to push something that doesn't typecheck, you can `git push --no-verify`. If the scripts get confused about whether a commit actually typechecks, you can clear out the type-checking result cache, which is in `/var/run/user/<your UID>/.mypy_toil_result_cache` on Linux and in `.mypy_toil_result_cache` in the Toil repo on Mac.

To uninstall the scripts, delete `.git/hooks/post-commit` and `.git/hooks/pre-push`.

24.5 Adding Retries to a Function

See toil.lib.retry .

retry() can be used to decorate any function based on the list of errors one wishes to retry on.

This list of errors can contain normal Exception objects, and/or RetryCondition objects wrapping Exceptions to include additional conditions.

For example, retrying on a one Exception (HTTPError):

```
from requests import get
from requests.exceptions import HTTPError

@retry(errors=[HTTPError])
def update_my_wallpaper():
    return get('https://www.deviantart.com/')
```

Or:

```
from requests import get
from requests.exceptions import HTTPError

@retry(errors=[HTTPError, ValueError])
```

(continues on next page)

(continued from previous page)

```
def update_my_wallpaper():
    return get('https://www.deviantart.com/')
```

The examples above will retry for the default interval on any errors specified the "errors=" arg list.

To retry on specifically 500/502/503/504 errors, you could specify an ErrorCondition object instead, for example:

```
from requests import get
from requests.exceptions import HTTPError

@retry(errors=[
    ErrorCondition(
                error=HTTPError,
                error_codes=[500, 502, 503, 504]
            )])
def update_my_wallpaper():
    return requests.get('https://www.deviantart.com/')
```

To retry on specifically errors containing the phrase "NotFound":

```
from requests import get
from requests.exceptions import HTTPError

@retry(errors=[
    ErrorCondition(
        error=HTTPError,
        error_message_must_include="NotFound"
    )])
def update_my_wallpaper():
    return requests.get('https://www.deviantart.com/')
```

To retry on all HTTPError errors EXCEPT an HTTPError containing the phrase "NotFound":

```
from requests import get
from requests.exceptions import HTTPError

@retry(errors=[
    HTTPError,
    ErrorCondition(
                error=HTTPError,
                error_message_must_include="NotFound",
                retry_on_this_condition=False
            )])
def update_my_wallpaper():
    return requests.get('https://www.deviantart.com/')
```

To retry on boto3's specific status errors, an example of the implementation is:

```
import boto3
from botocore.exceptions import ClientError

@retry(errors=[
    ErrorCondition(
```

(continues on next page)

(continued from previous page)

```
                error=ClientError,
                boto_error_codes=["BucketNotFound"]
            )])
def boto_bucket(bucket_name):
    boto_session = boto3.session.Session()
    s3_resource = boto_session.resource('s3')
    return s3_resource.Bucket(bucket_name)
```

Any combination of these will also work, provided the codes are matched to the correct exceptions. A ValueError will not return a 404, for example.

The retry function as a decorator should make retrying functions easier and clearer. It also encourages smaller independent functions, as opposed to lumping many different things that may need to be retried on different conditions in the same function.

The ErrorCondition object tries to take some of the heavy lifting of writing specific retry conditions and boil it down to an API that covers all common use-cases without the user having to write any new bespoke functions.

Use-cases covered currently:

1. Retrying on a normal error, like a KeyError.
2. Retrying on HTTP error codes (use ErrorCondition).
3. Retrying on boto's specific status errors, like "BucketNotFound" (use ErrorCondition).
4. Retrying when an error message contains a certain phrase (use ErrorCondition).
5. Explicitly NOT retrying on a condition (use ErrorCondition).

If new functionality is needed, it's currently best practice in Toil to add functionality to the ErrorCondition itself rather than making a new custom retry method.

CHAPTER

TWENTYFIVE

PULL REQUEST CHECKLISTS

This document contains checklists for dealing with PRs. More general PR information is available at *Pull Requests*.

25.1 Reviewing Pull Requests

This checklist is to be kept in sync with the checklist in the pull request template.

When reviewing a PR, do the following:

- **Make sure it is coming from `issues/XXXX-fix-the-thing` in the Toil repo, or from an external repo.**
  - If it is coming from an external repo, make sure to pull it in for CI with:

    ```
    contrib/admin/test-pr otheruser theirbranchname issues/XXXX-fix-the-thing
    ```

  - If there is no associated issue, create one.
- **Read through the code changes. Make sure that it doesn't have:**
  - Addition of trailing whitespace.
  - New variable or member names in `camelCase` that want to be in `snake_case`.
  - New functions without type hints.
  - New functions or classes without informative docstrings.
  - Changes to semantics not reflected in the relevant docstrings.
  - New or changed command line options for Toil workflows that are not reflected in `docs/running/cliOptions.rst`
  - New features without tests.
- Comment on the lines of code where problems exist with a review comment. You can shift-click the line numbers in the diff to select multiple lines.
- Finish the review with an overall description of your opinion.

25.2 Merging Pull Requests

This checklist is to be kept in sync with the checklist in the pull request template.

When merging a PR, do the following:

- Make sure the PR passes tests.
- Make sure the PR has been reviewed **since its last modification**. If not, review it.
- **Merge with the Github "Squash and merge" feature.**
  - **If there are multiple authors' commits, add Co-authored-by to give credit to all contributing authors.**
- Copy its recommended changelog entry to the Draft Changelog.
- Append the issue number in parentheses to the changelog entry.

CHAPTER

TWENTYSIX

TOIL ARCHITECTURE

The following diagram layouts out the software architecture of Toil.

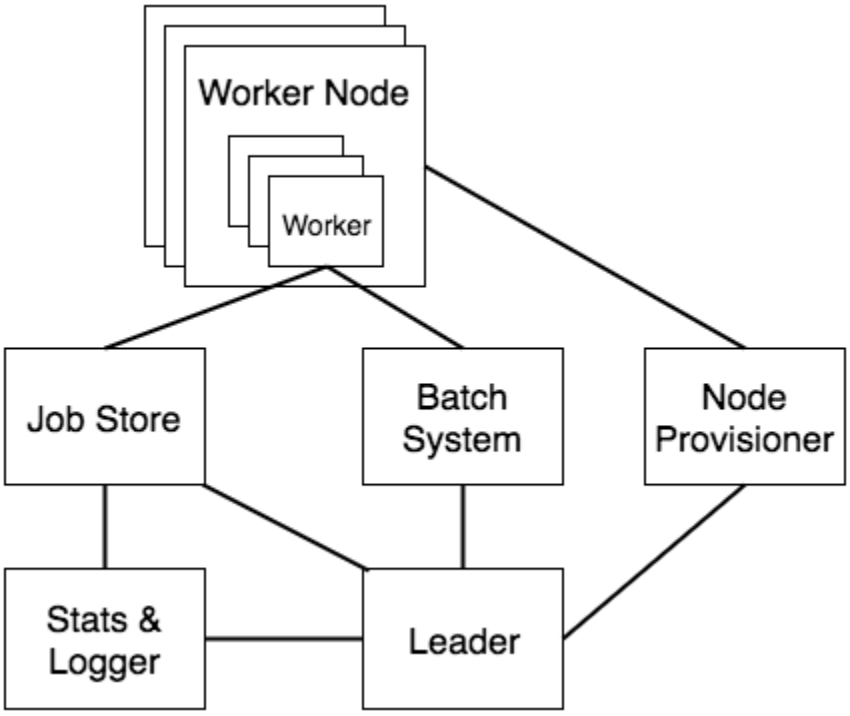


Fig. 1: Figure 1: The basic components of Toil's architecture.

These components are described below:

- **the leader:**
  The leader is responsible for deciding which jobs should be run. To do this it traverses the job graph. Currently this is a single threaded process, but we make aggressive steps to prevent it becoming a bottleneck (see *Read-only Leader* described below).

- **the job-store:**
  Handles all files shared between the components. Files in the job-store are the means by which the state of the workflow is maintained. Each job is backed by a file in the job store, and atomic updates to this state are used to ensure the workflow can always be resumed upon failure. The job-store can also store all user files, allowing them to be shared between jobs. The job-store is defined by the `AbstractJobStore` class. Multiple implementations of this class allow Toil to support different back-end file stores, e.g.: S3, network file systems, Google file store, etc.

- **workers:**
  The workers are temporary processes responsible for running jobs, one at a time per worker. Each worker process is invoked with a job argument that it is responsible for running. The worker monitors this job and reports back success or failure to the leader by editing the job's state in the file-store. If the job defines successor jobs the worker may choose to immediately run them (see *Job Chaining* below).
- **the batch-system:**
  Responsible for scheduling the jobs given to it by the leader, creating a worker command for each job. The batch-system is defined by the `AbstractBatchSystem` class. Toil uses multiple existing batch systems to schedule jobs, including Apache Mesos, GridEngine and a multi-process single node implementation that allows workflows to be run without any of these frameworks. Toil can therefore fairly easily be made to run a workflow using an existing cluster.
- **the node provisioner:**
  Creates worker nodes in which the batch system schedules workers. It is defined by the `AbstractProvisioner` class.
- **the statistics and logging monitor:**
  Monitors logging and statistics produced by the workers and reports them. Uses the job-store to gather this information.

26.1 Jobs and JobDescriptions

As noted in *Job Basics*, a job is the atomic unit of work in a Toil workflow. User scripts inherit from the `Job` class to define units of work. These jobs are pickled and stored in the job-store by the leader, and are retrieved and un-pickled by the worker when they are scheduled to run.

During scheduling, Toil does not work with the actual Job objects. Instead, `JobDescription` objects are used to store all the information that the Toil Leader ever needs to know about the Job. This includes requirements information, dependency information, commands to issue, etc.

Internally, the JobDescription object is referenced by its jobStoreID, which is often not human readable. However, the Job and JobDescription objects contain several human-readable names that are useful for logging and identification:

jobName	Name of the kind of job this is. This may be used in job store IDs and logging. Also used to let the cluster scaler learn a model for how long the job will take. Defaults to the job class's name if no real user-defined name is available. For a `FunctionWrappingJob`, the jobName is replaced by the wrapped function's name. For a CWL workflow, the jobName is the class name of the internal job that is running the CWL workflow, such as `"CWLJob"`.
unitName	Name of this *instance* of this kind of job. If set by the user, it will appear with the jobName in logging. For a CWL workflow, the unitName is set to a descriptive name that includes the CWL file name and the ID in the file if set.
displayName	A human-readable name to identify this particular job instance. Used as an identifier of the job class in the stats report. Defaults to the job class's name if no real user-defined name is available. For a CWL workflow, the displayName is the absolute workflow URI.

26.2 Optimizations

Toil implements lots of optimizations designed for scalability. Here we detail some of the key optimizations.

26.2.1 Read-only leader

The leader process is currently implemented as a single thread. Most of the leader's tasks revolve around processing the state of jobs, each stored as a file within the job-store. To minimise the load on this thread, each worker does as much work as possible to manage the state of the job it is running. As a result, with a couple of minor exceptions, the leader process never needs to write or update the state of a job within the job-store. For example, when a job is complete and has no further successors the responsible worker deletes the job from the job-store, marking it complete. The leader then only has to check for the existence of the file when it receives a signal from the batch-system to know that the job is complete. This off-loading of state management is orthogonal to future parallelization of the leader.

26.2.2 Job chaining

The scheduling of successor jobs is partially managed by the worker, reducing the number of individual jobs the leader needs to process. Currently this is very simple: if the there is a single next successor job to run and its resources fit within the resources of the current job and closely match the resources of the current job then the job is run immediately on the worker without returning to the leader. Further extensions of this strategy are possible, but for many workflows which define a series of serial successors (e.g. map sequencing reads, post-process mapped reads, etc.) this pattern is very effective at reducing leader workload.

26.2.3 Preemptable node support

Critical to running at large-scale is dealing with intermittent node failures. Toil is therefore designed to always be resumable providing the job-store does not become corrupt. This robustness allows Toil to run on preemptible nodes, which are only available when others are not willing to pay more to use them. Designing workflows that divide into many short individual jobs that can use preemptable nodes allows for workflows to be efficiently scheduled and executed.

26.2.4 Caching

Running bioinformatic pipelines often require the passing of large datasets between jobs. Toil caches the results from jobs such that child jobs running on the same node can directly use the same file objects, thereby eliminating the need for an intermediary transfer to the job store. Caching also reduces the burden on the local disks, because multiple jobs can share a single file. The resulting drop in I/O allows pipelines to run faster, and, by the sharing of files, allows users to run more jobs in parallel by reducing overall disk requirements.

To demonstrate the efficiency of caching, we ran an experimental internal pipeline on 3 samples from the TCGA Lung Squamous Carcinoma (LUSC) dataset. The pipeline takes the tumor and normal exome fastqs, and the tumor rna fastq and input, and predicts MHC presented neoepitopes in the patient that are potential targets for T-cell based immunotherapies. The pipeline was run individually on the samples on c3.8xlarge machines on AWS (60GB RAM,600GB SSD storage, 32 cores). The pipeline aligns the data to hg19-based references, predicts MHC haplotypes using PHLAT, calls mutations using 2 callers (MuTect and RADIA) and annotates them using SnpEff, then predicts MHC:peptide binding using the IEDB suite of tools before running an in-house rank boosting algorithm on the final calls.

To optimize time taken, The pipeline is written such that mutations are called on a per-chromosome basis from the whole-exome bams and are merged into a complete vcf. Running mutect in parallel on whole exome bams requires each mutect job to download the complete Tumor and Normal Bams to their working directories – An operation that quickly fills the disk and limits the parallelizability of jobs. The script was run in Toil, with and without caching, and Figure 2 shows that the workflow finishes faster in the cached case while using less disk on average than the uncached

run. We believe that benefits of caching arising from file transfers will be much higher on magnetic disk-based storage systems as compared to the SSD systems we tested this on.

26.3 Toil support for Common Workflow Language

The CWL document and input document are loaded using the 'cwltool.load_tool' module. This performs normalization and URI expansion (for example, relative file references are turned into absolute file URIs), validates the document against the CWL schema, initializes Python objects corresponding to major document elements (command line tools, workflows, workflow steps), and performs static type checking that sources and sinks have compatible types.

Input files referenced by the CWL document and input document are imported into the Toil file store. CWL documents may use any URI scheme supported by Toil file store, including local files and object storage.

The 'location' field of File references are updated to reflect the import token returned by the Toil file store.

For directory inputs, the directory listing is stored in Directory object. Each individual files is imported into Toil file store.

An initial workflow Job is created from the toplevel CWL document. Then, control passes to the Toil engine which schedules the initial workflow job to run.

When the toplevel workflow job runs, it traverses the CWL workflow and creates a toil job for each step. The dependency graph is expressed by making downstream jobs children of upstream jobs, and initializing the child jobs with an input object containing the promises of output from upstream jobs.

Because Toil jobs have a single output, but CWL permits steps to have multiple output parameters that may feed into multiple other steps, the input to a CWLJob is expressed with an "indirect dictionary". This is a dictionary of input parameters, where each entry value is a tuple of a promise and a promise key. When the job runs, the indirect dictionary is turned into a concrete input object by resolving each promise into its actual value (which is always a dict), and then looking up the promise key to get the actual value for the the input parameter.

If a workflow step specifies a scatter, then a scatter job is created and connected into the workflow graph as described above. When the scatter step runs, it creates child jobs for each parameterizations of the scatter. A gather job is added as a follow-on to gather the outputs into arrays.

When running a command line tool, it first creates output and temporary directories under the Toil local temp dir. It runs the command line tool using the single_job_executor from CWLTool, providing a Toil-specific constructor for filesystem access, and overriding the default PathMapper to use ToilPathMapper.

The ToilPathMapper keeps track of a file's symbolic identifier (the Toil FileID), its local path on the host (the value returned by readGlobalFile) and the the location of the file inside the Docker container.

After executing single_job_executor from CWLTool, it gets back the output object and status. If the underlying job failed, raise an exception. Files from the output object are added to the file store using writeGlobalFile and the 'location' field of File references are updated to reflect the token returned by the Toil file store.

When the workflow completes, it returns an indirect dictionary linking to the outputs of the job steps that contribute to the final output. This is the value returned by toil.start() or toil.restart(). This is resolved to get the final output object. The files in this object are exported from the file store to 'outdir' on the host file system, and the 'location' field of File references are updated to reflect the final exported location of the output files.

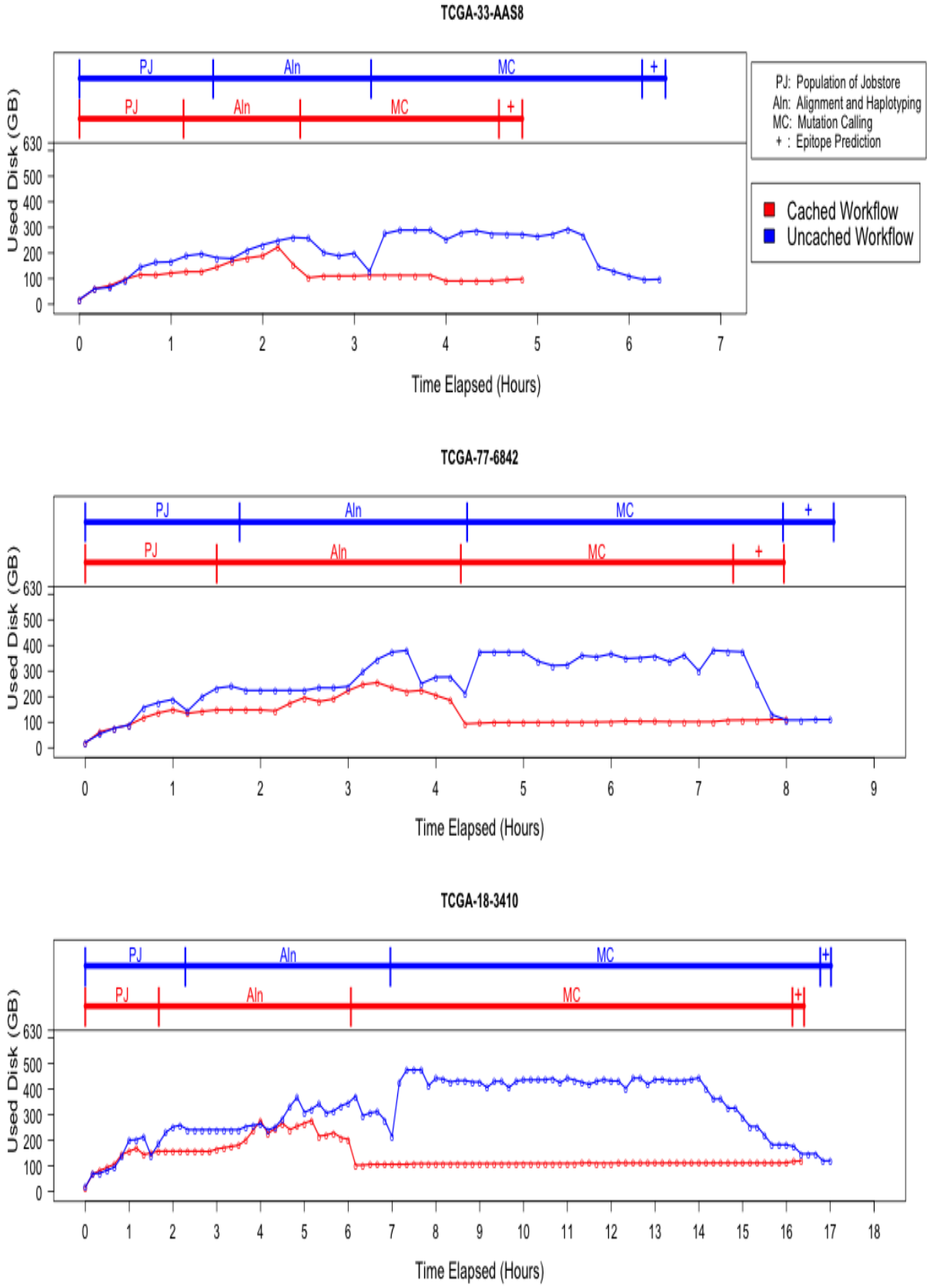


Fig. 2: Figure 2: Efficiency gain from caching. The lower half of each plot describes the disk used by the pipeline recorded every 10 minutes over the duration of the pipeline, and the upper half shows the corresponding stage of the pipeline that is being processed. Since jobs requesting the same file shared the same inode, the effective load on the disk is considerably lower than in the uncached case where every job downloads a personal copy of every file it needs. We see that in all cases, the uncached run uses almost 300-400GB more that the cached run in the resource heavy

CHAPTER

TWENTYSEVEN

MINIMUM AWS IAM PERMISSIONS

Toil requires at least the following permissions in an IAM role to operate on a cluster. These are added by default when launching a cluster. However, ensure that they are present if creating a custom IAM role when *launching a cluster* with the `--awsEc2ProfileArn` parameter.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ec2:*",
                "s3:*",
                "sdb:*",
                "iam:PassRole"
            ],
            "Resource": "*"
        }
    ]
}
```

CHAPTER

TWENTYEIGHT

AUTO-DEPLOYMENT

If you want to run your workflow in a distributed environment, on multiple worker machines, either in the cloud or on a bare-metal cluster, your script needs to be made available to those other machines. If your script imports other modules, those modules also need to be made available on the workers. Toil can automatically do that for you, with a little help on your part. We call this feature *auto-deployment* of a workflow.

Let's first examine various scenarios of auto-deploying a workflow, which, as we'll see shortly cannot be auto-deployed. Lastly, we'll deal with the issue of declaring *Toil as a dependency* of a workflow that is packaged as a setuptools distribution.

Toil can be easily deployed to a remote host. First, assuming you've followed our *Preparing your AWS environment* section to install Toil and use it to create a remote leader node on (in this example) AWS, you can now log into this into using *Ssh-Cluster Command* and once on the remote host, create and activate a virtualenv (noting to make sure to use the `--system-site-packages` option!):

```
$ virtualenv --system-site-packages venv
$ . venv/bin/activate
```

Note the `--system-site-packages` option, which ensures that globally-installed packages are accessible inside the virtualenv. Do not (re)install Toil after this! The `--system-site-packages` option has already transferred Toil and the dependencies from your local installation of Toil for you.

From here, you can install a project and its dependencies:

```
$ tree
.
├── util
│   ├── __init__.py
│   └── sort
│       ├── __init__.py
│       └── quick.py
└── workflow
    ├── __init__.py
    └── main.py

3 directories, 5 files
$ pip install matplotlib
$ cp -R workflow util venv/lib/python2.7/site-packages
```

Ideally, your project would have a `setup.py` file (see setuptools) which streamlines the installation process:

```
$ tree
.
```

(continues on next page)

(continued from previous page)

```
├── util
│   ├── __init__.py
│   └── sort
│       ├── __init__.py
│       └── quick.py
├── workflow
│   ├── __init__.py
│   └── main.py
└── setup.py

3 directories, 6 files
$ pip install .
```

Or, if your project has been published to PyPI:

```
$ pip install my-project
```

In each case, we have created a virtualenv with the `--system-site-packages` flag in the `venv` subdirectory then installed the `matplotlib` distribution from PyPI along with the two packages that our project consists of. (Again, both Python and Toil are assumed to be present on the leader and all worker nodes.)

We can now run our workflow:

```
$ python main.py --batchSystem=mesos ...
```

Important: If workflow's external dependencies contain native code (i.e. are not pure Python) then they must be manually installed on each worker.

Warning: Neither `python setup.py develop` nor `pip install -e .` can be used in this process as, instead of copying the source files, they create `.egg-link` files that Toil can't auto-deploy. Similarly, `python setup.py install` doesn't work either as it installs the project as a Python `.egg` which is also not currently supported by Toil (though it could be in the future).

Also note that using the `--single-version-externally-managed` flag with `setup.py` will prevent the installation of your package as an `.egg`. It will also disable the automatic installation of your project's dependencies.

28.1 Auto Deployment with Sibling Modules

This scenario applies if the user script imports modules that are its siblings:

```
$ cd my_project
$ ls
userScript.py utilities.py
$ ./userScript.py --batchSystem=mesos ...
```

Here `userScript.py` imports additional functionality from `utilities.py`. Toil detects that `userScript.py` has sibling modules and copies them to the workers, alongside the user script. Note that sibling modules will be auto-deployed regardless of whether they are actually imported by the user script–all .py files residing in the same directory as the user script will automatically be auto-deployed.

Sibling modules are a suitable method of organizing the source code of reasonably complicated workflows.

28.2 Auto-Deploying a Package Hierarchy

Recall that in Python, a package is a directory containing one or more `.py` files—one of which must be called `__init__.py`—and optionally other packages. For more involved workflows that contain a significant amount of code, this is the recommended way of organizing the source code. Because we use a package hierarchy, we can't really refer to the user script as such, we call it the user *module* instead. It is merely one of the modules in the package hierarchy. We need to inform Toil that we want to use a package hierarchy by invoking Python's `-m` option. That enables Toil to identify the entire set of modules belonging to the workflow and copy all of them to each worker. Note that while using the `-m` option is optional in the scenarios above, it is mandatory in this one.

The following shell session illustrates this:

```
$ cd my_project
$ tree
.
├── utils
│   ├── __init__.py
│   └── sort
│       ├── __init__.py
│       └── quick.py
└── workflow
    ├── __init__.py
    └── main.py

3 directories, 5 files
$ python -m workflow.main --batchSystem=mesos ...
```

Here the user module `main.py` does not reside in the current directory, but is part of a package called `util`, in a subdirectory of the current directory. Additional functionality is in a separate module called `util.sort.quick` which corresponds to `util/sort/quick.py`. Because we invoke the user module via `python -m workflow.main`, Toil can determine the root directory of the hierarchy–`my_project` in this case–and copy all Python modules underneath it to each worker. The `-m` option is documented here

When `-m` is passed, Python adds the current working directory to `sys.path`, the list of root directories to be considered when resolving a module name like `workflow.main`. Without that added convenience we'd have to run the workflow as `PYTHONPATH="$PWD" python -m workflow.main`. This also means that Toil can detect the root directory of the user module's package hierarchy even if it isn't the current working directory. In other words we could do this:

```
$ cd my_project
$ export PYTHONPATH="$PWD"
$ cd /some/other/dir
$ python -m workflow.main --batchSystem=mesos ...
```

Also note that the root directory itself must not be package, i.e. must not contain an `__init__.py`.

28.3 Relying on Shared Filesystems

Bare-metal clusters typically mount a shared file system like NFS on each node. If every node has that file system mounted at the same path, you can place your project on that shared filesystem and run your user script from there. Additionally, you can clone the Toil source tree into a directory on that shared file system and you won't even need to install Toil on every worker. Be sure to add both your project directory and the Toil clone to `PYTHONPATH`. Toil replicates `PYTHONPATH` from the leader to every worker.

Using a shared filesystem

Toil currently only supports a `tempdir` set to a local, non-shared directory.

28.3.1 Toil Appliance

The term Toil Appliance refers to the Mesos Docker image that Toil uses to simulate the machines in the virtual mesos cluster. It's easily deployed, only needs Docker, and allows for workflows to be run in single-machine mode and for clusters of VMs to be provisioned. To specify a different image, see the Toil *Environment Variables* section. For more information on the Toil Appliance, see the *Running in AWS* section.

CHAPTER

TWENTYNINE

ENVIRONMENT VARIABLES

There are several environment variables that affect the way Toil runs.

TOIL_CHECK_ENV	A flag that determines whether Toil will try to refer back to a Python virtual envir
TOIL_WORKDIR	An absolute path to a directory where Toil will write its temporary files. This dire
TOIL_WORKDIR_OVERRIDE	An absolute path to a directory where Toil will write its temporary files. This ove
TOIL_COORDINATION_DIR	An absolute path to a directory where Toil will write its lock files. This directory
TOIL_COORDINATION_DIR_OVERRIDE	An absolute path to a directory where Toil will write its lock files. This overrides
TOIL_BATCH_LOGS_DIR	A directory to save batch system logs into, where the leader can access them. The
TOIL_KUBERNETES_HOST_PATH	A path on Kubernetes hosts that will be mounted as the Toil work directory in the
TOIL_KUBERNETES_OWNER	A name prefix for easy identification of Kubernetes jobs. If not set, Toil will use t
TOIL_KUBERNETES_SERVICE_ACCOUNT	A service account name to apply when creating Kubernetes pods.
TOIL_KUBERNETES_POD_TIMEOUT	Seconds to wait for a scheduled Kubernetes pod to start running.
KUBE_WATCH_ENABLED	A boolean variable that allows for users to utilize kubernetes watch stream feature
TOIL_TES_ENDPOINT	URL to the TES server to run against when using the `tes` batch system.
TOIL_TES_USER	Username to use with HTTP Basic Authentication to log into the TES server.
TOIL_TES_PASSWORD	Password to use with HTTP Basic Authentication to log into the TES server.
TOIL_TES_BEARER_TOKEN	Token to use to authenticate to the TES server.
TOIL_APPLIANCE_SELF	The fully qualified reference for the Toil Appliance you wish to use, in the form R
TOIL_DOCKER_REGISTRY	The URL of the registry of the Toil Appliance image you wish to use. Docker wil
TOIL_DOCKER_NAME	The name of the Toil Appliance image you wish to use. Generally this is simply t
TOIL_AWS_SECRET_NAME	For the Kubernetes batch system, the name of a Kubernetes secret which contains
TOIL_AWS_ZONE	Zone to use when using AWS. Also determines region. Overrides TOIL_AWS_R
TOIL_AWS_REGION	Region to use when using AWS.
TOIL_AWS_AMI	ID of the AMI to use in node provisioning. If in doubt, don't set this variable.
TOIL_AWS_NODE_DEBUG	Determines whether to preserve nodes that have failed health checks. If set to `Tru`
TOIL_AWS_BATCH_QUEUE	Name or ARN of an AWS Batch Queue to use with the AWS Batch batch system.
TOIL_AWS_BATCH_JOB_ROLE_ARN	ARN of an IAM role to run AWS Batch jobs as with the AWS Batch batch system
TOIL_GOOGLE_PROJECTID	The Google project ID to use when generating Google job store names for tests o
TOIL_SLURM_ARGS	Arguments for sbatch for the slurm batch system. Do not pass CPU or memory sp
TOIL_SLURM_PE	Name of the slurm partition to use for parallel jobs. There is no default value for
TOIL_GRIDENGINE_ARGS	Arguments for qsub for the gridengine batch system. Do not pass CPU or memor
TOIL_GRIDENGINE_PE	Parallel environment arguments for qsub and for the gridengine batch system. Th
TOIL_TORQUE_ARGS	Arguments for qsub for the Torque batch system. Do not pass CPU or memory sp
TOIL_TORQUE_REQS	Arguments for the resource requirements for Torque batch system. Do not pass C
TOIL_LSF_ARGS	Additional arguments for the LSF's bsub command. Instead, define extra paramet
TOIL_HTCONDOR_PARAMS	Additional parameters to include in the HTCondor submit file passed to condor_s
TOIL_CUSTOM_DOCKER_INIT_COMMAND	Any custom bash command to run in the Toil docker container prior to running th
TOIL_CUSTOM_INIT_COMMAND	Any custom bash command to run prior to starting the Toil appliance. Can be use

TOIL_S3_HOST	the IP address or hostname to use for connecting to S3. Example: `TOIL_S3_HOST`
TOIL_S3_PORT	a port number to use for connecting to S3. Example: `TOIL_S3_PORT=9001`
TOIL_S3_USE_SSL	enable or disable the usage of SSL for connecting to S3 (`True` by default). Examp
TOIL_WES_BROKER_URL	An optional broker URL to use to communicate between the WES server and Cel
TOIL_WES_JOB_STORE_TYPE	Type of job store to use by default for workflows run via the WES server. Can be
TOIL_OWNER_TAG	This will tag cloud resources with a tag reading: "Owner: $TOIL_OWNER_TAG
TOIL_AWS_PROFILE	The name of an AWS profile to run TOIL with.
TOIL_AWS_TAGS	This will tag cloud resources with any arbitrary tags given in a JSON format. The
SINGULARITY_DOCKER_HUB_MIRROR	An http or https URL for the Singularity wrapper in the Toil Docker container to
OMP_NUM_THREADS	The number of cores set for OpenMP applications in the workers. If not set, Toil
GUNICORN_CMD_ARGS	Specify additional Gunicorn configurations for the Toil WES server. See Gunicor

CHAPTER

THIRTY

API REFERENCE

This page contains auto-generated API reference documentation[1].

30.1 toil

30.1.1 Subpackages

toil.batchSystems

Subpackages

toil.batchSystems.mesos

Subpackages

toil.batchSystems.mesos.test

Package Contents

Classes

ExceptionalThread	A thread whose join() method re-raises exceptions raised during run(). While join() is
MesosTestSupport	Mixin for test cases that need a running Mesos master and agent on the local host.

[1] Created with sphinx-autoapi

Functions

retry([intervals, infinite_retries, errors, ...])	Retry a function if it fails with any Exception defined in "errors".
cpu_count()	Get the rounded-up integer number of whole CPUs available.

Attributes

log

toil.batchSystems.mesos.test.**retry**(*intervals=None*, *infinite_retries=False*, *errors=None*, *log_message=None*, *prepare=None*)

Retry a function if it fails with any Exception defined in "errors".

Does so every x seconds, where x is defined by a list of numbers (ints or floats) in "intervals". Also accepts ErrorCondition events for more detailed retry attempts.

Parameters

- **intervals** (*Optional[List]*) – A list of times in seconds we keep retrying until returning failure. Defaults to retrying with the following exponential back-off before failing: 1s, 1s, 2s, 4s, 8s, 16s
- **infinite_retries** (*bool*) – If this is True, reset the intervals when they run out. Defaults to: False.
- **errors** (*Optional[Sequence[Union[ErrorCondition, Type[Exception]]]]*) – A list of exceptions OR ErrorCondition objects to catch and retry on. ErrorCondition objects describe more detailed error event conditions than a plain error. An ErrorCondition specifies: - Exception (required) - Error codes that must match to be retried (optional; defaults to not checking) - A string that must be in the error message to be retried (optional; defaults to not checking) - A bool that can be set to False to always error on this condition.

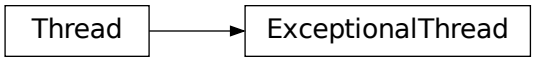
  If not specified, this will default to a generic Exception.
- **log_message** (*Optional[Tuple[Callable, str]]*) – Optional tuple of ("log/print function()", "message string") that will precede each attempt.
- **prepare** (*Optional[List[Callable]]*) – Optional list of functions to call, with the function's arguments, between retries, to reset state.

Returns
The result of the wrapped function or raise.

Return type
Callable[[Any], Any]

class toil.batchSystems.mesos.test.**ExceptionalThread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs=None*, *, *daemon=None*)

Bases: threading.Thread



A thread whose join() method re-raises exceptions raised during run(). While join() is idempotent, the exception is only during the first invocation of join() that successfully joined the thread. If join() times out, no exception will be re reraised even though an exception might already have occured in run().

When subclassing this thread, override tryRun() instead of run().

```
>>> def f():
...     assert 0
>>> t = ExceptionalThread(target=f)
>>> t.start()
>>> t.join()
Traceback (most recent call last):
...
AssertionError
```

```
>>> class MyThread(ExceptionalThread):
...     def tryRun( self ):
...         assert 0
>>> t = MyThread()
>>> t.start()
>>> t.join()
Traceback (most recent call last):
...
AssertionError
```

exc_info

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

Return type
None

tryRun()

Return type
None

join(**args*, ***kwargs*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

Parameters

- **args** (*Optional[float]*) –
- **kwargs** (*Optional[float]*) –

Return type
None

toil.batchSystems.mesos.test.**cpu_count**()

Get the rounded-up integer number of whole CPUs available.

Counts hyperthreads as CPUs.

Uses the system's actual CPU count, or the current v1 cgroup's quota per period, if the quota is set.

Ignores the cgroup's cpu shares value, because it's extremely difficult to interpret. See https://github.com/kubernetes/kubernetes/issues/81021.

Caches result for efficiency.

Returns
Integer count of available CPUs, minimum 1.

Return type
int

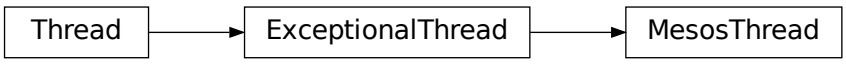
toil.batchSystems.mesos.test.**log**

class toil.batchSystems.mesos.test.**MesosTestSupport**

Mixin for test cases that need a running Mesos master and agent on the local host.

class MesosThread(*numCores*)

Bases: *toil.lib.threading.ExceptionalThread*



A thread whose join() method re-raises exceptions raised during run(). While join() is idempotent, the exception is only during the first invocation of join() that successfully joined the thread. If join() times out, no exception will be re reraised even though an exception might already have occured in run().

When subclassing this thread, override tryRun() instead of run().

```
>>> def f():
...     assert 0
>>> t = ExceptionalThread(target=f)
>>> t.start()
>>> t.join()
Traceback (most recent call last):
...
AssertionError
```

```
>>> class MyThread(ExceptionalThread):
...     def tryRun( self ):
...         assert 0
>>> t = MyThread()
>>> t.start()
>>> t.join()
Traceback (most recent call last):
...
AssertionError
```

lock

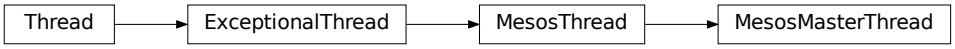
abstract mesosCommand()

tryRun()

findMesosBinary(*names*)

class MesosMasterThread(*numCores*)

Bases: *MesosTestSupport.MesosThread*



A thread whose join() method re-raises exceptions raised during run(). While join() is idempotent, the exception is only during the first invocation of join() that successfully joined the thread. If join() times out, no exception will be re reraised even though an exception might already have occured in run().

When subclassing this thread, override tryRun() instead of run().

```
>>> def f():
...     assert 0
>>> t = ExceptionalThread(target=f)
>>> t.start()
>>> t.join()
Traceback (most recent call last):
...
AssertionError
```

```
>>> class MyThread(ExceptionalThread):
...     def tryRun( self ):
```

(continues on next page)

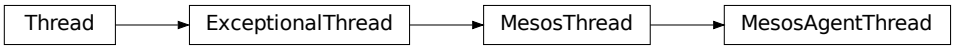
(continued from previous page)

```
...         assert 0
>>> t = MyThread()
>>> t.start()
>>> t.join()
Traceback (most recent call last):
...
AssertionError
```

mesosCommand()

class MesosAgentThread(*numCores*)

Bases: *MesosTestSupport.MesosThread*



A thread whose join() method re-raises exceptions raised during run(). While join() is idempotent, the exception is only during the first invocation of join() that successfully joined the thread. If join() times out, no exception will be re reraised even though an exception might already have occured in run().

When subclassing this thread, override tryRun() instead of run().

```
>>> def f():
...     assert 0
>>> t = ExceptionalThread(target=f)
>>> t.start()
>>> t.join()
Traceback (most recent call last):
...
AssertionError
```

```
>>> class MyThread(ExceptionalThread):
...     def tryRun( self ):
...         assert 0
>>> t = MyThread()
>>> t.start()
>>> t.join()
Traceback (most recent call last):
...
AssertionError
```

mesosCommand()

wait_for_master()

Submodules

toil.batchSystems.mesos.batchSystem

Module Contents

Classes

MesosBatchSystem	A Toil batch system implementation that uses Apache Mesos to distribute toil jobs as Mesos

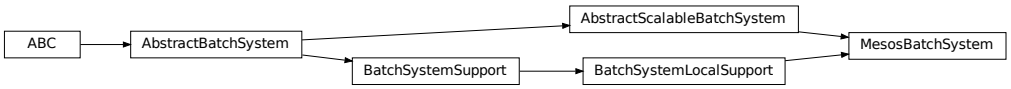
Attributes

log

toil.batchSystems.mesos.batchSystem.**log**

class toil.batchSystems.mesos.batchSystem.**MesosBatchSystem**(*config*, *maxCores*, *maxMemory*, *maxDisk*)

Bases: *toil.batchSystems.local_support.BatchSystemLocalSupport*, *toil.batchSystems.abstractBatchSystem.AbstractScalableBatchSystem*, pymesos.Scheduler



A Toil batch system implementation that uses Apache Mesos to distribute toil jobs as Mesos tasks over a cluster of agent nodes. A Mesos framework consists of a scheduler and an executor. This class acts as the scheduler and is typically run on the master node that also runs the Mesos master process with which the scheduler communicates via a driver component. The executor is implemented in a separate class. It is run on each agent node and communicates with the Mesos agent process via another driver object. The scheduler may also be run on a separate node from the master, which we then call somewhat ambiguously the driver node.

class ExecutorInfo(*nodeAddress*, *agentId*, *nodeInfo*, *lastSeen*)

userScript

Type
toil.resource.Resource

classmethod supportsAutoDeployment()

Whether this batch system supports auto-deployment of the user script itself.

If it does, the *setUserScript()* can be invoked to set the resource object representing the user script.

Note to implementors: If your implementation returns True here, it should also override

classmethod supportsWorkerCleanup()

Indicates whether this batch system invokes `BatchSystemSupport.workerCleanup()` after the last job for a particular workflow invocation finishes. Note that the term *worker* refers to an entire node, not just a worker process. A worker process may run more than one job sequentially, and more than one concurrent worker process may exist on a worker node, for the same workflow. The batch system is said to *shut down* after the last worker process terminates.

setUserScript(*userScript*)

Set the user script for this workflow. This method must be called before the first job is issued to this batch system, and only if *supportsAutoDeployment()* returns True, otherwise it will raise an exception.

Parameters
userScript – the resource object representing the user script or module and the modules it depends on.

ignoreNode(*nodeAddress*)

Stop sending jobs to this node. Used in autoscaling when the autoscaler is ready to terminate a node, but jobs are still running. This allows the node to be terminated after the current jobs have finished.

Parameters
nodeAddress – IP address of node to ignore.

unignoreNode(*nodeAddress*)

Stop ignoring this address, presumably after a node with this address has been terminated. This allows for the possibility of a new node having the same address as a terminated one.

issueBatchJob(*jobNode*, *job_environment=None*)

Issues the following command returning a unique jobID. Command is the string to run, memory is an int giving the number of bytes the job needs to run in and cores is the number of cpus needed for the job and error-file is the path of the file to place any std-err/std-out in.

Parameters

- **jobNode** (*toil.job.JobDescription*) –
- **job_environment** (*Optional[Dict[str, str]]*) –

killBatchJobs(*jobIDs*)

Kills the given job IDs. After returning, the killed jobs will not appear in the results of getRunningBatchJobIDs. The killed job will not be returned from getUpdatedBatchJob.

Parameters
jobIDs – list of IDs of jobs to kill

getIssuedBatchJobIDs()

Gets all currently issued jobs

Returns
A list of jobs (as jobIDs) currently issued (may be running, or may be waiting to be run). Despite the result being a list, the ordering should not be depended upon.

getRunningBatchJobIDs()

Gets a map of jobs as jobIDs that are currently running (not just waiting) and how long they have been running, in seconds.

Returns
dictionary with currently running jobID keys and how many seconds they have been running as the value

getUpdatedBatchJob(*maxWait*)

Returns information about job that has updated its status (i.e. ceased running, either successfully or with an error). Each such job will be returned exactly once.

Does not return info for jobs killed by killBatchJobs, although they may cause None to be returned earlier than maxWait.

Parameters
maxWait – the number of seconds to block, waiting for a result

Returns
If a result is available, returns UpdatedBatchJobInfo. Otherwise it returns None. wallTime is the number of seconds (a strictly positive float) in wall-clock time the job ran for, or None if this batch system does not support tracking wall time.

nodeInUse(*nodeIP*)

Can be used to determine if a worker node is running any tasks. If the node is doesn't exist, this function should simply return False.

Parameters
nodeIP (*str*) – The worker nodes private IP address

Returns
True if the worker node has been issued any tasks, else False

Return type
bool

getWaitDuration()

Gets the period of time to wait (floating point, in seconds) between checking for missing/overlong jobs.

shutdown()

Called at the completion of a toil invocation. Should cleanly terminate all worker threads.

Return type
None

registered(*driver*, *frameworkId*, *masterInfo*)

Invoked when the scheduler successfully registers with a Mesos master

resourceOffers(*driver*, *offers*)

Invoked when resources have been offered to this framework.

statusUpdate(*driver*, *update*)

Invoked when the status of a task has changed (e.g., a agent is lost and so the task is lost, a task finishes and an executor sends a status update saying so, etc). Note that returning from this callback _acknowledges_ receipt of this status update! If for whatever reason the scheduler aborts during this callback (or the process exits) another status update will be delivered (note, however, that this is currently not true if the agent sending the status update is lost/fails during that time).

frameworkMessage(*driver*, *executorId*, *agentId*, *message*)

Invoked when an executor sends a message.

getNodes(*preemptible=None*, *timeout=None*)

Return all nodes that match:

- preemptible status (None includes all)
- timeout period (seen within the last # seconds, or None for all)

Parameters

- **preemptible** (*Optional[bool]*) –
- **timeout** (*Optional[int]*) –

Return type
Dict[str, *toil.batchSystems.abstractBatchSystem.NodeInfo*]

reregistered(*driver*, *masterInfo*)

Invoked when the scheduler re-registers with a newly elected Mesos master.

executorLost(*driver*, *executorId*, *agentId*, *status*)

Invoked when an executor has exited/terminated abnormally.

classmethod get_default_mesos_endpoint()

Get the default IP/hostname and port that we will look for Mesos at.

Return type
str

classmethod add_options(*parser*)

If this batch system provides any command line options, add them to the given parser.

Parameters
parser (*Union[argparse.ArgumentParser, argparse._ArgumentGroup]*) –

Return type
None

classmethod setOptions(*setOption*)

Process command line or configuration options relevant to this batch system.

Parameters
setOption (toil.batchSystems.options.OptionSetter) – A function with signature setOption(option_name, parsing_function=None, check_function=None, default=None, env=None) returning nothing, used to update run configuration as a side effect.

toil.batchSystems.mesos.conftest

Module Contents

toil.batchSystems.mesos.conftest.**collect_ignore = []**

toil.batchSystems.mesos.executor

Module Contents

Classes

MesosExecutor	Part of Toil's Mesos framework, runs on a Mesos agent. A Toil job is passed to it via the

Functions

main()

Attributes

log

toil.batchSystems.mesos.executor.**log**

class toil.batchSystems.mesos.executor.**MesosExecutor**

Bases: pymesos.Executor

Part of Toil's Mesos framework, runs on a Mesos agent. A Toil job is passed to it via the task.data field, and launched via call(toil.command).

registered(*driver*, *executorInfo*, *frameworkInfo*, *agentInfo*)

Invoked once the executor driver has been able to successfully connect with Mesos.

reregistered(*driver*, *agentInfo*)

Invoked when the executor re-registers with a restarted agent.

disconnected(*driver*)

Invoked when the executor becomes "disconnected" from the agent (e.g., the agent is being restarted due to an upgrade).

killTask(*driver*, *taskId*)

Kill parent task process and all its spawned children

shutdown(*driver*)

error(*driver*, *message*)

Invoked when a fatal error has occurred with the executor and/or executor driver.

launchTask(*driver*, *task*)

Invoked by SchedulerDriver when a Mesos task should be launched by this executor

frameworkMessage(*driver*, *message*)

Invoked when a framework message has arrived for this executor.

toil.batchSystems.mesos.executor.**main**()

Package Contents

Classes

Shape	Represents a job or a node's "shape", in terms of the dimensions of memory, cores, disk and
JobQueue	
MesosShape	Represents a job or a node's "shape", in terms of the dimensions of memory, cores, disk and

Attributes

TaskData	
ToilJob	

class toil.batchSystems.mesos.**Shape**(*wallTime*, *memory*, *cores*, *disk*, *preemptible*)

Represents a job or a node's "shape", in terms of the dimensions of memory, cores, disk and wall-time allocation.

The wallTime attribute stores the number of seconds of a node allocation, e.g. 3600 for AWS. FIXME: and for jobs?

The memory and disk attributes store the number of bytes required by a job (or provided by a node) in RAM or on disk (SSD or HDD), respectively.

Parameters

- **wallTime** (*Union[int, float]*) –
- **memory** (*int*) –
- **cores** (*Union[int, float]*) –
- **disk** (*int*) –
- **preemptible** (*bool*) –

__eq__(*other*)

Return self==value.

Parameters
other (*Any*) –

Return type
bool

greater_than(*other*)

Parameters
other (*Any*) –

Return type
bool

__gt__(*other*)

Return self>value.

Parameters
other (*Any*) –

Return type
bool

__repr__()

Return repr(self).

Return type
str

__str__()

Return str(self).

Return type
str

__hash__()

Return hash(self).

Return type
int

toil.batchSystems.mesos.**TaskData**

class toil.batchSystems.mesos.**JobQueue**

insertJob(*job*, *jobType*)

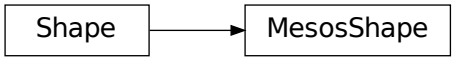
jobIDs()

nextJobOfType(*jobType*)

typeEmpty(*jobType*)

class toil.batchSystems.mesos.**MesosShape**(*wallTime*, *memory*, *cores*, *disk*, *preemptible*)

Bases: *toil.provisioners.abstractProvisioner.Shape*



Represents a job or a node's "shape", in terms of the dimensions of memory, cores, disk and wall-time allocation.

The wallTime attribute stores the number of seconds of a node allocation, e.g. 3600 for AWS. FIXME: and for jobs?

The memory and disk attributes store the number of bytes required by a job (or provided by a node) in RAM or on disk (SSD or HDD), respectively.

Parameters

- **wallTime** (*Union[int, float]*) –
- **memory** (*int*) –
- **cores** (*Union[int, float]*) –
- **disk** (*int*) –
- **preemptible** (*bool*) –

__gt__(*other*)

Inverted. Returns True if self is less than other, else returns False.

This is because jobTypes are sorted in decreasing order, and this was done to give expensive jobs priority.

toil.batchSystems.mesos.**ToilJob**

Submodules

toil.batchSystems.abstractBatchSystem

Module Contents

Classes

BatchJobExitReason	Enum where members are also (and must be) ints
UpdatedBatchJobInfo	Typed version of namedtuple.
WorkerCleanupInfo	Typed version of namedtuple.
AbstractBatchSystem	An abstract base class to represent the interface the batch system must provide to Toil.
BatchSystemSupport	Partial implementation of AbstractBatchSystem, support methods.
NodeInfo	The coresUsed attribute is a floating point value between 0 (all cores idle) and 1 (all cores
AbstractScalableBatchSystem	A batch system that supports a variable number of worker nodes. Used by :class:`toil.
ResourcePool	Represents an integral amount of a resource (such as memory bytes).
ResourceSet	Represents a collection of distinct resources (such as accelerators).

Attributes

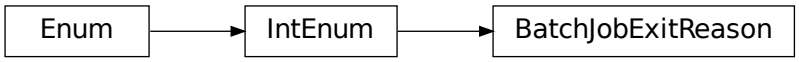
logger
EXIT_STATUS_UNAVAILABLE_VALUE

toil.batchSystems.abstractBatchSystem.**logger**

toil.batchSystems.abstractBatchSystem.**EXIT_STATUS_UNAVAILABLE_VALUE = 255**

class toil.batchSystems.abstractBatchSystem.**BatchJobExitReason**

Bases: enum.IntEnum



Enum where members are also (and must be) ints

FINISHED: int = 1

Successfully finished.

FAILED: int = 2

Job finished, but failed.

LOST: int = 3

Preemptable failure (job's executing host went away).

KILLED: int = 4

Job killed before finishing.

ERROR: int = 5

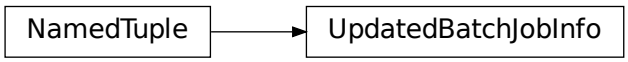
Internal error.

MEMLIMIT: int = 6

Job hit batch system imposed memory limit.

class toil.batchSystems.abstractBatchSystem.**UpdatedBatchJobInfo**

Bases: NamedTuple



Typed version of namedtuple.

Usage in Python versions >= 3.6:

```
class Employee(NamedTuple):
    name: str
    id: int
```

This is equivalent to:

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

The resulting class has extra __annotations__ and _field_types attributes, giving an ordered dict mapping field names to types. __annotations__ should be preferred, while _field_types is kept to maintain pre PEP 526 compatibility. (The field names are in the _fields attribute, which is part of the namedtuple API.) Alternative equivalent keyword syntax is also accepted:

```
Employee = NamedTuple('Employee', name=str, id=int)
```

In Python versions <= 3.5 use:

```
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

jobID: int

exitStatus: int

The exit status (integer value) of the job. 0 implies successful.

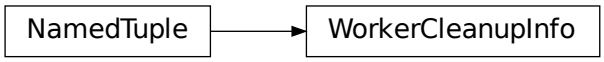
EXIT_STATUS_UNAVAILABLE_VALUE is used when the exit status is not available (e.g. job is lost).

exitReason: Optional[*BatchJobExitReason*]

wallTime: Union[float, int, None]

class toil.batchSystems.abstractBatchSystem.**WorkerCleanupInfo**

Bases: NamedTuple



Typed version of namedtuple.

Usage in Python versions >= 3.6:

```
class Employee(NamedTuple):
    name: str
    id: int
```

This is equivalent to:

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

The resulting class has extra __annotations__ and _field_types attributes, giving an ordered dict mapping field names to types. __annotations__ should be preferred, while _field_types is kept to maintain pre PEP 526 compatibility. (The field names are in the _fields attribute, which is part of the namedtuple API.) Alternative equivalent keyword syntax is also accepted:

```
Employee = NamedTuple('Employee', name=str, id=int)
```

In Python versions <= 3.5 use:

```
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

work_dir: Optional[str]

Work directory path (where the cache would go) if specified by user

coordination_dir: Optional[str]

Coordination directory path (where lock files would go) if specified by user

workflow_id: str

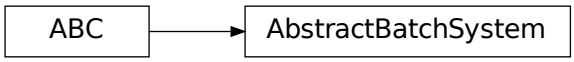
Used to identify files specific to this workflow

clean_work_dir: str

When to clean up the work and coordination directories for a job ('always', 'onSuccess', 'onError', 'never')

class toil.batchSystems.abstractBatchSystem.**AbstractBatchSystem**

Bases: abc.ABC



An abstract base class to represent the interface the batch system must provide to Toil.

abstract classmethod supportsAutoDeployment()

Whether this batch system supports auto-deployment of the user script itself.

If it does, the *setUserScript()* can be invoked to set the resource object representing the user script.

Note to implementors: If your implementation returns True here, it should also override

Return type
bool

abstract classmethod supportsWorkerCleanup()

Indicates whether this batch system invokes *BatchSystemSupport.workerCleanup()* after the last job for a particular workflow invocation finishes. Note that the term *worker* refers to an entire node, not just a worker process. A worker process may run more than one job sequentially, and more than one concurrent worker process may exist on a worker node, for the same workflow. The batch system is said to *shut down* after the last worker process terminates.

Return type
bool

abstract setUserScript(*userScript*)

Set the user script for this workflow. This method must be called before the first job is issued to this batch system, and only if *supportsAutoDeployment()* returns True, otherwise it will raise an exception.

Parameters
userScript (toil.resource.Resource) – the resource object representing the user script or module and the modules it depends on.

Return type
None

set_message_bus(*message_bus*)

Give the batch system an opportunity to connect directly to the message bus, so that it can send informational messages about the jobs it is running to other Toil components.

Parameters
message_bus (toil.bus.MessageBus) –

Return type
None

abstract issueBatchJob(*jobDesc*, *job_environment=None*)

Issues a job with the specified command to the batch system and returns a unique jobID.

Parameters

- **jobDesc** (toil.job.JobDescription) – a toil.job.JobDescription
- **job_environment** (*Optional[Dict[str, str]]*) – a collection of job-specific environment variables to be set on the worker.

Returns
a unique jobID that can be used to reference the newly issued job

Return type
int

abstract killBatchJobs(*jobIDs*)

Kills the given job IDs. After returning, the killed jobs will not appear in the results of getRunningBatchJobIDs. The killed job will not be returned from getUpdatedBatchJob.

Parameters
jobIDs (*List[int]*) – list of IDs of jobs to kill

Return type
None

abstract getIssuedBatchJobIDs()

Gets all currently issued jobs

Returns
A list of jobs (as jobIDs) currently issued (may be running, or may be waiting to be run). Despite the result being a list, the ordering should not be depended upon.

Return type
List[int]

abstract getRunningBatchJobIDs()

Gets a map of jobs as jobIDs that are currently running (not just waiting) and how long they have been running, in seconds.

Returns
dictionary with currently running jobID keys and how many seconds they have been running as the value

Return type
Dict[int, float]

abstract getUpdatedBatchJob(*maxWait*)

Returns information about job that has updated its status (i.e. ceased running, either successfully or with an error). Each such job will be returned exactly once.

Does not return info for jobs killed by killBatchJobs, although they may cause None to be returned earlier than maxWait.

Parameters
maxWait (*int*) – the number of seconds to block, waiting for a result

Returns
If a result is available, returns UpdatedBatchJobInfo. Otherwise it returns None. wallTime is the number of seconds (a strictly positive float) in wall-clock time the job ran for, or None if this batch system does not support tracking wall time.

Return type
Optional[*UpdatedBatchJobInfo*]

getSchedulingStatusMessage()

Get a log message fragment for the user about anything that might be going wrong in the batch system, if available.

If no useful message is available, return None.

This can be used to report what resource is the limiting factor when scheduling jobs, for example. If the leader thinks the workflow is stuck, the message can be displayed to the user to help them diagnose why it might be stuck.

Returns
User-directed message about scheduling state.

Return type
Optional[str]

abstract shutdown()

Called at the completion of a toil invocation. Should cleanly terminate all worker threads.

Return type
None

abstract setEnv(*name*, *value=None*)

Set an environment variable for the worker process before it is launched.

The worker process will typically inherit the environment of the machine it is running on but this method makes it possible to override specific variables in that inherited environment before the worker is launched. Note that this mechanism is different to the one used by the worker internally to set up the environment of a job. A call to this method affects all jobs issued after this method returns. Note to implementors: This means that you would typically need to copy the variables before enqueuing a job.

If no value is provided it will be looked up from the current environment.

Parameters

- **name** (*str*) –
- **value** (*Optional[str]*) –

Return type
None

classmethod add_options(*parser*)

If this batch system provides any command line options, add them to the given parser.

Parameters
parser (*Union[argparse.ArgumentParser, argparse._ArgumentGroup]*) –

Return type
None

classmethod setOptions(*setOption*)

Process command line or configuration options relevant to this batch system.

Parameters
setOption (toil.batchSystems.options.OptionSetter) – A function with signature setOption(option_name, parsing_function=None, check_function=None, default=None, env=None) returning nothing, used to update run configuration as a side effect.

Return type
None

getWorkerContexts()

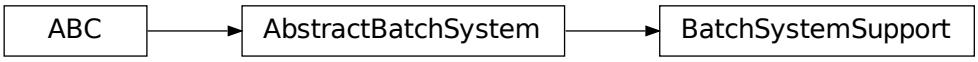
Get a list of picklable context manager objects to wrap worker work in, in order.

Can be used to ask the Toil worker to do things in-process (such as configuring environment variables, hot-deploying user scripts, or cleaning up a node) that would otherwise require a wrapping "executor" process.

Return type
List[ContextManager[Any]]

class toil.batchSystems.abstractBatchSystem.**BatchSystemSupport**(*config*, *maxCores*, *maxMemory*, *maxDisk*)

Bases: *AbstractBatchSystem*



Partial implementation of AbstractBatchSystem, support methods.

Parameters

- **config** (toil.common.Config) –
- **maxCores** (*float*) –
- **maxMemory** (*int*) –
- **maxDisk** (*int*) –

check_resource_request(*requirer*)

Check resource request is not greater than that available or allowed.

Parameters

- **requirer** (toil.job.Requirer) – Object whose requirements are being checked
- **job_name** (*str*) – Name of the job being checked, for generating a useful error report.
- **detail** (*str*) – Batch-system-specific message to include in the error.

Raises
InsufficientSystemResources – raised when a resource is requested in an amount greater than allowed

Return type
None

setEnv(*name*, *value=None*)

Set an environment variable for the worker process before it is launched. The worker process will typically inherit the environment of the machine it is running on but this method makes it possible to override specific variables in that inherited environment before the worker is launched. Note that this mechanism is different to the one used by the worker internally to set up the environment of a job. A call to this method affects all jobs issued after this method returns. Note to implementors: This means that you would typically need to copy the variables before enqueuing a job.

If no value is provided it will be looked up from the current environment.

Parameters

- **name** (*str*) – the environment variable to be set on the worker.
- **value** (*Optional[str]*) – if given, the environment variable given by name will be set to this value.

Return type
None

if None, the variable's current value will be used as the value on the worker

Raises
RuntimeError – if value is None and the name cannot be found in the environment

Parameters

- **name** (*str*) –
- **value** (*Optional[str]*) –

Return type
None

set_message_bus(*message_bus*)

Give the batch system an opportunity to connect directly to the message bus, so that it can send informational messages about the jobs it is running to other Toil components.

Parameters
message_bus (toil.bus.MessageBus) –

Return type
None

get_batch_logs_dir()

Get the directory where the backing batch system should save its logs.

Only really makes sense if the backing batch system actually saves logs to a filesystem; Kubernetes for example does not. Ought to be a directory shared between the leader and the workers, if the backing batch system writes logs onto the worker's view of the filesystem, like many HPC schedulers do.

Return type
str

format_std_out_err_path(*toil_job_id*, *cluster_job_id*, *std*)

Format path for batch system standard output/error and other files generated by the batch system itself.

Files will be written to the batch logs directory (–batchLogsDir, defaulting to the Toil work directory) with names containing both the Toil and batch system job IDs, for ease of debugging job failures.

Param
int toil_job_id : The unique id that Toil gives a job.

Param
cluster_job_id : What the cluster, for example, GridEngine, uses as its internal job id.

Param
string std : The provenance of the stream (for example: 'err' for 'stderr' or 'out' for 'stdout')

Return type
string : Formatted filename; however if self.config.noStdOutErr is true, returns '/dev/null' or equivalent.

Parameters

- **toil_job_id** (*int*) –
- **cluster_job_id** (*str*) –
- **std** (*str*) –

`format_std_out_err_glob`(*toil_job_id*)

Get a glob string that will match all file paths generated by format_std_out_err_path for a job.

Parameters
toil_job_id (*int*) –

Return type
str

`static workerCleanup`(*info*)

Cleans up the worker node on batch system shutdown. Also see `supportsWorkerCleanup()`.

Parameters
info (WorkerCleanupInfo) – A named tuple consisting of all the relevant information for cleaning up the worker.

Return type
None

`class toil.batchSystems.abstractBatchSystem.NodeInfo`(*coresUsed*, *memoryUsed*, *coresTotal*, *memoryTotal*, *requestedCores*, *requestedMemory*, *workers*)

The coresUsed attribute is a floating point value between 0 (all cores idle) and 1 (all cores busy), reflecting the CPU load of the node.

The memoryUsed attribute is a floating point value between 0 (no memory used) and 1 (all memory used), reflecting the memory pressure on the node.

The coresTotal and memoryTotal attributes are the node's resources, not just the used resources

The requestedCores and requestedMemory attributes are all the resources that Toil Jobs have reserved on the node, regardless of whether the resources are actually being used by the Jobs.

The workers attribute is an integer reflecting the number of workers currently active workers on the node.

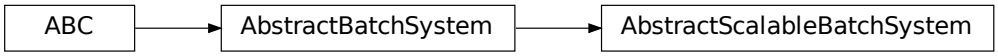
Parameters

- **coresUsed** (*float*) –
- **memoryUsed** (*float*) –
- **coresTotal** (*float*) –
- **memoryTotal** (*int*) –
- **requestedCores** (*float*) –

- **requestedMemory** (*int*) –
- **workers** (*int*) –

class toil.batchSystems.abstractBatchSystem.**AbstractScalableBatchSystem**

Bases: *AbstractBatchSystem*



A batch system that supports a variable number of worker nodes. Used by `toil.provisioners.clusterScaler.ClusterScaler` to scale the number of worker nodes in the cluster up or down depending on overall load.

abstract getNodes(*preemptible=None*, *timeout=600*)

Returns a dictionary mapping node identifiers of preemptible or non-preemptible nodes to NodeInfo objects, one for each node.

Parameters

- **preemptible** (*Optional[bool]*) – If True (False) only (non-)preemptible nodes will be returned. If None, all nodes will be returned.
- **timeout** (*int*) –

Return type
Dict[str, *NodeInfo*]

abstract nodeInUse(*nodeIP*)

Can be used to determine if a worker node is running any tasks. If the node is doesn't exist, this function should simply return False.

Parameters
nodeIP (*str*) – The worker nodes private IP address

Returns
True if the worker node has been issued any tasks, else False

Return type
bool

abstract ignoreNode(*nodeAddress*)

Stop sending jobs to this node. Used in autoscaling when the autoscaler is ready to terminate a node, but jobs are still running. This allows the node to be terminated after the current jobs have finished.

Parameters
nodeAddress (*str*) – IP address of node to ignore.

Return type
None

abstract unignoreNode(*nodeAddress*)

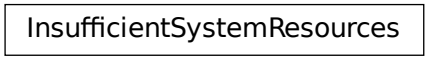
Stop ignoring this address, presumably after a node with this address has been terminated. This allows for the possibility of a new node having the same address as a terminated one.

Parameters
nodeAddress (*str*) –

Return type
None

exception toil.batchSystems.abstractBatchSystem.**InsufficientSystemResources**(*requirer*, *resource*, *available=None*, *batch_system=None*, *source=None*, *details=[]*)

Bases: Exception



Common base class for all non-exit exceptions.

Parameters

- **requirer** (toil.job.Requirer) –
- **resource** (*str*) –
- **available** (*Optional[toil.job.ParsedRequirement]*) –
- **batch_system** (*Optional[str]*) –
- **source** (*Optional[str]*) –
- **details** (*List[str]*) –

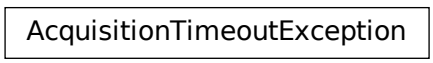
__str__()

Explain the exception.

Return type
str

exception toil.batchSystems.abstractBatchSystem.**AcquisitionTimeoutException**(*resource*, *requested*, *available*)

Bases: Exception



To be raised when a resource request times out.

Parameters

- **resource** (*str*) –
- **requested** (*Union[int, float, Set[int]]*) –
- **available** (*Union[int, float, Set[int]]*) –

class toil.batchSystems.abstractBatchSystem.**ResourcePool**(*initial_value*, *resource_type*, *timeout=5*)

Represents an integral amount of a resource (such as memory bytes). Amounts can be acquired immediately or with a timeout, and released. Provides a context manager to do something with an amount of resource acquired.

Parameters

- **initial_value** (*int*) –
- **resource_type** (*str*) –
- **timeout** (*float*) –

acquireNow(*amount*)

Reserve the given amount of the given resource. Returns True if successful and False if this is not possible immediately.

Parameters
amount (*int*) –

Return type
bool

acquire(*amount*)

Reserve the given amount of the given resource. Raises AcquisitionTimeoutException if this is not possible in under self.timeout time.

Parameters
amount (*int*) –

Return type
None

release(*amount*)

Parameters
amount (*int*) –

Return type
None

__str__()

Return str(self).

Return type
str

__repr__()

Return repr(self).

Return type
str

acquisitionOf(*amount*)

Parameters
amount (*int*) –

Return type
Iterator[None]

class toil.batchSystems.abstractBatchSystem.**ResourceSet**(*initial_value*, *resource_type*, *timeout=5*)

Represents a collection of distinct resources (such as accelerators). Subsets can be acquired immediately or with a timeout, and released. Provides a context manager to do something with a set of of resources acquired.

Parameters

- **initial_value** (*Set[int]*) –
- **resource_type** (*str*) –
- **timeout** (*float*) –

acquireNow(*subset*)

Reserve the given amount of the given resource. Returns True if successful and False if this is not possible immediately.

Parameters
subset (*Set[int]*) –

Return type
bool

acquire(*subset*)

Reserve the given amount of the given resource. Raises AcquisitionTimeoutException if this is not possible in under self.timeout time.

Parameters
subset (*Set[int]*) –

Return type
None

release(*subset*)

Parameters
subset (*Set[int]*) –

Return type
None

get_free_snapshot()

Get a snapshot of what items are free right now. May be stale as soon as you get it, but you will need some kind of hint to try and do an acquire.

Return type
Set[int]

__str__()

Return str(self).

Return type
str

__repr__()

Return repr(self).

Return type
str

acquisitionOf(*subset*)

Parameters
subset (*Set[int]*) –

Return type
Iterator[None]

toil.batchSystems.abstractGridEngineBatchSystem

Module Contents

Classes

AbstractGridEngineBatchSystem	A partial implementation of BatchSystemSupport for batch systems run on a

Attributes

logger
JobTuple

toil.batchSystems.abstractGridEngineBatchSystem.**logger**

toil.batchSystems.abstractGridEngineBatchSystem.**JobTuple**

class toil.batchSystems.abstractGridEngineBatchSystem.**AbstractGridEngineBatchSystem**(*config*, *maxCores*, *maxMemory*, *maxDisk*)

Bases: *toil.batchSystems.cleanup_support.BatchSystemCleanupSupport*

A partial implementation of BatchSystemSupport for batch systems run on a standard HPC cluster. By default auto-deployment is not implemented.

class Worker(*newJobsQueue*, *updatedJobsQueue*, *killQueue*, *killedJobsQueue*, *boss*)

Bases: threading.Thread

A class that represents a thread of control.

This class can be safely subclassed in a limited fashion. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the run() method in a subclass.

Parameters

- **newJobsQueue** (*queue.Queue*) –
- **updatedJobsQueue** (*queue.Queue*) –
- **killQueue** (*queue.Queue*) –
- **killedJobsQueue** (*queue.Queue*) –
- **boss** (AbstractGridEngineBatchSystem) –

getBatchSystemID(*jobID*)

Get batch system-specific job ID

Note: for the moment this is the only consistent way to cleanly get the batch system job ID

Parameters
jobID (*int*) – Toil BatchSystem numerical job ID
Return type
str

forgetJob(*jobID*)

Remove jobID passed

Parameters
jobID (*int*) – toil job ID
Return type
None

createJobs(*newJob*)

Create a new job with the given attributes.

Implementation-specific; called by AbstractGridEngineWorker.run()

Parameters
newJob (*JobTuple*) –
Return type
bool

killJobs()

Kill any running jobs within worker

checkOnJobs()

Check and update status of all running jobs.

Respects statePollingWait and will return cached results if not within time period to talk with the scheduler.

run()

Run any new jobs

abstract coalesce_job_exit_codes(*batch_job_id_list*)

Returns exit codes for a list of jobs.

Called by AbstractGridEngineWorker.checkOnJobs().

This is an optional part of the interface. It should raise NotImplementedError if not actually implemented for a particular scheduler.

Parameters
batch_job_id_list (*string*) – List of batch system job ID

Return type
list

abstract prepareSubmission(*cpu*, *memory*, *jobID*, *command*, *jobName*, *job_environment=None*, *gpus=None*)

Preparation in putting together a command-line string for submitting to batch system (via submitJob().)

Param
int cpu

Param
int memory

Param
int jobID: Toil job ID

Param
string subLine: the command line string to be called

Param
string jobName: the name of the Toil job, to provide metadata to batch systems if desired

Param
dict job_environment: the environment variables to be set on the worker

Return type
List[str]

Parameters
- **cpu** (*int*) –
- **memory** (*int*) –
- **jobID** (*int*) –
- **command** (*str*) –
- **jobName** (*str*) –
- **job_environment** (*Optional[Dict[str, str]]*) –
- **gpus** (*Optional[int]*) –

abstract submitJob(*subLine*)

Wrapper routine for submitting the actual command-line call, then processing the output to get the batch system job ID

Param
string subLine: the literal command line string to be called

Return type
string: batch system job ID, which will be stored internally

abstract getRunningJobIDs()

Get a list of running job IDs. Implementation-specific; called by boss AbstractGridEngineBatchSystem implementation via AbstractGridEngineBatchSystem.getRunningBatchJobIDs()

Return type
list

abstract killJob(*jobID*)

Kill specific job with the Toil job ID. Implementation-specific; called by AbstractGridEngineWorker.killJobs()

Parameters

jobID (*string*) – Toil job ID

abstract getJobExitCode(*batchJobID*)

Returns job exit code or an instance of abstractBatchSystem.BatchJobExitReason. if something else happened other than the job exiting. Implementation-specific; called by AbstractGridEngineWorker.checkOnJobs()

Parameters

batchjobID (*string*) – batch system job ID

Return type

int|toil.batchSystems.abstractBatchSystem.BatchJobExitReason: exit code int or BatchJobExitReason if something else happened other than job exiting.

classmethod supportsWorkerCleanup()

Indicates whether this batch system invokes `BatchSystemSupport.workerCleanup()` after the last job for a particular workflow invocation finishes. Note that the term *worker* refers to an entire node, not just a worker process. A worker process may run more than one job sequentially, and more than one concurrent worker process may exist on a worker node, for the same workflow. The batch system is said to *shut down* after the last worker process terminates.

classmethod supportsAutoDeployment()

Whether this batch system supports auto-deployment of the user script itself.

If it does, the `setUserScript()` can be invoked to set the resource object representing the user script.

Note to implementors: If your implementation returns True here, it should also override

issueBatchJob(*jobDesc*, *job_environment=None*)

Issues a job with the specified command to the batch system and returns a unique jobID.

Parameters

- **jobDesc** – a toil.job.JobDescription
- **job_environment** (*Optional[Dict[str, str]]*) – a collection of job-specific environment variables to be set on the worker.

Returns

a unique jobID that can be used to reference the newly issued job

killBatchJobs(*jobIDs*)

Kills the given jobs, represented as Job ids, then checks they are dead by checking they are not in the list of issued jobs.

getIssuedBatchJobIDs()

Gets the list of issued jobs

getRunningBatchJobIDs()

Retrieve running job IDs from local and batch scheduler.

Respects statePollingWait and will return cached results if not within time period to talk with the scheduler.

getUpdatedBatchJob(*maxWait*)

Returns information about job that has updated its status (i.e. ceased running, either successfully or with an error). Each such job will be returned exactly once.

Does not return info for jobs killed by killBatchJobs, although they may cause None to be returned earlier than maxWait.

Parameters
maxWait – the number of seconds to block, waiting for a result

Returns
If a result is available, returns UpdatedBatchJobInfo. Otherwise it returns None. wallTime is the number of seconds (a strictly positive float) in wall-clock time the job ran for, or None if this batch system does not support tracking wall time.

shutdown()

Signals worker to shutdown (via sentinel) then cleanly joins the thread

Return type
None

setEnv(*name*, *value=None*)

Set an environment variable for the worker process before it is launched. The worker process will typically inherit the environment of the machine it is running on but this method makes it possible to override specific variables in that inherited environment before the worker is launched. Note that this mechanism is different to the one used by the worker internally to set up the environment of a job. A call to this method affects all jobs issued after this method returns. Note to implementors: This means that you would typically need to copy the variables before enqueuing a job.

If no value is provided it will be looked up from the current environment.

Parameters

- **name** – the environment variable to be set on the worker.
- **value** – if given, the environment variable given by name will be set to this value.

if None, the variable's current value will be used as the value on the worker

Raises
RuntimeError – if value is None and the name cannot be found in the environment

classmethod getWaitDuration()

sleepSeconds(*sleeptime=1*)

Helper function to drop on all state-querying functions to avoid over-querying.

with_retries(*operation*, **args*, ***kwargs*)

Call operation with args and kwargs. If one of the calls to an SGE command fails, sleep and try again for a set number of times.

toil.batchSystems.awsBatch

Batch system for running Toil workflows on AWS Batch.

Useful with the AWS job store.

AWS Batch has no means for scheduling based on disk usage, so the backing machines need to have "enough" disk and other constraints need to guarantee that disk does not fill.

Assumes that an AWS Batch Queue name or ARN is already provided.

Handles creating and destroying a JobDefinition for the workflow run.

Additional containers should be launched with Singularity, not Docker.

Module Contents

Classes

AWSBatchBatchSystem	Adds cleanup support when the last running job leaves a node, for batch

Attributes

logger
STATE_TO_EXIT_REASON
MAX_POLL_COUNT
MIN_REQUESTABLE_MIB
MIN_REQUESTABLE_CORES

toil.batchSystems.awsBatch.**logger**

toil.batchSystems.awsBatch.**STATE_TO_EXIT_REASON: Dict[str, *toil.batchSystems.abstractBatchSystem.BatchJobExitReason*]**

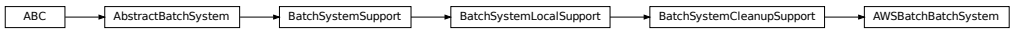
toil.batchSystems.awsBatch.**MAX_POLL_COUNT = 100**

toil.batchSystems.awsBatch.**MIN_REQUESTABLE_MIB = 4**

toil.batchSystems.awsBatch.**MIN_REQUESTABLE_CORES = 1**

class toil.batchSystems.awsBatch.**AWSBatchBatchSystem**(*config*, *maxCores*, *maxMemory*, *maxDisk*)

Bases: *toil.batchSystems.cleanup_support.BatchSystemCleanupSupport*



Adds cleanup support when the last running job leaves a node, for batch systems that can't provide it using the backing scheduler.

Parameters

- **config** (*toil.common.Config*) –
- **maxCores** (*float*) –
- **maxMemory** (*int*) –
- **maxDisk** (*int*) –

classmethod supportsAutoDeployment()

Whether this batch system supports auto-deployment of the user script itself.

If it does, the *setUserScript()* can be invoked to set the resource object representing the user script.

Note to implementors: If your implementation returns True here, it should also override

Return type
bool

setUserScript(*user_script*)

Set the user script for this workflow. This method must be called before the first job is issued to this batch system, and only if *supportsAutoDeployment()* returns True, otherwise it will raise an exception.

Parameters

- **userScript** – the resource object representing the user script or module and the modules it depends on.
- **user_script** (toil.resource.Resource) –

Return type
None

issueBatchJob(*job_desc*, *job_environment=None*)

Issues a job with the specified command to the batch system and returns a unique jobID.

Parameters

- **jobDesc** – a toil.job.JobDescription
- **job_environment** (*Optional[Dict[str, str]]*) – a collection of job-specific environment variables to be set on the worker.
- **job_desc** (toil.job.JobDescription) –

Returns
a unique jobID that can be used to reference the newly issued job

Return type
int

getUpdatedBatchJob(*maxWait*)

Returns information about job that has updated its status (i.e. ceased running, either successfully or with an error). Each such job will be returned exactly once.

Does not return info for jobs killed by killBatchJobs, although they may cause None to be returned earlier than maxWait.

Parameters
maxWait (*int*) – the number of seconds to block, waiting for a result

Returns
If a result is available, returns UpdatedBatchJobInfo. Otherwise it returns None. wallTime is the number of seconds (a strictly positive float) in wall-clock time the job ran for, or None if this batch system does not support tracking wall time.

Return type
Optional[*toil.batchSystems.abstractBatchSystem.UpdatedBatchJobInfo*]

shutdown()

Called at the completion of a toil invocation. Should cleanly terminate all worker threads.

Return type
None

`getIssuedBatchJobIDs()`

Gets all currently issued jobs

Returns
A list of jobs (as jobIDs) currently issued (may be running, or may be waiting to be run). Despite the result being a list, the ordering should not be depended upon.

Return type
List[int]

`getRunningBatchJobIDs()`

Gets a map of jobs as jobIDs that are currently running (not just waiting) and how long they have been running, in seconds.

Returns
dictionary with currently running jobID keys and how many seconds they have been running as the value

Return type
Dict[int, float]

`killBatchJobs(job_ids)`

Kills the given job IDs. After returning, the killed jobs will not appear in the results of getRunningBatchJobIDs. The killed job will not be returned from getUpdatedBatchJob.

Parameters

- **jobIDs** – list of IDs of jobs to kill
- **job_ids** (*List[int]*) –

Return type
None

`classmethod add_options(parser)`

If this batch system provides any command line options, add them to the given parser.

Parameters
parser (*Union[argparse.ArgumentParser, argparse._ArgumentGroup]*) –

Return type
None

`classmethod setOptions(setOption)`

Process command line or configuration options relevant to this batch system.

Parameters
setOption (toil.batchSystems.options.OptionSetter) – A function with signature setOption(option_name, parsing_function=None, check_function=None, default=None, env=None) returning nothing, used to update run configuration as a side effect.

Return type
None

toil.batchSystems.cleanup_support

Module Contents

Classes

BatchSystemCleanupSupport	Adds cleanup support when the last running job leaves a node, for batch
WorkerCleanupContext	Context manager used by *BatchSystemCleanupSupport* to implement

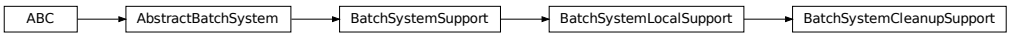
Attributes

logger

toil.batchSystems.cleanup_support.**logger**

class toil.batchSystems.cleanup_support.**BatchSystemCleanupSupport**(*config*, *maxCores*, *maxMemory*, *maxDisk*)

Bases: *toil.batchSystems.local_support.BatchSystemLocalSupport*



Adds cleanup support when the last running job leaves a node, for batch systems that can't provide it using the backing scheduler.

Parameters

- **config** (toil.common.Config) –
- **maxCores** (*float*) –
- **maxMemory** (*int*) –
- **maxDisk** (*int*) –

classmethod supportsWorkerCleanup()

Indicates whether this batch system invokes `BatchSystemSupport.workerCleanup()` after the last job for a particular workflow invocation finishes. Note that the term *worker* refers to an entire node, not just a worker process. A worker process may run more than one job sequentially, and more than one concurrent worker process may exist on a worker node, for the same workflow. The batch system is said to *shut down* after the last worker process terminates.

Return type
bool

getWorkerContexts()

Get a list of picklable context manager objects to wrap worker work in, in order.

Can be used to ask the Toil worker to do things in-process (such as configuring environment variables, hot-deploying user scripts, or cleaning up a node) that would otherwise require a wrapping "executor" process.

Return type
List[ContextManager[Any]]

class toil.batchSystems.cleanup_support.**WorkerCleanupContext**(*workerCleanupInfo*)

Context manager used by *BatchSystemCleanupSupport* to implement cleanup on a node after the last worker is done working.

Gets wrapped around the worker's work.

Parameters
workerCleanupInfo (toil.batchSystems.abstractBatchSystem.WorkerCleanupInfo) –

__enter__()

Return type
None

__exit__(*type*, *value*, *traceback*)

Parameters

- **type** (*Optional[Type[BaseException]]*) –
- **value** (*Optional[BaseException]*) –
- **traceback** (*Optional[types.TracebackType]*) –

Return type
None

toil.batchSystems.contained_executor

Executor for running inside a container.

Useful for Kubernetes and TES batch systems.

Module Contents

Functions

pack_job(job_desc[, user_script, environment])	Create a command that, when run, will execute the given job.
executor()	Main function of the _toil_contained_executor entry-point.

Attributes

logger

toil.batchSystems.contained_executor.**logger**

toil.batchSystems.contained_executor.**pack_job**(*job_desc*, *user_script=None*, *environment=None*)

Create a command that, when run, will execute the given job.

Parameters

- **job_desc** (toil.job.JobDescription) – Job description for the job to run.
- **user_script** (*Optional[*toil.resource.Resource*]*) – User script that will be loaded before the job is run.
- **environment** (*Optional[Dict[str, str]]*) – Environment variable dict that will be applied before

Return type
List[str]

the job is run.

Returns
Command to run the job, as an argument list that can be run

Parameters

- **job_desc** (toil.job.JobDescription) –
- **user_script** (*Optional[*toil.resource.Resource*]*) –
- **environment** (*Optional[Dict[str, str]]*) –

Return type
List[str]

inside the Toil appliance container.

toil.batchSystems.contained_executor.**executor**()

Main function of the _toil_contained_executor entrypoint.

Runs inside the Toil container.

Responsible for setting up the user script and running the command for the job (which may in turn invoke the Toil worker entrypoint).

Return type
None

toil.batchSystems.gridengine

Module Contents

Classes

GridEngineBatchSystem	A partial implementation of BatchSystemSupport for batch systems run on a

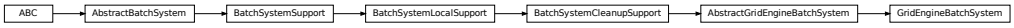
Attributes

logger

toil.batchSystems.gridengine.**logger**

class toil.batchSystems.gridengine.**GridEngineBatchSystem**(*config*, *maxCores*, *maxMemory*, *maxDisk*)

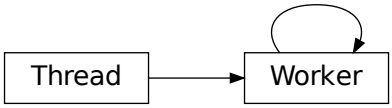
Bases: *toil.batchSystems.abstractGridEngineBatchSystem.AbstractGridEngineBatchSystem*



A partial implementation of BatchSystemSupport for batch systems run on a standard HPC cluster. By default auto-deployment is not implemented.

class Worker(*newJobsQueue*, *updatedJobsQueue*, *killQueue*, *killedJobsQueue*, *boss*)

Bases: *toil.batchSystems.abstractGridEngineBatchSystem.AbstractGridEngineBatchSystem.Worker*



Grid Engine-specific AbstractGridEngineWorker methods

Parameters

- **newJobsQueue** (*queue.Queue*) –
- **updatedJobsQueue** (*queue.Queue*) –
- **killQueue** (*queue.Queue*) –
- **killedJobsQueue** (*queue.Queue*) –

- **boss** (AbstractGridEngineBatchSystem) –

getRunningJobIDs()

Get a list of running job IDs. Implementation-specific; called by boss AbstractGridEngineBatchSystem implementation via AbstractGridEngineBatchSystem.getRunningBatchJobIDs()

Return type
list

killJob(*jobID*)

Kill specific job with the Toil job ID. Implementation-specific; called by AbstractGridEngineWorker.killJobs()

Parameters
jobID (*string*) – Toil job ID

prepareSubmission(*cpu*, *memory*, *jobID*, *command*, *jobName*, *job_environment=None*, *gpus=None*)

Preparation in putting together a command-line string for submitting to batch system (via submitJob().)

Param
int cpu
Param
int memory
Param
int jobID: Toil job ID
Param
string subLine: the command line string to be called
Param
string jobName: the name of the Toil job, to provide metadata to batch systems if desired
Param
dict job_environment: the environment variables to be set on the worker
Return type
List[str]
Parameters
- **cpu** (*int*) –
- **memory** (*int*) –
- **jobID** (*int*) –
- **command** (*str*) –
- **jobName** (*str*) –
- **job_environment** (*Optional[Dict[str, str]]*) –
- **gpus** (*Optional[int]*) –

submitJob(*subLine*)

Wrapper routine for submitting the actual command-line call, then processing the output to get the batch system job ID

Param
string subLine: the literal command line string to be called
Return type
string: batch system job ID, which will be stored internally

getJobExitCode(*sgeJobID*)

Get job exist code, checking both qstat and qacct. Return None if still running. Higher level should retry on CalledProcessErrorStderr, for the case the job has finished and qacct result is stale.

prepareQsub(*cpu*, *mem*, *jobID*, *job_environment=None*)

Parameters
- **cpu** (*int*) –
- **mem** (*int*) –
- **jobID** (*int*) –

• **job_environment** (*Optional[Dict[str, str]]*) –

Return type
List[str]

classmethod getWaitDuration()

toil.batchSystems.htcondor

Module Contents

Classes

HTCondorBatchSystem	A partial implementation of BatchSystemSupport for batch systems run on a

Attributes

logger
JobTuple
schedd_lock

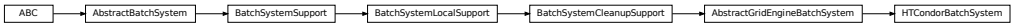
toil.batchSystems.htcondor.**logger**

toil.batchSystems.htcondor.**JobTuple**

toil.batchSystems.htcondor.**schedd_lock**

class toil.batchSystems.htcondor.**HTCondorBatchSystem**(*config*, *maxCores*, *maxMemory*, *maxDisk*)

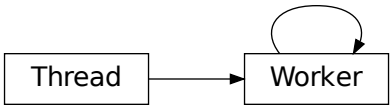
Bases: *toil.batchSystems.abstractGridEngineBatchSystem.AbstractGridEngineBatchSystem*



A partial implementation of BatchSystemSupport for batch systems run on a standard HPC cluster. By default auto-deployment is not implemented.

class Worker(*newJobsQueue*, *updatedJobsQueue*, *killQueue*, *killedJobsQueue*, *boss*)

Bases: *toil.batchSystems.abstractGridEngineBatchSystem.AbstractGridEngineBatchSystem.Worker*



A class that represents a thread of control.

This class can be safely subclassed in a limited fashion. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the run() method in a subclass.

Parameters

- **newJobsQueue** (*queue.Queue*) –
- **updatedJobsQueue** (*queue.Queue*) –
- **killQueue** (*queue.Queue*) –
- **killedJobsQueue** (*queue.Queue*) –
- **boss** (AbstractGridEngineBatchSystem) –

createJobs(*newJob*)

Create a new job with the given attributes.

Implementation-specific; called by AbstractGridEngineWorker.run()

Parameters
newJob (*JobTuple*) –
Return type
bool

prepareSubmission(*cpu*, *memory*, *disk*, *jobID*, *jobName*, *command*, *environment*)

Preparation in putting together a command-line string for submitting to batch system (via submitJob().)

Param
int cpu
Param
int memory
Param
int jobID: Toil job ID
Param
string subLine: the command line string to be called
Param
string jobName: the name of the Toil job, to provide metadata to batch systems if desired
Param
dict job_environment: the environment variables to be set on the worker
Return type
List[str]
Parameters
- **cpu** (*int*) –
- **memory** (*int*) –
- **disk** (*int*) –
- **jobID** (*int*) –
- **jobName** (*str*) –

- **command** (*str*) –
- **environment** (*Dict[str, str]*) –

submitJob(*submitObj*)

Wrapper routine for submitting the actual command-line call, then processing the output to get the batch system job ID

Param

string subLine: the literal command line string to be called

Return type

string: batch system job ID, which will be stored internally

getRunningJobIDs()

Get a list of running job IDs. Implementation-specific; called by boss AbstractGridEngineBatchSystem implementation via AbstractGridEngineBatchSystem.getRunningBatchJobIDs()

Return type

list

killJob(*jobID*)

Kill specific job with the Toil job ID. Implementation-specific; called by AbstractGridEngineWorker.killJobs()

Parameters

jobID (*string*) – Toil job ID

getJobExitCode(*batchJobID*)

Returns job exit code or an instance of abstractBatchSystem.BatchJobExitReason. if something else happened other than the job exiting. Implementation-specific; called by AbstractGridEngineWorker.checkOnJobs()

Parameters

batchjobID (*string*) – batch system job ID

Return type

int|toil.batchSystems.abstractBatchSystem.BatchJobExitReason: exit code int or BatchJobExitReason if something else happened other than job exiting.

connectSchedd()

Connect to HTCondor Schedd and yield a Schedd object.

You can only use it inside the context. Handles locking to make sure that only one thread is trying to do this at a time.

duplicate_quotes(*value*)

Escape a string by doubling up all single and double quotes.

This is used for arguments we pass to htcondor that need to be inside both double and single quote enclosures.

Parameters

value (*str*) –

Return type

str

getEnvString(*overrides*)

Build an environment string that a HTCondor Submit object can use.

For examples of valid strings, see: http://research.cs.wisc.edu/htcondor/manual/current/condor_submit.html#man-condor-submit-environment

Parameters

overrides (*Dict[str, str]*) –

Return type

str

issueBatchJob(*jobNode*, *job_environment=None*)

Issues a job with the specified command to the batch system and returns a unique jobID.

Parameters

- **jobDesc** – a toil.job.JobDescription
- **job_environment** (*Optional[Dict[str, str]]*) – a collection of job-specific environment variables to be set on the worker.

Returns

a unique jobID that can be used to reference the newly issued job

toil.batchSystems.kubernetes

Batch system for running Toil workflows on Kubernetes.

Ony useful with network-based job stores, like AWSJobStore.

Within non-privileged Kubernetes containers, additional Docker containers cannot yet be launched. That functionality will need to wait for user-mode Docker

Module Contents

Classes

KubernetesBatchSystem	Adds cleanup support when the last running job leaves a node, for batch

Functions

is_retryable_kubernetes_error(e)	A function that determines whether or not Toil should retry or stop given

Attributes

logger
retryable_kubernetes_errors
KeyValuesList

toil.batchSystems.kubernetes.**logger**

toil.batchSystems.kubernetes.**retryable_kubernetes_errors: List[Union[Type[Exception], *toil.lib.retry.ErrorCondition*]]**

toil.batchSystems.kubernetes.**is_retryable_kubernetes_error**(*e*)

A function that determines whether or not Toil should retry or stop given exceptions thrown by Kubernetes.

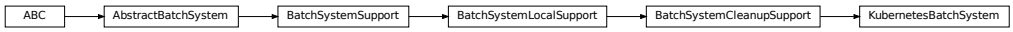
Parameters
e (*Exception*) –

Return type
bool

toil.batchSystems.kubernetes.**KeyValuesList**

class toil.batchSystems.kubernetes.**KubernetesBatchSystem**(*config*, *maxCores*, *maxMemory*, *maxDisk*)

Bases: *toil.batchSystems.cleanup_support.BatchSystemCleanupSupport*



Adds cleanup support when the last running job leaves a node, for batch systems that can't provide it using the backing scheduler.

Parameters

- **config** (toil.common.Config) –
- **maxCores** (*int*) –
- **maxMemory** (*int*) –
- **maxDisk** (*int*) –

class DecoratorWrapper(*to_wrap*, *decorator*)

Class to wrap an object so all its methods are decorated.

Parameters

- **to_wrap** (*Any*) –
- **decorator** (*Callable[[Callable[P, Any]], Callable[P, Any]]*) –

P

__getattr__(*name*)

Get a member as if we are actually the wrapped object. If it looks callable, we will decorate it.

Parameters
name (*str*) –

Return type
Any

class Placement

Internal format for pod placement constraints and preferences.

required_labels: KeyValuesList = []

Labels which are required to be present (with these values).

desired_labels: KeyValuesList = []

Labels which are optional, but preferred to be present (with these values).

prohibited_labels: KeyValuesList = []

Labels which are not allowed to be present (with these values).

tolerated_taints: KeyValuesList = []

Taints which are allowed to be present (with these values).

set_preemptible(*preemptible*)

Add constraints for a job being preemptible or not.

Preemptible jobs will be able to run on preemptible or non-preemptible nodes, and will prefer preemptible nodes if available.

Non-preemptible jobs will not be allowed to run on nodes that are marked as preemptible.

Understands the labeling scheme used by EKS, and the taint scheme used by GCE. The Toil-managed Kubernetes setup will mimic at least one of these.

Parameters
preemptible (*bool*) –
Return type
None

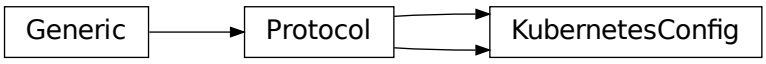
apply(*pod_spec*)

Set `affinity` and/or `tolerations` fields on pod_spec, so that it runs on the right kind of nodes for the constraints we represent.

Parameters
pod_spec (*kubernetes.client.V1PodSpec*) –
Return type
None

class KubernetesConfig

Bases: `Protocol`



Type-enforcing protocol for Toil configs that have the extra Kubernetes batch system fields.

TODO: Until MyPY lets protocols inherit form non-protocols, we will have to let the fact that this also has to be a Config just be manually enforced.

kubernetes_host_path: Optional[str]

kubernetes_owner: str

kubernetes_service_account: Optional[str]

kubernetes_pod_timeout: float

ItemT

CovItemT

P

R

OptionType

classmethod supportsAutoDeployment()

Whether this batch system supports auto-deployment of the user script itself.

If it does, the *setUserScript()* can be invoked to set the resource object representing the user script.

Note to implementors: If your implementation returns True here, it should also override

Return type
bool

setUserScript(*userScript*)

Set the user script for this workflow. This method must be called before the first job is issued to this batch system, and only if *supportsAutoDeployment()* returns True, otherwise it will raise an exception.

Parameters
userScript (*toil.resource.Resource*) – the resource object representing the user script or module and the modules it depends on.

Return type
None

issueBatchJob(*job_desc*, *job_environment=None*)

Issues a job with the specified command to the batch system and returns a unique jobID.

Parameters

- **jobDesc** – a toil.job.JobDescription
- **job_environment** (*Optional[Dict[str, str]]*) – a collection of job-specific environment variables to be set on the worker.
- **job_desc** (*toil.job.JobDescription*) –

Returns
a unique jobID that can be used to reference the newly issued job

Return type
int

getUpdatedBatchJob(*maxWait*)

Returns information about job that has updated its status (i.e. ceased running, either successfully or with an error). Each such job will be returned exactly once.

Does not return info for jobs killed by killBatchJobs, although they may cause None to be returned earlier than maxWait.

Parameters
maxWait (*float*) – the number of seconds to block, waiting for a result

Returns
If a result is available, returns UpdatedBatchJobInfo. Otherwise it returns None. wallTime is the number of seconds (a strictly positive float) in wall-clock time the job ran for, or None if this batch system does not support tracking wall time.

Return type
Optional[*toil.batchSystems.abstractBatchSystem.UpdatedBatchJobInfo*]

shutdown()

Called at the completion of a toil invocation. Should cleanly terminate all worker threads.

Return type
None

getIssuedBatchJobIDs()

Gets all currently issued jobs

Returns
A list of jobs (as jobIDs) currently issued (may be running, or may be waiting to be run). Despite the result being a list, the ordering should not be depended upon.

Return type
List[int]

getRunningBatchJobIDs()

Gets a map of jobs as jobIDs that are currently running (not just waiting) and how long they have been running, in seconds.

Returns
dictionary with currently running jobID keys and how many seconds they have been running as the value

Return type
Dict[int, float]

killBatchJobs(*jobIDs*)

Kills the given job IDs. After returning, the killed jobs will not appear in the results of getRunningBatchJobIDs. The killed job will not be returned from getUpdatedBatchJob.

Parameters
jobIDs (*List[int]*) – list of IDs of jobs to kill

Return type
None

classmethod get_default_kubernetes_owner()

Get the default Kubernetes-acceptable username string to tack onto jobs.

Return type
str

classmethod add_options(*parser*)

If this batch system provides any command line options, add them to the given parser.

Parameters
parser (*Union[argparse.ArgumentParser, argparse._ArgumentGroup]*) –

Return type
None

classmethod setOptions(*setOption*)

Process command line or configuration options relevant to this batch system.

Parameters
setOption (toil.batchSystems.options.OptionSetter) – A function with signature setOption(option_name, parsing_function=None, check_function=None, default=None, env=None) returning nothing, used to update run configuration as a side effect.

Return type
None

toil.batchSystems.local_support

Module Contents

Classes

BatchSystemLocalSupport	Adds a local queue for helper jobs, useful for CWL & others.

Attributes

logger

toil.batchSystems.local_support.**logger**

class toil.batchSystems.local_support.**BatchSystemLocalSupport**(*config*, *maxCores*, *maxMemory*, *maxDisk*)

Bases: *toil.batchSystems.abstractBatchSystem.BatchSystemSupport*

Adds a local queue for helper jobs, useful for CWL & others.

Parameters

- **config** (toil.common.Config) –
- **maxCores** (*float*) –
- **maxMemory** (*int*) –
- **maxDisk** (*int*) –

handleLocalJob(*jobDesc*)

To be called by issueBatchJobs.

Returns the jobID if the jobDesc has been submitted to the local queue, otherwise returns None

Parameters
jobDesc (toil.job.JobDescription) –

Return type
Optional[int]

killLocalJobs(*jobIDs*)

Will kill all local jobs that match the provided jobIDs.

To be called by killBatchJobs.

Parameters
jobIDs (*List[int]*) –

Return type
None

`getIssuedLocalJobIDs()`

To be called by getIssuedBatchJobIDs.

Return type
List[int]

`getRunningLocalJobIDs()`

To be called by getRunningBatchJobIDs().

Return type
Dict[int, float]

`getUpdatedLocalJob(maxWait)`

To be called by getUpdatedBatchJob().

Parameters
maxWait (*int*) –

Return type
Optional[*toil.batchSystems.abstractBatchSystem.UpdatedBatchJobInfo*]

`getNextJobID()`

Must be used to get job IDs so that the local and batch jobs do not conflict.

Return type
int

`shutdownLocal()`

To be called from shutdown().

Return type
None

toil.batchSystems.lsf

Module Contents

Classes

LSFBatchSystem	A partial implementation of BatchSystemSupport for batch systems run on a

Attributes

logger

toil.batchSystems.lsf.**logger**

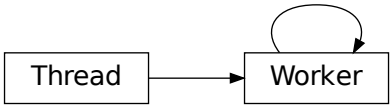
class toil.batchSystems.lsf.**LSFBatchSystem**(*config*, *maxCores*, *maxMemory*, *maxDisk*)

Bases: *toil.batchSystems.abstractGridEngineBatchSystem.AbstractGridEngineBatchSystem*

A partial implementation of BatchSystemSupport for batch systems run on a standard HPC cluster. By default auto-deployment is not implemented.

class Worker(*newJobsQueue*, *updatedJobsQueue*, *killQueue*, *killedJobsQueue*, *boss*)

Bases: *toil.batchSystems.abstractGridEngineBatchSystem.AbstractGridEngineBatchSystem.Worker*



LSF specific AbstractGridEngineWorker methods.

Parameters

- **newJobsQueue** (*queue.Queue*) –
- **updatedJobsQueue** (*queue.Queue*) –
- **killQueue** (*queue.Queue*) –
- **killedJobsQueue** (*queue.Queue*) –
- **boss** (*AbstractGridEngineBatchSystem*) –

getRunningJobIDs()

Get a list of running job IDs. Implementation-specific; called by boss AbstractGridEngineBatchSystem implementation via AbstractGridEngineBatchSystem.getRunningBatchJobIDs()

Return type

list

fallbackRunningJobIDs(*currentjobs*)

killJob(*jobID*)

Kill specific job with the Toil job ID. Implementation-specific; called by AbstractGridEngineWorker.killJobs()

Parameters
jobID (*string*) – Toil job ID

prepareSubmission(*cpu*, *memory*, *jobID*, *command*, *jobName*, *job_environment=None*, *gpus=None*)

Preparation in putting together a command-line string for submitting to batch system (via submitJob().)

Param
int cpu
Param
int memory
Param
int jobID: Toil job ID
Param
string subLine: the command line string to be called
Param
string jobName: the name of the Toil job, to provide metadata to batch systems if desired
Param
dict job_environment: the environment variables to be set on the worker
Return type
List[str]
Parameters
- **cpu** (*int*) –
- **memory** (*int*) –
- **jobID** (*int*) –
- **command** (*str*) –
- **jobName** (*str*) –
- **job_environment** (*Optional[Dict[str, str]]*) –
- **gpus** (*Optional[int]*) –

submitJob(*subLine*)

Wrapper routine for submitting the actual command-line call, then processing the output to get the batch system job ID

Param
string subLine: the literal command line string to be called
Return type
string: batch system job ID, which will be stored internally

coalesce_job_exit_codes(*batch_job_id_list*)

Returns exit codes for a list of jobs.

Called by AbstractGridEngineWorker.checkOnJobs().

This is an optional part of the interface. It should raise NotImplementedError if not actually implemented for a particular scheduler.

Parameters
batch_job_id_list (*string*) – List of batch system job ID
Return type
list

getJobExitCode(*lsfJobID*)

Returns job exit code or an instance of abstractBatchSystem.BatchJobExitReason. if something else happened other than the job exiting. Implementation-specific; called by AbstractGridEngineWorker.checkOnJobs()

Parameters
batchjobID (*string*) – batch system job ID
Return type
int|toil.batchSystems.abstractBatchSystem.BatchJobExitReason: exit code int or BatchJobExitReason if something else happened other than job exiting.

parse_bjobs_record(*bjobs_record*, *job*)

Helper functions for getJobExitCode and to parse the bjobs status record
Parameters
- **bjobs_record** (*dict*) –
- **job** (*int*) –

Return type
Union[int, None]

getJobExitCodeBACCT(*job*)

fallbackGetJobExitCode(*job*)

prepareBsub(*cpu*, *mem*, *jobID*)

Make a bsub commandline to execute.
params:
cpu: number of cores needed mem: number of bytes of memory needed jobID: ID number of the job

Parameters
- **cpu** (*int*) –
- **mem** (*int*) –
- **jobID** (*int*) –

Return type
List[str]

parseBjobs(*bjobs_output_str*)

Parse records from bjobs json type output
Params bjobs_output_str
stdout of bjobs json type output

parseMaxMem(*jobID*)

Parse the maximum memory from job.
Parameters
jobID – ID number of the job

getWaitDuration()

We give LSF a second to catch its breath (in seconds)

toil.batchSystems.lsfHelper

Module Contents

Functions

find(basedir, string)	walk basedir and return all files matching string
find_first_match(basedir, string)	return the first file that matches string starting from basedir
get_conf_file(filename, env)	
apply_conf_file(fn, conf_filename)	
per_core_reserve_from_stream(stream)	
get_lsf_units_from_stream(stream)	
tokenize_conf_stream(conf_handle)	convert the key=val pairs in a LSF config stream to tuples of tokens
apply_bparams(fn)	apply fn to each line of bparams, returning the result
apply_lsadmin(fn)	apply fn to each line of lsadmin, returning the result
get_lsf_units([resource])	check if we can find LSF_UNITS_FOR_LIMITS in lsadmin and lsf.conf
parse_mem_and_cmd_from_output(output)	Use regex to find "MAX MEM" and "Command" inside of an output.
get_lsf_version()	Get current LSF version
check_lsf_json_output_supported()	Check if the current LSF system supports bjobs json output.
parse_memory(mem)	Parse memory parameter.
per_core_reservation()	returns True if the cluster is configured for reservations to be per core,

Attributes

LSB_PARAMS_FILENAME
LSF_CONF_FILENAME
LSF_CONF_ENV
DEFAULT_LSF_UNITS
DEFAULT_RESOURCE_UNITS
LSF_JSON_OUTPUT_MIN_VERSION
logger

toil.batchSystems.lsfHelper.**LSB_PARAMS_FILENAME = 'lsb.params'**

toil.batchSystems.lsfHelper.**LSF_CONF_FILENAME = 'lsf.conf'**

toil.batchSystems.lsfHelper.**LSF_CONF_ENV = ['LSF_CONFDIR', 'LSF_ENVDIR']**

toil.batchSystems.lsfHelper.**DEFAULT_LSF_UNITS = 'KB'**

toil.batchSystems.lsfHelper.**DEFAULT_RESOURCE_UNITS = 'MB'**

toil.batchSystems.lsfHelper.**LSF_JSON_OUTPUT_MIN_VERSION = '10.1.0.2'**

toil.batchSystems.lsfHelper.**logger**

toil.batchSystems.lsfHelper.**find**(*basedir*, *string*)
walk basedir and return all files matching string

toil.batchSystems.lsfHelper.**find_first_match**(*basedir*, *string*)
return the first file that matches string starting from basedir

toil.batchSystems.lsfHelper.**get_conf_file**(*filename*, *env*)

toil.batchSystems.lsfHelper.**apply_conf_file**(*fn*, *conf_filename*)

toil.batchSystems.lsfHelper.**per_core_reserve_from_stream**(*stream*)

toil.batchSystems.lsfHelper.**get_lsf_units_from_stream**(*stream*)

toil.batchSystems.lsfHelper.**tokenize_conf_stream**(*conf_handle*)
convert the key=val pairs in a LSF config stream to tuples of tokens

toil.batchSystems.lsfHelper.**apply_bparams**(*fn*)
apply fn to each line of bparams, returning the result

toil.batchSystems.lsfHelper.**apply_lsadmin**(*fn*)
apply fn to each line of lsadmin, returning the result

toil.batchSystems.lsfHelper.**get_lsf_units**(*resource=False*)
check if we can find LSF_UNITS_FOR_LIMITS in lsadmin and lsf.conf files, preferring the value in bparams, then lsadmin, then the lsf.conf file

Parameters
resource (*bool*) –

Return type
str

toil.batchSystems.lsfHelper.**parse_mem_and_cmd_from_output**(*output*)
Use regex to find "MAX MEM" and "Command" inside of an output.

Parameters
output (*str*) –

toil.batchSystems.lsfHelper.**get_lsf_version**()
Get current LSF version

toil.batchSystems.lsfHelper.**check_lsf_json_output_supported**()
Check if the current LSF system supports bjobs json output.

toil.batchSystems.lsfHelper.**parse_memory**(*mem*)
Parse memory parameter.

Parameters
mem (*float*) –

Return type
str

toil.batchSystems.lsfHelper.**per_core_reservation**()

returns True if the cluster is configured for reservations to be per core, False if it is per job

toil.batchSystems.options

Module Contents

Classes

OptionSetter	Protocol for the setOption function we get to let us set up CLI options for

Functions

set_batchsystem_options(batch_system, set_option)	Call set_option for all the options for the given named batch system, or
add_all_batchsystem_options(parser)	
set_batchsystem_config_defaults(config)	Set default and environment-based options for builtin batch systems. This

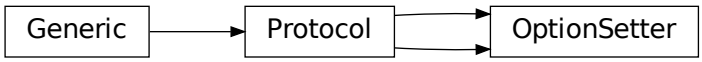
Attributes

logger

toil.batchSystems.options.**logger**

class toil.batchSystems.options.**OptionSetter**

Bases: Protocol



Protocol for the setOption function we get to let us set up CLI options for each batch system.

Actual functionality is defined in the Config class.

OptionType

__call__(*option_name*, *parsing_function=None*, *check_function=None*, *default=None*, *env=None*, *old_names=None*)

Parameters

- **option_name** (*str*) –
- **parsing_function** (*Optional[Callable[[Any], OptionType]]*) –
- **check_function** (*Optional[Callable[[OptionType], Union[None, bool]]]*) –
- **default** (*Optional[OptionType]*) –
- **env** (*Optional[List[str]]*) –
- **old_names** (*Optional[List[str]]*) –

Return type
bool

toil.batchSystems.options.**set_batchsystem_options**(*batch_system*, *set_option*)

Call set_option for all the options for the given named batch system, or all batch systems if no name is provided.

Parameters

- **batch_system** (*Optional[str]*) –
- **set_option** (OptionSetter) –

Return type
None

toil.batchSystems.options.**add_all_batchsystem_options**(*parser*)

Parameters
parser (*Union[argparse.ArgumentParser, argparse._ArgumentGroup]*) –

Return type
None

toil.batchSystems.options.**set_batchsystem_config_defaults**(*config*)

Set default and environment-based options for builtin batch systems. This is required if a Config object is not constructed from an Options object.

Return type
None

toil.batchSystems.parasol

Module Contents

Classes

ParasolBatchSystem	The interface for Parasol.

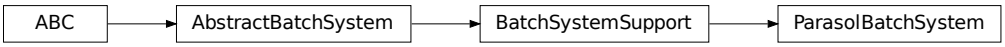
Attributes

logger

toil.batchSystems.parasol.**logger**

class toil.batchSystems.parasol.**ParasolBatchSystem**(*config*, *maxCores*, *maxMemory*, *maxDisk*)

Bases: *toil.batchSystems.abstractBatchSystem.BatchSystemSupport*



The interface for Parasol.

parasolOutputPattern

runningPattern

classmethod supportsWorkerCleanup()

Indicates whether this batch system invokes `BatchSystemSupport.workerCleanup()` after the last job for a particular workflow invocation finishes. Note that the term *worker* refers to an entire node, not just a worker process. A worker process may run more than one job sequentially, and more than one concurrent worker process may exist on a worker node, for the same workflow. The batch system is said to *shut down* after the last worker process terminates.

classmethod supportsAutoDeployment()

Whether this batch system supports auto-deployment of the user script itself.

If it does, the `setUserScript()` can be invoked to set the resource object representing the user script.

Note to implementors: If your implementation returns True here, it should also override

issueBatchJob(*jobDesc*, *job_environment=None*)

Issue parasol with job commands.

Parameters

job_environment (*Optional[Dict[str, str]]*) –

setEnv(*name*, *value=None*)

Set an environment variable for the worker process before it is launched. The worker process will typically inherit the environment of the machine it is running on but this method makes it possible to override specific variables in that inherited environment before the worker is launched. Note that this mechanism is different to the one used by the worker internally to set up the environment of a job. A call to this method affects all jobs issued after this method returns. Note to implementors: This means that you would typically need to copy the variables before enqueuing a job.

If no value is provided it will be looked up from the current environment.

Parameters

- **name** – the environment variable to be set on the worker.
- **value** – if given, the environment variable given by name will be set to this value.

if None, the variable's current value will be used as the value on the worker

Raises
RuntimeError – if value is None and the name cannot be found in the environment

killBatchJobs(*jobIDs*)

Kills the given jobs, represented as Job ids, then checks they are dead by checking they are not in the list of issued jobs.

getJobIDsForResultsFile(*resultsFile*)

Get all queued and running jobs for a results file.

getIssuedBatchJobIDs()

Gets the list of jobs issued to parasol in all results files, but not including jobs created by other users.

getRunningBatchJobIDs()

Returns map of running jobIDs and the time they have been running.

getUpdatedBatchJob(*maxWait*)

Returns information about job that has updated its status (i.e. ceased running, either successfully or with an error). Each such job will be returned exactly once.

Does not return info for jobs killed by killBatchJobs, although they may cause None to be returned earlier than maxWait.

Parameters
maxWait – the number of seconds to block, waiting for a result

Returns
If a result is available, returns UpdatedBatchJobInfo. Otherwise it returns None. wallTime is the number of seconds (a strictly positive float) in wall-clock time the job ran for, or None if this batch system does not support tracking wall time.

updatedJobWorker()

We use the parasol results to update the status of jobs, adding them to the list of updated jobs.

Results have the following structure.. (thanks Mark D!)

int status;	Job status - wait() return format. 0 is good.
char host;	Machine job ran on.
char jobId;	Job queuing system job ID
char exe;	Job executable file (no path)
int usrTicks;	'User' CPU time in ticks.
int sysTicks;	'System' CPU time in ticks.
unsigned submitTime;	Job submission time in seconds since 1/1/1970
unsigned startTime;	Job start time in seconds since 1/1/1970
unsigned endTime;	Job end time in seconds since 1/1/1970
char user;	User who ran job
char errFile;	Location of stderr file on host

Plus you finally have the command name.

shutdown()

Called at the completion of a toil invocation. Should cleanly terminate all worker threads.

Return type
None

classmethod add_options(*parser*)

If this batch system provides any command line options, add them to the given parser.

Parameters
parser (*Union[argparse.ArgumentParser, argparse._ArgumentGroup]*) –

Return type
None

classmethod setOptions(*setOption*)

Process command line or configuration options relevant to this batch system.

Parameters
setOption (toil.batchSystems.options.OptionSetter) – A function with signature setOption(option_name, parsing_function=None, check_function=None, default=None, env=None) returning nothing, used to update run configuration as a side effect.

toil.batchSystems.registry

Module Contents

Functions

aws_batch_batch_system_factory()	
gridengine_batch_system_factory()	
parasol_batch_system_factory()	
lsf_batch_system_factory()	
single_machine_batch_system_factory()	
mesos_batch_system_factory()	
slurm_batch_system_factory()	
tes_batch_system_factory()	
torque_batch_system_factory()	
htcondor_batch_system_factory()	
kubernetes_batch_system_factory()	
addBatchSystemFactory(key, batchSystemFactory)	Adds a batch system to the registry for workflow-supplied batch systems.
save_batch_system_plugin_state()	Return a snapshot of the plugin registry that can be restored to remove
restore_batch_system_plugin_state(snapshot)	Restore the batch system registry state to a snapshot from

Attributes

logger

BATCH_SYSTEM_FACTORY_REGISTRY

BATCH_SYSTEMS

DEFAULT_BATCH_SYSTEM

toil.batchSystems.registry.**logger**

toil.batchSystems.registry.**aws_batch_batch_system_factory**()

toil.batchSystems.registry.**gridengine_batch_system_factory**()

toil.batchSystems.registry.**parasol_batch_system_factory**()

toil.batchSystems.registry.**lsf_batch_system_factory**()

toil.batchSystems.registry.**single_machine_batch_system_factory**()

toil.batchSystems.registry.**mesos_batch_system_factory**()

toil.batchSystems.registry.**slurm_batch_system_factory**()

toil.batchSystems.registry.**tes_batch_system_factory**()

toil.batchSystems.registry.**torque_batch_system_factory**()

toil.batchSystems.registry.**htcondor_batch_system_factory**()

toil.batchSystems.registry.**kubernetes_batch_system_factory**()

toil.batchSystems.registry.**BATCH_SYSTEM_FACTORY_REGISTRY: Dict[str, Callable[[], Type[*toil.batchSystems.abstractBatchSystem.AbstractBatchSystem*]]]**

toil.batchSystems.registry.**BATCH_SYSTEMS**

toil.batchSystems.registry.**DEFAULT_BATCH_SYSTEM = 'single_machine'**

toil.batchSystems.registry.**addBatchSystemFactory**(*key*, *batchSystemFactory*)

Adds a batch system to the registry for workflow-supplied batch systems.

Parameters

- **key** (*str*) –
- **batchSystemFactory** (*Callable[[], Type[toil.batchSystems.abstractBatchSystem.AbstractBatchSystem]]*) –

toil.batchSystems.registry.**save_batch_system_plugin_state**()

Return a snapshot of the plugin registry that can be restored to remove added plugins. Useful for testing the plugin system in-process with other tests.

Return type

Tuple[List[str], Dict[str, Callable[[], Type[*toil.batchSystems.abstractBatchSystem.AbstractBatchSystem*]]]]

toil.batchSystems.registry.**restore_batch_system_plugin_state**(*snapshot*)

Restore the batch system registry state to a snapshot from save_batch_system_plugin_state().

Parameters

snapshot (*Tuple[List[str], Dict[str, Callable[[], Type[toil.batchSystems.abstractBatchSystem.AbstractBatchSystem]]]]*) –

toil.batchSystems.singleMachine

Module Contents

Classes

SingleMachineBatchSystem	The interface for running jobs on a single machine, runs all the jobs you
Info	Record for a running job.

Attributes

logger

toil.batchSystems.singleMachine.**logger**

class toil.batchSystems.singleMachine.**SingleMachineBatchSystem**(*config*, *maxCores*, *maxMemory*, *maxDisk*, *max_jobs=None*)

Bases: *toil.batchSystems.abstractBatchSystem.BatchSystemSupport*

The interface for running jobs on a single machine, runs all the jobs you give it as they come in, but in parallel.

Uses a single "daddy" thread to manage a fleet of child processes.

Communication with the daddy thread happens via two queues: one queue of jobs waiting to be run (the input queue), and one queue of jobs that are finished/stopped and need to be returned by getUpdatedBatchJob (the output queue).

When the batch system is shut down, the daddy thread is stopped.

If running in debug-worker mode, jobs are run immediately as they are sent to the batch system, in the sending thread, and the daddy thread is not run. But the queues are still used.

Parameters

- **config** (toil.common.Config) –
- **maxCores** (*float*) –

- **maxMemory** (*int*) –
- **maxDisk** (*int*) –
- **max_jobs** (*Optional[int]*) –

numCores

minCores = 0.1

The minimal fractional CPU. Tasks with a smaller core requirement will be rounded up to this value.

physicalMemory

classmethod supportsAutoDeployment()

Whether this batch system supports auto-deployment of the user script itself.

If it does, the `setUserScript()` can be invoked to set the resource object representing the user script.

Note to implementors: If your implementation returns True here, it should also override

classmethod supportsWorkerCleanup()

Indicates whether this batch system invokes `BatchSystemSupport.workerCleanup()` after the last job for a particular workflow invocation finishes. Note that the term *worker* refers to an entire node, not just a worker process. A worker process may run more than one job sequentially, and more than one concurrent worker process may exist on a worker node, for the same workflow. The batch system is said to *shut down* after the last worker process terminates.

daddy()

Be the "daddy" thread.

Our job is to look at jobs from the input queue.

If a job fits in the available resources, we allocate resources for it and kick off a child process.

We also check on our children.

When a child finishes, we reap it, release its resources, and put its information in the output queue.

getSchedulingStatusMessage()

Get a log message fragment for the user about anything that might be going wrong in the batch system, if available.

If no useful message is available, return None.

This can be used to report what resource is the limiting factor when scheduling jobs, for example. If the leader thinks the workflow is stuck, the message can be displayed to the user to help them diagnose why it might be stuck.

Returns
User-directed message about scheduling state.

check_resource_request(*requirer*)

Check resource request is not greater than that available or allowed.

Parameters

- **requirer** (*toil.job.Requirer*) – Object whose requirements are being checked
- **job_name** (*str*) – Name of the job being checked, for generating a useful error report.
- **detail** (*str*) – Batch-system-specific message to include in the error.

Raises
InsufficientSystemResources – raised when a resource is requested in an amount greater than allowed

Return type
None

issueBatchJob(*jobDesc*, *job_environment=None*)

Adds the command and resources to a queue to be run.

Parameters
- **jobDesc** (toil.job.JobDescription) –
- **job_environment** (*Optional[Dict[str, str]]*) –

Return type
int

killBatchJobs(*jobIDs*)

Kills jobs by ID.

Parameters
jobIDs (*List[int]*) –

Return type
None

getIssuedBatchJobIDs()

Just returns all the jobs that have been run, but not yet returned as updated.

Return type
List[int]

getRunningBatchJobIDs()

Gets a map of jobs as jobIDs that are currently running (not just waiting) and how long they have been running, in seconds.

Returns
dictionary with currently running jobID keys and how many seconds they have been running as the value

Return type
Dict[int, float]

shutdown()

Terminate cleanly and join daddy thread.

Return type
None

getUpdatedBatchJob(*maxWait*)

Returns a tuple of a no-longer-running job, the return value of its process, and its runtime, or None.

Parameters
maxWait (*int*) –

Return type
Optional[*toil.batchSystems.abstractBatchSystem.UpdatedBatchJobInfo*]

classmethod add_options(*parser*)

If this batch system provides any command line options, add them to the given parser.

Parameters
parser (*Union[argparse.ArgumentParser, argparse._ArgumentGroup]*) –

Return type
None

classmethod setOptions(*setOption*)

Process command line or configuration options relevant to this batch system.

Parameters
setOption (toil.batchSystems.options.OptionSetter) – A function with signature setOption(option_name, parsing_function=None, check_function=None, default=None, env=None) returning nothing, used to update run configuration as a side effect.

class toil.batchSystems.singleMachine.**Info**(*startTime*, *popen*, *resources*, *killIntended*)

Record for a running job.

Stores the start time of the job, the Popen object representing its child (or None), the tuple of (coreFractions, memory, disk) it is using (or None), and whether the job is supposed to be being killed.

toil.batchSystems.slurm

Module Contents

Classes

SlurmBatchSystem	A partial implementation of BatchSystemSupport for batch systems run on a

Attributes

logger

toil.batchSystems.slurm.**logger**

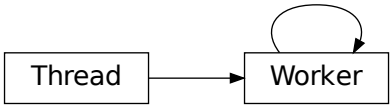
class toil.batchSystems.slurm.**SlurmBatchSystem**(*config*, *maxCores*, *maxMemory*, *maxDisk*)

Bases: *toil.batchSystems.abstractGridEngineBatchSystem.AbstractGridEngineBatchSystem*

A partial implementation of BatchSystemSupport for batch systems run on a standard HPC cluster. By default auto-deployment is not implemented.

class Worker(*newJobsQueue*, *updatedJobsQueue*, *killQueue*, *killedJobsQueue*, *boss*)

Bases: *toil.batchSystems.abstractGridEngineBatchSystem.AbstractGridEngineBatchSystem.Worker*



A class that represents a thread of control.

This class can be safely subclassed in a limited fashion. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the run() method in a subclass.

Parameters

- **newJobsQueue** (*queue.Queue*) –
- **updatedJobsQueue** (*queue.Queue*) –
- **killQueue** (*queue.Queue*) –
- **killedJobsQueue** (*queue.Queue*) –
- **boss** (AbstractGridEngineBatchSystem) –

getRunningJobIDs()

Get a list of running job IDs. Implementation-specific; called by boss AbstractGridEngineBatchSystem implementation via AbstractGridEngineBatchSystem.getRunningBatchJobIDs()

Return type
list

killJob(*jobID*)

Kill specific job with the Toil job ID. Implementation-specific; called by AbstractGridEngineWorker.killJobs()

Parameters
jobID (*string*) – Toil job ID

prepareSubmission(*cpu*, *memory*, *jobID*, *command*, *jobName*, *job_environment=None*, *gpus=None*)

Preparation in putting together a command-line string for submitting to batch system (via submitJob().)

Param
int cpu
Param
int memory
Param
int jobID: Toil job ID
Param
string subLine: the command line string to be called
Param
string jobName: the name of the Toil job, to provide metadata to batch systems if desired
Param
dict job_environment: the environment variables to be set on the worker

Return type
List[str]
Parameters
- **cpu** (*int*) –
- **memory** (*int*) –
- **jobID** (*int*) –
- **command** (*str*) –
- **jobName** (*str*) –
- **job_environment** (*Optional[Dict[str, str]]*) –
- **gpus** (*Optional[int]*) –

submitJob(*subLine*)

Wrapper routine for submitting the actual command-line call, then processing the output to get the batch system job ID

Param
string subLine: the literal command line string to be called
Return type
string: batch system job ID, which will be stored internally

coalesce_job_exit_codes(*batch_job_id_list*)

Collect all job exit codes in a single call. :param batch_job_id_list: list of Job ID strings, where each string has the form "<job>[.<task>]". :return: list of job exit codes, associated with the list of job IDs.

Parameters
batch_job_id_list (*list*) –
Return type
list

getJobExitCode(*batchJobID*)

Get job exit code for given batch job ID. :param batchJobID: string of the form "<job>[.<task>]". :return: integer job exit code.

Parameters
batchJobID (*str*) –
Return type
int

prepareSbatch(*cpu*, *mem*, *jobID*, *jobName*, *job_environment*, *gpus*)

Parameters
- **cpu** (*int*) –
- **mem** (*int*) –
- **jobID** (*int*) –
- **jobName** (*str*) –
- **job_environment** (*Optional[Dict[str, str]]*) –
- **gpus** (*Optional[int]*) –

Return type
List[str]

parse_elapsed(*elapsed*)

OptionType

classmethod add_options(*parser*)

If this batch system provides any command line options, add them to the given parser.

Parameters
parser (*Union[argparse.ArgumentParser, argparse._ArgumentGroup]*) –

classmethod setOptions(*setOption*)

Process command line or configuration options relevant to this batch system.

Parameters
setOption (toil.batchSystems.options.OptionSetter) – A function with signature setOption(option_name, parsing_function=None, check_function=None, default=None, env=None) returning nothing, used to update run configuration as a side effect.

Return type
None

toil.batchSystems.tes

Batch system for running Toil workflows on GA4GH TES.

Useful with network-based job stores when the TES server provides tasks with credentials, and filesystem-based job stores when the TES server lets tasks mount the job store.

Additional containers should be launched with Singularity, not Docker.

Module Contents

Classes

TESBatchSystem	Adds cleanup support when the last running job leaves a node, for batch

Attributes

logger
STATE_TO_EXIT_REASON

toil.batchSystems.tes.**logger**

toil.batchSystems.tes.**STATE_TO_EXIT_REASON: Dict[str, *toil.batchSystems.abstractBatchSystem.BatchJobExitReason*]**

class toil.batchSystems.tes.**TESBatchSystem**(*config*, *maxCores*, *maxMemory*, *maxDisk*)

Bases: *toil.batchSystems.cleanup_support.BatchSystemCleanupSupport*

Adds cleanup support when the last running job leaves a node, for batch systems that can't provide it using the backing scheduler.

Parameters

- **config** (toil.common.Config) –
- **maxCores** (*float*) –
- **maxMemory** (*int*) –
- **maxDisk** (*int*) –

classmethod supportsAutoDeployment()

Whether this batch system supports auto-deployment of the user script itself.

If it does, the *setUserScript()* can be invoked to set the resource object representing the user script.

Note to implementors: If your implementation returns True here, it should also override

Return type
bool

classmethod get_default_tes_endpoint()

Get the default TES endpoint URL to use.

(unless overridden by an option or environment variable)

Return type
str

setUserScript(*user_script*)

Set the user script for this workflow. This method must be called before the first job is issued to this batch system, and only if *supportsAutoDeployment()* returns True, otherwise it will raise an exception.

Parameters

- **userScript** – the resource object representing the user script or module and the modules it depends on.
- **user_script** (toil.resource.Resource) –

Return type
None

issueBatchJob(*job_desc*, *job_environment=None*)

Issues a job with the specified command to the batch system and returns a unique jobID.

Parameters

- **jobDesc** – a toil.job.JobDescription
- **job_environment** (*Optional[Dict[str, str]]*) – a collection of job-specific environment variables to be set on the worker.
- **job_desc** (toil.job.JobDescription) –

Returns
a unique jobID that can be used to reference the newly issued job

Return type
int

getUpdatedBatchJob(*maxWait*)

Returns information about job that has updated its status (i.e. ceased running, either successfully or with an error). Each such job will be returned exactly once.

Does not return info for jobs killed by killBatchJobs, although they may cause None to be returned earlier than maxWait.

Parameters
maxWait (*int*) – the number of seconds to block, waiting for a result

Returns
If a result is available, returns UpdatedBatchJobInfo. Otherwise it returns None. wallTime is the number of seconds (a strictly positive float) in wall-clock time the job ran for, or None if this batch system does not support tracking wall time.

Return type
Optional[*toil.batchSystems.abstractBatchSystem.UpdatedBatchJobInfo*]

shutdown()

Called at the completion of a toil invocation. Should cleanly terminate all worker threads.

Return type
None

getIssuedBatchJobIDs()

Gets all currently issued jobs

Returns
A list of jobs (as jobIDs) currently issued (may be running, or may be waiting to be run). Despite the result being a list, the ordering should not be depended upon.

Return type
List[int]

getRunningBatchJobIDs()

Gets a map of jobs as jobIDs that are currently running (not just waiting) and how long they have been running, in seconds.

Returns
dictionary with currently running jobID keys and how many seconds they have been running as the value

Return type
Dict[int, float]

killBatchJobs(*job_ids*)

Kills the given job IDs. After returning, the killed jobs will not appear in the results of getRunningBatchJobIDs. The killed job will not be returned from getUpdatedBatchJob.

Parameters

- **jobIDs** – list of IDs of jobs to kill
- **job_ids** (*List[int]*) –

Return type
None

classmethod add_options(*parser*)

If this batch system provides any command line options, add them to the given parser.

Parameters
parser (*Union[argparse.ArgumentParser, argparse._ArgumentGroup]*) –

Return type
None

classmethod setOptions(*setOption*)

Process command line or configuration options relevant to this batch system.

Parameters

setOption (toil.batchSystems.options.OptionSetter) – A function with signature setOption(option_name, parsing_function=None, check_function=None, default=None, env=None) returning nothing, used to update run configuration as a side effect.

Return type

None

toil.batchSystems.torque

Module Contents

Classes

TorqueBatchSystem	A partial implementation of BatchSystemSupport for batch systems run on a

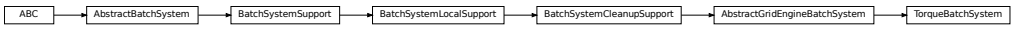
Attributes

logger

toil.batchSystems.torque.**logger**

class toil.batchSystems.torque.**TorqueBatchSystem**(*config*, *maxCores*, *maxMemory*, *maxDisk*)

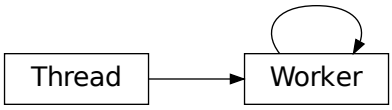
Bases: *toil.batchSystems.abstractGridEngineBatchSystem.AbstractGridEngineBatchSystem*



A partial implementation of BatchSystemSupport for batch systems run on a standard HPC cluster. By default auto-deployment is not implemented.

class Worker(*newJobsQueue*, *updatedJobsQueue*, *killQueue*, *killedJobsQueue*, *boss*)

Bases: *toil.batchSystems.abstractGridEngineBatchSystem.AbstractGridEngineBatchSystem.Worker*



A class that represents a thread of control.

This class can be safely subclassed in a limited fashion. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the run() method in a subclass.

getRunningJobIDs()

Get a list of running job IDs. Implementation-specific; called by boss AbstractGridEngineBatchSystem implementation via AbstractGridEngineBatchSystem.getRunningBatchJobIDs()

Return type
list

getUpdatedBatchJob(*maxWait*)

killJob(*jobID*)

Kill specific job with the Toil job ID. Implementation-specific; called by AbstractGridEngineWorker.killJobs()

Parameters
jobID (*string*) – Toil job ID

prepareSubmission(*cpu*, *memory*, *jobID*, *command*, *jobName*, *job_environment=None*, *gpus=None*)

Preparation in putting together a command-line string for submitting to batch system (via submitJob().)

Param
int cpu
Param
int memory
Param
int jobID: Toil job ID
Param
string subLine: the command line string to be called
Param
string jobName: the name of the Toil job, to provide metadata to batch systems if desired
Param
dict job_environment: the environment variables to be set on the worker
Return type
List[str]
Parameters
- **cpu** (*int*) –
- **memory** (*int*) –
- **jobID** (*int*) –
- **command** (*str*) –
- **jobName** (*str*) –
- **job_environment** (*Optional[Dict[str, str]]*) –
- **gpus** (*Optional[int]*) –

submitJob(*subLine*)

Wrapper routine for submitting the actual command-line call, then processing the output to get the batch system job ID

Param
string subLine: the literal command line string to be called

Return type
string: batch system job ID, which will be stored internally

getJobExitCode(*torqueJobID*)

Returns job exit code or an instance of abstractBatchSystem.BatchJobExitReason. if something else happened other than the job exiting. Implementation-specific; called by AbstractGridEngineWorker.checkOnJobs()

Parameters
batchjobID (*string*) – batch system job ID

Return type
int|toil.batchSystems.abstractBatchSystem.BatchJobExitReason: exit code int or BatchJobExitReason if something else happened other than job exiting.

prepareQsub(*cpu*, *mem*, *jobID*, *job_environment*)

Parameters
- **cpu** (*int*) –
- **mem** (*int*) –
- **jobID** (*int*) –
- **job_environment** (*Optional[Dict[str, str]]*) –

Return type
List[str]

generateTorqueWrapper(*command*, *jobID*)

A very simple script generator that just wraps the command given; for now this goes to default tempdir

Package Contents

exception toil.batchSystems.**DeadlockException**(*msg*)

Bases: Exception

Exception thrown by the Leader or BatchSystem when a deadlock is encountered due to insufficient resources to run the workflow

__str__()

Stringify the exception, including the message.

toil.cwl

Submodules

toil.cwl.conftest

Module Contents

toil.cwl.conftest.**collect_ignore** = []

toil.cwl.cwltoil

Implemented support for Common Workflow Language (CWL) for Toil.

Module Contents

Classes

UnresolvedDict	Tag to indicate a dict contains promises that must be resolved.
SkipNull	Internal sentinel object.
Conditional	Object holding conditional expression until we are ready to evaluate it.
ResolveSource	Apply linkMerge and pickValue operators to values coming into a port.
StepValueFrom	A workflow step input which has a valueFrom expression attached to it.
DefaultWithSource	A workflow step input that has both a source and a default value.
JustAValue	A simple value masquerading as a 'resolve'-able object.
ToilPathMapper	Keeps track of files in a Toil way.
ToilSingleJobExecutor	A SingleJobExecutor that does not assume it is at the top level of the workflow.
ToilTool	Mixin to hook Toil into a cwltool tool type.
ToilCommandLineTool	Subclass the cwltool command line tool to provide the custom ToilPathMapper.
ToilExpressionTool	Subclass the cwltool expression tool to provide the custom ToilPathMapper.
ToilFsAccess	Custom filesystem access class which handles toil filestore references.
CWLNamedJob	Base class for all CWL jobs that do user work, to give them useful names.
ResolveIndirect	Helper Job.
CWLJobWrapper	Wrap a CWL job that uses dynamic resources requirement.
CWLJob	Execute a CWL tool using cwltool.executors.SingleJobExecutor.
CWLScatter	Implement workflow scatter step.
CWLGather	Follows on to a scatter Job.
SelfJob	Fake job object to facilitate implementation of CWLWorkflow.run().
CWLWorkflow	Toil Job to convert a CWL workflow graph into a Toil job graph.

Functions

cwltoil_was_removed()	Complain about deprecated entrypoint.
filter_skip_null(name, value)	Recursively filter out SkipNull objects from 'value'.
ensure_no_collisions(directory[, dir_description])	Make sure no items in the given CWL Directory have the same name.
resolve_dict_w_promises(dict_w_promises[, file_store])	Resolve a dictionary of promises evaluate expressions to produce the actual values.
simplify_list(maybe_list)	Turn a length one list loaded by cwltool into a scalar.
toil_make_tool(toolpath_object, loadingContext)	Emit custom ToilCommandLineTools.
check_directory_dict_invariants(contents)	Make sure a directory structure dict makes sense. Throws an error
decode_directory(dir_path)	Decode a directory from a "toildir:" path to a directory (or a file in it).
encode_directory(contents)	Encode a directory from a "toildir:" path to a directory (or a file in it).
toil_get_file(file_store, index, existing, file_store_id)	Set up the given file or directory from the Toil jobstore at a file URI
write_file(writeFunc, index, existing, file_uri)	Write a file into the Toil jobstore.
path_to_loc(obj)	Make a path into a location.
import_files(import_function, fs_access, fileindex, ...)	Prepare all files and directories.
upload_directory(directory_metadata, directory_contents)	Upload a Directory object.
upload_file(uploadfunc, fileindex, existing, file_metadata)	Update a file object so that the location is a reference to the toil file store.
writeGlobalFileWrapper(file_store, fileuri)	Wrap writeGlobalFile to accept file:// URIs.
remove_empty_listings(rec)	
toilStageFiles(toil, cwljob, outdir[, destBucket])	Copy input files out of the global file store and update location and path.
get_container_engine(runtime_context)	
makeJob(tool, jobobj, runtime_context, parent_name, ...)	Create the correct Toil Job object for the CWL tool.
remove_pickle_problems(obj)	Doc_loader does not pickle correctly, causing Toil errors, remove from objects.
visitSteps(cmdline_tool, op)	Iterate over a CWL Process object, running the op on each tool description
rm_unprocessed_secondary_files(job_params)	
filtered_secondary_files(unfiltered_secondary_files)	Remove unprocessed secondary files.
scan_for_unsupported_requirements(tool[, ...])	Scan the given CWL tool for any unsupported optional features.
determine_load_listing(tool)	Determine the directory.listing feature in CWL.
generate_default_job_store(batch_system_name, ...)	Choose a default job store appropriate to the requested batch system and
main([args, stdout])	Run the main loop for toil-cwl-runner.
find_default_container(args, builder)	Find the default constructor by consulting a Toil.options object.

Attributes

logger
DEFAULT_TMPDIR
DEFAULT_TMPDIR_PREFIX
DirectoryContents
ProcessType
usage_message

toil.cwl.cwltoil.**logger**

toil.cwl.cwltoil.**DEFAULT_TMPDIR**

toil.cwl.cwltoil.**DEFAULT_TMPDIR_PREFIX**

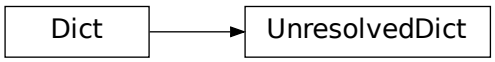
toil.cwl.cwltoil.**cwltoil_was_removed**()

Complain about deprecated entrypoint.

Return type
None

class toil.cwl.cwltoil.**UnresolvedDict**

Bases: Dict[Any, Any]



Tag to indicate a dict contains promises that must be resolved.

class toil.cwl.cwltoil.**SkipNull**

Internal sentinel object.

Indicates a null value produced by each port of a skipped conditional step. The CWL 1.2 specification calls for treating this the exactly the same as a null value.

toil.cwl.cwltoil.**filter_skip_null**(*name*, *value*)

Recursively filter out SkipNull objects from 'value'.

Parameters

- **name** (*str*) – Name of port producing this value. Only used when we find an unhandled null from a conditional step and we print out a warning. The name allows the user to better localize which step/port was responsible for the unhandled null.
- **value** (*Any*) – port output value object

Return type
Any

toil.cwl.cwltoil.**ensure_no_collisions**(*directory*, *dir_description=None*)

Make sure no items in the given CWL Directory have the same name.

If any do, raise a WorkflowException about a "File staging conflict".

Does not recurse into subdirectories.

Parameters

- **directory** (*cwltool.utils.DirectoryType*) –
- **dir_description** (*Optional[str]*) –

Return type
None

class toil.cwl.cwltoil.**Conditional**(*expression=None*, *outputs=None*, *requirements=None*, *container_engine='docker'*)

Object holding conditional expression until we are ready to evaluate it.

Evaluation occurs at the moment the encloses step is ready to run.

Parameters

- **expression** (*Optional[str]*) –
- **outputs** (*Union[Dict[str, cwltool.utils.CWLOutputType], None]*) –
- **requirements** (*Optional[List[cwltool.utils.CWLObjectType]]*) –
- **container_engine** (*str*) –

is_false(*job*)

Determine if expression evaluates to False given completed step inputs.

Parameters
job (*cwltool.utils.CWLObjectType*) – job output object

Returns
bool

Return type
bool

skipped_outputs()

Generate a dict of SkipNull objects corresponding to the output structure.

Return type
Dict[str, *SkipNull*]

class toil.cwl.cwltoil.**ResolveSource**(*name*, *input*, *source_key*, *promises*)

Apply linkMerge and pickValue operators to values coming into a port.

Parameters

- **name** (*str*) –
- **input** (*Dict[str, cwltool.utils.CWLObjectType]*) –
- **source_key** (*str*) –
- **promises** (*Dict[str, toil.job.Job]*) –

promise_tuples: Union[List[Tuple[str, *toil.job.Promise*]], Tuple[str, *toil.job.Promise*]]

__repr__()

Allow for debug printing.

Return type
str

resolve()

First apply linkMerge then pickValue if either present.

Return type
Any

link_merge(*values*)

Apply linkMerge operator to *values* object.

Parameters
values (*cwltool.utils.CWLObjectType*) – result of step

Return type
Union[List[cwltool.utils.CWLOutputType], cwltool.utils.CWLOutputType]

pick_value(*values*)

Apply pickValue operator to *values* object.

Parameters
values (*Union[List[Union[str, SkipNull]], Any]*) – Intended to be a list, but other types will be returned without modification.

Returns

Return type
Any

class toil.cwl.cwltoil.**StepValueFrom**(*expr*, *source*, *req*, *container_engine*)

A workflow step input which has a valueFrom expression attached to it.

The valueFrom expression will be evaluated to produce the actual input object for the step.

Parameters

- **expr** (*str*) –
- **source** (*Any*) –
- **req** (*List[cwltool.utils.CWLObjectType]*) –
- **container_engine** (*str*) –

eval_prep(*step_inputs*, *file_store*)

Resolve the contents of any file in a set of inputs.

The inputs must be associated with the StepValueFrom object's self.source.

Called when loadContents is specified.

Parameters

- **step_inputs** (*cwltool.utils.CWLObjectType*) – Workflow step inputs.
- **file_store** (toil.fileStores.abstractFileStore.AbstractFileStore) – A toil file store, needed to resolve toilfile:// paths.

Return type
None

resolve()

Resolve the promise in the valueFrom expression's context.

Returns
object that will serve as expression context

Return type
Any

do_eval(*inputs*)

Evaluate the valueFrom expression with the given input object.

Parameters
inputs (*cwltool.utils.CWLObjectType*) –

Returns
object

Return type
Any

class toil.cwl.cwltoil.**DefaultWithSource**(*default*, *source*)

A workflow step input that has both a source and a default value.

Parameters

- **default** (*Any*) –
- **source** (*Any*) –

resolve()

Determine the final input value when the time is right.

(when the source can be resolved)

Returns
dict

Return type
Any

class toil.cwl.cwltoil.**JustAValue**(*val*)

A simple value masquerading as a 'resolve'-able object.

Parameters
val (*Any*) –

resolve()

Return the value.

Return type
Any

toil.cwl.cwltoil.**resolve_dict_w_promises**(*dict_w_promises*, *file_store=None*)

Resolve a dictionary of promises evaluate expressions to produce the actual values.

Parameters

- **dict_w_promises** (*Union[UnresolvedDict, cwltool.utils.CWLObjectType, Dict[str, Union[str, StepValueFrom]]]*) – input dict for these values

- **file_store** (*Optional[toil.fileStores.abstractFileStore.AbstractFileStore]*) –

Returns
dictionary of actual values

Return type
cwltool.utils.CWLObjectType

toil.cwl.cwltoil.**simplify_list**(*maybe_list*)

Turn a length one list loaded by cwltool into a scalar.

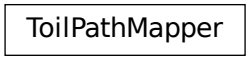
Anything else is passed as-is, by reference.

Parameters
maybe_list (*Any*) –

Return type
Any

class toil.cwl.cwltoil.**ToilPathMapper**(*referenced_files*, *basedir*, *stagedir*, *separateDirs=True*, *get_file=None*, *stage_listing=False*, *streaming_allowed=True*)

Bases: cwltool.pathmapper.PathMapper



Keeps track of files in a Toil way.

Maps between the symbolic identifier of a file (the Toil FileID), its local path on the host (the value returned by readGlobalFile) and the location of the file inside the software container.

Parameters

- **referenced_files** (*List[cwltool.utils.CWLObjectType]*) –
- **basedir** (*str*) –
- **stagedir** (*str*) –
- **separateDirs** (*bool*) –
- **get_file** (*Union[Any, None]*) –
- **stage_listing** (*bool*) –
- **streaming_allowed** (*bool*) –

visit(*obj*, *stagedir*, *basedir*, *copy=False*, *staged=False*)

Iterate over a CWL object, resolving File and Directory path references.

This is called on each File or Directory CWL object. The Files and Directories all have "location" fields. For the Files, these are from upload_file(), and for the Directories, these are from upload_directory(), with their children being assigned locations based on listing the Directories using ToilFsAccess.

Parameters

- **obj** (*cwltool.utils.CWLObjectType*) – The CWL File or Directory to process

- **stagedir** (*str*) – The base path for target paths to be generated under,
- **basedir** (*str*) –
- **copy** (*bool*) –
- **staged** (*bool*) –

Return type
None

except when a File or Directory has an overriding parent directory in dirname

Parameters

- **basedir** (*str*) – The directory from which relative paths should be
- **obj** (*cwltool.utils.CWLObjectType*) –
- **stagedir** (*str*) –
- **copy** (*bool*) –
- **staged** (*bool*) –

Return type
None

resolved; used as the base directory for the StdFsAccess that generated the listing being processed.

Parameters

- **copy** (*bool*) – If set, use writable types for Files and Directories.
- **staged** (*bool*) – Starts as True at the top of the recursion. Set to False
- **obj** (*cwltool.utils.CWLObjectType*) –
- **stagedir** (*str*) –
- **basedir** (*str*) –

Return type
None

when entering a directory that we can actually download, so we don't stage files and subdirectories separately from the directory as a whole. Controls the staged flag on generated mappings, and therefore whether files and directories are actually placed at their mapped-to target locations. If stage_listing is True, we will leave this True throughout and stage everything.

Produces one MapperEnt for every unique location for a File or Directory. These MapperEnt objects are instructions to cwltool's stage_files function: https://github.com/common-workflow-language/cwltool/blob/a3e3a5720f7b0131fa4f9c0b3f73b62a347278a6/cwltool/process.py#L254

The MapperEnt has fields:

resolved: An absolute local path anywhere on the filesystem where the file/directory can be found, or the contents of a file to populate it with if type is CreateWritableFile or CreateFile. Or, a URI understood by the StdFsAccess in use (for example, toilfile:).

target: An absolute path under stagedir that the file or directory will then be placed at by cwltool. Except if a File or Directory has a dirname field, giving its parent path, that is used instead.

type: One of:

File: cwltool will copy or link the file from resolved to target, if possible.

CreateFile: cwltool will create the file at target, treating resolved as the contents.

WritableFile: cwltool will copy the file from resolved to target, making it writable.

CreateWritableFile: cwltool will create the file at target, treating resolved as the contents, and make it writable.

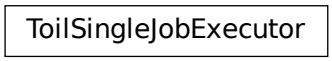
Directory: cwltool will copy or link the directory from resolved to target, if possible. Otherwise, cwltool will make the directory at target if resolved starts with "_:". Otherwise it will do nothing.

WritableDirectory: cwltool will copy the directory from resolved to target, if possible. Otherwise, cwltool will make the directory at target if resolved starts with "_:". Otherwise it will do nothing.

staged: if set to False, cwltool will not make or copy anything for this entry

class toil.cwl.cwltoil.**ToilSingleJobExecutor**

Bases: cwltool.executors.SingleJobExecutor



A SingleJobExecutor that does not assume it is at the top level of the workflow.

We need this because otherwise every job thinks it is top level and tries to discover secondary files, which may exist when they haven't actually been passed at the top level and thus aren't supposed to be visible.

run_jobs(*process*, *job_order_object*, *logger*, *runtime_context*)

run_jobs from SingleJobExecutor, but not in a top level runtime context.

Parameters

- **process** (*cwltool.process.Process*) –
- **job_order_object** (*cwltool.utils.CWLObjectType*) –
- **logger** (*logging.Logger*) –
- **runtime_context** (*cwltool.context.RuntimeContext*) –

Return type
None

class toil.cwl.cwltoil.**ToilTool**

Mixin to hook Toil into a cwltool tool type.

make_path_mapper(*reffiles*, *stagedir*, *runtimeContext*, *separateDirs*)

Create the appropriate PathMapper for the situation.

Parameters

- **reffiles** (*List[Any]*) –
- **stagedir** (*str*) –
- **runtimeContext** (*cwltool.context.RuntimeContext*) –

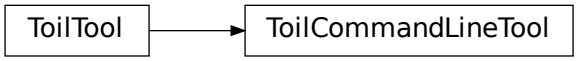
- **separateDirs** (*bool*) –

Return type
cwltool.pathmapper.PathMapper

__str__()
Return string representation of this tool type.

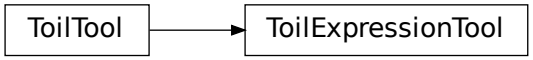
Return type
str

class toil.cwl.cwltoil.**ToilCommandLineTool**
Bases: *ToilTool*, cwltool.command_line_tool.CommandLineTool



Subclass the cwltool command line tool to provide the custom ToilPathMapper.

class toil.cwl.cwltoil.**ToilExpressionTool**
Bases: *ToilTool*, cwltool.command_line_tool.ExpressionTool



Subclass the cwltool expression tool to provide the custom ToilPathMapper.

toil.cwl.cwltoil.**toil_make_tool**(*toolpath_object*, *loadingContext*)
Emit custom ToilCommandLineTools.

This factory function is meant to be passed to cwltool.load_tool().

Parameters

- **toolpath_object** (*ruamel.yaml.comments.CommentedMap*) –
- **loadingContext** (*cwltool.context.LoadingContext*) –

Return type
cwltool.process.Process

toil.cwl.cwltoil.**DirectoryContents**

toil.cwl.cwltoil.**check_directory_dict_invariants**(*contents*)
Make sure a directory structure dict makes sense. Throws an error otherwise.

Currently just checks to make sure no empty-string keys exist.

Parameters
contents (*DirectoryContents*) –

Return type
None

toil.cwl.cwltoil.**decode_directory**(*dir_path*)

Decode a directory from a "toildir:" path to a directory (or a file in it).

Returns the decoded directory dict, the remaining part of the path (which may be None), and the deduplication key string that uniquely identifies the directory.

Parameters
dir_path (*str*) –

Return type
Tuple[DirectoryContents, Optional[str], str]

toil.cwl.cwltoil.**encode_directory**(*contents*)

Encode a directory from a "toildir:" path to a directory (or a file in it).

Takes the directory dict, which is a dict from name to URI for a file or dict for a subdirectory.

Parameters
contents (*DirectoryContents*) –

Return type
str

class toil.cwl.cwltoil.**ToilFsAccess**(*basedir*, *file_store=None*)

Bases: cwltool.stdfsaccess.StdFsAccess

Custom filesystem access class which handles toil filestore references.

Normal file paths will be resolved relative to basedir, but 'toilfile:' and 'toildir:' URIs will be fulfilled from the Toil file store.

Also supports URLs supported by Toil job store implementations.

Parameters

- **basedir** (*str*) –
- **file_store** (*Optional[toil.fileStores.abstractFileStore.AbstractFileStore]*) –

glob(*pattern*)

Parameters
pattern (*str*) –

Return type
List[str]

open(*fn*, *mode*)

Parameters

- **fn** (*str*) –
- **mode** (*str*) –

Return type
IO[Any]

exists(*path*)

Test for file existence.

Parameters
path (*str*) –

Return type
bool

size(*path*)

Parameters
path (*str*) –

Return type
int

isfile(*fn*)

Parameters
fn (*str*) –

Return type
bool

isdir(*fn*)

Parameters
fn (*str*) –

Return type
bool

listdir(*fn*)

Parameters
fn (*str*) –

Return type
List[str]

join(*path*, **paths*)

Parameters

- **path** (*str*) –
- **paths** (*str*) –

Return type
str

realpath(*fn*)

Parameters
fn (*str*) –

Return type
str

toil.cwl.cwltoil.**toil_get_file**(*file_store*, *index*, *existing*, *file_store_id*, *streamable=False*, *streaming_allowed=True*, *pipe_threads=None*)

Set up the given file or directory from the Toil jobstore at a file URI where it can be accessed locally.

Run as part of the tool setup, inside jobs on the workers. Also used as part of reorganizing files to get them uploaded at the end of a tool.

Parameters

- **file_store** (toil.fileStores.abstractFileStore.AbstractFileStore) – The Toil file store to download from.
- **index** (*Dict[str, str]*) – Maps from downloaded file path back to input Toil URI.
- **existing** (*Dict[str, str]*) – Maps from file_store_id URI to downloaded file path.
- **file_store_id** (*str*) – The URI for the file to download.
- **streamable** (*bool*) – If the file is has 'streamable' flag set
- **streaming_allowed** (*bool*) – If streaming is allowed
- **pipe_threads** (*Optional[List[Tuple[threading.Thread, int]]]*) – List of threads responsible for streaming the data

Return type
str

and open file descriptors corresponding to those files. Caller is responsible to close the file descriptors (to break the pipes) and join the threads

toil.cwl.cwltoil.**write_file**(*writeFunc*, *index*, *existing*, *file_uri*)

Write a file into the Toil jobstore.

'existing' is a set of files retrieved as inputs from toil_get_file. This ensures they are mapped back as the same name if passed through.

Returns a toil uri path to the object.

Parameters

- **writeFunc** (*Callable[[str], toil.fileStores.FileID]*) –
- **index** (*Dict[str, str]*) –
- **existing** (*Dict[str, str]*) –
- **file_uri** (*str*) –

Return type
str

toil.cwl.cwltoil.**path_to_loc**(*obj*)

Make a path into a location.

(If a CWL object has a "path" and not a "location")

Parameters
obj (*cwltool.utils.CWLObjectType*) –

Return type
None

toil.cwl.cwltoil.**import_files**(*import_function*, *fs_access*, *fileindex*, *existing*, *cwl_object*, *skip_broken=False*, *bypass_file_store=False*)

Prepare all files and directories.

Will be executed from the leader or worker in the context of the given CWL tool, order, or output object to be used on the workers. Make sure their sizes are set and import all the files.

Recurses inside directories using the fs_access to find files to upload and subdirectory structure to encode, even if their listings are not set or not recursive.

Preserves any listing fields.

If a file cannot be found (like if it is an optional secondary file that doesn't exist), fails, unless skip_broken is set, in which case it leaves the location it was supposed to have been at.

Also does some miscelaneous normalization.

Parameters

- **import_function** (*Callable[[str], toil.fileStores.FileID]*) – The function used to upload a URI and get a
- **fs_access** (*cwltool.stdfsaccess.StdFsAccess*) –
- **fileindex** (*Dict[str, str]*) –
- **existing** (*Dict[str, str]*) –
- **cwl_object** (*Optional[cwltool.utils.CWLObjectType]*) –
- **skip_broken** (*bool*) –
- **bypass_file_store** (*bool*) –

Return type
None

Toil FileID for it.

Parameters

- **fs_access** (*cwltool.stdfsaccess.StdFsAccess*) – the CWL FS access object we use to access the filesystem
- **import_function** (*Callable[[str], toil.fileStores.FileID]*) –
- **fileindex** (*Dict[str, str]*) –
- **existing** (*Dict[str, str]*) –
- **cwl_object** (*Optional[cwltool.utils.CWLObjectType]*) –
- **skip_broken** (*bool*) –
- **bypass_file_store** (*bool*) –

Return type
None

to find files to import. Needs to support the URI schemes used.

Parameters

- **fileindex** (*Dict[str, str]*) – Forward map to fill in from file URI to Toil storage
- **import_function** (*Callable[[str], toil.fileStores.FileID]*) –
- **fs_access** (*cwltool.stdfsaccess.StdFsAccess*) –
- **existing** (*Dict[str, str]*) –
- **cwl_object** (*Optional[cwltool.utils.CWLObjectType]*) –
- **skip_broken** (*bool*) –
- **bypass_file_store** (*bool*) –

Return type
None

location, used by write_file to deduplicate writes.

Parameters

- **existing** (*Dict[str, str]*) – Reverse map to fill in from Toil storage location to file
- **import_function** (*Callable[[str], toil.fileStores.FileID]*) –
- **fs_access** (*cwltool.stdfsaccess.StdFsAccess*) –
- **fileindex** (*Dict[str, str]*) –
- **cwl_object** (*Optional[cwltool.utils.CWLObjectType]*) –
- **skip_broken** (*bool*) –
- **bypass_file_store** (*bool*) –

Return type
None

URI. Not read from.

Parameters

- **cwl_object** (*Optional[cwltool.utils.CWLObjectType]*) – CWL tool (or workflow order) we are importing files for
- **skip_broken** (*bool*) – If True, when files can't be imported because they e.g.
- **import_function** (*Callable[[str], toil.fileStores.FileID]*) –
- **fs_access** (*cwltool.stdfsaccess.StdFsAccess*) –
- **fileindex** (*Dict[str, str]*) –
- **existing** (*Dict[str, str]*) –
- **bypass_file_store** (*bool*) –

Return type
None

don't exist, leave their locations alone rather than failing with an error.

Parameters

- **bypass_file_store** (*bool*) – If True, leave file:// URIs in place instead of
- **import_function** (*Callable[[str], toil.fileStores.FileID]*) –
- **fs_access** (*cwltool.stdfsaccess.StdFsAccess*) –

- **fileindex** (*Dict[str, str]*) –
- **existing** (*Dict[str, str]*) –
- **cwl_object** (*Optional[cwltool.utils.CWLObjectType]*) –
- **skip_broken** (*bool*) –

Return type
None

importing files and directories.

toil.cwl.cwltoil.**upload_directory**(*directory_metadata*, *directory_contents*, *skip_broken=False*)

Upload a Directory object.

Ignores the listing (which may not be recursive and isn't safe or efficient to touch), and instead uses directory_contents, which is a recursive dict structure from filename to file URI or subdirectory contents dict.

Makes sure the directory actually exists, and rewrites its location to be something we can use on another machine.

We can't rely on the directory's listing as visible to the next tool as a complete recursive description of the files we will need to present to the tool, since some tools require it to be cleared or single-level but still expect to see its contents in the filesystem.

Parameters

- **directory_metadata** (*cwltool.utils.CWLObjectType*) –
- **directory_contents** (*DirectoryContents*) –
- **skip_broken** (*bool*) –

Return type
None

toil.cwl.cwltoil.**upload_file**(*uploadfunc*, *fileindex*, *existing*, *file_metadata*, *skip_broken=False*)

Update a file object so that the location is a reference to the toil file store.

Write the file object to the file store if necessary.

Parameters

- **uploadfunc** (*Callable[[str], toil.fileStores.FileID]*) –
- **fileindex** (*Dict[str, str]*) –
- **existing** (*Dict[str, str]*) –
- **file_metadata** (*cwltool.utils.CWLObjectType*) –
- **skip_broken** (*bool*) –

Return type
None

toil.cwl.cwltoil.**writeGlobalFileWrapper**(*file_store*, *fileuri*)

Wrap writeGlobalFile to accept file:// URIs.

Parameters

- **file_store** (*toil.fileStores.abstractFileStore.AbstractFileStore*) –
- **fileuri** (*str*) –

Return type
toil.fileStores.FileID

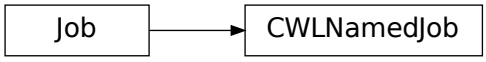
toil.cwl.cwltoil.**remove_empty_listings**(*rec*)

Parameters
rec (*cwltool.utils.CWLObjectType*) –

Return type
None

class toil.cwl.cwltoil.**CWLNamedJob**(*cores=1*, *memory='1GiB'*, *disk='1MiB'*, *accelerators=None*, *tool_id=None*, *parent_name=None*, *subjob_name=None*, *local=None*)

Bases: *toil.job.Job*



Base class for all CWL jobs that do user work, to give them useful names.

Parameters

- **cores** (*Union[float, None]*) –
- **memory** (*Union[int, str, None]*) –
- **disk** (*Union[int, str, None]*) –
- **accelerators** (*Optional[List[toil.job.AcceleratorRequirement]]*) –
- **tool_id** (*Optional[str]*) –
- **parent_name** (*Optional[str]*) –
- **subjob_name** (*Optional[str]*) –
- **local** (*Optional[bool]*) –

class toil.cwl.cwltoil.**ResolveIndirect**(*cwljob*, *parent_name=None*)

Bases: *CWLNamedJob*

Helper Job.

Accepts an unresolved dict (containing promises) and produces a dictionary of actual values.

Parameters

- **cwljob** (*toil.job.Promised[cwltool.utils.CWLObjectType]*) –
- **parent_name** (*Optional[str]*) –

run(*file_store*)

Evaluate the promises and return their values.

Parameters
file_store (toil.fileStores.abstractFileStore.AbstractFileStore) –

Return type
cwltool.utils.CWLObjectType

toil.cwl.cwltoil.**toilStageFiles**(*toil*, *cwljob*, *outdir*, *destBucket=None*)

Copy input files out of the global file store and update location and path.

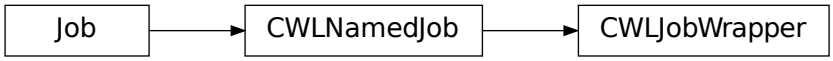
Parameters

- **destBucket** (*Union[str, None]*) – If set, export to this base URL instead of to the local filesystem.
- **toil** (toil.common.Toil) –
- **cwljob** (*Union[cwltool.utils.CWLObjectType, List[cwltool.utils.CWLObjectType]]*) –
- **outdir** (*str*) –

Return type
None

class toil.cwl.cwltoil.**CWLJobWrapper**(*tool*, *cwljob*, *runtime_context*, *parent_name*, *conditional=None*)

Bases: *CWLNamedJob*



Wrap a CWL job that uses dynamic resources requirement.

When executed, this creates a new child job which has the correct resource requirement set.

Parameters

- **tool** (*cwltool.process.Process*) –
- **cwljob** (*cwltool.utils.CWLObjectType*) –
- **runtime_context** (*cwltool.context.RuntimeContext*) –
- **parent_name** (*Optional[str]*) –
- **conditional** (*Union[Conditional, None]*) –

run(*file_store*)

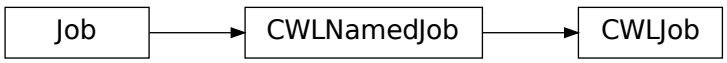
Create a child job with the correct resource requirements set.

Parameters
file_store (toil.fileStores.abstractFileStore.AbstractFileStore) –

Return type
Any

class toil.cwl.cwltoil.**CWLJob**(*tool*, *cwljob*, *runtime_context*, *parent_name=None*, *conditional=None*)

Bases: *CWLNamedJob*



Execute a CWL tool using cwltool.executors.SingleJobExecutor.

Parameters

- **tool** (*cwltool.process.Process*) –
- **cwljob** (*cwltool.utils.CWLObjectType*) –
- **runtime_context** (*cwltool.context.RuntimeContext*) –
- **parent_name** (*Optional[str]*) –
- **conditional** (*Union[Conditional, None]*) –

required_env_vars(*cwljob*)

Yield environment variables from EnvVarRequirement.

Parameters
cwljob (*Any*) –

Return type
Iterator[Tuple[str, str]]

populate_env_vars(*cwljob*)

Prepare environment variables necessary at runtime for the job.

Env vars specified in the CWL "requirements" section should already be loaded in self.cwltool.requirements, however those specified with "EnvVarRequirement" take precedence and are only populated here. Therefore, this not only returns a dictionary with all evaluated "EnvVarRequirement" env vars, but checks self.cwltool.requirements for any env vars with the same name and replaces their value with that found in the "EnvVarRequirement" env var if it exists.

Parameters
cwljob (*cwltool.utils.CWLObjectType*) –

Return type
Dict[str, str]

run(*file_store*)

Execute the CWL document.

Parameters
file_store (*toil.fileStores.abstractFileStore.AbstractFileStore*) –

Return type
Any

toil.cwl.cwltoil.**get_container_engine**(*runtime_context*)

Parameters
runtime_context (*cwltool.context.RuntimeContext*) –

Return type
str

toil.cwl.cwltoil.**makeJob**(*tool*, *jobobj*, *runtime_context*, *parent_name*, *conditional*)

Create the correct Toil Job object for the CWL tool.

Types: workflow, job, or job wrapper for dynamic resource requirements.

Returns
"wfjob, followOn" if the input tool is a workflow, and "job, job" otherwise

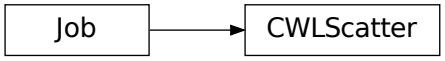
Parameters

- **tool** (*cwltool.process.Process*) –
- **jobobj** (*cwltool.utils.CWLObjectType*) –
- **runtime_context** (*cwltool.context.RuntimeContext*) –
- **parent_name** (*Optional[str]*) –
- **conditional** (*Union[Conditional, None]*) –

Return type
Union[Tuple[*CWLWorkflow*, *ResolveIndirect*], Tuple[*CWLJob*, *CWLJob*], Tuple[*CWLJobWrapper*, *CWLJobWrapper*]]

class toil.cwl.cwltoil.**CWLScatter**(*step*, *cwljob*, *runtime_context*, *parent_name*, *conditional*)

Bases: *toil.job.Job*



Implement workflow scatter step.

When run, this creates a child job for each parameterization of the scatter.

Parameters

- **step** (*cwltool.workflow.WorkflowStep*) –
- **cwljob** (*cwltool.utils.CWLObjectType*) –
- **runtime_context** (*cwltool.context.RuntimeContext*) –
- **parent_name** (*Optional[str]*) –
- **conditional** (*Union[Conditional, None]*) –

flat_crossproduct_scatter(*joborder*, *scatter_keys*, *outputs*, *postScatterEval*)

Cartesian product of the inputs, then flattened.

Parameters

- **joborder** (*cwltool.utils.CWLObjectType*) –
- **scatter_keys** (*List[str]*) –
- **outputs** (*List[toil.job.Promised[cwltool.utils.CWLObjectType]]*) –

- **postScatterEval** (*Callable[[cwltool.utils.CWLObjectType], cwltool.utils.CWLObjectType]*) –

Return type
None

nested_crossproduct_scatter(*joborder*, *scatter_keys*, *postScatterEval*)

Cartesian product of the inputs.

Parameters

- **joborder** (*cwltool.utils.CWLObjectType*) –
- **scatter_keys** (*List[str]*) –
- **postScatterEval** (*Callable[[cwltool.utils.CWLObjectType], cwltool.utils.CWLObjectType]*) –

Return type
List[toil.job.Promised[cwltool.utils.CWLObjectType]]

run(*file_store*)

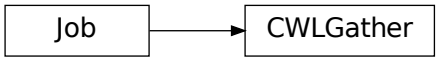
Generate the follow on scatter jobs.

Parameters
file_store (toil.fileStores.abstractFileStore.AbstractFileStore) –

Return type
List[toil.job.Promised[cwltool.utils.CWLObjectType]]

class toil.cwl.cwltoil.**CWLGather**(*step*, *outputs*)

Bases: *toil.job.Job*



Follows on to a scatter Job.

This gathers the outputs of each job in the scatter into an array for each output parameter.

Parameters

- **step** (*cwltool.workflow.WorkflowStep*) –
- **outputs** (*toil.job.Promised[Union[cwltool.utils.CWLObjectType, List[cwltool.utils.CWLObjectType]]]*) –

static extract(*obj*, *k*)

Extract the given key from the obj.

If the object is a list, extract it from all members of the list.

Parameters

- **obj** (*Union[cwltool.utils.CWLObjectType, List[cwltool.utils.CWLObjectType]]*) –

- **k** (*str*) –

Return type
Union[cwltool.utils.CWLOutputType, List[cwltool.utils.CWLObjectType]]

run(*file_store*)

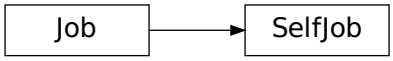
Gather all the outputs of the scatter.

Parameters
file_store (toil.fileStores.abstractFileStore.AbstractFileStore) –

Return type
Dict[str, Any]

class toil.cwl.cwltoil.**SelfJob**(*j*, *v*)

Bases: *toil.job.Job*



Fake job object to facilitate implementation of CWLWorkflow.run().

Parameters

- **j** (CWLWorkflow) –
- **v** (*cwltool.utils.CWLObjectType*) –

rv(**path*)

Return our properties dictionary.

Parameters
path (*Any*) –

Return type
Any

addChild(*c*)

Add a child to our workflow.

Parameters
c (toil.job.Job) –

Return type
Any

hasChild(*c*)

Check if the given child is in our workflow.

Parameters
c (toil.job.Job) –

Return type
Any

toil.cwl.cwltoil.**ProcessType**

toil.cwl.cwltoil.**remove_pickle_problems**(*obj*)

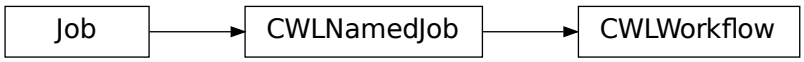
Doc_loader does not pickle correctly, causing Toil errors, remove from objects.

Parameters
obj (*ProcessType*) –

Return type
ProcessType

class toil.cwl.cwltoil.**CWLWorkflow**(*cwlwf*, *cwljob*, *runtime_context*, *parent_name=None*, *conditional=None*)

Bases: *CWLNamedJob*



Toil Job to convert a CWL workflow graph into a Toil job graph.

The Toil job graph will include the appropriate dependencies.

Parameters

- **cwlwf** (*cwltool.workflow.Workflow*) –
- **cwljob** (*cwltool.utils.CWLObjectType*) –
- **runtime_context** (*cwltool.context.RuntimeContext*) –
- **parent_name** (*Optional[str]*) –
- **conditional** (*Union[Conditional, None]*) –

run(*file_store*)

Convert a CWL Workflow graph into a Toil job graph.

Always runs on the leader, because the batch system knows to schedule it as a local job.

Parameters
file_store (toil.fileStores.abstractFileStore.AbstractFileStore) –

Return type
Union[*UnresolvedDict*, Dict[str, *SkipNull*]]

toil.cwl.cwltoil.**visitSteps**(*cmdline_tool*, *op*)

Iterate over a CWL Process object, running the op on each tool description CWL object.

Parameters

- **cmdline_tool** (*cwltool.process.Process*) –
- **op** (*Callable[[ruamel.yaml.comments.CommentedMap], None]*) –

Return type
None

toil.cwl.cwltoil.**rm_unprocessed_secondary_files**(*job_params*)

Parameters
job_params (*Any*) –

Return type
None

toil.cwl.cwltoil.**filtered_secondary_files**(*unfiltered_secondary_files*)

Remove unprocessed secondary files.

Interpolated strings and optional inputs in secondary files were added to CWL in version 1.1.

The CWL libraries we call do successfully resolve the interpolated strings, but add the resolved fields to the list of unresolved fields so we remove them here after the fact.

We keep secondary files using the 'toildir:', or '_:' protocols, or using the 'file:' protocol and indicating files or directories that actually exist. The 'required' logic seems to be handled deeper in cwltool.builder.Builder(), and correctly determines which files should be imported. Therefore we remove the files here and if this file is SUPPOSED to exist, it will still give the appropriate file does not exist error, but just a bit further down the track.

Parameters
unfiltered_secondary_files (*cwltool.utils.CWLObjectType*) –

Return type
List[cwltool.utils.CWLObjectType]

toil.cwl.cwltoil.**scan_for_unsupported_requirements**(*tool*, *bypass_file_store=False*)

Scan the given CWL tool for any unsupported optional features.

If it has them, raise an informative UnsupportedRequirement.

Parameters

- **tool** (*cwltool.process.Process*) – The CWL tool to check for unsupported requirements.
- **bypass_file_store** (*bool*) – True if the Toil file store is not being used to

Return type
None

transport files between nodes, and raw origin node file:// URIs are exposed to tools instead.

toil.cwl.cwltoil.**determine_load_listing**(*tool*)

Determine the directory.listing feature in CWL.

In CWL, any input directory can have a DIRECTORY_NAME.listing (where DIRECTORY_NAME is any variable name) set to one of the following three options:

no_listing: DIRECTORY_NAME.listing will be undefined.
e.g. inputs.DIRECTORY_NAME.listing == unspecified

shallow_listing: DIRECTORY_NAME.listing will return a list one level

deep of DIRECTORY_NAME's contents.

e.g. inputs.DIRECTORY_NAME.listing == [items in directory]
inputs.DIRECTORY_NAME.listing[0].listing == undefined inputs.DIRECTORY_NAME.listing.length == # of items in directory

deep_listing: DIRECTORY_NAME.listing will return a list of the entire

contents of DIRECTORY_NAME.

e.g. inputs.DIRECTORY_NAME.listing == [items in directory]

inputs.DIRECTORY_NAME.listing[0].listing == [items in subdirectory if it exists and is the first item listed]

inputs.DIRECTORY_NAME.listing.length == # of items in directory

See: https://www.commonwl.org/v1.1/CommandLineTool.html#LoadListingRequirement
https://www.commonwl.org/v1.1/CommandLineTool.html#LoadListingEnum

DIRECTORY_NAME.listing should be determined first from loadListing. If that's not specified, from LoadListingRequirement. Else, default to "no_listing" if unspecified.

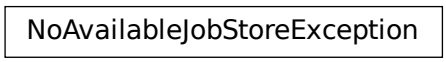
Parameters
tool (*cwltool.process.Process*) – ToilCommandLineTool

Return str
One of 'no_listing', 'shallow_listing', or 'deep_listing'.

Return type
typing_extensions.Literal[no_listing, shallow_listing, deep_listing]

exception toil.cwl.cwltoil.**NoAvailableJobStoreException**

Bases: Exception



Indicates that no job store name is available.

toil.cwl.cwltoil.**generate_default_job_store**(*batch_system_name*, *provisioner_name*, *local_directory*)

Choose a default job store appropriate to the requested batch system and provisioner, and installed modules. Raises an error if no good default is available and the user must choose manually.

Parameters

- **batch_system_name** (*Optional[str]*) – Registry name of the batch system the user has requested, if any. If no name has been requested, should be None.
- **provisioner_name** (*Optional[str]*) – Name of the provisioner the user has requested, if any. Recognized provisioners include 'aws' and 'gce'. None indicates that no provisioner is in use.
- **local_directory** (*str*) – Path to a nonexistent local directory suitable for use as a file job store.

Return str
Job store specifier for a usable job store.

Return type
str

toil.cwl.cwltoil.**usage_message**

toil.cwl.cwltoil.**main**(*args=None*, *stdout=sys.stdout*)

Run the main loop for toil-cwl-runner.

Parameters

- **args** (*Optional[List[str]]*) –
- **stdout** (*TextIO*) –

Return type

int

toil.cwl.cwltoil.**find_default_container**(*args*, *builder*)

Find the default constructor by consulting a Toil.options object.

Parameters

- **args** (*argparse.Namespace*) –
- **builder** (*cwltool.builder.Builder*) –

Return type

Optional[str]

toil.cwl.utils

Utility functions used for Toil's CWL interpreter.

Module Contents

Functions

visit_top_cwl_class(rec, classes, op)	Apply the given operation to all top-level CWL objects with the given named CWL class.
visit_cwl_class_and_reduce(rec, classes, op_down, op_up)	Apply the given operations to all CWL objects with the given named CWL class.
download_structure(file_store, index, existing, ...)	Download nested dictionary from the Toil file store to a local path.

Attributes

logger
CWL_UNSUPPORTED_REQUIREMENT_EXIT_CODE
CWL_UNSUPPORTED_REQUIREMENT_EXCEPTION
DownReturnType
UpReturnType
DirectoryStructure

toil.cwl.utils.**logger**

toil.cwl.utils.**CWL_UNSUPPORTED_REQUIREMENT_EXIT_CODE = 33**

exception toil.cwl.utils.**CWLUnsupportedException**

Bases: Exception

Fallback exception.

toil.cwl.utils.**CWL_UNSUPPORTED_REQUIREMENT_EXCEPTION: Union[Type[cwltool.errors.UnsupportedRequirement], Type[*CWLUnsupportedException*]]**

toil.cwl.utils.**visit_top_cwl_class**(*rec*, *classes*, *op*)

Apply the given operation to all top-level CWL objects with the given named CWL class.

Like cwltool's visit_class but doesn't look inside any object visited.

Parameters

- **rec** (*Any*) –
- **classes** (*Iterable[str]*) –
- **op** (*Callable[[Any], Any]*) –

Return type

None

toil.cwl.utils.**DownReturnType**

toil.cwl.utils.**UpReturnType**

toil.cwl.utils.**visit_cwl_class_and_reduce**(*rec*, *classes*, *op_down*, *op_up*)

Apply the given operations to all CWL objects with the given named CWL class.

Applies the down operation top-down, and the up operation bottom-up, and passes the down operation's result and a list of the up operation results for all child keys (flattening across lists and collapsing nodes of non-matching classes) to the up operation.

Returns
The flattened list of up operation results from all calls.

Parameters

- **rec** (*Any*) –
- **classes** (*Iterable[str]*) –
- **op_down** (*Callable[[Any], DownReturnType]*) –
- **op_up** (*Callable[[Any, DownReturnType, List[UpReturnType]], UpReturnType]*) –

Return type
List[UpReturnType]

toil.cwl.utils.**DirectoryStructure**

toil.cwl.utils.**download_structure**(*file_store*, *index*, *existing*, *dir_dict*, *into_dir*)

Download nested dictionary from the Toil file store to a local path.

Parameters

- **file_store** (toil.fileStores.abstractFileStore.AbstractFileStore) – The Toil file store to download from.
- **index** (*Dict[str, str]*) – Maps from downloaded file path back to input Toil URI.
- **existing** (*Dict[str, str]*) – Maps from file_store_id URI to downloaded file path.
- **dir_dict** (*DirectoryStructure*) – a dict from string to string (for files) or dict (for
- **into_dir** (*str*) –

Return type
None

subdirectories) describing a directory structure.

Parameters

- **into_dir** (*str*) – The directory to download the top-level dict's files
- **file_store** (toil.fileStores.abstractFileStore.AbstractFileStore) –
- **index** (*Dict[str, str]*) –
- **existing** (*Dict[str, str]*) –
- **dir_dict** (*DirectoryStructure*) –

Return type
None

into.

Package Contents

Functions

check_cwltool_version()	Check if the installed cwltool version matches Toil's expected version. A

Attributes

cwltool_version
logger

exception toil.cwl.**InvalidVersion**

Bases: Exception

Common base class for all non-exit exceptions.

toil.cwl.**cwltool_version = '3.1.20230425144158'**

toil.cwl.**logger**

toil.cwl.**check_cwltool_version**()

Check if the installed cwltool version matches Toil's expected version. A warning is printed if the versions differ.

Return type
None

toil.fileStores

Submodules

toil.fileStores.abstractFileStore

Module Contents

Classes

AbstractFileStore	Interface used to allow user code run by Toil to read and write files.

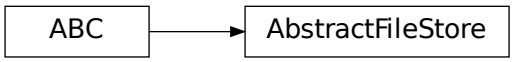
Attributes

logger

toil.fileStores.abstractFileStore.**logger**

class toil.fileStores.abstractFileStore.**AbstractFileStore**(*jobStore*, *jobDesc*, *file_store_dir*, *waitForPreviousCommit*)

Bases: abc.ABC



Interface used to allow user code run by Toil to read and write files.

Also provides the interface to other Toil facilities used by user code, including:

- normal (non-real-time) logging
- finding the correct temporary directory for scratch work
- importing and exporting files into and out of the workflow

Stores user files in the jobStore, but keeps them separate from actual jobs.

May implement caching.

Passed as argument to the *toil.job.Job.run()* method.

Access to files is only permitted inside the context manager provided by *toil.fileStores.abstractFileStore.AbstractFileStore.open()*.

Also responsible for committing completed jobs back to the job store with an update operation, and allowing that commit operation to be waited for.

Parameters

- **jobStore** (toil.jobStores.abstractJobStore.AbstractJobStore) –
- **jobDesc** (toil.job.JobDescription) –
- **file_store_dir** (*str*) –
- **waitForPreviousCommit** (*Callable[[], Any]*) –

static createFileStore(*jobStore*, *jobDesc*, *file_store_dir*, *waitForPreviousCommit*, *caching*)

Create a concreate FileStore.

Parameters

- **jobStore** (toil.jobStores.abstractJobStore.AbstractJobStore) –
- **jobDesc** (toil.job.JobDescription) –
- **file_store_dir** (*str*) –

- **waitForPreviousCommit** (*Callable[[], Any]*) –
- **caching** (*Optional[bool]*) –

Return type
Union[*toil.fileStores.nonCachingFileStore.NonCachingFileStore*, *toil.fileStores.cachingFileStore.CachingFileStore*]

static shutdownFileStore(*workflowID*, *config_work_dir*, *config_coordination_dir*)

Carry out any necessary filestore-specific cleanup.

This is a destructive operation and it is important to ensure that there are no other running processes on the system that are modifying or using the file store for this workflow.

This is the intended to be the last call to the file store in a Toil run, called by the batch system cleanup function upon batch system shutdown.

Parameters

- **workflowID** (*str*) – The workflow ID for this invocation of the workflow
- **config_work_dir** (*Optional[str]*) – The path to the work directory in the Toil Config.
- **config_coordination_dir** (*Optional[str]*) – The path to the coordination directory in the Toil Config.

Return type
None

open(*job*)

Create the context manager around tasks prior and after a job has been run.

File operations are only permitted inside the context manager.

Implementations must only yield from within *with super().open(job):*.

Parameters
job (toil.job.Job) – The job instance of the toil job to run.

Return type
Generator[None, None, None]

getLocalTempDir()

Get a new local temporary directory in which to write files.

The directory will only persist for the duration of the job.

Returns
The absolute path to a new local temporary directory. This directory will exist for the duration of the job only, and is guaranteed to be deleted once the job terminates, removing all files it contains recursively.

Return type
str

getLocalTempFile(*suffix=None*, *prefix=None*)

Get a new local temporary file that will persist for the duration of the job.

Parameters

- **suffix** (*Optional[str]*) – If not None, the file name will end with this string. Otherwise, default value ".tmp" will be used
- **prefix** (*Optional[str]*) – If not None, the file name will start with this string. Otherwise, default value "tmp" will be used

Returns
The absolute path to a local temporary file. This file will exist for the duration of the job only, and is guaranteed to be deleted once the job terminates.

Return type
str

getLocalTempFileName(*suffix=None*, *prefix=None*)

Get a valid name for a new local file. Don't actually create a file at the path.

Parameters

- **suffix** (*Optional[str]*) – If not None, the file name will end with this string. Otherwise, default value ".tmp" will be used
- **prefix** (*Optional[str]*) – If not None, the file name will start with this string. Otherwise, default value "tmp" will be used

Returns
Path to valid file

Return type
str

abstract writeGlobalFile(*localFileName*, *cleanup=False*)

Upload a file (as a path) to the job store.

If the file is in a FileStore-managed temporary directory (i.e. from *toil.fileStores.abstractFileStore.AbstractFileStore.getLocalTempDir()*), it will become a local copy of the file, eligible for deletion by *toil.fileStores.abstractFileStore.AbstractFileStore.deleteLocalFile()*.

If an executable file on the local filesystem is uploaded, its executability will be preserved when it is downloaded again.

Parameters

- **localFileName** (*str*) – The path to the local file to upload. The last path component (basename of the file) will remain associated with the file in the file store, if supported by the backing JobStore, so that the file can be searched for by name or name glob.
- **cleanup** (*bool*) – if True then the copy of the global file will be deleted once the job and all its successors have completed running. If not the global file must be deleted manually.

Returns
an ID that can be used to retrieve the file.

Return type
toil.fileStores.FileID

writeGlobalFileStream(*cleanup=False*, *basename=None*, *encoding=None*, *errors=None*)

Similar to writeGlobalFile, but allows the writing of a stream to the job store. The yielded file handle does not need to and should not be closed explicitly.

Parameters

- **encoding** (*Optional[str]*) – The name of the encoding used to decode the file. Encodings are the same as for decode(). Defaults to None which represents binary mode.
- **errors** (*Optional[str]*) – Specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

- **cleanup** (*bool*) – is as in *toil.fileStores.abstractFileStore.AbstractFileStore.writeGlobalFile()*.
- **basename** (*Optional[str]*) – If supported by the backing JobStore, use the given file basename so that when searching the job store with a query matching that basename, the file will be detected.

Returns
A context manager yielding a tuple of 1) a file handle which can be written to and 2) the toil.fileStores.FileID of the resulting file in the job store.

Return type
Iterator[Tuple[*toil.lib.io.WriteWatchingStream*, *toil.fileStores.FileID*]]

logAccess(*fileStoreID*, *destination=None*)

Record that the given file was read by the job.

(to be announced if the job fails)

If destination is not None, it gives the path that the file was downloaded to. Otherwise, assumes that the file was streamed.

Must be called by *readGlobalFile()* and *readGlobalFileStream()* implementations.

Parameters

- **fileStoreID** (*Union[toil.fileStores.FileID, str]*) –
- **destination** (*Union[str, None]*) –

Return type
None

abstract readGlobalFile(*fileStoreID*, *userPath=None*, *cache=True*, *mutable=False*, *symlink=False*)

Make the file associated with fileStoreID available locally.

If mutable is True, then a copy of the file will be created locally so that the original is not modified and does not change the file for other jobs. If mutable is False, then a link can be created to the file, saving disk resources. The file that is downloaded will be executable if and only if it was originally uploaded from an executable file on the local filesystem.

If a user path is specified, it is used as the destination. If a user path isn't specified, the file is stored in the local temp directory with an encoded name.

The destination file must not be deleted by the user; it can only be deleted through deleteLocalFile.

Implementations must call *logAccess()* to report the download.

Parameters

- **fileStoreID** (*str*) – job store id for the file
- **userPath** (*Optional[str]*) – a path to the name of file to which the global file will be copied or hard-linked (see below).
- **cache** (*bool*) – Described in `toil.fileStores.CachingFileStore.readGlobalFile()`
- **mutable** (*bool*) – Described in `toil.fileStores.CachingFileStore.readGlobalFile()`
- **symlink** (*bool*) – True if caller can accept symlink, False if caller can only accept a normal file or hardlink

Returns
An absolute path to a local, temporary copy of the file keyed by fileStoreID.

Return type
str

abstract readGlobalFileStream(*fileStoreID*, *encoding=None*, *errors=None*)

Read a stream from the job store; similar to readGlobalFile.

The yielded file handle does not need to and should not be closed explicitly.

Parameters

- **encoding** (*Optional[str]*) – the name of the encoding used to decode the file. Encodings are the same as for decode(). Defaults to None which represents binary mode.
- **errors** (*Optional[str]*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.
- **fileStoreID** (*str*) –

Return type
ContextManager[Union[IO[bytes], IO[str]]]

Implementations must call *logAccess()* to report the download.

Returns
a context manager yielding a file handle which can be read from.

Parameters

- **fileStoreID** (*str*) –
- **encoding** (*Optional[str]*) –
- **errors** (*Optional[str]*) –

Return type
ContextManager[Union[IO[bytes], IO[str]]]

getGlobalFileSize(*fileStoreID*)

Get the size of the file pointed to by the given ID, in bytes.

If a FileID or something else with a non-None 'size' field, gets that.

Otherwise, asks the job store to poll the file's size.

Note that the job store may overestimate the file's size, for example if it is encrypted and had to be augmented with an IV or other encryption framing.

Parameters
fileStoreID (*Union[toil.fileStores.FileID, str]*) – File ID for the file

Returns
File's size in bytes, as stored in the job store

Return type
int

abstract deleteLocalFile(*fileStoreID*)

Delete local copies of files associated with the provided job store ID.

Raises an OSError with an errno of errno.ENOENT if no such local copies exist. Thus, cannot be called multiple times in succession.

The files deleted are all those previously read from this file ID via readGlobalFile by the current job into the job's file-store-provided temp directory, plus the file that was written to create the given file ID, if it was written by the current job from the job's file-store-provided temp directory.

Parameters
fileStoreID (*Union[toil.fileStores.FileID, str]*) – File Store ID of the file to be deleted.

Return type
None

abstract deleteGlobalFile(*fileStoreID*)

Delete local files and then permanently deletes them from the job store.

To ensure that the job can be restarted if necessary, the delete will not happen until after the job's run method has completed.

Parameters
fileStoreID (*Union[toil.fileStores.FileID, str]*) – the File Store ID of the file to be deleted.

Return type
None

importFile(*srcUrl*, *sharedFileName=None*)

Parameters

- **srcUrl** (*str*) –
- **sharedFileName** (*Optional[str]*) –

Return type
Optional[*toil.fileStores.FileID*]

import_file(*src_uri*, *shared_file_name=None*)

Parameters

- **src_uri** (*str*) –
- **shared_file_name** (*Optional[str]*) –

Return type
Optional[*toil.fileStores.FileID*]

exportFile(*jobStoreFileID*, *dstUrl*)

Parameters

- **jobStoreFileID** (*toil.fileStores.FileID*) –
- **dstUrl** (*str*) –

Return type
None

abstract export_file(*file_id*, *dst_uri*)

Parameters

- **file_id** (*toil.fileStores.FileID*) –
- **dst_uri** (*str*) –

Return type
None

`logToMaster`(*text*, *level=logging.INFO*)

Send a logging message to the leader. The message will also be logged by the worker at the same level.

Parameters

- **text** (*str*) – The string to log.
- **level** (*int*) – The logging level.

Return type
None

`abstract startCommit`(*jobState=False*)

Update the status of the job on the disk.

May start an asynchronous process. Call waitForCommit() to wait on that process.

Parameters
jobState (*bool*) – If True, commit the state of the FileStore's job, and file deletes. Otherwise, commit only file creates/updates.

Return type
None

`abstract waitForCommit`()

Blocks while startCommit is running.

This function is called by this job's successor to ensure that it does not begin modifying the job store until after this job has finished doing so.

Might be called when startCommit is never called on a particular instance, in which case it does not block.

Returns
Always returns True

Return type
bool

`abstract classmethod shutdown`(*shutdown_info*)

Shutdown the filestore on this node.

This is intended to be called on batch system shutdown.

Parameters
shutdown_info (*Any*) – The implementation-specific shutdown information, for shutting down the file store and removing all its state and all job local temp directories from the node.

Return type
None

toil.fileStores.cachingFileStore

Module Contents

Classes

CachingFileStore	A cache-enabled file store.

Attributes

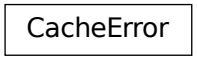
logger
SQLITE_TIMEOUT_SECS

toil.fileStores.cachingFileStore.**logger**

toil.fileStores.cachingFileStore.**SQLITE_TIMEOUT_SECS = 60.0**

exception toil.fileStores.cachingFileStore.**CacheError**(*message*)

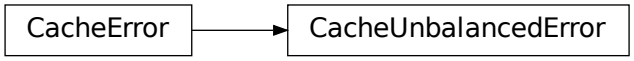
Bases: Exception



Error Raised if the user attempts to add a non-local file to cache

exception toil.fileStores.cachingFileStore.**CacheUnbalancedError**

Bases: *CacheError*



Raised if file store can't free enough space for caching

message = 'Unable unable to free enough space for caching. This error frequently arises due to jobs using...'

exception toil.fileStores.cachingFileStore.**IllegalDeletionCacheError**(*deletedFile*)

Bases: *CacheError*

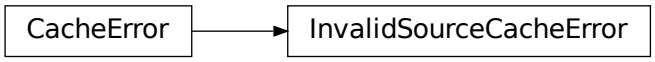
Error raised if the caching code discovers a file that represents a reference to a cached file to have gone missing.

This can be a big problem if a hard link is moved, because then the cache will be unable to evict the file it links to.

Remember that files read with readGlobalFile may not be deleted by the user and need to be deleted with deleteLocalFile.

exception toil.fileStores.cachingFileStore.**InvalidSourceCacheError**(*message*)

Bases: *CacheError*



Error raised if the user attempts to add a non-local file to cache

class toil.fileStores.cachingFileStore.**CachingFileStore**(*jobStore*, *jobDesc*, *file_store_dir*, *waitForPreviousCommit*)

Bases: *toil.fileStores.abstractFileStore.AbstractFileStore*

A cache-enabled file store.

Provides files that are read out as symlinks or hard links into a cache directory for the node, if permitted by the workflow.

Also attempts to write files back to the backing JobStore asynchronously, after quickly taking them into the cache. Writes are only required to finish when the job's actual state after running is committed back to the job store.

Internaly, manages caching using a database. Each node has its own database, shared between all the workers on the node. The database contains several tables:

files contains one entry for each file in the cache. Each entry knows the path to its data on disk. It also knows its global file ID, its state, and its owning worker PID. If the owning worker dies, another worker will pick it up. It also knows its size.

File states are:

- "cached": happily stored in the cache. Reads can happen immediately. Owner is null. May be adopted and moved to state "deleting" by anyone, if it has no outstanding immutable references.
- "downloading": in the process of being saved to the cache by a non-null owner. Reads must wait for the state to become "cached". If the worker dies, goes to state "deleting", because we don't know if it was fully downloaded or if anyone still needs it. No references can be created to a "downloading" file except by the worker responsible for downloading it.
- "uploadable": stored in the cache and ready to be written to the job store by a non-null owner. Transitions to "uploading" when a (thread of) the owning worker process picks it up and begins uploading it, to free cache space or to commit a completed job. If the worker dies, goes to state "cached", because it may have outstanding immutable references from the dead-but-not-cleaned-up job that was going to write it.
- "uploading": stored in the cache and being written to the job store by a non-null owner. Transitions to "cached" when successfully uploaded. If the worker dies, goes to state "cached", because it may have outstanding immutable references from the dead-but-not-cleaned-up job that was writing it.
- "deleting": in the process of being removed from the cache by a non-null owner. Will eventually be removed from the database.

refs contains one entry for each outstanding reference to a cached file (hard link, symlink, or full copy). The table name is refs instead of references because references is an SQL reserved word. It remembers what job ID has the reference, and the path the reference is at. References have three states:

- "immutable": represents a hardlink or symlink to a file in the cache. Dedicates the file's size in bytes of the job's disk requirement to the cache, to be used to cache this file or to keep around other files without references. May be upgraded to "copying" if the link can't actually be created.
- "copying": records that a file in the cache is in the process of being copied to a path. Will be upgraded to a mutable reference eventually.
- "mutable": records that a file from the cache was copied to a certain path. Exist only to support deleteLocalFile's API. Only files with only mutable references (or no references) are eligible for eviction.

jobs contains one entry for each job currently running. It keeps track of the job's ID, the worker that is supposed to be running the job, the job's disk requirement, and the job's local temp dir path that will need to be cleaned up. When workers check for jobs whose workers have died, they null out the old worker, and grab ownership of and clean up jobs and their references until the null-worker jobs are gone.

properties contains key, value pairs for tracking total space available, and whether caching is free for this run.

Parameters

- **jobStore** (*toil.jobStores.abstractJobStore.AbstractJobStore*) –
- **jobDesc** (*toil.job.JobDescription*) –
- **file_store_dir** (*str*) –
- **waitForPreviousCommit** (*Callable[[], Any]*) –

getCacheLimit()

Return the total number of bytes to which the cache is limited.

If no limit is available, raises an error.

getCacheUsed()

Return the total number of bytes used in the cache.

If no value is available, raises an error.

getCacheExtraJobSpace()

Return the total number of bytes of disk space requested by jobs running against this cache but not yet used.

We can get into a situation where the jobs on the node take up all its space, but then they want to write to or read from the cache. So when that happens, we need to debit space from them somehow…

If no value is available, raises an error.

getCacheAvailable()

Return the total number of free bytes available for caching, or, if negative, the total number of bytes of cached files that need to be evicted to free up enough space for all the currently scheduled jobs.

If no value is available, raises an error.

getSpaceUsableForJobs()

Return the total number of bytes that are not taken up by job requirements, ignoring files and file usage. We can't ever run more jobs than we actually have room for, even with caching.

If not retrievable, raises an error.

getCacheUnusedJobRequirement()

Return the total number of bytes of disk space requested by the current job and not used by files the job is using in the cache.

Mutable references don't count, but immutable/uploading ones do.

If no value is available, raises an error.

adjustCacheLimit(*newTotalBytes*)

Adjust the total cache size limit to the given number of bytes.

fileIsCached(*fileID*)

Return true if the given file is currently cached, and false otherwise.

Note that this can't really be relied upon because a file may go cached -> deleting after you look at it. If you need to do something with the file you need to do it in a transaction.

getFileReaderCount(*fileID*)

Return the number of current outstanding reads of the given file.

Counts mutable references too.

cachingIsFree()

Return true if files can be cached for free, without taking up space. Return false otherwise.

This will be true when working with certain job stores in certain configurations, most notably the FileJobStore.

open(*job*)

This context manager decorated method allows cache-specific operations to be conducted before and after the execution of a job in worker.py

Parameters
job (toil.job.Job) –

Return type
Generator[None, None, None]

writeGlobalFile(*localFileName*, *cleanup=False*, *executable=False*)

Creates a file in the jobstore and returns a FileID reference.

readGlobalFile(*fileStoreID*, *userPath=None*, *cache=True*, *mutable=False*, *symlink=False*)

Make the file associated with fileStoreID available locally.

If mutable is True, then a copy of the file will be created locally so that the original is not modified and does not change the file for other jobs. If mutable is False, then a link can be created to the file, saving disk resources. The file that is downloaded will be executable if and only if it was originally uploaded from an executable file on the local filesystem.

If a user path is specified, it is used as the destination. If a user path isn't specified, the file is stored in the local temp directory with an encoded name.

The destination file must not be deleted by the user; it can only be deleted through deleteLocalFile.

Implementations must call `logAccess()` to report the download.

Parameters

- **fileStoreID** – job store id for the file
- **userPath** – a path to the name of file to which the global file will be copied or hard-linked (see below).
- **cache** – Described in `toil.fileStores.CachingFileStore.readGlobalFile()`
- **mutable** – Described in `toil.fileStores.CachingFileStore.readGlobalFile()`
- **symlink** – True if caller can accept symlink, False if caller can only accept a normal file or hardlink

Returns

An absolute path to a local, temporary copy of the file keyed by fileStoreID.

readGlobalFileStream(*fileStoreID*, *encoding=None*, *errors=None*)

Read a stream from the job store; similar to readGlobalFile.

The yielded file handle does not need to and should not be closed explicitly.

Parameters

- **encoding** – the name of the encoding used to decode the file. Encodings are the same as for decode(). Defaults to None which represents binary mode.
- **errors** – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Implementations must call `logAccess()` to report the download.

Returns

a context manager yielding a file handle which can be read from.

deleteLocalFile(*fileStoreID*)

Delete local copies of files associated with the provided job store ID.

Raises an OSError with an errno of errno.ENOENT if no such local copies exist. Thus, cannot be called multiple times in succession.

The files deleted are all those previously read from this file ID via readGlobalFile by the current job into the job's file-store-provided temp directory, plus the file that was written to create the given file ID, if it was written by the current job from the job's file-store-provided temp directory.

Parameters

fileStoreID – File Store ID of the file to be deleted.

deleteGlobalFile(*fileStoreID*)

Delete local files and then permanently deletes them from the job store.

To ensure that the job can be restarted if necessary, the delete will not happen until after the job's run method has completed.

Parameters
fileStoreID – the File Store ID of the file to be deleted.

exportFile(*jobStoreFileID*, *dstUrl*)

Parameters

- **jobStoreFileID** (toil.fileStores.FileID) –
- **dstUrl** (*str*) –

Return type
None

export_file(*file_id*, *dst_uri*)

Parameters

- **file_id** (toil.fileStores.FileID) –
- **dst_uri** (*str*) –

Return type
None

waitForCommit()

Blocks while startCommit is running.

This function is called by this job's successor to ensure that it does not begin modifying the job store until after this job has finished doing so.

Might be called when startCommit is never called on a particular instance, in which case it does not block.

Returns
Always returns True

Return type
bool

startCommit(*jobState=False*)

Update the status of the job on the disk.

May start an asynchronous process. Call waitForCommit() to wait on that process.

Parameters
jobState – If True, commit the state of the FileStore's job, and file deletes. Otherwise, commit only file creates/updates.

startCommitThread(*jobState*)

Run in a thread to actually commit the current job.

classmethod shutdown(*shutdown_info*)

Parameters
shutdown_info (*Tuple[str, str]*) – Tuple of the coordination directory (where the cache database is) and the cache directory (where the cached data is).

Return type
None

Job local temp directories will be removed due to their appearance in the database.

`__del__()`

Cleanup function that is run when destroying the class instance that ensures that all the file writing threads exit.

toil.fileStores.nonCachingFileStore

Module Contents

Classes

NonCachingFileStore	Interface used to allow user code run by Toil to read and write files.

Attributes

logger

toil.fileStores.nonCachingFileStore.**logger**: logging.Logger

class toil.fileStores.nonCachingFileStore.**NonCachingFileStore**(*jobStore*, *jobDesc*, *file_store_dir*, *waitForPreviousCommit*)

Bases: *toil.fileStores.abstractFileStore.AbstractFileStore*

Interface used to allow user code run by Toil to read and write files.

Also provides the interface to other Toil facilities used by user code, including:

- normal (non-real-time) logging
- finding the correct temporary directory for scratch work
- importing and exporting files into and out of the workflow

Stores user files in the jobStore, but keeps them separate from actual jobs.

May implement caching.

Passed as argument to the *toil.job.Job.run()* method.

Access to files is only permitted inside the context manager provided by *toil.fileStores.abstractFileStore.AbstractFileStore.open()*.

Also responsible for committing completed jobs back to the job store with an update operation, and allowing that commit operation to be waited for.

Parameters

- **jobStore** (toil.jobStores.abstractJobStore.AbstractJobStore) –
- **jobDesc** (toil.job.JobDescription) –
- **file_store_dir** (*str*) –
- **waitForPreviousCommit** (*Callable[[], Any]*) –

static check_for_coordination_corruption(*coordination_dir*)

Make sure the coordination directory hasn't been deleted unexpectedly.

Slurm has been known to delete XDG_RUNTIME_DIR out from under processes it was promised to, so it is possible that in certain misconfigured environments the coordination directory and everything in it could go away unexpectedly. We are going to regularly make sure that the things we think should exist actually exist, and we are going to abort if they do not.

Parameters
coordination_dir (*Optional[str]*) –

Return type
None

check_for_state_corruption()

Make sure state tracking information hasn't been deleted unexpectedly.

Return type
None

open(*job*)

Create the context manager around tasks prior and after a job has been run.

File operations are only permitted inside the context manager.

Implementations must only yield from within *with super().open(job):*.

Parameters
job (toil.job.Job) – The job instance of the toil job to run.

Return type
Generator[None, None, None]

writeGlobalFile(*localFileName*, *cleanup=False*)

Upload a file (as a path) to the job store.

If the file is in a FileStore-managed temporary directory (i.e. from *toil.fileStores.abstractFileStore.AbstractFileStore.getLocalTempDir()*), it will become a local copy of the file, eligible for deletion by *toil.fileStores.abstractFileStore.AbstractFileStore.deleteLocalFile()*.

If an executable file on the local filesystem is uploaded, its executability will be preserved when it is downloaded again.

Parameters

- **localFileName** (*str*) – The path to the local file to upload. The last path component (basename of the file) will remain associated with the file in the file store, if supported by the backing JobStore, so that the file can be searched for by name or name glob.
- **cleanup** (*bool*) – if True then the copy of the global file will be deleted once the job and all its successors have completed running. If not the global file must be deleted manually.

Returns
an ID that can be used to retrieve the file.

Return type
toil.fileStores.FileID

readGlobalFile(*fileStoreID*, *userPath=None*, *cache=True*, *mutable=False*, *symlink=False*)

Make the file associated with fileStoreID available locally.

If mutable is True, then a copy of the file will be created locally so that the original is not modified and does not change the file for other jobs. If mutable is False, then a link can be created to the file, saving disk resources. The file that is downloaded will be executable if and only if it was originally uploaded from an executable file on the local filesystem.

If a user path is specified, it is used as the destination. If a user path isn't specified, the file is stored in the local temp directory with an encoded name.

The destination file must not be deleted by the user; it can only be deleted through deleteLocalFile.

Implementations must call `logAccess()` to report the download.

Parameters

- **fileStoreID** (*str*) – job store id for the file
- **userPath** (*Optional[str]*) – a path to the name of file to which the global file will be copied or hard-linked (see below).
- **cache** (*bool*) – Described in `toil.fileStores.CachingFileStore.readGlobalFile()`
- **mutable** (*bool*) – Described in `toil.fileStores.CachingFileStore.readGlobalFile()`
- **symlink** (*bool*) – True if caller can accept symlink, False if caller can only accept a normal file or hardlink

Returns
An absolute path to a local, temporary copy of the file keyed by fileStoreID.

Return type
str

readGlobalFileStream(*fileStoreID*, *encoding=None*, *errors=None*)

Read a stream from the job store; similar to readGlobalFile.

The yielded file handle does not need to and should not be closed explicitly.

Parameters

- **encoding** (*Optional[str]*) – the name of the encoding used to decode the file. Encodings are the same as for decode(). Defaults to None which represents binary mode.
- **errors** (*Optional[str]*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

- **fileStoreID** (*str*) –

Return type
Iterator[Union[IO[bytes], IO[str]]]

Implementations must call `logAccess()` to report the download.

Returns
a context manager yielding a file handle which can be read from.

Parameters

- **fileStoreID** (*str*) –
- **encoding** (*Optional[str]*) –
- **errors** (*Optional[str]*) –

Return type
Iterator[Union[IO[bytes], IO[str]]]

exportFile(*jobStoreFileID*, *dstUrl*)

Parameters

- **jobStoreFileID** (toil.fileStores.FileID) –
- **dstUrl** (*str*) –

Return type
None

export_file(*file_id*, *dst_uri*)

Parameters

- **file_id** (toil.fileStores.FileID) –
- **dst_uri** (*str*) –

Return type
None

deleteLocalFile(*fileStoreID*)

Delete local copies of files associated with the provided job store ID.

Raises an OSError with an errno of errno.ENOENT if no such local copies exist. Thus, cannot be called multiple times in succession.

The files deleted are all those previously read from this file ID via readGlobalFile by the current job into the job's file-store-provided temp directory, plus the file that was written to create the given file ID, if it was written by the current job from the job's file-store-provided temp directory.

Parameters
fileStoreID (*str*) – File Store ID of the file to be deleted.

Return type
None

deleteGlobalFile(*fileStoreID*)

Delete local files and then permanently deletes them from the job store.

To ensure that the job can be restarted if necessary, the delete will not happen until after the job's run method has completed.

Parameters
fileStoreID (*str*) – the File Store ID of the file to be deleted.

Return type
None

waitForCommit()

Blocks while startCommit is running.

This function is called by this job's successor to ensure that it does not begin modifying the job store until after this job has finished doing so.

Might be called when startCommit is never called on a particular instance, in which case it does not block.

Returns
Always returns True

Return type
bool

startCommit(*jobState=False*)

Update the status of the job on the disk.

May start an asynchronous process. Call waitForCommit() to wait on that process.

Parameters
jobState (*bool*) – If True, commit the state of the FileStore's job, and file deletes. Otherwise, commit only file creates/updates.

Return type
None

__del__()

Cleanup function that is run when destroying the class instance. Nothing to do since there are no async write events.

Return type
None

classmethod shutdown(*shutdown_info*)

Parameters
shutdown_info (*str*) – The coordination directory.

Return type
None

Package Contents

Classes

FileID	A small wrapper around Python's builtin string class.

class toil.fileStores.**FileID**(*fileStoreID*, *size*, *executable=False*)

Bases: str

A small wrapper around Python's builtin string class.

It is used to represent a file's ID in the file store, and has a size attribute that is the file's size in bytes. This object is returned by importFile and writeGlobalFile.

Calls into the file store can use bare strings; size will be queried from the job store if unavailable in the ID.

Parameters

- **fileStoreID** (*str*) –
- **size** (*int*) –
- **executable** (*bool*) –

pack()

Pack the FileID into a string so it can be passed through external code.

Return type
str

classmethod forPath(*fileStoreID*, *filePath*)

Parameters

- **fileStoreID** (*str*) –
- **filePath** (*str*) –

Return type
FileID

classmethod unpack(*packedFileStoreID*)

Unpack the result of pack() into a FileID object.

Parameters
packedFileStoreID (*str*) –

Return type
FileID

toil.jobStores

Subpackages

toil.jobStores.aws

Submodules

toil.jobStores.aws.jobStore

Module Contents

Classes

AWSJobStore	A job store that uses Amazon's S3 for file storage and SimpleDB for storing job info and

Attributes

boto3_session
s3_boto3_resource
s3_boto3_client
logger
CONSISTENCY_TICKS
CONSISTENCY_TIME
aRepr
custom_repr

toil.jobStores.aws.jobStore.**boto3_session**

toil.jobStores.aws.jobStore.**s3_boto3_resource**

toil.jobStores.aws.jobStore.**s3_boto3_client**

toil.jobStores.aws.jobStore.**logger**

toil.jobStores.aws.jobStore.**CONSISTENCY_TICKS = 5**

toil.jobStores.aws.jobStore.**CONSISTENCY_TIME = 1**

exception toil.jobStores.aws.jobStore.**ChecksumError**

Bases: Exception

Raised when a download from AWS does not contain the correct data.

class toil.jobStores.aws.jobStore.**AWSJobStore**(*locator*, *partSize=50 << 20*)

Bases: *toil.jobStores.abstractJobStore.AbstractJobStore*

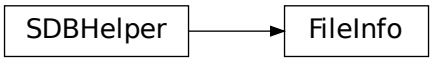
A job store that uses Amazon's S3 for file storage and SimpleDB for storing job info and enforcing strong consistency on the S3 file storage. There will be SDB domains for jobs and files and a versioned S3 bucket for file contents. Job objects are pickled, compressed, partitioned into chunks of 1024 bytes and each chunk is stored as a an attribute of the SDB item representing the job. UUIDs are used to identify jobs and files.

Parameters

- **locator** (*str*) –
- **partSize** (*int*) –

class **FileInfo**(*fileID*, *ownerID*, *encrypted*, *version=None*, *content=None*, *numContentChunks=0*, *checksum=None*)

Bases: *toil.jobStores.aws.utils.SDBHelper*



Represents a file in this job store.

property fileID

property ownerID

property version

property previousVersion

property content

property checksum

outer

Type
AWSJobStore

classmethod create(*ownerID*)

classmethod presenceIndicator()

The key that is guaranteed to be present in the return value of binaryToAttributes(). Assuming that binaryToAttributes() is used with SDB's PutAttributes, the return value of this method could be used to detect the presence/absence of an item in SDB.

classmethod exists(*jobStoreFileID*)

classmethod load(*jobStoreFileID*)

classmethod loadOrCreate(*jobStoreFileID*, *ownerID*, *encrypted*)

classmethod loadOrFail(*jobStoreFileID*, *customName=None*)

Return type
AWSJobStore.FileInfo

Returns
an instance of this class representing the file with the given ID

Raises
NoSuchFileException – if given file does not exist

classmethod fromItem(*item*)

Convert an SDB item to an instance of this class.

toItem()

Convert this instance to an attribute dictionary suitable for SDB put_attributes().

Return type
(dict,int)

Returns
the attributes dict and an integer specifying the the number of chunk attributes in the dictionary that are used for storing inlined content.

static maxInlinedSize()

save()

upload(*localFilePath*, *calculateChecksum=True*)

uploadStream(*multipart=True*, *allowInlining=True*, *encoding=None*, *errors=None*)

Context manager that gives out a binary or text mode upload stream to upload data.

copyFrom(*srcObj*)

Copies contents of source key into this file.

Parameters
srcObj (*S3.Object*) – The key (object) that will be copied from

copyTo(*dstObj*)

Copies contents of this file to the given key.

Parameters
dstObj (*S3.Object*) – The key (object) to copy this file's content to

download(*localFilePath*, *verifyChecksum=True*)

downloadStream(*verifyChecksum=True*, *encoding=None*, *errors=None*)

Context manager that gives out a download stream to download data.

delete()

getSize()

Return the size of the referenced item in bytes.

__repr__()

Return repr(self).

property sseKeyPath

bucketNameRe

minBucketNameLen = 3

maxBucketNameLen = 63

maxNameLen = 10

nameSeparator = '--'

jobsPerBatchInsert = 25

itemsPerBatchDelete = 25

sharedFileOwnerID

statsFileOwnerID

readStatsFileOwnerID

versionings

initialize(*config*)

Initialize this job store.

Create the physical storage for this job store, allocate a workflow ID and persist the given Toil configuration to the store.

Parameters

config – the Toil configuration to initialize this job store with. The given configuration will be updated with the newly allocated workflow ID.

Raises

JobStoreExistsException – if the physical storage for this job store already exists

resume()

Connect this instance to the physical storage it represents and load the Toil configuration into the AbstractJobStore.config attribute.

Raises

NoSuchJobStoreException – if the physical storage for this job store doesn't exist

batch()

If supported by the batch system, calls to create() with this context manager active will be performed in a batch after the context manager is released.

assign_job_id(*job_description*)

Get a new jobStoreID to be used by the described job, and assigns it to the JobDescription.

Files associated with the assigned ID will be accepted even if the JobDescription has never been created or updated.

Parameters
job_description (toil.job.JobDescription) – The JobDescription to give an ID to

create_job(*job_description*)

Writes the given JobDescription to the job store. The job must have an ID assigned already.

Must call jobDescription.pre_update_hook()

Returns
The JobDescription passed.

Return type
toil.job.JobDescription

job_exists(*job_id*)

Indicates whether a description of the job with the specified jobStoreID exists in the job store

Return type
bool

jobs()

Best effort attempt to return iterator on JobDescriptions for all jobs in the store. The iterator may not return all jobs and may also contain orphaned jobs that have already finished successfully and should not be rerun. To guarantee you get any and all jobs that can be run instead construct a more expensive ToilState object

Returns
Returns iterator on jobs in the store. The iterator may or may not contain all jobs and may contain invalid jobs

Return type
Iterator[toil.job.jobDescription]

load_job(*job_id*)

Loads the description of the job referenced by the given ID, assigns it the job store's config, and returns it.

May declare the job to have failed (see *toil.job.JobDescription.setupJobAfterFailure()*) if there is evidence of a failed update attempt.

Parameters
job_id – the ID of the job to load

Raises
NoSuchJobException – if there is no job with the given ID

update_job(*job_description*)

Persists changes to the state of the given JobDescription in this store atomically.

Must call jobDescription.pre_update_hook()

Parameters
job (toil.job.JobDescription) – the job to write to this job store

delete_job(*job_id*)

Removes the JobDescription from the store atomically. You may not then subsequently call load(), write(), update(), etc. with the same jobStoreID or any JobDescription bearing it.

This operation is idempotent, i.e. deleting a job twice or deleting a non-existent job will succeed silently.

Parameters
job_id (*str*) – the ID of the job to delete from this job store

get_empty_file_store_id(*jobStoreID=None*, *cleanup=False*, *basename=None*)

Creates an empty file in the job store and returns its ID. Call to fileExists(getEmptyFileStoreID(jobStoreID)) will return True.

Parameters

- **job_id** (*str*) – the id of a job, or None. If specified, the may be associated with that job in a job-store-specific way. This may influence the returned ID.
- **cleanup** (*bool*) – Whether to attempt to delete the file when the job whose jobStoreID was given as jobStoreID is deleted with jobStore.delete(job). If jobStoreID was not given, does nothing.
- **basename** (*str*) – If supported by the implementation, use the given file basename so that when searching the job store with a query matching that basename, the file will be detected.

Returns

a jobStoreFileID that references the newly created file and can be used to reference the file in the future.

Return type

str

classmethod get_size(*url*)

Get the size in bytes of the file at the given URL, or None if it cannot be obtained.

Parameters

src_uri – URL that points to a file or object in the storage mechanism of a supported URL scheme e.g. a blob in an AWS s3 bucket.

write_file(*local_path*, *job_id=None*, *cleanup=False*)

Takes a file (as a path) and places it in this job store. Returns an ID that can be used to retrieve the file at a later time. The file is written in a atomic manner. It will not appear in the jobStore until the write has successfully completed.

Parameters

- **local_path** (*str*) – the path to the local file that will be uploaded to the job store. The last path component (basename of the file) will remain associated with the file in the file store, if supported, so that the file can be searched for by name or name glob.
- **job_id** (*str*) – the id of a job, or None. If specified, the may be associated with that job in a job-store-specific way. This may influence the returned ID.
- **cleanup** (*bool*) – Whether to attempt to delete the file when the job whose jobStoreID was given as jobStoreID is deleted with jobStore.delete(job). If jobStoreID was not given, does nothing.

Raises

- *ConcurrentFileModificationException* – if the file was modified concurrently during an invocation of this method
- *NoSuchJobException* – if the job specified via jobStoreID does not exist

FIXME: some implementations may not raise this

Returns

an ID referencing the newly created file and can be used to read the file in the future.

Return type

str

write_file_stream(*job_id=None*, *cleanup=False*, *basename=None*, *encoding=None*, *errors=None*)

Similar to writeFile, but returns a context manager yielding a tuple of 1) a file handle which can be written to and 2) the ID of the resulting file in the job store. The yielded file handle does not need to and should not be closed explicitly. The file is written in a atomic manner. It will not appear in the jobStore until the write has successfully completed.

Parameters

- **job_id** (*str*) – the id of a job, or None. If specified, the may be associated with that job in a job-store-specific way. This may influence the returned ID.
- **cleanup** (*bool*) – Whether to attempt to delete the file when the job whose jobStoreID was given as jobStoreID is deleted with jobStore.delete(job). If jobStoreID was not given, does nothing.
- **basename** (*str*) – If supported by the implementation, use the given file basename so that when searching the job store with a query matching that basename, the file will be detected.
- **encoding** (*str*) – the name of the encoding used to encode the file. Encodings are the same as for encode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Raises

- *ConcurrentFileModificationException* – if the file was modified concurrently during an invocation of this method
- *NoSuchJobException* – if the job specified via jobStoreID does not exist

FIXME: some implementations may not raise this

Returns

a context manager yielding a file handle which can be written to and an ID that references the newly created file and can be used to read the file in the future.

Return type

Iterator[Tuple[IO[bytes], str]]

write_shared_file_stream(*shared_file_name*, *encrypted=None*, *encoding=None*, *errors=None*)

Returns a context manager yielding a writable file handle to the global file referenced by the given name. File will be created in an atomic manner.

Parameters

- **shared_file_name** (*str*) – A file name matching AbstractJobStore.fileNameRegex, unique within this job store
- **encrypted** (*bool*) – True if the file must be encrypted, None if it may be encrypted or False if it must be stored in the clear.
- **encoding** (*str*) – the name of the encoding used to encode the file. Encodings are the same as for encode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Raises

ConcurrentFileModificationException – if the file was modified concurrently during an invocation of this method

Returns
a context manager yielding a writable file handle

Return type
Iterator[IO[bytes]]

update_file(*file_id*, *local_path*)

Replaces the existing version of a file in the job store.

Throws an exception if the file does not exist.

Parameters

- **file_id** – the ID of the file in the job store to be updated
- **local_path** – the local path to a file that will overwrite the current version in the job store

Raises

- *ConcurrentFileModificationException* – if the file was modified concurrently during an invocation of this method
- *NoSuchFileException* – if the specified file does not exist

update_file_stream(*file_id*, *encoding=None*, *errors=None*)

Replaces the existing version of a file in the job store. Similar to writeFile, but returns a context manager yielding a file handle which can be written to. The yielded file handle does not need to and should not be closed explicitly.

Parameters

- **file_id** (*str*) – the ID of the file in the job store to be updated
- **encoding** (*str*) – the name of the encoding used to encode the file. Encodings are the same as for encode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Raises

- *ConcurrentFileModificationException* – if the file was modified concurrently during an invocation of this method
- *NoSuchFileException* – if the specified file does not exist

file_exists(*file_id*)

Determine whether a file exists in this job store.

Parameters
file_id – an ID referencing the file to be checked

get_file_size(*file_id*)

Get the size of the given file in bytes, or 0 if it does not exist when queried.

Note that job stores which encrypt files might return overestimates of file sizes, since the encrypted file may have been padded to the nearest block, augmented with an initialization vector, etc.

Parameters
file_id (*str*) – an ID referencing the file to be checked

Return type
int

read_file(*file_id*, *local_path*, *symlink=False*)

Copies or hard links the file referenced by jobStoreFileID to the given local file path. The version will be consistent with the last copy of the file written/updated. If the file in the job store is later modified via updateFile or updateFileStream, it is implementation-defined whether those writes will be visible at localFilePath. The file is copied in an atomic manner. It will not appear in the local file system until the copy has completed.

The file at the given local path may not be modified after this method returns!

Note! Implementations of readFile need to respect/provide the executable attribute on FileIDs.

Parameters

- **file_id** (*str*) – ID of the file to be copied
- **local_path** (*str*) – the local path indicating where to place the contents of the given file in the job store
- **symlink** (*bool*) – whether the reader can tolerate a symlink. If set to true, the job store may create a symlink instead of a full copy of the file or a hard link.

read_file_stream(*file_id*, *encoding=None*, *errors=None*)

Similar to readFile, but returns a context manager yielding a file handle which can be read from. The yielded file handle does not need to and should not be closed explicitly.

Parameters

- **file_id** (*str*) – ID of the file to get a readable file handle for
- **encoding** (*str*) – the name of the encoding used to decode the file. Encodings are the same as for decode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Returns
a context manager yielding a file handle which can be read from

Return type
Iterator[Union[IO[bytes], IO[str]]]

read_shared_file_stream(*shared_file_name*, *encoding=None*, *errors=None*)

Returns a context manager yielding a readable file handle to the global file referenced by the given name.

Parameters

- **shared_file_name** (*str*) – A file name matching AbstractJobStore.fileNameRegex, unique within this job store
- **encoding** (*str*) – the name of the encoding used to decode the file. Encodings are the same as for decode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Returns
a context manager yielding a readable file handle

Return type
Iterator[IO[bytes]]

delete_file(*file_id*)

Deletes the file with the given ID from this job store. This operation is idempotent, i.e. deleting a file twice or deleting a non-existent file will succeed silently.

Parameters
file_id (*str*) – ID of the file to delete

write_logs(*msg*)

Stores a message as a log in the jobstore.

Parameters
msg (*str*) – the string to be written

Raises
ConcurrentFileModificationException – if the file was modified concurrently during an invocation of this method

read_logs(*callback*, *read_all=False*)

Reads logs accumulated by the write_logs() method. For each log this method calls the given callback function with the message as an argument (rather than returning logs directly, this method must be supplied with a callback which will process log messages).

Only unread logs will be read unless the read_all parameter is set.

Parameters

- **callback** (*Callable*) – a function to be applied to each of the stats file handles found
- **read_all** (*bool*) – a boolean indicating whether to read the already processed stats files in addition to the unread stats files

Raises
ConcurrentFileModificationException – if the file was modified concurrently during an invocation of this method

Returns
the number of stats files processed

Return type
int

get_public_url(*jobStoreFileID*)

Returns a publicly accessible URL to the given file in the job store. The returned URL may expire as early as 1h after its been returned. Throw an exception if the file does not exist.

Parameters
file_name (*str*) – the jobStoreFileID of the file to generate a URL for

Raises
NoSuchFileException – if the specified file does not exist in this job store

Return type
str

get_shared_public_url(*shared_file_name*)

Differs from `getPublicUrl()` in that this method is for generating URLs for shared files written by `writeSharedFileStream()`.

Returns a publicly accessible URL to the given file in the job store. The returned URL starts with 'http:', 'https:' or 'file:'. The returned URL may expire as early as 1h after its been returned. Throw an exception if the file does not exist.

Parameters
shared_file_name (*str*) – The name of the shared file to generate a publically accessible url for.

Raises
NoSuchFileException – raised if the specified file does not exist in the store

Return type
str

destroy()

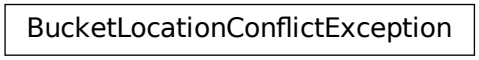
The inverse of *initialize()*, this method deletes the physical storage represented by this instance. While not being atomic, this method *is* at least idempotent, as a means to counteract potential issues with eventual consistency exhibited by the underlying storage mechanisms. This means that if the method fails (raises an exception), it may (and should be) invoked again. If the underlying storage mechanism is eventually consistent, even a successful invocation is not an ironclad guarantee that the physical storage vanished completely and immediately. A successful invocation only guarantees that the deletion will eventually happen. It is therefore recommended to not immediately reuse the same job store location for a new Toil workflow.

toil.jobStores.aws.jobStore.**aRepr**

toil.jobStores.aws.jobStore.**custom_repr**

exception toil.jobStores.aws.jobStore.**BucketLocationConflictException**(*bucketRegion*)

Bases: Exception



Common base class for all non-exit exceptions.

toil.jobStores.aws.utils

Module Contents

Classes

SDBHelper	A mixin with methods for storing limited amounts of binary data in an SDB item

Functions

fileSizeAndTime(localFilePath)	
uploadFromPath(localFilePath, resource, bucketName, fileID)	Uploads a file to s3, using multipart uploading if applicable
uploadFile(readable, resource, bucketName, fileID[, ...])	Upload a readable object to s3, using multipart uploading if applicable.
copyKeyMultipart(resource, srcBucketName, srcKeyName, ...)	Copies a key from a source key to a destination key in multiple parts. Note that if the
monkeyPatchSdbConnection(sdb)	**type sdb** SDBConnection
sdb_unavailable(e)	
no_such_sdb_domain(e)	
retryable_ssl_error(e)	
retryable_sdb_errors(e)	
retry_sdb([delays, timeout, predicate])	

Attributes

logger
DIAL_SPECIFIC_REGION_CONFIG

toil.jobStores.aws.utils.**logger**

toil.jobStores.aws.utils.**DIAL_SPECIFIC_REGION_CONFIG**

class toil.jobStores.aws.utils.**SDBHelper**

A mixin with methods for storing limited amounts of binary data in an SDB item

```
>>> import os
>>> H=SDBHelper
>>> H.presenceIndicator()
u'numChunks'
>>> H.binaryToAttributes(None)['numChunks']
0
>>> H.attributesToBinary({u'numChunks': 0})
(None, 0)
>>> H.binaryToAttributes(b'')
{u'000': b'VQ==', u'numChunks': 1}
>>> H.attributesToBinary({u'numChunks': 1, u'000': b'VQ=='})
(b'', 1)
```

Good pseudo-random data is very likely smaller than its bzip2ed form. Subtract 1 for the type character, i.e 'C' or 'U', with which the string is prefixed. We should get one full chunk:

```
>>> s = os.urandom(H.maxRawValueSize-1)
>>> d = H.binaryToAttributes(s)
>>> len(d), len(d['000'])
(2, 1024)
>>> H.attributesToBinary(d) == (s, 1)
True
```

One byte more and we should overflow four bytes into the second chunk, two bytes for base64-encoding the additional character and two bytes for base64-padding to the next quartet.

```
>>> s += s[0:1]
>>> d = H.binaryToAttributes(s)
>>> len(d), len(d['000']), len(d['001'])
(3, 1024, 4)
>>> H.attributesToBinary(d) == (s, 2)
True
```

maxAttributesPerItem = 256

maxValueSize = 1024

maxRawValueSize

classmethod maxBinarySize(*extraReservedChunks=0*)

classmethod binaryToAttributes(*binary*)

Turn a bytestring, or None, into SimpleDB attributes.

classmethod presenceIndicator()

The key that is guaranteed to be present in the return value of binaryToAttributes(). Assuming that binaryToAttributes() is used with SDB's PutAttributes, the return value of this method could be used to detect the presence/absence of an item in SDB.

classmethod attributesToBinary(*attributes*)

Return type
(str|None,int)

Returns
the binary data and the number of chunks it was composed from

toil.jobStores.aws.utils.**fileSizeAndTime**(*localFilePath*)

toil.jobStores.aws.utils.**uploadFromPath**(*localFilePath*, *resource*, *bucketName*, *fileID*, *headerArgs=None*, *partSize=50 << 20*)

Uploads a file to s3, using multipart uploading if applicable

Parameters

- **localFilePath** (*str*) – Path of the file to upload to s3
- **resource** (*S3.Resource*) – boto3 resource
- **bucketName** (*str*) – name of the bucket to upload to
- **fileID** (*str*) – the name of the file to upload to

- **headerArgs** (*dict*) – http headers to use when uploading - generally used for encryption purposes
- **partSize** (*int*) – max size of each part in the multipart upload, in bytes

Returns
version of the newly uploaded file

toil.jobStores.aws.utils.**uploadFile**(*readable*, *resource*, *bucketName*, *fileID*, *headerArgs=None*, *partSize=50 << 20*)

Upload a readable object to s3, using multipart uploading if applicable. :param readable: a readable stream or a file path to upload to s3 :param S3.Resource resource: boto3 resource :param str bucketName: name of the bucket to upload to :param str fileID: the name of the file to upload to :param dict headerArgs: http headers to use when uploading - generally used for encryption purposes :param int partSize: max size of each part in the multipart upload, in bytes :return: version of the newly uploaded file

Parameters

- **bucketName** (*str*) –
- **fileID** (*str*) –
- **headerArgs** (*Optional[dict]*) –
- **partSize** (*int*) –

exception toil.jobStores.aws.utils.**ServerSideCopyProhibitedError**

Bases: RuntimeError

Raised when AWS refuses to perform a server-side copy between S3 keys, and insists that you pay to download and upload the data yourself instead.

toil.jobStores.aws.utils.**copyKeyMultipart**(*resource*, *srcBucketName*, *srcKeyName*, *srcKeyVersion*, *dstBucketName*, *dstKeyName*, *sseAlgorithm=None*, *sseKey=None*, *copySourceSseAlgorithm=None*, *copySourceSseKey=None*)

Copies a key from a source key to a destination key in multiple parts. Note that if the destination key exists it will be overwritten implicitly, and if it does not exist a new key will be created. If the destination bucket does not exist an error will be raised.

This function will always do a fast, server-side copy, at least until/unless <https://github.com/boto/boto3/issues/3270> is fixed. In some situations, a fast, server-side copy is not actually possible. For example, when residing in an AWS VPC with an S3 VPC Endpoint configured, copying from a bucket in another region to a bucket in your own region cannot be performed server-side. This is because the VPC Endpoint S3 API servers refuse to perform server-side copies between regions, the source region's API servers refuse to initiate the copy and refer you to the destination bucket's region's API servers, and the VPC routing tables are configured to redirect all access to the current region's S3 API servers to the S3 Endpoint API servers instead.

If a fast server-side copy is not actually possible, a ServerSideCopyProhibitedError will be raised.

Parameters

- **resource** (*mypy_boto3_s3.S3ServiceResource*) – boto3 resource
- **srcBucketName** (*str*) – The name of the bucket to be copied from.
- **srcKeyName** (*str*) – The name of the key to be copied from.
- **srcKeyVersion** (*str*) – The version of the key to be copied from.
- **dstBucketName** (*str*) – The name of the destination bucket for the copy.
- **dstKeyName** (*str*) – The name of the destination key that will be created or overwritten.
- **sseAlgorithm** (*str*) – Server-side encryption algorithm for the destination.
- **sseKey** (*str*) – Server-side encryption key for the destination.
- **copySourceSseAlgorithm** (*str*) – Server-side encryption algorithm for the source.
- **copySourceSseKey** (*str*) – Server-side encryption key for the source.

Return type
str

Returns
The version of the copied file (or None if versioning is not enabled for dstBucket).

toil.jobStores.aws.utils.**monkeyPatchSdbConnection**(*sdb*)

toil.jobStores.aws.utils.**sdb_unavailable**(*e*)

toil.jobStores.aws.utils.**no_such_sdb_domain**(*e*)

toil.jobStores.aws.utils.**retryable_ssl_error**(*e*)

toil.jobStores.aws.utils.**retryable_sdb_errors**(*e*)

toil.jobStores.aws.utils.**retry_sdb**(*delays=DEFAULT_DELAYS*, *timeout=DEFAULT_TIMEOUT*, *predicate=retryable_sdb_errors*)

Submodules

toil.jobStores.abstractJobStore

Module Contents

Classes

AbstractJobStore	Represents the physical storage for the jobs and files in a Toil workflow.
JobStoreSupport	A mostly fake JobStore to access URLs not really associated with real job

Attributes

logger

toil.jobStores.abstractJobStore.**logger**

exception toil.jobStores.abstractJobStore.**ProxyConnectionError**

Bases: BaseException

Dummy class.

exception toil.jobStores.abstractJobStore.**InvalidImportExportUrlException**(*url*)

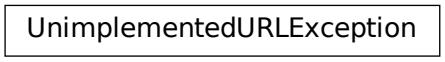
Bases: Exception

Common base class for all non-exit exceptions.

Parameters
url (*urllib.parse.ParseResult*) –

exception toil.jobStores.abstractJobStore.**UnimplementedURLException**(*url*, *operation*)

Bases: RuntimeError



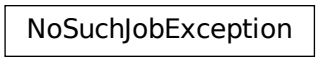
Unspecified run-time error.

Parameters

- **url** (*urllib.parse.ParseResult*) –
- **operation** (*str*) –

exception toil.jobStores.abstractJobStore.**NoSuchJobException**(*jobStoreID*)

Bases: Exception

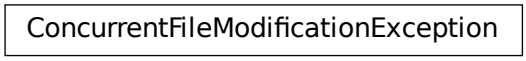


Indicates that the specified job does not exist.

Parameters
jobStoreID (toil.fileStores.FileID) –

exception toil.jobStores.abstractJobStore.**ConcurrentFileModificationException**(*jobStoreFileID*)

Bases: Exception



Indicates that the file was attempted to be modified by multiple processes at once.

Parameters
jobStoreFileID (toil.fileStores.FileID) –

exception toil.jobStores.abstractJobStore.**NoSuchFileException**(*jobStoreFileID*, *customName=None*, **extra*)

Bases: Exception

Indicates that the specified file does not exist.

Parameters

- **jobStoreFileID** (toil.fileStores.FileID) –
- **customName** (*Optional[str]*) –
- **extra** (*Any*) –

exception toil.jobStores.abstractJobStore.**NoSuchJobStoreException**(*locator*)

Bases: Exception

Indicates that the specified job store does not exist.

Parameters
locator (*str*) –

exception toil.jobStores.abstractJobStore.**JobStoreExistsException**(*locator*)

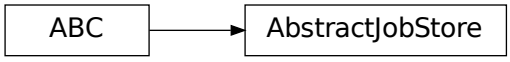
Bases: Exception

Indicates that the specified job store already exists.

Parameters
locator (*str*) –

class toil.jobStores.abstractJobStore.**AbstractJobStore**(*locator*)

Bases: abc.ABC



Represents the physical storage for the jobs and files in a Toil workflow.

JobStores are responsible for storing *toil.job.JobDescription* (which relate jobs to each other) and files.

Actual *toil.job.Job* objects are stored in files, referenced by JobDescriptions. All the non-file CRUD methods the JobStore provides deal in JobDescriptions and not full, executable Jobs.

To actually get ahold of a *toil.job.Job*, use *toil.job.Job.loadJob()* with a JobStore and the relevant JobDescription.

Parameters
locator (*str*) –

property config: *toil.common.Config*

Return the Toil configuration associated with this job store.

Return type
toil.common.Config

property locator: str

Get the locator that defines the job store, which can be used to connect to it.

Return type
str

rootJobStoreIDFileName = 'rootJobStoreID'

publicUrlExpiration

sharedFileNameRegex

initialize(*config*)

Initialize this job store.

Create the physical storage for this job store, allocate a workflow ID and persist the given Toil configuration to the store.

Parameters
config (toil.common.Config) – the Toil configuration to initialize this job store with. The given configuration will be updated with the newly allocated workflow ID.

Raises
JobStoreExistsException – if the physical storage for this job store already exists

Return type
None

writeConfig()

Return type
None

write_config()

Persists the value of the *AbstractJobStore.config* attribute to the job store, so that it can be retrieved later by other instances of this class.

Return type
None

resume()

Connect this instance to the physical storage it represents and load the Toil configuration into the *AbstractJobStore.config* attribute.

Raises
NoSuchJobStoreException – if the physical storage for this job store doesn't exist

Return type
None

setRootJob(*rootJobStoreID*)

Set the root job of the workflow backed by this job store.

Parameters
rootJobStoreID (toil.fileStores.FileID) –

Return type
None

set_root_job(*job_id*)

Set the root job of the workflow backed by this job store.

Parameters
job_id (toil.fileStores.FileID) – The ID of the job to set as root

Return type
None

loadRootJob()

Return type
toil.job.JobDescription

load_root_job()

Loads the JobDescription for the root job in the current job store.

Raises
toil.job.JobException – If no root job is set or if the root job doesn't exist in this job store

Returns
The root job.

Return type
toil.job.JobDescription

createRootJob(*desc*)

Parameters
desc (toil.job.JobDescription) –

Return type
toil.job.JobDescription

create_root_job(*job_description*)

Create the given JobDescription and set it as the root job in this job store.

Parameters
job_description (toil.job.JobDescription) – JobDescription to save and make the root job.

Return type
toil.job.JobDescription

getRootJobReturnValue()

Return type
Any

get_root_job_return_value()

Parse the return value from the root job.

Raises an exception if the root job hasn't fulfilled its promise yet.

Return type
Any

importFile(*srcUrl: str*, *sharedFileName: str*, *hardlink: bool = False*, *symlink: bool = True*) → None

importFile(*srcUrl: str*, *sharedFileName: None = None*, *hardlink: bool = False*, *symlink: bool = True*) → *toil.fileStores.FileID*

import_file(*src_uri: str*, *shared_file_name: str*, *hardlink: bool = False*, *symlink: bool = True*) → None

import_file(*src_uri: str*, *shared_file_name: None = None*, *hardlink: bool = False*, *symlink: bool = True*) → *toil.fileStores.FileID*

Imports the file at the given URL into job store. The ID of the newly imported file is returned. If the name of a shared file name is provided, the file will be imported as such and None is returned. If an executable file on the local filesystem is uploaded, its executability will be preserved when it is downloaded.

Currently supported schemes are:

- **'s3' for objects in Amazon S3**
  e.g. s3://bucket/key
- **'file' for local files**
  e.g. file:///local/file/path
- **'http'**
  e.g. http://someurl.com/path
- **'gs'**
  e.g. gs://bucket/file

Parameters

- **src_uri** (*str*) – URL that points to a file or object in the storage mechanism of a supported URL scheme e.g. a blob in an AWS s3 bucket. It must be a file, not a directory or prefix.
- **shared_file_name** (*str*) – Optional name to assign to the imported file within the job store

Returns
The jobStoreFileID of the imported file or None if shared_file_name was given

Return type
toil.fileStores.FileID or None

exportFile(*jobStoreFileID*, *dstUrl*)

Parameters

- **jobStoreFileID** (toil.fileStores.FileID) –
- **dstUrl** (*str*) –

Return type
None

export_file(*file_id*, *dst_uri*)

Exports file to destination pointed at by the destination URL. The exported file will be executable if and only if it was originally uploaded from an executable file on the local filesystem.

Refer to *AbstractJobStore.import_file()* documentation for currently supported URL schemes.

Note that the helper method _exportFile is used to read from the source and write to destination. To implement any optimizations that circumvent this, the _exportFile method should be overridden by subclasses of AbstractJobStore.

Parameters

- **file_id** (*str*) – The id of the file in the job store that should be exported.

- **dst_uri** (*str*) – URL that points to a file or object in the storage mechanism of a supported URL scheme e.g. a blob in an AWS s3 bucket.

Return type
None

classmethod list_url(*src_uri*)

List the directory at the given URL. Returned path components can be joined with '/' onto the passed URL to form new URLs. Those that end in '/' correspond to directories. The provided URL may or may not end with '/'.

Currently supported schemes are:

- **'s3' for objects in Amazon S3**
  e.g. s3://bucket/prefix/
- **'file' for local files**
  e.g. file:///local/dir/path/

Parameters
src_uri (*str*) – URL that points to a directory or prefix in the storage mechanism of a supported URL scheme e.g. a prefix in an AWS s3 bucket.

Returns
A list of URL components in the given directory, already URL-encoded.

Return type
List[str]

classmethod get_is_directory(*src_uri*)

Return True if the thing at the given URL is a directory, and False if it is a file. The URL may or may not end in '/'.

Parameters
src_uri (*str*) –

Return type
bool

classmethod read_from_url(*src_uri*, *writable*)

Read the given URL and write its content into the given writable stream.

Returns
The size of the file in bytes and whether the executable permission bit is set

Return type
Tuple[int, bool]

Parameters

- **src_uri** (*str*) –
- **writable** (*IO[bytes]*) –

classmethod getSize(*url*)

Parameters
url (*urllib.parse.ParseResult*) –

Return type
None

abstract classmethod get_size(*src_uri*)

Get the size in bytes of the file at the given URL, or None if it cannot be obtained.

Parameters
src_uri (*urllib.parse.ParseResult*) – URL that points to a file or object in the storage mechanism of a supported URL scheme e.g. a blob in an AWS s3 bucket.

Return type
None

abstract destroy()

The inverse of *initialize()*, this method deletes the physical storage represented by this instance. While not being atomic, this method *is* at least idempotent, as a means to counteract potential issues with eventual consistency exhibited by the underlying storage mechanisms. This means that if the method fails (raises an exception), it may (and should be) invoked again. If the underlying storage mechanism is eventually consistent, even a successful invocation is not an ironclad guarantee that the physical storage vanished completely and immediately. A successful invocation only guarantees that the deletion will eventually happen. It is therefore recommended to not immediately reuse the same job store location for a new Toil workflow.

Return type
None

getEnv()

Return type
Dict[str, str]

get_env()

Returns a dictionary of environment variables that this job store requires to be set in order to function properly on a worker.

Return type
dict[str,str]

clean(*jobCache=None*)

Function to cleanup the state of a job store after a restart.

Fixes jobs that might have been partially updated. Resets the try counts and removes jobs that are not successors of the current root job.

Parameters
jobCache (*Optional[Dict[Union[str, toil.job.TemporaryID], toil.job.JobDescription]]*) – if a value it must be a dict from job ID keys to JobDescription object values. Jobs will be loaded from the cache (which can be downloaded from the job store in a batch) instead of piecemeal when recursed into.

Return type
toil.job.JobDescription

assignID(*jobDescription*)

Parameters
jobDescription (toil.job.JobDescription) –

Return type
None

abstract assign_job_id(*job_description*)

Get a new jobStoreID to be used by the described job, and assigns it to the JobDescription.

Files associated with the assigned ID will be accepted even if the JobDescription has never been created or updated.

Parameters
job_description (toil.job.JobDescription) – The JobDescription to give an ID to

Return type
None

batch()

If supported by the batch system, calls to create() with this context manager active will be performed in a batch after the context manager is released.

Return type
Iterator[None]

create(*jobDescription*)

Parameters
jobDescription (toil.job.JobDescription) –

Return type
toil.job.JobDescription

abstract create_job(*job_description*)

Writes the given JobDescription to the job store. The job must have an ID assigned already.

Must call jobDescription.pre_update_hook()

Returns
The JobDescription passed.

Return type
toil.job.JobDescription

Parameters
job_description (toil.job.JobDescription) –

exists(*jobStoreID*)

Parameters
jobStoreID (*str*) –

Return type
bool

abstract job_exists(*job_id*)

Indicates whether a description of the job with the specified jobStoreID exists in the job store

Return type
bool

Parameters
job_id (*str*) –

getPublicUrl(*fileName*)

Parameters
fileName (*str*) –

Return type
str

abstract get_public_url(*file_name*)

Returns a publicly accessible URL to the given file in the job store. The returned URL may expire as early as 1h after its been returned. Throw an exception if the file does not exist.

Parameters
file_name (*str*) – the jobStoreFileID of the file to generate a URL for

Raises
NoSuchFileException – if the specified file does not exist in this job store

Return type
str

getSharedPublicUrl(*sharedFileName*)

Parameters
sharedFileName (*str*) –

Return type
str

abstract get_shared_public_url(*shared_file_name*)

Differs from *getPublicUrl()* in that this method is for generating URLs for shared files written by *writeSharedFileStream()*.

Returns a publicly accessible URL to the given file in the job store. The returned URL starts with 'http:', 'https:' or 'file:'. The returned URL may expire as early as 1h after its been returned. Throw an exception if the file does not exist.

Parameters
shared_file_name (*str*) – The name of the shared file to generate a publically accessible url for.

Raises
NoSuchFileException – raised if the specified file does not exist in the store

Return type
str

load(*jobStoreID*)

Parameters
jobStoreID (*str*) –

Return type
toil.job.JobDescription

abstract load_job(*job_id*)

Loads the description of the job referenced by the given ID, assigns it the job store's config, and returns it.

May declare the job to have failed (see *toil.job.JobDescription.setupJobAfterFailure()*) if there is evidence of a failed update attempt.

Parameters
job_id (*str*) – the ID of the job to load

Raises
NoSuchJobException – if there is no job with the given ID

Return type
toil.job.JobDescription

update(*jobDescription*)

Parameters
jobDescription (toil.job.JobDescription) –

Return type
None

abstract update_job(*job_description*)

Persists changes to the state of the given JobDescription in this store atomically.

Must call jobDescription.pre_update_hook()

Parameters

- **job** (toil.job.JobDescription) – the job to write to this job store
- **job_description** (toil.job.JobDescription) –

Return type
None

delete(*jobStoreID*)

Parameters
jobStoreID (*str*) –

Return type
None

abstract delete_job(*job_id*)

Removes the JobDescription from the store atomically. You may not then subsequently call load(), write(), update(), etc. with the same jobStoreID or any JobDescription bearing it.

This operation is idempotent, i.e. deleting a job twice or deleting a non-existent job will succeed silently.

Parameters
job_id (*str*) – the ID of the job to delete from this job store

Return type
None

abstract jobs()

Best effort attempt to return iterator on JobDescriptions for all jobs in the store. The iterator may not return all jobs and may also contain orphaned jobs that have already finished successfully and should not be rerun. To guarantee you get any and all jobs that can be run instead construct a more expensive ToilState object

Returns
Returns iterator on jobs in the store. The iterator may or may not contain all jobs and may contain invalid jobs

Return type
Iterator[toil.job.jobDescription]

writeFile(*localFilePath*, *jobStoreID=None*, *cleanup=False*)

Parameters

- **localFilePath** (*str*) –
- **jobStoreID** (*Optional[str]*) –

- **cleanup** (*bool*) –

Return type
str

abstract write_file(*local_path*, *job_id=None*, *cleanup=False*)

Takes a file (as a path) and places it in this job store. Returns an ID that can be used to retrieve the file at a later time. The file is written in a atomic manner. It will not appear in the jobStore until the write has successfully completed.

Parameters

- **local_path** (*str*) – the path to the local file that will be uploaded to the job store. The last path component (basename of the file) will remain associated with the file in the file store, if supported, so that the file can be searched for by name or name glob.
- **job_id** (*str*) – the id of a job, or None. If specified, the may be associated with that job in a job-store-specific way. This may influence the returned ID.
- **cleanup** (*bool*) – Whether to attempt to delete the file when the job whose jobStoreID was given as jobStoreID is deleted with jobStore.delete(job). If jobStoreID was not given, does nothing.

Raises

- ***ConcurrentFileModificationException*** – if the file was modified concurrently during an invocation of this method
- ***NoSuchJobException*** – if the job specified via jobStoreID does not exist

Return type
str

FIXME: some implementations may not raise this

Returns
an ID referencing the newly created file and can be used to read the file in the future.

Return type
str

Parameters

- **local_path** (*str*) –
- **job_id** (*Optional[str]*) –
- **cleanup** (*bool*) –

writeFileStream(*jobStoreID=None*, *cleanup=False*, *basename=None*, *encoding=None*, *errors=None*)

Parameters

- **jobStoreID** (*Optional[str]*) –
- **cleanup** (*bool*) –
- **basename** (*Optional[str]*) –
- **encoding** (*Optional[str]*) –
- **errors** (*Optional[str]*) –

Return type
ContextManager[Tuple[IO[bytes], str]]

abstract write_file_stream(*job_id=None*, *cleanup=False*, *basename=None*, *encoding=None*, *errors=None*)

Similar to writeFile, but returns a context manager yielding a tuple of 1) a file handle which can be written to and 2) the ID of the resulting file in the job store. The yielded file handle does not need to and should not be closed explicitly. The file is written in a atomic manner. It will not appear in the jobStore until the write has successfully completed.

Parameters

- **job_id** (*str*) – the id of a job, or None. If specified, the may be associated with that job in a job-store-specific way. This may influence the returned ID.
- **cleanup** (*bool*) – Whether to attempt to delete the file when the job whose jobStoreID was given as jobStoreID is deleted with jobStore.delete(job). If jobStoreID was not given, does nothing.
- **basename** (*str*) – If supported by the implementation, use the given file basename so that when searching the job store with a query matching that basename, the file will be detected.
- **encoding** (*str*) – the name of the encoding used to encode the file. Encodings are the same as for encode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Raises

- *ConcurrentFileModificationException* – if the file was modified concurrently during an invocation of this method
- *NoSuchJobException* – if the job specified via jobStoreID does not exist

Return type
Iterator[Tuple[IO[bytes], str]]

FIXME: some implementations may not raise this

Returns
a context manager yielding a file handle which can be written to and an ID that references the newly created file and can be used to read the file in the future.

Return type
Iterator[Tuple[IO[bytes], str]]

Parameters

- **job_id** (*Optional[str]*) –
- **cleanup** (*bool*) –
- **basename** (*Optional[str]*) –
- **encoding** (*Optional[str]*) –
- **errors** (*Optional[str]*) –

getEmptyFileStoreID(*jobStoreID=None*, *cleanup=False*, *basename=None*)

Parameters

- **jobStoreID** (*Optional[str]*) –
- **cleanup** (*bool*) –
- **basename** (*Optional[str]*) –

Return type
str

abstract get_empty_file_store_id(*job_id=None*, *cleanup=False*, *basename=None*)

Creates an empty file in the job store and returns its ID. Call to fileExists(getEmptyFileStoreID(jobStoreID)) will return True.

Parameters

- **job_id** (*str*) – the id of a job, or None. If specified, the may be associated with that job in a job-store-specific way. This may influence the returned ID.
- **cleanup** (*bool*) – Whether to attempt to delete the file when the job whose jobStoreID was given as jobStoreID is deleted with jobStore.delete(job). If jobStoreID was not given, does nothing.
- **basename** (*str*) – If supported by the implementation, use the given file basename so that when searching the job store with a query matching that basename, the file will be detected.

Returns
a jobStoreFileID that references the newly created file and can be used to reference the file in the future.

Return type
str

readFile(*jobStoreFileID*, *localFilePath*, *symlink=False*)

Parameters

- **jobStoreFileID** (*str*) –
- **localFilePath** (*str*) –
- **symlink** (*bool*) –

Return type
None

abstract read_file(*file_id*, *local_path*, *symlink=False*)

Copies or hard links the file referenced by jobStoreFileID to the given local file path. The version will be consistent with the last copy of the file written/updated. If the file in the job store is later modified via updateFile or updateFileStream, it is implementation-defined whether those writes will be visible at localFilePath. The file is copied in an atomic manner. It will not appear in the local file system until the copy has completed.

The file at the given local path may not be modified after this method returns!

Note! Implementations of readFile need to respect/provide the executable attribute on FileIDs.

Parameters

- **file_id** (*str*) – ID of the file to be copied
- **local_path** (*str*) – the local path indicating where to place the contents of the given file in the job store
- **symlink** (*bool*) – whether the reader can tolerate a symlink. If set to true, the job store may create a symlink instead of a full copy of the file or a hard link.

Return type
None

readFileStream(*jobStoreFileID*, *encoding=None*, *errors=None*)

Parameters

- **jobStoreFileID** (*str*) –
- **encoding** (*Optional[str]*) –
- **errors** (*Optional[str]*) –

Return type
Union[ContextManager[IO[bytes]], ContextManager[IO[str]]]

read_file_stream(*file_id: Union[toil.fileStores.FileID, str]*, *encoding: Literal[None] = None*, *errors: Optional[str] = None*) → ContextManager[IO[bytes]]

read_file_stream(*file_id: Union[toil.fileStores.FileID, str]*, *encoding: str*, *errors: Optional[str] = None*) → ContextManager[IO[str]]

Similar to readFile, but returns a context manager yielding a file handle which can be read from. The yielded file handle does not need to and should not be closed explicitly.

Parameters

- **file_id** (*str*) – ID of the file to get a readable file handle for
- **encoding** (*str*) – the name of the encoding used to decode the file. Encodings are the same as for decode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Returns
a context manager yielding a file handle which can be read from

Return type
Iterator[Union[IO[bytes], IO[str]]]

deleteFile(*jobStoreFileID*)

Parameters
jobStoreFileID (*str*) –

Return type
None

abstract delete_file(*file_id*)

Deletes the file with the given ID from this job store. This operation is idempotent, i.e. deleting a file twice or deleting a non-existent file will succeed silently.

Parameters
file_id (*str*) – ID of the file to delete

Return type
None

fileExists(*jobStoreFileID*)

Determine whether a file exists in this job store.

Parameters
jobStoreFileID (*str*) –

Return type
bool

abstract file_exists(*file_id*)

Determine whether a file exists in this job store.

Parameters
file_id (*str*) – an ID referencing the file to be checked

Return type
bool

getFileSize(*jobStoreFileID*)

Get the size of the given file in bytes.

Parameters
jobStoreFileID (*str*) –

Return type
int

abstract get_file_size(*file_id*)

Get the size of the given file in bytes, or 0 if it does not exist when queried.

Note that job stores which encrypt files might return overestimates of file sizes, since the encrypted file may have been padded to the nearest block, augmented with an initialization vector, etc.

Parameters
file_id (*str*) – an ID referencing the file to be checked

Return type
int

updateFile(*jobStoreFileID*, *localFilePath*)

Replaces the existing version of a file in the job store.

Parameters

- **jobStoreFileID** (*str*) –
- **localFilePath** (*str*) –

Return type
None

abstract update_file(*file_id*, *local_path*)

Replaces the existing version of a file in the job store.

Throws an exception if the file does not exist.

Parameters

- **file_id** (*str*) – the ID of the file in the job store to be updated
- **local_path** (*str*) – the local path to a file that will overwrite the current version in the job store

Raises

- *ConcurrentFileModificationException* – if the file was modified concurrently during an invocation of this method
- *NoSuchFileException* – if the specified file does not exist

Return type
None

updateFileStream(*jobStoreFileID*, *encoding=None*, *errors=None*)

Parameters

- **jobStoreFileID** (*str*) –
- **encoding** (*Optional[str]*) –
- **errors** (*Optional[str]*) –

Return type
ContextManager[IO[Any]]

abstract update_file_stream(*file_id*, *encoding=None*, *errors=None*)

Replaces the existing version of a file in the job store. Similar to writeFile, but returns a context manager yielding a file handle which can be written to. The yielded file handle does not need to and should not be closed explicitly.

Parameters

- **file_id** (*str*) – the ID of the file in the job store to be updated
- **encoding** (*str*) – the name of the encoding used to encode the file. Encodings are the same as for encode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Raises

- ***ConcurrentFileModificationException*** – if the file was modified concurrently during an invocation of this method
- ***NoSuchFileException*** – if the specified file does not exist

Return type
Iterator[IO[Any]]

writeSharedFileStream(*sharedFileName*, *isProtected=None*, *encoding=None*, *errors=None*)

Parameters

- **sharedFileName** (*str*) –
- **isProtected** (*Optional[bool]*) –
- **encoding** (*Optional[str]*) –
- **errors** (*Optional[str]*) –

Return type
ContextManager[IO[bytes]]

abstract write_shared_file_stream(*shared_file_name*, *encrypted=None*, *encoding=None*, *errors=None*)

Returns a context manager yielding a writable file handle to the global file referenced by the given name. File will be created in an atomic manner.

Parameters

- **shared_file_name** (*str*) – A file name matching AbstractJobStore.fileNameRegex, unique within this job store
- **encrypted** (*bool*) – True if the file must be encrypted, None if it may be encrypted or False if it must be stored in the clear.

- **encoding** (*str*) – the name of the encoding used to encode the file. Encodings are the same as for encode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Raises
ConcurrentFileModificationException – if the file was modified concurrently during an invocation of this method

Returns
a context manager yielding a writable file handle

Return type
Iterator[IO[bytes]]

readSharedFileStream(*sharedFileName*, *encoding=None*, *errors=None*)

Parameters

- **sharedFileName** (*str*) –
- **encoding** (*Optional[str]*) –
- **errors** (*Optional[str]*) –

Return type
ContextManager[IO[bytes]]

abstract read_shared_file_stream(*shared_file_name*, *encoding=None*, *errors=None*)

Returns a context manager yielding a readable file handle to the global file referenced by the given name.

Parameters

- **shared_file_name** (*str*) – A file name matching AbstractJobStore.fileNameRegex, unique within this job store
- **encoding** (*str*) – the name of the encoding used to decode the file. Encodings are the same as for decode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Returns
a context manager yielding a readable file handle

Return type
Iterator[IO[bytes]]

writeStatsAndLogging(*statsAndLoggingString*)

Parameters
statsAndLoggingString (*str*) –

Return type
None

abstract write_logs(*msg*)

Stores a message as a log in the jobstore.

Parameters
msg (*str*) – the string to be written

Raises
ConcurrentFileModificationException – if the file was modified concurrently during an invocation of this method

Return type
None

readStatsAndLogging(*callback*, *readAll=False*)

Parameters

- **callback** (*Callable[Ellipsis, Any]*) –
- **readAll** (*bool*) –

Return type
int

abstract read_logs(*callback*, *read_all=False*)

Reads logs accumulated by the write_logs() method. For each log this method calls the given callback function with the message as an argument (rather than returning logs directly, this method must be supplied with a callback which will process log messages).

Only unread logs will be read unless the read_all parameter is set.

Parameters

- **callback** (*Callable*) – a function to be applied to each of the stats file handles found
- **read_all** (*bool*) – a boolean indicating whether to read the already processed stats files in addition to the unread stats files

Raises
ConcurrentFileModificationException – if the file was modified concurrently during an invocation of this method

Returns
the number of stats files processed

Return type
int

write_leader_pid()

Write the pid of this process to a file in the job store.

Overwriting the current contents of pid.log is a feature, not a bug of this method. Other methods will rely on always having the most current pid available. So far there is no reason to store any old pids.

Return type
None

read_leader_pid()

Read the pid of the leader process to a file in the job store.

Raises
NoSuchFileException – If the PID file doesn't exist.

Return type
int

write_leader_node_id()

Write the leader node id to the job store. This should only be called by the leader.

Return type
None

`read_leader_node_id()`

Read the leader node id stored in the job store.

Raises
NoSuchFileException – If the node ID file doesn't exist.

Return type
str

`write_kill_flag(kill=False)`

Write a file inside the job store that serves as a kill flag.

The initialized file contains the characters "NO". This should only be changed when the user runs the "toil kill" command.

Changing this file to a "YES" triggers a kill of the leader process. The workers are expected to be cleaned up by the leader.

Parameters
kill (*bool*) –

Return type
None

`read_kill_flag()`

Read the kill flag from the job store, and return True if the leader has been killed. False otherwise.

Return type
bool

`default_caching()`

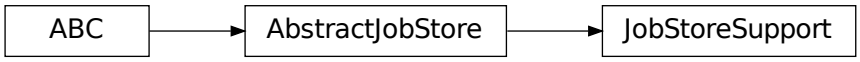
Jobstore's preference as to whether it likes caching or doesn't care about it. Some jobstores benefit from caching, however on some local configurations it can be flaky.

see https://github.com/DataBiosphere/toil/issues/4218

Return type
bool

`class toil.jobStores.abstractJobStore.JobStoreSupport(locator)`

Bases: *AbstractJobStore*



A mostly fake JobStore to access URLs not really associated with real job stores.

Parameters
locator (*str*) –

classmethod get_size(*url*)

Get the size in bytes of the file at the given URL, or None if it cannot be obtained.

Parameters

- **src_uri** – URL that points to a file or object in the storage mechanism of a supported URL scheme e.g. a blob in an AWS s3 bucket.
- **url** (*urllib.parse.ParseResult*) –

Return type
Optional[int]

toil.jobStores.conftest

Module Contents

toil.jobStores.conftest.**collect_ignore** = []

toil.jobStores.fileJobStore

Module Contents

Classes

FileJobStore	A job store that uses a directory on a locally attached file system. To be compatible with

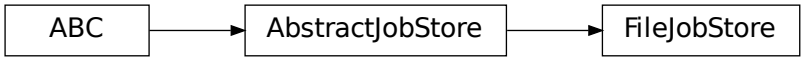
Attributes

logger

toil.jobStores.fileJobStore.**logger**

class toil.jobStores.fileJobStore.**FileJobStore**(*path*, *fanOut=1000*)

Bases: *toil.jobStores.abstractJobStore.AbstractJobStore*



A job store that uses a directory on a locally attached file system. To be compatible with distributed batch systems, that file system must be shared by all worker nodes.

Parameters

- **path** (*str*) –
- **fanOut** (*int*) –

validDirs = 'abcdefghijklmnopqrstuvwxyz0123456789'

validDirsSet

JOB_DIR_PREFIX = 'instance-'

JOB_NAME_DIR_PREFIX = 'kind-'

BUFFER_SIZE = 10485760

default_caching()

Jobstore's preference as to whether it likes caching or doesn't care about it. Some jobstores benefit from caching, however on some local configurations it can be flaky.

see https://github.com/DataBiosphere/toil/issues/4218

Return type
bool

__repr__()

Return repr(self).

initialize(*config*)

Initialize this job store.

Create the physical storage for this job store, allocate a workflow ID and persist the given Toil configuration to the store.

Parameters
config – the Toil configuration to initialize this job store with. The given configuration will be updated with the newly allocated workflow ID.

Raises
JobStoreExistsException – if the physical storage for this job store already exists

resume()

Connect this instance to the physical storage it represents and load the Toil configuration into the `AbstractJobStore.config` attribute.

Raises
NoSuchJobStoreException – if the physical storage for this job store doesn't exist

destroy()

The inverse of *initialize()*, this method deletes the physical storage represented by this instance. While not being atomic, this method *is* at least idempotent, as a means to counteract potential issues with eventual consistency exhibited by the underlying storage mechanisms. This means that if the method fails (raises an exception), it may (and should be) invoked again. If the underlying storage mechanism is eventually consistent, even a successful invocation is not an ironclad guarantee that the physical storage vanished completely and immediately. A successful invocation only guarantees that the deletion will eventually happen. It is therefore recommended to not immediately reuse the same job store location for a new Toil workflow.

assign_job_id(*job_description*)

Get a new jobStoreID to be used by the described job, and assigns it to the JobDescription.

Files associated with the assigned ID will be accepted even if the JobDescription has never been created or updated.

Parameters
job_description (toil.job.JobDescription) – The JobDescription to give an ID to

create_job(*job_description*)

Writes the given JobDescription to the job store. The job must have an ID assigned already.

Must call jobDescription.pre_update_hook()

Returns
The JobDescription passed.

Return type
toil.job.JobDescription

batch()

If supported by the batch system, calls to create() with this context manager active will be performed in a batch after the context manager is released.

job_exists(*job_id*)

Indicates whether a description of the job with the specified jobStoreID exists in the job store

Return type
bool

get_public_url(*jobStoreFileID*)

Returns a publicly accessible URL to the given file in the job store. The returned URL may expire as early as 1h after its been returned. Throw an exception if the file does not exist.

Parameters
file_name (*str*) – the jobStoreFileID of the file to generate a URL for

Raises
NoSuchFileException – if the specified file does not exist in this job store

Return type
str

get_shared_public_url(*sharedFileName*)

Differs from `getPublicUrl()` in that this method is for generating URLs for shared files written by `writeSharedFileStream()`.

Returns a publicly accessible URL to the given file in the job store. The returned URL starts with 'http:', 'https:' or 'file:'. The returned URL may expire as early as 1h after its been returned. Throw an exception if the file does not exist.

Parameters
shared_file_name (*str*) – The name of the shared file to generate a publically accessible url for.

Raises
NoSuchFileException – raised if the specified file does not exist in the store

Return type
str

load_job(*job_id*)

Loads the description of the job referenced by the given ID, assigns it the job store's config, and returns it.

May declare the job to have failed (see *toil.job.JobDescription.setupJobAfterFailure()*) if there is evidence of a failed update attempt.

Parameters
job_id – the ID of the job to load

Raises
NoSuchJobException – if there is no job with the given ID

update_job(*job*)

Persists changes to the state of the given JobDescription in this store atomically.

Must call jobDescription.pre_update_hook()

Parameters
job (toil.job.JobDescription) – the job to write to this job store

delete_job(*job_id*)

Removes the JobDescription from the store atomically. You may not then subsequently call load(), write(), update(), etc. with the same jobStoreID or any JobDescription bearing it.

This operation is idempotent, i.e. deleting a job twice or deleting a non-existent job will succeed silently.

Parameters
job_id (*str*) – the ID of the job to delete from this job store

jobs()

Best effort attempt to return iterator on JobDescriptions for all jobs in the store. The iterator may not return all jobs and may also contain orphaned jobs that have already finished successfully and should not be rerun. To guarantee you get any and all jobs that can be run instead construct a more expensive ToilState object

Returns
Returns iterator on jobs in the store. The iterator may or may not contain all jobs and may contain invalid jobs

Return type
Iterator[toil.job.jobDescription]

optional_hard_copy(*hardlink*)

classmethod get_size(*url*)

Get the size in bytes of the file at the given URL, or None if it cannot be obtained.

Parameters
src_uri – URL that points to a file or object in the storage mechanism of a supported URL scheme e.g. a blob in an AWS s3 bucket.

write_file(*local_path*, *job_id=None*, *cleanup=False*)

Takes a file (as a path) and places it in this job store. Returns an ID that can be used to retrieve the file at a later time. The file is written in a atomic manner. It will not appear in the jobStore until the write has successfully completed.

Parameters

- **local_path** (*str*) – the path to the local file that will be uploaded to the job store. The last path component (basename of the file) will remain associated with the file in the file store, if supported, so that the file can be searched for by name or name glob.
- **job_id** (*str*) – the id of a job, or None. If specified, the may be associated with that job in a job-store-specific way. This may influence the returned ID.
- **cleanup** (*bool*) – Whether to attempt to delete the file when the job whose jobStoreID was given as jobStoreID is deleted with jobStore.delete(job). If jobStoreID was not given, does nothing.

Raises

- *ConcurrentFileModificationException* – if the file was modified concurrently during an invocation of this method
- *NoSuchJobException* – if the job specified via jobStoreID does not exist

FIXME: some implementations may not raise this

Returns
an ID referencing the newly created file and can be used to read the file in the future.

Return type
str

write_file_stream(*job_id=None*, *cleanup=False*, *basename=None*, *encoding=None*, *errors=None*)

Similar to writeFile, but returns a context manager yielding a tuple of 1) a file handle which can be written to and 2) the ID of the resulting file in the job store. The yielded file handle does not need to and should not be closed explicitly. The file is written in a atomic manner. It will not appear in the jobStore until the write has successfully completed.

Parameters

- **job_id** (*str*) – the id of a job, or None. If specified, the may be associated with that job in a job-store-specific way. This may influence the returned ID.
- **cleanup** (*bool*) – Whether to attempt to delete the file when the job whose jobStoreID was given as jobStoreID is deleted with jobStore.delete(job). If jobStoreID was not given, does nothing.
- **basename** (*str*) – If supported by the implementation, use the given file basename so that when searching the job store with a query matching that basename, the file will be detected.
- **encoding** (*str*) – the name of the encoding used to encode the file. Encodings are the same as for encode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Raises

- *ConcurrentFileModificationException* – if the file was modified concurrently during an invocation of this method
- *NoSuchJobException* – if the job specified via jobStoreID does not exist

FIXME: some implementations may not raise this

Returns
a context manager yielding a file handle which can be written to and an ID that references the newly created file and can be used to read the file in the future.

Return type
Iterator[Tuple[IO[bytes], str]]

get_empty_file_store_id(*jobStoreID=None*, *cleanup=False*, *basename=None*)

Creates an empty file in the job store and returns its ID. Call to fileExists(getEmptyFileStoreID(jobStoreID)) will return True.

Parameters

- **job_id** (*str*) – the id of a job, or None. If specified, the may be associated with that job in a job-store-specific way. This may influence the returned ID.

- **cleanup** (*bool*) – Whether to attempt to delete the file when the job whose jobStoreID was given as jobStoreID is deleted with jobStore.delete(job). If jobStoreID was not given, does nothing.
- **basename** (*str*) – If supported by the implementation, use the given file basename so that when searching the job store with a query matching that basename, the file will be detected.

Returns
a jobStoreFileID that references the newly created file and can be used to reference the file in the future.

Return type
str

update_file(*file_id*, *local_path*)

Replaces the existing version of a file in the job store.

Throws an exception if the file does not exist.

Parameters

- **file_id** – the ID of the file in the job store to be updated
- **local_path** – the local path to a file that will overwrite the current version in the job store

Raises

- *ConcurrentFileModificationException* – if the file was modified concurrently during an invocation of this method
- *NoSuchFileException* – if the specified file does not exist

read_file(*file_id*, *local_path*, *symlink=False*)

Copies or hard links the file referenced by jobStoreFileID to the given local file path. The version will be consistent with the last copy of the file written/updated. If the file in the job store is later modified via updateFile or updateFileStream, it is implementation-defined whether those writes will be visible at localFilePath. The file is copied in an atomic manner. It will not appear in the local file system until the copy has completed.

The file at the given local path may not be modified after this method returns!

Note! Implementations of readFile need to respect/provide the executable attribute on FileIDs.

Parameters

- **file_id** (*str*) – ID of the file to be copied
- **local_path** (*str*) – the local path indicating where to place the contents of the given file in the job store
- **symlink** (*bool*) – whether the reader can tolerate a symlink. If set to true, the job store may create a symlink instead of a full copy of the file or a hard link.

delete_file(*file_id*)

Deletes the file with the given ID from this job store. This operation is idempotent, i.e. deleting a file twice or deleting a non-existent file will succeed silently.

Parameters
file_id (*str*) – ID of the file to delete

file_exists(*file_id*)

Determine whether a file exists in this job store.

Parameters
file_id – an ID referencing the file to be checked

get_file_size(*file_id*)

Get the size of the given file in bytes, or 0 if it does not exist when queried.

Note that job stores which encrypt files might return overestimates of file sizes, since the encrypted file may have been padded to the nearest block, augmented with an initialization vector, etc.

Parameters
file_id (*str*) – an ID referencing the file to be checked

Return type
int

update_file_stream(*file_id*, *encoding=None*, *errors=None*)

Replaces the existing version of a file in the job store. Similar to writeFile, but returns a context manager yielding a file handle which can be written to. The yielded file handle does not need to and should not be closed explicitly.

Parameters

- **file_id** (*str*) – the ID of the file in the job store to be updated
- **encoding** (*str*) – the name of the encoding used to encode the file. Encodings are the same as for encode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Raises

- ***ConcurrentFileModificationException*** – if the file was modified concurrently during an invocation of this method
- ***NoSuchFileException*** – if the specified file does not exist

read_file_stream(*file_id: Union[str,* toil.fileStores.FileID*]*, *encoding: Literal[None] = None*, *errors: Optional[str] = None*) → Iterator[IO[bytes]]

read_file_stream(*file_id: Union[str,* toil.fileStores.FileID*]*, *encoding: str*, *errors: Optional[str] = None*) → Iterator[IO[str]]

read_file_stream(*file_id: Union[str,* toil.fileStores.FileID*]*, *encoding: Optional[str] = None*, *errors: Optional[str] = None*) → Union[Iterator[IO[bytes]], Iterator[IO[str]]]

Similar to readFile, but returns a context manager yielding a file handle which can be read from. The yielded file handle does not need to and should not be closed explicitly.

Parameters

- **file_id** (*str*) – ID of the file to get a readable file handle for
- **encoding** (*str*) – the name of the encoding used to decode the file. Encodings are the same as for decode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Returns
a context manager yielding a file handle which can be read from

Return type
Iterator[Union[IO[bytes], IO[str]]]

write_shared_file_stream(*shared_file_name*, *encrypted=None*, *encoding=None*, *errors=None*)

Returns a context manager yielding a writable file handle to the global file referenced by the given name. File will be created in an atomic manner.

Parameters

- **shared_file_name** (*str*) – A file name matching AbstractJobStore.fileNameRegex, unique within this job store
- **encrypted** (*bool*) – True if the file must be encrypted, None if it may be encrypted or False if it must be stored in the clear.
- **encoding** (*str*) – the name of the encoding used to encode the file. Encodings are the same as for encode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Raises

ConcurrentFileModificationException – if the file was modified concurrently during an invocation of this method

Returns

a context manager yielding a writable file handle

Return type

Iterator[IO[bytes]]

read_shared_file_stream(*shared_file_name*, *encoding=None*, *errors=None*)

Returns a context manager yielding a readable file handle to the global file referenced by the given name.

Parameters

- **shared_file_name** (*str*) – A file name matching AbstractJobStore.fileNameRegex, unique within this job store
- **encoding** (*str*) – the name of the encoding used to decode the file. Encodings are the same as for decode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Returns

a context manager yielding a readable file handle

Return type

Iterator[IO[bytes]]

write_logs(*msg*)

Stores a message as a log in the jobstore.

Parameters

msg (*str*) – the string to be written

Raises

ConcurrentFileModificationException – if the file was modified concurrently during an invocation of this method

read_logs(*callback*, *read_all=False*)

Reads logs accumulated by the write_logs() method. For each log this method calls the given callback function with the message as an argument (rather than returning logs directly, this method must be supplied with a callback which will process log messages).

Only unread logs will be read unless the read_all parameter is set.

Parameters

- **callback** (*Callable*) – a function to be applied to each of the stats file handles found
- **read_all** (*bool*) – a boolean indicating whether to read the already processed stats files in addition to the unread stats files

Raises

ConcurrentFileModificationException – if the file was modified concurrently during an invocation of this method

Returns

the number of stats files processed

Return type

int

toil.jobStores.googleJobStore

Module Contents

Classes

GoogleJobStore	Represents the physical storage for the jobs and files in a Toil workflow.

Functions

google_retry_predicate(e)	necessary because under heavy load google may throw
google_retry(f)	This decorator retries the wrapped function if google throws any angry service

Attributes

log
GOOGLE_STORAGE
MAX_BATCH_SIZE

toil.jobStores.googleJobStore.**log**

toil.jobStores.googleJobStore.**GOOGLE_STORAGE = 'gs'**

toil.jobStores.googleJobStore.**MAX_BATCH_SIZE = 1000**

toil.jobStores.googleJobStore.**google_retry_predicate**(*e*)

necessary because under heavy load google may throw
TooManyRequests: 429 The project exceeded the rate limit for creating and deleting buckets.

or numerous other server errors which need to be retried.

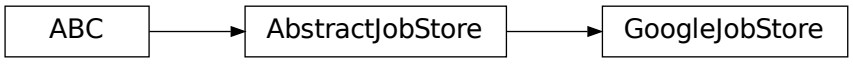
toil.jobStores.googleJobStore.**google_retry**(*f*)

This decorator retries the wrapped function if google throws any angry service errors.

It should wrap any function that makes use of the Google Client API

class toil.jobStores.googleJobStore.**GoogleJobStore**(*locator*)

Bases: *toil.jobStores.abstractJobStore.AbstractJobStore*



Represents the physical storage for the jobs and files in a Toil workflow.

JobStores are responsible for storing *toil.job.JobDescription* (which relate jobs to each other) and files.

Actual *toil.job.Job* objects are stored in files, referenced by JobDescriptions. All the non-file CRUD methods the JobStore provides deal in JobDescriptions and not full, executable Jobs.

To actually get ahold of a *toil.job.Job*, use *toil.job.Job.loadJob()* with a JobStore and the relevant JobDescription.

Parameters
locator (*str*) –

nodeServiceAccountJson = '/root/service_account.json'

initialize(*config=None*)

Initialize this job store.

Create the physical storage for this job store, allocate a workflow ID and persist the given Toil configuration to the store.

Parameters
config – the Toil configuration to initialize this job store with. The given configuration will be updated with the newly allocated workflow ID.

Raises
JobStoreExistsException – if the physical storage for this job store already exists

resume()

Connect this instance to the physical storage it represents and load the Toil configuration into the AbstractJobStore.config attribute.

Raises
NoSuchJobStoreException – if the physical storage for this job store doesn't exist

destroy()

The inverse of *initialize()*, this method deletes the physical storage represented by this instance. While not being atomic, this method *is* at least idempotent, as a means to counteract potential issues with eventual consistency exhibited by the underlying storage mechanisms. This means that if the method fails (raises an exception), it may (and should be) invoked again. If the underlying storage mechanism is eventually consistent, even a successful invocation is not an ironclad guarantee that the physical storage vanished completely and immediately. A successful invocation only guarantees that the deletion will eventually happen. It is therefore recommended to not immediately reuse the same job store location for a new Toil workflow.

assign_job_id(*job_description*)

Get a new jobStoreID to be used by the described job, and assigns it to the JobDescription.

Files associated with the assigned ID will be accepted even if the JobDescription has never been created or updated.

Parameters
job_description (toil.job.JobDescription) – The JobDescription to give an ID to

batch()

If supported by the batch system, calls to create() with this context manager active will be performed in a batch after the context manager is released.

create_job(*job_description*)

Writes the given JobDescription to the job store. The job must have an ID assigned already.

Must call jobDescription.pre_update_hook()

Returns
The JobDescription passed.

Return type
toil.job.JobDescription

job_exists(*job_id*)

Indicates whether a description of the job with the specified jobStoreID exists in the job store

Return type
bool

get_public_url(*fileName*)

Returns a publicly accessible URL to the given file in the job store. The returned URL may expire as early as 1h after its been returned. Throw an exception if the file does not exist.

Parameters
file_name (*str*) – the jobStoreFileID of the file to generate a URL for

Raises
NoSuchFileException – if the specified file does not exist in this job store

Return type
str

get_shared_public_url(*sharedFileName*)

Differs from `getPublicUrl()` in that this method is for generating URLs for shared files written by `writeSharedFileStream()`.

Returns a publicly accessible URL to the given file in the job store. The returned URL starts with 'http:', 'https:' or 'file:'. The returned URL may expire as early as 1h after its been returned. Throw an exception if the file does not exist.

Parameters
shared_file_name (*str*) – The name of the shared file to generate a publically accessible url for.

Raises
NoSuchFileException – raised if the specified file does not exist in the store

Return type
str

load_job(*job_id*)

Loads the description of the job referenced by the given ID, assigns it the job store's config, and returns it.

May declare the job to have failed (see *toil.job.JobDescription.setupJobAfterFailure()*) if there is evidence of a failed update attempt.

Parameters
job_id – the ID of the job to load

Raises
NoSuchJobException – if there is no job with the given ID

update_job(*job*)

Persists changes to the state of the given JobDescription in this store atomically.

Must call jobDescription.pre_update_hook()

Parameters
job (*toil.job.JobDescription*) – the job to write to this job store

delete_job(*job_id*)

Removes the JobDescription from the store atomically. You may not then subsequently call load(), write(), update(), etc. with the same jobStoreID or any JobDescription bearing it.

This operation is idempotent, i.e. deleting a job twice or deleting a non-existent job will succeed silently.

Parameters
job_id (*str*) – the ID of the job to delete from this job store

get_env()

Return a dict of environment variables to send out to the workers so they can load the job store.

jobs()

Best effort attempt to return iterator on JobDescriptions for all jobs in the store. The iterator may not return all jobs and may also contain orphaned jobs that have already finished successfully and should not be rerun. To guarantee you get any and all jobs that can be run instead construct a more expensive ToilState object

Returns
Returns iterator on jobs in the store. The iterator may or may not contain all jobs and may contain invalid jobs

Return type
Iterator[toil.job.jobDescription]

write_file(*local_path*, *job_id=None*, *cleanup=False*)

Takes a file (as a path) and places it in this job store. Returns an ID that can be used to retrieve the file at a later time. The file is written in a atomic manner. It will not appear in the jobStore until the write has successfully completed.

Parameters

- **local_path** (*str*) – the path to the local file that will be uploaded to the job store. The last path component (basename of the file) will remain associated with the file in the file store, if supported, so that the file can be searched for by name or name glob.
- **job_id** (*str*) – the id of a job, or None. If specified, the may be associated with that job in a job-store-specific way. This may influence the returned ID.
- **cleanup** (*bool*) – Whether to attempt to delete the file when the job whose jobStoreID was given as jobStoreID is deleted with jobStore.delete(job). If jobStoreID was not given, does nothing.

Raises

- *ConcurrentFileModificationException* – if the file was modified concurrently during an invocation of this method
- *NoSuchJobException* – if the job specified via jobStoreID does not exist

FIXME: some implementations may not raise this

Returns
an ID referencing the newly created file and can be used to read the file in the future.

Return type
str

write_file_stream(*job_id=None*, *cleanup=False*, *basename=None*, *encoding=None*, *errors=None*)

Similar to writeFile, but returns a context manager yielding a tuple of 1) a file handle which can be written to and 2) the ID of the resulting file in the job store. The yielded file handle does not need to and should not be closed explicitly. The file is written in a atomic manner. It will not appear in the jobStore until the write has successfully completed.

Parameters

- **job_id** (*str*) – the id of a job, or None. If specified, the may be associated with that job in a job-store-specific way. This may influence the returned ID.
- **cleanup** (*bool*) – Whether to attempt to delete the file when the job whose jobStoreID was given as jobStoreID is deleted with jobStore.delete(job). If jobStoreID was not given, does nothing.
- **basename** (*str*) – If supported by the implementation, use the given file basename so that when searching the job store with a query matching that basename, the file will be detected.
- **encoding** (*str*) – the name of the encoding used to encode the file. Encodings are the same as for encode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Raises

- *ConcurrentFileModificationException* – if the file was modified concurrently during an invocation of this method
- *NoSuchJobException* – if the job specified via jobStoreID does not exist

FIXME: some implementations may not raise this

Returns
a context manager yielding a file handle which can be written to and an ID that references the newly created file and can be used to read the file in the future.

Return type
Iterator[Tuple[IO[bytes], str]]

get_empty_file_store_id(*jobStoreID=None*, *cleanup=False*, *basename=None*)

Creates an empty file in the job store and returns its ID. Call to fileExists(getEmptyFileStoreID(jobStoreID)) will return True.

Parameters

- **job_id** (*str*) – the id of a job, or None. If specified, the may be associated with that job in a job-store-specific way. This may influence the returned ID.
- **cleanup** (*bool*) – Whether to attempt to delete the file when the job whose jobStoreID was given as jobStoreID is deleted with jobStore.delete(job). If jobStoreID was not given, does nothing.
- **basename** (*str*) – If supported by the implementation, use the given file basename so that when searching the job store with a query matching that basename, the file will be detected.

Returns
a jobStoreFileID that references the newly created file and can be used to reference the file in the future.

Return type
str

read_file(*file_id*, *local_path*, *symlink=False*)

Copies or hard links the file referenced by jobStoreFileID to the given local file path. The version will be consistent with the last copy of the file written/updated. If the file in the job store is later modified via updateFile or updateFileStream, it is implementation-defined whether those writes will be visible at localFilePath. The file is copied in an atomic manner. It will not appear in the local file system until the copy has completed.

The file at the given local path may not be modified after this method returns!

Note! Implementations of readFile need to respect/provide the executable attribute on FileIDs.

Parameters

- **file_id** (*str*) – ID of the file to be copied
- **local_path** (*str*) – the local path indicating where to place the contents of the given file in the job store
- **symlink** (*bool*) – whether the reader can tolerate a symlink. If set to true, the job store may create a symlink instead of a full copy of the file or a hard link.

read_file_stream(*file_id*, *encoding=None*, *errors=None*)

Similar to readFile, but returns a context manager yielding a file handle which can be read from. The yielded file handle does not need to and should not be closed explicitly.

Parameters

- **file_id** (*str*) – ID of the file to get a readable file handle for
- **encoding** (*str*) – the name of the encoding used to decode the file. Encodings are the same as for decode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Returns
a context manager yielding a file handle which can be read from

Return type
Iterator[Union[IO[bytes], IO[str]]]

`delete_file`(*file_id*)

Deletes the file with the given ID from this job store. This operation is idempotent, i.e. deleting a file twice or deleting a non-existent file will succeed silently.

Parameters
file_id (*str*) – ID of the file to delete

`file_exists`(*file_id*)

Determine whether a file exists in this job store.

Parameters
file_id – an ID referencing the file to be checked

`get_file_size`(*file_id*)

Get the size of the given file in bytes, or 0 if it does not exist when queried.

Note that job stores which encrypt files might return overestimates of file sizes, since the encrypted file may have been padded to the nearest block, augmented with an initialization vector, etc.

Parameters
file_id (*str*) – an ID referencing the file to be checked

Return type
int

`update_file`(*file_id*, *local_path*)

Replaces the existing version of a file in the job store.

Throws an exception if the file does not exist.

Parameters

- **file_id** – the ID of the file in the job store to be updated
- **local_path** – the local path to a file that will overwrite the current version in the job store

Raises

- *ConcurrentFileModificationException* – if the file was modified concurrently during an invocation of this method
- *NoSuchFileException* – if the specified file does not exist

`update_file_stream`(*file_id*, *encoding=None*, *errors=None*)

Replaces the existing version of a file in the job store. Similar to writeFile, but returns a context manager yielding a file handle which can be written to. The yielded file handle does not need to and should not be closed explicitly.

Parameters

- **file_id** (*str*) – the ID of the file in the job store to be updated
- **encoding** (*str*) – the name of the encoding used to encode the file. Encodings are the same as for encode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Raises

- *ConcurrentFileModificationException* – if the file was modified concurrently during an invocation of this method
- *NoSuchFileException* – if the specified file does not exist

write_shared_file_stream(*shared_file_name*, *encrypted=True*, *encoding=None*, *errors=None*)

Returns a context manager yielding a writable file handle to the global file referenced by the given name. File will be created in an atomic manner.

Parameters

- **shared_file_name** (*str*) – A file name matching AbstractJobStore.fileNameRegex, unique within this job store
- **encrypted** (*bool*) – True if the file must be encrypted, None if it may be encrypted or False if it must be stored in the clear.
- **encoding** (*str*) – the name of the encoding used to encode the file. Encodings are the same as for encode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Raises

ConcurrentFileModificationException – if the file was modified concurrently during an invocation of this method

Returns

a context manager yielding a writable file handle

Return type

Iterator[IO[bytes]]

read_shared_file_stream(*shared_file_name*, *isProtected=True*, *encoding=None*, *errors=None*)

Returns a context manager yielding a readable file handle to the global file referenced by the given name.

Parameters

- **shared_file_name** (*str*) – A file name matching AbstractJobStore.fileNameRegex, unique within this job store
- **encoding** (*str*) – the name of the encoding used to decode the file. Encodings are the same as for decode(). Defaults to None which represents binary mode.
- **errors** (*str*) – an optional string that specifies how encoding errors are to be handled. Errors are the same as for open(). Defaults to 'strict' when an encoding is specified.

Returns

a context manager yielding a readable file handle

Return type

Iterator[IO[bytes]]

classmethod get_size(*url*)

Get the size in bytes of the file at the given URL, or None if it cannot be obtained.

Parameters

src_uri – URL that points to a file or object in the storage mechanism of a supported URL scheme e.g. a blob in an AWS s3 bucket.

write_logs(*msg*)

Stores a message as a log in the jobstore.

Parameters
msg (*str*) – the string to be written

Raises
ConcurrentFileModificationException – if the file was modified concurrently during an invocation of this method

Return type
None

read_logs(*callback*, *read_all=False*)

Reads logs accumulated by the write_logs() method. For each log this method calls the given callback function with the message as an argument (rather than returning logs directly, this method must be supplied with a callback which will process log messages).

Only unread logs will be read unless the read_all parameter is set.

Parameters

- **callback** (*Callable*) – a function to be applied to each of the stats file handles found
- **read_all** (*bool*) – a boolean indicating whether to read the already processed stats files in addition to the unread stats files

Raises
ConcurrentFileModificationException – if the file was modified concurrently during an invocation of this method

Returns
the number of stats files processed

Return type
int

toil.jobStores.utils

Module Contents

Classes

WritablePipe	An object-oriented wrapper for os.pipe. Clients should subclass it, implement
ReadablePipe	An object-oriented wrapper for os.pipe. Clients should subclass it, implement
ReadableTransformingPipe	A pipe which is constructed around a readable stream, and which provides a

Functions

generate_locator(job_store_type[, local_suggestion, ...])	Generate a random locator for a job store of the given type. Raises an

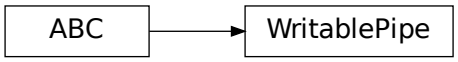
Attributes

log

toil.jobStores.utils.**log**

class toil.jobStores.utils.**WritablePipe**(*encoding=None*, *errors=None*)

Bases: abc.ABC



An object-oriented wrapper for os.pipe. Clients should subclass it, implement *readFrom()* to consume the readable end of the pipe, then instantiate the class as a context manager to get the writable end. See the example below.

```
>>> import sys, shutil
>>> class MyPipe(WritablePipe):
...     def readFrom(self, readable):
...         shutil.copyfileobj(codecs.getreader('utf-8')(readable), sys.stdout)
>>> with MyPipe() as writable:
...     _ = writable.write('Hello, world!\n'.encode('utf-8'))
Hello, world!
```

Each instance of this class creates a thread and invokes the readFrom method in that thread. The thread will be join()ed upon normal exit from the context manager, i.e. the body of the *with* statement. If an exception occurs, the thread will not be joined but a well-behaved *readFrom()* implementation will terminate shortly thereafter due to the pipe having been closed.

Now, exceptions in the reader thread will be reraised in the main thread:

```
>>> class MyPipe(WritablePipe):
...     def readFrom(self, readable):
...         raise RuntimeError('Hello, world!')
>>> with MyPipe() as writable:
...     pass
Traceback (most recent call last):
...
RuntimeError: Hello, world!
```

More complicated, less illustrative tests:

Same as above, but proving that handles are closed:

```
>>> x = os.dup(0); os.close(x)
>>> class MyPipe(WritablePipe):
...     def readFrom(self, readable):
...         raise RuntimeError('Hello, world!')
>>> with MyPipe() as writable:
...     pass
Traceback (most recent call last):
...
RuntimeError: Hello, world!
>>> y = os.dup(0); os.close(y); x == y
True
```

Exceptions in the body of the with statement aren't masked, and handles are closed:

```
>>> x = os.dup(0); os.close(x)
>>> class MyPipe(WritablePipe):
...     def readFrom(self, readable):
...         pass
>>> with MyPipe() as writable:
...     raise RuntimeError('Hello, world!')
Traceback (most recent call last):
...
RuntimeError: Hello, world!
>>> y = os.dup(0); os.close(y); x == y
True
```

abstract readFrom(*readable*)

Implement this method to read data from the pipe. This method should support both binary and text mode output.

Parameters

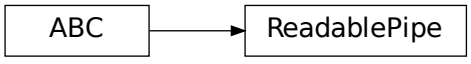
readable (*file*) – the file object representing the readable end of the pipe. Do not explicitly invoke the close() method of the object, that will be done automatically.

__enter__()

__exit__(*exc_type*, *exc_val*, *exc_tb*)

class toil.jobStores.utils.**ReadablePipe**(*encoding=None*, *errors=None*)

Bases: abc.ABC



An object-oriented wrapper for os.pipe. Clients should subclass it, implement *writeTo()* to place data into the writable end of the pipe, then instantiate the class as a context manager to get the writable end. See the example

below.

```
>>> import sys, shutil
>>> class MyPipe(ReadablePipe):
...     def writeTo(self, writable):
...         writable.write('Hello, world!\n'.encode('utf-8'))
>>> with MyPipe() as readable:
...     shutil.copyfileobj(codecs.getreader('utf-8')(readable), sys.stdout)
Hello, world!
```

Each instance of this class creates a thread and invokes the *writeTo()* method in that thread. The thread will be join()ed upon normal exit from the context manager, i.e. the body of the *with* statement. If an exception occurs, the thread will not be joined but a well-behaved *writeTo()* implementation will terminate shortly thereafter due to the pipe having been closed.

Now, exceptions in the reader thread will be reraised in the main thread:

```
>>> class MyPipe(ReadablePipe):
...     def writeTo(self, writable):
...         raise RuntimeError('Hello, world!')
>>> with MyPipe() as readable:
...     pass
Traceback (most recent call last):
...
RuntimeError: Hello, world!
```

More complicated, less illustrative tests:

Same as above, but proving that handles are closed:

```
>>> x = os.dup(0); os.close(x)
>>> class MyPipe(ReadablePipe):
...     def writeTo(self, writable):
...         raise RuntimeError('Hello, world!')
>>> with MyPipe() as readable:
...     pass
Traceback (most recent call last):
...
RuntimeError: Hello, world!
>>> y = os.dup(0); os.close(y); x == y
True
```

Exceptions in the body of the with statement aren't masked, and handles are closed:

```
>>> x = os.dup(0); os.close(x)
>>> class MyPipe(ReadablePipe):
...     def writeTo(self, writable):
...         pass
>>> with MyPipe() as readable:
...     raise RuntimeError('Hello, world!')
Traceback (most recent call last):
...
RuntimeError: Hello, world!
>>> y = os.dup(0); os.close(y); x == y
True
```

abstract writeTo(*writable*)

Implement this method to write data from the pipe. This method should support both binary and text mode input.

Parameters

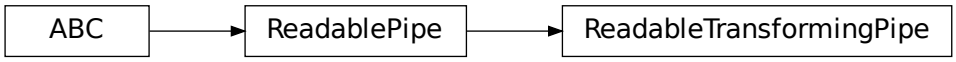
writable (*file*) – the file object representing the writable end of the pipe. Do not explicitly invoke the close() method of the object, that will be done automatically.

__enter__()

__exit__(*exc_type*, *exc_val*, *exc_tb*)

class toil.jobStores.utils.**ReadableTransformingPipe**(*source*, *encoding=None*, *errors=None*)

Bases: *ReadablePipe*



A pipe which is constructed around a readable stream, and which provides a context manager that gives a readable stream.

Useful as a base class for pipes which have to transform or otherwise visit bytes that flow through them, instead of just consuming or producing data.

Clients should subclass it and implement *transform()*, like so:

```
>>> import sys, shutil
>>> class MyPipe(ReadableTransformingPipe):
...     def transform(self, readable, writable):
...         writable.write(readable.read().decode('utf-8').upper().encode('utf-8'))
>>> class SourcePipe(ReadablePipe):
...     def writeTo(self, writable):
...         writable.write('Hello, world!\n'.encode('utf-8'))
>>> with SourcePipe() as source:
...     with MyPipe(source) as transformed:
...         shutil.copyfileobj(codecs.getreader('utf-8')(transformed), sys.stdout)
HELLO, WORLD!
```

The *transform()* method runs in its own thread, and should move data chunk by chunk instead of all at once. It should finish normally if it encounters either an EOF on the readable, or a BrokenPipeError on the writable. This means that it should make sure to actually catch a BrokenPipeError when writing.

See also: toil.lib.misc.WriteWatchingStream.

abstract transform(*readable*, *writable*)

Implement this method to ship data through the pipe.

Parameters

- **readable** (*file*) – the input stream file object to transform.
- **writable** (*file*) – the file object representing the writable end of the pipe. Do not

explicitly invoke the close() method of the object, that will be done automatically.

writeTo(*writable*)

Implement this method to write data from the pipe. This method should support both binary and text mode input.

Parameters

writable (*file*) – the file object representing the writable end of the pipe. Do not

explicitly invoke the close() method of the object, that will be done automatically.

exception toil.jobStores.utils.**JobStoreUnavailableException**

Bases: RuntimeError

Raised when a particular type of job store is requested but can't be used.

toil.jobStores.utils.**generate_locator**(*job_store_type*, *local_suggestion=None*, *decoration=None*)

Generate a random locator for a job store of the given type. Raises an JobStoreUnavailableException if that job store cannot be used.

Parameters

- **job_store_type** (*str*) – Registry name of the job store to use.
- **local_suggestion** (*Optional[str]*) – Path to a nonexistent local directory suitable for
- **decoration** (*Optional[str]*) –

Return type

str

use as a file job store. :param decoration: Extra string to add to the job store locator, if convenient.

Return str

Job store locator for a usable job store.

Parameters

- **job_store_type** (*str*) –
- **local_suggestion** (*Optional[str]*) –
- **decoration** (*Optional[str]*) –

Return type

str

toil.lib

Subpackages

toil.lib.aws

Submodules

toil.lib.aws.ami

Module Contents

Functions

get_flatcar_ami(ec2_client[, architecture])	Retrieve the flatcar AMI image to use as the base for all Toil autoscaling instances.
flatcar_release_feed_amis(region[, architecture, source])	Yield AMI IDs for the given architecture from the Flatcar release feed.
feed_flatcar_ami_release(ec2_client[, architecture, ...])	Check a Flatcar release feed for the latest flatcar AMI.
aws_marketplace_flatcar_ami_search(ec2_client[, ...])	Query AWS for all AMI names matching 'Flatcar-stable-*' and return the most recent one.

Attributes

logger

toil.lib.aws.ami.**logger**

toil.lib.aws.ami.**get_flatcar_ami**(*ec2_client*, *architecture='amd64'*)

Retrieve the flatcar AMI image to use as the base for all Toil autoscaling instances.

AMI must be available to the user on AWS (attempting to launch will return a 403 otherwise).

Priority is:

1. User specified AMI via TOIL_AWS_AMI
2. Official AMI from stable.release.flatcar-linux.net
3. Search the AWS Marketplace

If all of these sources fail, we raise an error to complain.

Parameters

- **ec2_client** (*botocore.client.BaseClient*) – Boto3 EC2 Client
- **architecture** (*str*) – The architecture type for the new AWS machine. Can be either amd64 or arm64

Return type

str

toil.lib.aws.ami.**flatcar_release_feed_amis**(*region*, *architecture='amd64'*, *source='stable'*)

Yield AMI IDs for the given architecture from the Flatcar release feed.

Parameters

- **source** (*str*) – can be set to a Flatcar release channel ('stable', 'beta', or 'alpha'), 'archive' to check the Internet Archive for a feed, and 'toil' to check if the Toil project has put up a feed.
- **region** (*str*) –
- **architecture** (*str*) –

Return type
Iterator[str]

Retries if the release feed cannot be fetched. If the release feed has a permanent error, yields nothing. If some entries in the release feed are unparseable, yields the others.

toil.lib.aws.ami.**feed_flatcar_ami_release**(*ec2_client*, *architecture='amd64'*, *source='stable'*)

Check a Flatcar release feed for the latest flatcar AMI.

Verify it's on AWS.

Parameters

- **ec2_client** (*botocore.client.BaseClient*) – Boto3 EC2 Client
- **architecture** (*str*) – The architecture type for the new AWS machine. Can be either amd64 or arm64
- **source** (*str*) – can be set to a Flatcar release channel ('stable', 'beta', or 'alpha'), 'archive' to check the Internet Archive for a feed, and 'toil' to check if the Toil project has put up a feed.

Return type
Optional[str]

toil.lib.aws.ami.**aws_marketplace_flatcar_ami_search**(*ec2_client*, *architecture='amd64'*)

Query AWS for all AMI names matching 'Flatcar-stable-*' and return the most recent one.

Parameters

- **ec2_client** (*botocore.client.BaseClient*) –
- **architecture** (*str*) –

Return type
Optional[str]

toil.lib.aws.iam

Module Contents

Functions

init_action_collection()	Initialization of an action collection, an action collection contains allowed Actions and NotActions
add_to_action_collection(a, b)	Combines two action collections
policy_permissions_allow(given_permissions[, ...])	Check whether given set of actions are a subset of another given set of actions, returns true if they are
permission_matches_any(perm, list_perms)	Takes a permission and checks whether it's contained within a list of given permissions
get_actions_from_policy_document(policy_doc)	Given a policy document, go through each statement and create an AllowedActionCollection representing the
allowed_actions_attached(iam, attached_policies)	Go through all attached policy documents and create an AllowedActionCollection representing granted permissions.
allowed_actions_roles(iam, policy_names, role_name)	Returns a dictionary containing a list of all aws actions allowed for a given role.
allowed_actions_users(iam, policy_names, user_name)	Gets all allowed actions for a user given by user_name, returns a dictionary, keyed by resource,
get_policy_permissions(region)	Returns an action collection containing lists of all permission grant patterns keyed by resource
get_aws_account_num()	Returns AWS account num

Attributes

logger
CLUSTER_LAUNCHING_PERMISSIONS
AllowedActionCollection

toil.lib.aws.iam.**logger**

toil.lib.aws.iam.**CLUSTER_LAUNCHING_PERMISSIONS = ['iam:CreateRole', 'iam:CreateInstanceProfile', 'iam:TagInstanceProfile', 'iam:DeleteRole',...**

toil.lib.aws.iam.**AllowedActionCollection**

toil.lib.aws.iam.**init_action_collection**()

Initialization of an action collection, an action collection contains allowed Actions and NotActions by resource, these are patterns containing wildcards, an Action explicitly allows a matched pattern, eg ec2:* will explicitly allow all ec2 permissions

A NotAction will explicitly allow all actions that don't match a specific pattern eg iam:* allows all non iam actions

Return type
AllowedActionCollection

toil.lib.aws.iam.**add_to_action_collection**(*a*, *b*)

Combines two action collections

Parameters

- **a** (*AllowedActionCollection*) –
- **b** (*AllowedActionCollection*) –

Return type
AllowedActionCollection

toil.lib.aws.iam.**policy_permissions_allow**(*given_permissions*, *required_permissions=[]*)

Check whether given set of actions are a subset of another given set of actions, returns true if they are otherwise false and prints a warning.

Parameters

- **required_permissions** (*List[str]*) – Dictionary containing actions required, keyed by resource
- **given_permissions** (*AllowedActionCollection*) – Set of actions that are granted to a user or role

Return type
bool

toil.lib.aws.iam.**permission_matches_any**(*perm*, *list_perms*)

Takes a permission and checks whether it's contained within a list of given permissions Returns True if it is otherwise False

Parameters

- **perm** (*str*) – Permission to check in string form
- **list_perms** (*List[str]*) – Permission list to check against

Return type
bool

toil.lib.aws.iam.**get_actions_from_policy_document**(*policy_doc*)

Given a policy document, go through each statement and create an AllowedActionCollection representing the permissions granted in the policy document.

Parameters
policy_doc (*Dict[str, Any]*) – A policy document to examine

Return type
AllowedActionCollection

toil.lib.aws.iam.**allowed_actions_attached**(*iam*, *attached_policies*)

Go through all attached policy documents and create an AllowedActionCollection representing granted permissions.

Parameters

- **iam** (*mypy_boto3_iam.IAMClient*) – IAM client to use
- **attached_policies** (*List[mypy_boto3_iam.type_defs.AttachedPolicyTypeDef]*) – Attached policies

Return type
AllowedActionCollection

toil.lib.aws.iam.**allowed_actions_roles**(*iam*, *policy_names*, *role_name*)

Returns a dictionary containing a list of all aws actions allowed for a given role. This dictionary is keyed by resource and gives a list of policies allowed on that resource.

Parameters

- **iam** (*mypy_boto3_iam.IAMClient*) – IAM client to use
- **policy_names** (*List[str]*) – Name of policy document associated with a role
- **role_name** (*str*) – Name of role to get associated policies

Return type
AllowedActionCollection

toil.lib.aws.iam.**allowed_actions_users**(*iam*, *policy_names*, *user_name*)

Gets all allowed actions for a user given by user_name, returns a dictionary, keyed by resource, with a list of permissions allowed for each given resource.

Parameters

- **iam** (*mypy_boto3_iam.IAMClient*) – IAM client to use
- **policy_names** (*List[str]*) – Name of policy document associated with a user
- **user_name** (*str*) – Name of user to get associated policies

Return type
AllowedActionCollection

toil.lib.aws.iam.**get_policy_permissions**(*region*)

Returns an action collection containing lists of all permission grant patterns keyed by resource that they are allowed upon. Requires AWS credentials to be associated with a user or assumed role.

Parameters

- **zone** – AWS zone to connect to
- **region** (*str*) –

Return type
AllowedActionCollection

toil.lib.aws.iam.**get_aws_account_num**()

Returns AWS account num

Return type
Optional[str]

toil.lib.aws.session

Module Contents

Classes

AWSConnectionManager	Class that represents a connection to AWS. Caches Boto 3 and Boto 2 objects

Functions

establish_boto3_session([region_name])	Get a Boto 3 session usable by the current thread.
client(service_name[, region_name, endpoint_url, config])	Get a Boto 3 client for a particular AWS service, usable by the current thread.
resource(service_name[, region_name, endpoint_url])	Get a Boto 3 resource for a particular AWS service, usable by the current thread.

Attributes

logger

toil.lib.aws.session.**logger**

class toil.lib.aws.session.**AWSConnectionManager**

Class that represents a connection to AWS. Caches Boto 3 and Boto 2 objects by region.

Access to any kind of item goes through the particular method for the thing you want (session, resource, service, Boto2 Context), and then you pass the region you want to work in, and possibly the type of thing you want, as arguments.

This class is intended to eventually enable multi-region clusters, where connections to multiple regions may need to be managed in the same provisioner.

We also support None for a region, in which case no region will be passed to Boto/Boto3. The caller is responsible for implementing e.g. TOIL_AWS_REGION support.

Since connection objects may not be thread safe (see <https://boto3.amazonaws.com/v1/documentation/api/1.14.31/guide/session.html#multithreading-or-multiprocessing-with-sessions>), one is created for each thread that calls the relevant lookup method.

session(*region*)

Get the Boto3 Session to use for the given region.

Parameters
region (*Optional[str]*) –

Return type
boto3.session.Session

resource(*region*, *service_name*, *endpoint_url=None*)

Get the Boto3 Resource to use with the given service (like 'ec2') in the given region.

Parameters

- **endpoint_url** (*Optional[str]*) – AWS endpoint URL to use for the client. If not specified, a default is used.
- **region** (*Optional[str]*) –
- **service_name** (*str*) –

Return type
boto3.resources.base.ServiceResource

client(*region*, *service_name*, *endpoint_url=None*, *config=None*)

Get the Boto3 Client to use with the given service (like 'ec2') in the given region.

Parameters

- **endpoint_url** (*Optional[str]*) – AWS endpoint URL to use for the client. If not specified, a default is used.
- **config** (*Optional[botocore.client.Config]*) – Custom configuration to use for the client.
- **region** (*Optional[str]*) –
- **service_name** (*str*) –

Return type
botocore.client.BaseClient

boto2(*region*, *service_name*)

Get the connected boto2 connection for the given region and service.

Parameters

- **region** (*Optional[str]*) –
- **service_name** (*str*) –

Return type
boto.connection.AWSAuthConnection

toil.lib.aws.session.**establish_boto3_session**(*region_name=None*)

Get a Boto 3 session usable by the current thread.

This function may not always establish a *new* session; it can be memoized.

Parameters
region_name (*Optional[str]*) –

Return type
boto3.Session

toil.lib.aws.session.**client**(*service_name*, *region_name=None*, *endpoint_url=None*, *config=None*)

Get a Boto 3 client for a particular AWS service, usable by the current thread.

Global alternative to AWSConnectionManager.

Parameters

- **service_name** (*str*) –
- **region_name** (*Optional[str]*) –
- **endpoint_url** (*Optional[str]*) –
- **config** (*Optional[botocore.client.Config]*) –

Return type
botocore.client.BaseClient

toil.lib.aws.session.**resource**(*service_name*, *region_name=None*, *endpoint_url=None*)

Get a Boto 3 resource for a particular AWS service, usable by the current thread.

Global alternative to AWSConnectionManager.

Parameters

- **service_name** (*str*) –
- **region_name** (*Optional[str]*) –
- **endpoint_url** (*Optional[str]*) –

Return type
boto3.resources.base.ServiceResource

toil.lib.aws.utils

Module Contents

Functions

delete_iam_role(role_name[, region, quiet])	
delete_iam_instance_profile(instance_profile_name[, ...])	
delete_sdb_domain(sdb_domain_name[, region, quiet])	
connection_reset(e)	Return true if an error is a connection reset error.
retryable_s3_errors(e)	Return true if this is an error from S3 that looks like we ought to retry our request.
retry_s3([delays, timeout, predicate])	Retry iterator of context managers specifically for S3 operations.
delete_s3_bucket(s3_resource, bucket[, quiet])	Delete the given S3 bucket.
create_s3_bucket(s3_resource, bucket_name, region)	Create an AWS S3 bucket, using the given Boto3 S3 session, with the
enable_public_objects(bucket_name)	Enable a bucket to contain objects which are public.
get_bucket_region(bucket_name[, endpoint_url, ...])	Get the AWS region name associated with the given S3 bucket.
region_to_bucket_location(region)	
bucket_location_to_region(location)	
get_object_for_url(url[, existing])	Extracts a key (object) from a given parsed s3:// URL.
list_objects_for_url(url)	Extracts a key (object) from a given parsed s3:// URL. The URL will be
flatten_tags(tags)	Convert tags from a key to value dict into a list of 'Key': xxx, 'Value': xxx dicts.

Attributes

BotoServerError
logger
THROTTLED_ERROR_CODES

toil.lib.aws.utils.**BotoServerError**

toil.lib.aws.utils.**logger**

toil.lib.aws.utils.**THROTTLED_ERROR_CODES = ['Throttling', 'ThrottlingException', 'ThrottledException', 'RequestThrottledException',...**

toil.lib.aws.utils.**delete_iam_role**(*role_name*, *region=None*, *quiet=True*)

Parameters

- **role_name** (*str*) –
- **region** (*Optional[str]*) –
- **quiet** (*bool*) –

Return type
None

toil.lib.aws.utils.**delete_iam_instance_profile**(*instance_profile_name*, *region=None*, *quiet=True*)

Parameters

- **instance_profile_name** (*str*) –
- **region** (*Optional[str]*) –
- **quiet** (*bool*) –

Return type
None

toil.lib.aws.utils.**delete_sdb_domain**(*sdb_domain_name*, *region=None*, *quiet=True*)

Parameters

- **sdb_domain_name** (*str*) –
- **region** (*Optional[str]*) –
- **quiet** (*bool*) –

Return type
None

toil.lib.aws.utils.**connection_reset**(*e*)

Return true if an error is a connection reset error.

Parameters
e (*Exception*) –

Return type
bool

toil.lib.aws.utils.**retryable_s3_errors**(*e*)

Return true if this is an error from S3 that looks like we ought to retry our request.

Parameters
e (*Exception*) –

Return type
bool

toil.lib.aws.utils.**retry_s3**(*delays=DEFAULT_DELAYS*, *timeout=DEFAULT_TIMEOUT*, *predicate=retryable_s3_errors*)

Retry iterator of context managers specifically for S3 operations.

Parameters

- **delays** (*Iterable[float]*) –
- **timeout** (*float*) –
- **predicate** (*Callable[[Exception], bool]*) –

Return type
Iterator[ContextManager[None]]

toil.lib.aws.utils.**delete_s3_bucket**(*s3_resource*, *bucket*, *quiet=True*)

Delete the given S3 bucket.

Parameters

- **s3_resource** (*mypy_boto3_s3.S3ServiceResource*) –
- **bucket** (*str*) –
- **quiet** (*bool*) –

Return type
None

toil.lib.aws.utils.**create_s3_bucket**(*s3_resource*, *bucket_name*, *region*)

Create an AWS S3 bucket, using the given Boto3 S3 session, with the given name, in the given region.

Supports the us-east-1 region, where bucket creation is special.

ALL S3 bucket creation should use this function.

Parameters

- **s3_resource** (*mypy_boto3_s3.S3ServiceResource*) –
- **bucket_name** (*str*) –
- **region** (*Union[mypy_boto3_s3.literals.BucketLocationConstraintType, Literal[us-east-1]]*) –

Return type
mypy_boto3_s3.service_resource.Bucket

toil.lib.aws.utils.**enable_public_objects**(*bucket_name*)

Enable a bucket to contain objects which are public.

This adjusts the bucket's Public Access Block setting to not block all public access, and also adjusts the bucket's Object Ownership setting to a setting which enables object ACLs.

Does *not* touch the *account*'s Public Access Block setting, which can also interfere here. That is probably best left to the account administrator.

This configuration used to be the default, and is what most of Toil's code is written to expect, but it was changed so that new buckets default to the more restrictive setting <https://aws.amazon.com/about-aws/whats-new/2022/12/amazon-s3-automatically-enable-block-public-access-disable-access-control-lists-buckets-april-2023/>, with the expectation that people would write IAM policies for the buckets to allow public access if needed. Toil expects to be able to make arbitrary objects in arbitrary places public, and naming them all in an IAM policy would be a very awkward way to do it. So we restore the old behavior.

Parameters
bucket_name (*str*) –

Return type
None

toil.lib.aws.utils.**get_bucket_region**(*bucket_name*, *endpoint_url=None*, *only_strategies=None*)

Get the AWS region name associated with the given S3 bucket.

Takes an optional S3 API URL override.

Parameters

- **only_strategies** (*Optional[Set[int]]*) – For testing, use only strategies with 1-based numbers in this set.
- **bucket_name** (*str*) –
- **endpoint_url** (*Optional[str]*) –

Return type
str

toil.lib.aws.utils.**region_to_bucket_location**(*region*)

Parameters
region (*str*) –

Return type
str

toil.lib.aws.utils.**bucket_location_to_region**(*location*)

Parameters
location (*Optional[str]*) –

Return type
str

toil.lib.aws.utils.**get_object_for_url**(*url*, *existing=None*)

Extracts a key (object) from a given parsed s3:// URL.

Parameters

- **existing** (*bool*) – If True, key is expected to exist. If False, key is expected not to exists and it will be created. If None, the key will be created if it doesn't exist.
- **url** (*urllib.parse.ParseResult*) –

Return type
mypy_boto3_s3.service_resource.Object

toil.lib.aws.utils.**list_objects_for_url**(*url*)

Extracts a key (object) from a given parsed s3:// URL. The URL will be supplemented with a trailing slash if it is missing.

Parameters
url (*urllib.parse.ParseResult*) –

Return type
List[str]

toil.lib.aws.utils.**flatten_tags**(*tags*)

Convert tags from a key to value dict into a list of 'Key': xxx, 'Value': xxx dicts.

Parameters
tags (*Dict[str, str]*) –

Return type
List[Dict[str, str]]

Package Contents

Functions

get_current_aws_region()	Return the AWS region that the currently configured AWS zone (see
get_aws_zone_from_environment()	Get the AWS zone from TOIL_AWS_ZONE if set.
get_aws_zone_from_metadata()	Get the AWS zone from instance metadata, if on EC2 and the boto module is
get_aws_zone_from_boto()	Get the AWS zone from the Boto config file, if it is configured and the
get_aws_zone_from_environment_region()	Pick an AWS zone in the region defined by TOIL_AWS_REGION, if it is set.
get_current_aws_zone()	Get the currently configured or occupied AWS zone to use.
zone_to_region(zone)	Get a region (e.g. us-west-2) from a zone (e.g. us-west-1c).
running_on_ec2()	Return True if we are currently running on EC2, and false otherwise.
running_on_ecs()	Return True if we are currently running on Amazon ECS, and false otherwise.
build_tag_dict_from_env([environment])	

Attributes

logger

toil.lib.aws.**logger**

toil.lib.aws.**get_current_aws_region**()

Return the AWS region that the currently configured AWS zone (see get_current_aws_zone()) is in.

Return type
Optional[str]

toil.lib.aws.**get_aws_zone_from_environment**()

Get the AWS zone from TOIL_AWS_ZONE if set.

Return type
Optional[str]

toil.lib.aws.**get_aws_zone_from_metadata**()

Get the AWS zone from instance metadata, if on EC2 and the boto module is available. Otherwise, gets the AWS zone from ECS task metadata, if on ECS.

Return type
Optional[str]

toil.lib.aws.**get_aws_zone_from_boto**()

Get the AWS zone from the Boto config file, if it is configured and the boto module is available.

Return type
Optional[str]

toil.lib.aws.**get_aws_zone_from_environment_region**()

Pick an AWS zone in the region defined by TOIL_AWS_REGION, if it is set.

Return type
Optional[str]

toil.lib.aws.**get_current_aws_zone**()

Get the currently configured or occupied AWS zone to use.

Reports the TOIL_AWS_ZONE environment variable if set.

Otherwise, if we have boto and are running on EC2, or if we are on ECS, reports the zone we are running in.

Otherwise, if we have the TOIL_AWS_REGION variable set, chooses a zone in that region.

Finally, if we have boto2, and a default region is configured in Boto 2, chooses a zone in that region.

Returns None if no method can produce a zone to use.

Return type
Optional[str]

toil.lib.aws.**zone_to_region**(*zone*)

Get a region (e.g. us-west-2) from a zone (e.g. us-west-1c).

Parameters
zone (*str*) –

Return type
str

toil.lib.aws.**running_on_ec2**()

Return True if we are currently running on EC2, and false otherwise.

Return type
bool

toil.lib.aws.**running_on_ecs**()

Return True if we are currently running on Amazon ECS, and false otherwise.

Return type
bool

toil.lib.aws.**build_tag_dict_from_env**(*environment=os.environ*)

Parameters
environment (*MutableMapping[str, str]*) –

Return type
Dict[str, str]

toil.lib.encryption

Submodules

toil.lib.encryption.conftest

Module Contents

toil.lib.encryption.conftest.**collect_ignore = []**

Submodules

toil.lib.accelerators

Accelerator (i.e. GPU) utilities for Toil

Module Contents

Functions

have_working_nvidia_smi()	Return True if the nvidia-smi binary, from nvidia's CUDA userspace
have_working_nvidia_docker_runtime()	Return True if Docker exists and can handle an "nvidia" runtime and the "--gpus" option.
count_nvidia_gpus()	Return the number of nvidia GPUs seen by nvidia-smi, or 0 if it is not working.
get_individual_local_accelerators()	Determine all the local accelerators available. Report each with count 1,
get_restrictive_environment_for_local_accelerators(...)	Get environment variables which can be applied to a process to restrict it

toil.lib.accelerators.**have_working_nvidia_smi**()

Return True if the nvidia-smi binary, from nvidia's CUDA userspace utilities, is installed and can be run successfully.

TODO: This isn't quite the same as the check that cwltool uses to decide if it can fulfill a CUDARequirement.

Return type
bool

toil.lib.accelerators.**have_working_nvidia_docker_runtime**()

Return True if Docker exists and can handle an "nvidia" runtime and the "–gpus" option.

Return type
bool

toil.lib.accelerators.**count_nvidia_gpus**()

Return the number of nvidia GPUs seen by nvidia-smi, or 0 if it is not working.

Return type
int

toil.lib.accelerators.**get_individual_local_accelerators**()

Determine all the local accelerators available. Report each with count 1, in the order of the number that can be used to assign them.

TODO: How will numbers work with multiple types of accelerator? We need an accelerator assignment API.

Return type
List[*toil.job.AcceleratorRequirement*]

toil.lib.accelerators.**get_restrictive_environment_for_local_accelerators**(*accelerator_numbers*)

Get environment variables which can be applied to a process to restrict it to using only the given accelerator numbers.

The numbers are in the space of accelerators returned by get_individual_local_accelerators().

Parameters
accelerator_numbers (*Set[int]*) –

Return type
Dict[str, str]

toil.lib.bioio

Module Contents

Functions

system(command)	A convenience wrapper around subprocess.check_call that logs the command before passing it
getLogLevelString([logger])	
setLoggingFromOptions(options)	
getTempFile([suffix, rootDir])	

toil.lib.bioio.**system**(*command*)

A convenience wrapper around subprocess.check_call that logs the command before passing it on. The command can be either a string or a sequence of strings. If it is a string shell=True will be passed to subprocess.check_call. :type command: str | sequence[string]

toil.lib.bioio.**getLogLevelString**(*logger=None*)

toil.lib.bioio.**setLoggingFromOptions**(*options*)

toil.lib.bioio.**getTempFile**(*suffix=''*, *rootDir=None*)

toil.lib.compatibility

Module Contents

Functions

deprecated(new_function_name)	
compat_bytes(s)	
compat_bytes_recursive(data)	Convert a tree of objects over bytes to objects over strings.

toil.lib.compatibility.**deprecated**(*new_function_name*)

Parameters
new_function_name (*str*) –

Return type
Callable[Ellipsis, Any]

toil.lib.compatibility.**compat_bytes**(*s*)

Parameters
s (*Union[bytes, str]*) –

Return type
str

toil.lib.compatibility.**compat_bytes_recursive**(*data*)

Convert a tree of objects over bytes to objects over strings.

Parameters
data (*Any*) –

Return type
Any

toil.lib.conversions

Conversion utilities for mapping memory, disk, core declarations from strings to numbers and vice versa. Also contains general conversion functions

Module Contents

Functions

bytes_in_unit([unit])	
convert_units(num, src_unit[, dst_unit])	Returns a float representing the converted input in dst_units.
parse_memory_string(string)	Given a string representation of some memory (i.e. '1024 Mib'), return the
human2bytes(string)	Given a string representation of some memory (i.e. '1024 Mib'), return the
bytes2human(n)	Return a binary value as a human readable string with units.
b_to_mib(n)	Convert a number from bytes to mibibytes.
mib_to_b(n)	Convert a number from mibibytes to bytes.
hms_duration_to_seconds(hms)	Parses a given time string in hours:minutes:seconds,

Attributes

BINARY_PREFIXES
DECIMAL_PREFIXES
VALID_PREFIXES

toil.lib.conversions.**BINARY_PREFIXES = ['ki', 'mi', 'gi', 'ti', 'pi', 'ei', 'kib', 'mib', 'gib', 'tib', 'pib', 'eib']**

toil.lib.conversions.**DECIMAL_PREFIXES = ['b', 'k', 'm', 'g', 't', 'p', 'e', 'kb', 'mb', 'gb', 'tb', 'pb', 'eb']**

toil.lib.conversions.**VALID_PREFIXES**

toil.lib.conversions.**bytes_in_unit**(*unit='B'*)

Parameters
unit (*str*) –

Return type
int

toil.lib.conversions.**convert_units**(*num*, *src_unit*, *dst_unit='B'*)

Returns a float representing the converted input in dst_units.

Parameters

- **num** (*float*) –
- **src_unit** (*str*) –
- **dst_unit** (*str*) –

Return type
float

toil.lib.conversions.**parse_memory_string**(*string*)

Given a string representation of some memory (i.e. '1024 Mib'), return the number and unit.

Parameters
string (*str*) –

Return type
Tuple[float, str]

toil.lib.conversions.**human2bytes**(*string*)

Given a string representation of some memory (i.e. '1024 Mib'), return the integer number of bytes.

Parameters
string (*str*) –

Return type
int

toil.lib.conversions.**bytes2human**(*n*)

Return a binary value as a human readable string with units.

Parameters
n (*SupportsInt*) –

Return type
str

toil.lib.conversions.**b_to_mib**(*n*)

Convert a number from bytes to mibibytes.

Parameters
n (*Union[int, float]*) –

Return type
float

toil.lib.conversions.**mib_to_b**(*n*)

Convert a number from mibibytes to bytes.

Parameters
n (*Union[int, float]*) –

Return type
float

toil.lib.conversions.**hms_duration_to_seconds**(*hms*)

Parses a given time string in hours:minutes:seconds, returns an equivalent total seconds value

Parameters
hms (*str*) –

Return type
float

toil.lib.docker

Module Contents

Functions

dockerCheckOutput(*args, **kwargs)	
dockerCall(*args, **kwargs)	
subprocessDockerCall(*args, **kwargs)	
apiDockerCall(job, image[, parameters, deferParam, ...])	A toil wrapper for the python docker API.
dockerKill(container_name[, gentleKill, remove, timeout])	Immediately kills a container. Equivalent to "docker kill":
dockerStop(container_name[, remove])	Gracefully kills a container. Equivalent to "docker stop":
containerIsRunning(container_name[, timeout])	Checks whether the container is running or not.
getContainerName(job)	Create a random string including the job name, and return it. Name will

Attributes

logger
FORGO
STOP
RM

toil.lib.docker.**logger**

toil.lib.docker.**FORGO = 0**

toil.lib.docker.**STOP = 1**

toil.lib.docker.**RM = 2**

toil.lib.docker.**dockerCheckOutput**(**args*, ***kwargs*)

toil.lib.docker.**dockerCall**(**args*, ***kwargs*)

toil.lib.docker.**subprocessDockerCall**(**args*, ***kwargs*)

toil.lib.docker.**apiDockerCall**(*job*, *image*, *parameters=None*, *deferParam=None*, *volumes=None*, *working_dir=None*, *containerName=None*, *entrypoint=None*, *detach=False*, *log_config=None*, *auto_remove=None*, *remove=False*, *user=None*, *environment=None*, *stdout=None*, *stderr=False*, *stream=False*, *demux=False*, *streamfile=None*, *timeout=365 * 24 * 60 * 60*, ***kwargs*)

A toil wrapper for the python docker API.

Docker API Docs: https://docker-py.readthedocs.io/en/stable/index.html Docker API Code: https://github.com/docker/docker-py

This implements docker's python API within toil so that calls are run as jobs, with the intention that failed/orphaned docker jobs be handled appropriately.

Example of using dockerCall in toil to index a FASTA file with SAMtools: def toil_job(job):

working_dir = job.fileStore.getLocalTempDir() path = job.fileStore.readGlobalFile(ref_id,

os.path.join(working_dir, 'ref.fasta')

parameters = ['faidx', path] apiDockerCall(job,

image='quay.io/ucgc_cgl/samtools:latest', working_dir=working_dir, parameters=parameters)

Note that when run with detach=False, or with detach=True and stdout=True or stderr=True, this is a blocking call. When run with detach=True and without output capture, the container is started and returned without waiting for it to finish.

Parameters

- **job** (*toil.Job.job*) – The Job instance for the calling function.
- **image** (*str*) – Name of the Docker image to be used. (e.g. 'quay.io/ucsc_cgl/samtools:latest')
- **parameters** (*list[str]*) – A list of string elements. If there are multiple elements, these will be joined with spaces. This handling of multiple elements provides backwards compatibility with previous versions which called docker using subprocess.check_call(). **If list of lists: list[list[str]], then treat as successive commands chained with pipe.
- **working_dir** (*str*) – The working directory.
- **deferParam** (*int*) – Action to take on the container upon job completion. FORGO (0) leaves the container untouched and running. STOP (1) Sends SIGTERM, then SIGKILL if necessary to the container. RM (2) Immediately send SIGKILL to the container. This is the default behavior if deferParam is set to None.
- **name** (*str*) – The name/ID of the container.
- **entrypoint** (*str*) – Prepends commands sent to the container. See: https://docker-py.readthedocs.io/en/stable/containers.html
- **detach** (*bool*) – Run the container in detached mode. (equivalent to '-d')
- **stdout** (*bool*) – Return logs from STDOUT when detach=False (default: True). Block and capture stdout to a file when detach=True (default: False). Output capture defaults to output.log, and can be specified with the "streamfile" kwarg.
- **stderr** (*bool*) – Return logs from STDERR when detach=False (default: False). Block and capture stderr to a file when detach=True (default: False). Output capture defaults to output.log, and can be specified with the "streamfile" kwarg.
- **stream** (*bool*) – If True and detach=False, return a log generator instead of a string. Ignored if detach=True. (default: False).
- **demux** (*bool*) – Similar to *demux* in container.exec_run(). If True and detach=False, returns a tuple of (stdout, stderr). If stream=True, returns a log generator with tuples of (stdout, stderr). Ignored if detach=True. (default: False).

- **streamfile** (*str*) – Collect container output to this file if detach=True and stderr and/or stdout are True. Defaults to "output.log".
- **log_config** (*dict*) – Specify the logs to return from the container. See: https://docker-py.readthedocs.io/en/stable/containers.html
- **remove** (*bool*) – Remove the container on exit or not.
- **user** (*str*) – The container will be run with the privileges of the user specified. Can be an actual name, such as 'root' or 'lifeisaboutfishtacos', or it can be the uid or gid of the user ('0' is root; '1000' is an example of a less privileged uid or gid), or a complement of the uid:gid (RECOMMENDED), such as '0:0' (root user : root group) or '1000:1000' (some other user : some other user group).
- **environment** – Allows one to set environment variables inside of the container, such as:
- **timeout** (*int*) – Use the given timeout in seconds for interactions with the Docker daemon. Note that the underlying docker module is not always able to abort ongoing reads and writes in order to respect the timeout. Defaults to 1 year (i.e. wait essentially indefinitely).
- **kwargs** – Additional keyword arguments supplied to the docker API's run command. The list is 75 keywords total, for examples and full documentation see: https://docker-py.readthedocs.io/en/stable/containers.html

Returns

Returns the standard output/standard error text, as requested, when detach=False. Returns the underlying docker.models.containers.Container object from the Docker API when detach=True.

toil.lib.docker.**dockerKill**(*container_name*, *gentleKill=False*, *remove=False*, *timeout=365 * 24 * 60 * 60*)

Immediately kills a container. Equivalent to "docker kill": https://docs.docker.com/engine/reference/commandline/kill/

Parameters

- **container_name** (*str*) – Name of the container being killed.
- **gentleKill** (*bool*) – If True, trigger a graceful shutdown.
- **remove** (*bool*) – If True, remove the container after it exits.
- **timeout** (*int*) – Use the given timeout in seconds for interactions with the Docker daemon. Note that the underlying docker module is not always able to abort ongoing reads and writes in order to respect the timeout. Defaults to 1 year (i.e. wait essentially indefinitely).

Return type

None

toil.lib.docker.**dockerStop**(*container_name*, *remove=False*)

Gracefully kills a container. Equivalent to "docker stop": https://docs.docker.com/engine/reference/commandline/stop/

Parameters

- **container_name** (*str*) – Name of the container being stopped.
- **remove** (*bool*) – If True, remove the container after it exits.

Return type

None

toil.lib.docker.**containerIsRunning**(*container_name*, *timeout=365 * 24 * 60 * 60*)

Checks whether the container is running or not.

Parameters

- **container_name** (*str*) – Name of the container being checked.
- **timeout** (*int*) – Use the given timeout in seconds for interactions with the Docker daemon. Note that the underlying docker module is not always able to abort ongoing reads and writes in order to respect the timeout. Defaults to 1 year (i.e. wait essentially indefinitely).

Returns

True if status is 'running', False if status is anything else,

and None if the container does not exist.

`toil.lib.docker.getContainerName(job)`

Create a random string including the job name, and return it. Name will match [a-zA-Z0-9][a-zA-Z0-9_.-]

toil.lib.ec2

Module Contents

Functions

not_found(e)	
inconsistencies_detected(e)	
retry_ec2([t, retry_for, retry_while])	
wait_transition(resource, from_states, to_state[, ...])	Wait until the specified EC2 resource (instance, image, volume, ...) transitions from any
wait_instances_running(ec2, instances)	Wait until no instance in the given iterable is 'pending'. Yield every instance that
wait_spot_requests_active(ec2, requests[, timeout, ...])	Wait until no spot request in the given iterator is in the 'open' state or, optionally,
create_spot_instances(ec2, price, image_id, spec[, ...])	Create instances on the spot market.
create_ondemand_instances(ec2, image_id, spec[, ...])	Requests the RunInstances EC2 API call but accounts for the race between recently created
prune(bushy)	Prune entries in the given dict with false-y values.
wait_until_instance_profile_arn_exists(...)	
create_instances(ec2_resource, image_id, key_name, ...)	Replaces create_ondemand_instances. Uses boto3 and returns a list of Boto3 instance dicts.
create_launch_template(ec2_client, template_name, ...)	Creates a launch template with the given name for launching instances with the given parameters.
create_auto_scaling_group(autoscaling_client, ...[, ...])	Create a new Auto Scaling Group with the given name (which is also its

Attributes

a_short_time
a_long_time
logger
INCONSISTENCY_ERRORS
iam_client

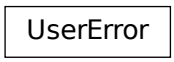
toil.lib.ec2.**a_short_time = 5**

toil.lib.ec2.**a_long_time**

toil.lib.ec2.**logger**

exception toil.lib.ec2.**UserError**(*message=None*, *cause=None*)

Bases: RuntimeError



Unspecified run-time error.

toil.lib.ec2.**not_found**(*e*)

toil.lib.ec2.**inconsistencies_detected**(*e*)

toil.lib.ec2.**INCONSISTENCY_ERRORS**

toil.lib.ec2.**retry_ec2**(*t=a_short_time*, *retry_for=10 * a_short_time*, *retry_while=not_found*)

exception toil.lib.ec2.**UnexpectedResourceState**(*resource*, *to_state*, *state*)

Bases: Exception

Common base class for all non-exit exceptions.

toil.lib.ec2.**wait_transition**(*resource*, *from_states*, *to_state*, *state_getter=attrgetter('state')*)

Wait until the specified EC2 resource (instance, image, volume, ...) transitions from any of the given 'from' states to the specified 'to' state. If the instance is found in a state other that the to state or any of the from states, an exception will be thrown.

Parameters

- **resource** – the resource to monitor
- **from_states** – a set of states that the resource is expected to be in before the transition occurs
- **to_state** – the state of the resource when this method returns

toil.lib.ec2.**wait_instances_running**(*ec2*, *instances*)

Wait until no instance in the given iterable is 'pending'. Yield every instance that entered the running state as soon as it does.

Parameters

- **ec2** (*boto.ec2.connection.EC2Connection*) – the EC2 connection to use for making requests
- **instances** (*Iterable[Boto2Instance]*) – the instances to wait on

Return type
Iterable[Boto2Instance]

toil.lib.ec2.**wait_spot_requests_active**(*ec2*, *requests*, *timeout=None*, *tentative=False*)

Wait until no spot request in the given iterator is in the 'open' state or, optionally, a timeout occurs. Yield spot requests as soon as they leave the 'open' state.

Parameters

- **requests** (*Iterable[boto.ec2.spotinstancerequest.SpotInstanceRequest]*) – The requests to wait on.
- **timeout** (*float*) – Maximum time in seconds to spend waiting or None to wait forever. If a
- **tentative** (*bool*) –

Return type
Iterable[List[boto.ec2.spotinstancerequest.SpotInstanceRequest]]

timeout occurs, the remaining open requests will be cancelled.

Parameters

- **tentative** (*bool*) – if True, give up on a spot request at the earliest indication of it
- **requests** (*Iterable[boto.ec2.spotinstancerequest.SpotInstanceRequest]*) –
- **timeout** (*float*) –

Return type
Iterable[List[boto.ec2.spotinstancerequest.SpotInstanceRequest]]

not being fulfilled immediately

toil.lib.ec2.**create_spot_instances**(*ec2*, *price*, *image_id*, *spec*, *num_instances=1*, *timeout=None*, *tentative=False*, *tags=None*)

Create instances on the spot market.

Return type
Iterable[List[boto.ec2.instance.Instance]]

toil.lib.ec2.**create_ondemand_instances**(*ec2*, *image_id*, *spec*, *num_instances=1*)

Requests the RunInstances EC2 API call but accounts for the race between recently created instance profiles, IAM roles and an instance creation that refers to them.

Return type
List[Boto2Instance]

toil.lib.ec2.**prune**(*bushy*)

Prune entries in the given dict with false-y values. Boto3 may not like None and instead wants no key.

Parameters
bushy (*dict*) –

Return type
dict

toil.lib.ec2.**iam_client**

toil.lib.ec2.**wait_until_instance_profile_arn_exists**(*instance_profile_arn*)

Parameters
instance_profile_arn (*str*) –

toil.lib.ec2.**create_instances**(*ec2_resource*, *image_id*, *key_name*, *instance_type*, *num_instances=1*, *security_group_ids=None*, *user_data=None*, *block_device_map=None*, *instance_profile_arn=None*, *placement_az=None*, *subnet_id=None*, *tags=None*)

Replaces create_ondemand_instances. Uses boto3 and returns a list of Boto3 instance dicts.

See "create_instances" (returns a list of ec2.Instance objects):
https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/ec2.html#EC2.ServiceResource.create_instances

Not to be confused with "run_instances" (same input args; returns a dictionary):
https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/ec2.html#EC2.Client.run_instances

Tags, if given, are applied to the instances, and all volumes.

Parameters

- **ec2_resource** (*boto3.resources.base.ServiceResource*) –
- **image_id** (*str*) –
- **key_name** (*str*) –
- **instance_type** (*str*) –
- **num_instances** (*int*) –
- **security_group_ids** (*Optional[List]*) –
- **user_data** (*Optional[Union[str, bytes]]*) –
- **block_device_map** (*Optional[List[Dict]]*) –
- **instance_profile_arn** (*Optional[str]*) –
- **placement_az** (*Optional[str]*) –
- **subnet_id** (*str*) –

- **tags** (*Optional[Dict[str, str]]*) –

Return type
List[dict]

toil.lib.ec2.**create_launch_template**(*ec2_client*, *template_name*, *image_id*, *key_name*, *instance_type*, *security_group_ids=None*, *user_data=None*, *block_device_map=None*, *instance_profile_arn=None*, *placement_az=None*, *subnet_id=None*, *tags=None*)

Creates a launch template with the given name for launching instances with the given parameters.

We only ever use the default version of any launch template.

Internally calls https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/ec2.html?highlight=create_launch_template#EC2.Client.create_launch_template

Parameters

- **tags** (*Optional[Dict[str, str]]*) – Tags, if given, are applied to the template itself, all instances, and all volumes.
- **user_data** (*Optional[Union[str, bytes]]*) – non-base64-encoded user data to pass to the instances.
- **ec2_client** (*botocore.client.BaseClient*) –
- **template_name** (*str*) –
- **image_id** (*str*) –
- **key_name** (*str*) –
- **instance_type** (*str*) –
- **security_group_ids** (*Optional[List]*) –
- **block_device_map** (*Optional[List[Dict]]*) –
- **instance_profile_arn** (*Optional[str]*) –
- **placement_az** (*Optional[str]*) –
- **subnet_id** (*Optional[str]*) –

Returns
the ID of the launch template.

Return type
str

toil.lib.ec2.**create_auto_scaling_group**(*autoscaling_client*, *asg_name*, *launch_template_ids*, *vpc_subnets*, *min_size*, *max_size*, *instance_types=None*, *spot_bid=None*, *spot_cheapest=False*, *tags=None*)

Create a new Auto Scaling Group with the given name (which is also its unique identifier).

Parameters

- **autoscaling_client** (*botocore.client.BaseClient*) – Boto3 client for autoscaling.
- **asg_name** (*str*) – Unique name for the autoscaling group.
- **launch_template_ids** (*Dict[str, str]*) – ID of the launch template to make instances from, for each instance type.
- **vpc_subnets** (*List[str]*) – One or more subnet IDs to place instances in the group into. Determine the availability zone(s) instances will launch into.

- **min_size** (*int*) – Minimum number of instances to have in the group at all times.
- **max_size** (*int*) – Maximum number of instances to allow in the group at any time.
- **instance_types** (*Optional[List[str]]*) – Use a pool over the given instance types, instead of the type given in the launch template. For on-demand groups, this is a prioritized list. For spot groups, we let AWS balance according to spot_strategy. Must be 20 types or shorter.
- **spot_bid** (*Optional[float]*) – If set, the ASG will be a spot market ASG. Bid is in dollars per instance hour. All instance types in the group are bid on equivalently.
- **spot_cheapest** (*bool*) – If true, use the cheapest spot instances available out of instance_types, instead of the spot instances that minimize eviction probability.
- **tags** (*Optional[Dict[str, str]]*) – Tags to apply to the ASG only. Tags for the instances should be added to the launch template instead.

Return type
None

The default version of the launch template is used.

toil.lib.ec2nodes

Module Contents

Classes

InstanceType	

Functions

isNumber(s)	Determines if a unicode string (that may include commas) is a number.
parseStorage(storageData)	Parses EC2 JSON storage param string into a number.
parseMemory(memAttribute)	Returns EC2 'memory' string as a float.
fetchEC2Index(filename)	Downloads and writes the AWS Billing JSON to a file using the AWS pricing API.
fetchEC2InstanceDict(awsBillingJson, region)	Takes a JSON and returns a list of InstanceType objects representing EC2 instance params.
updateStaticEC2Instances()	Generates a new python file of fetchable EC2 Instances by region with current prices and specs.

Attributes

logger
dirname
EC2Regions

toil.lib.ec2nodes.**logger**

toil.lib.ec2nodes.**dirname**

toil.lib.ec2nodes.**EC2Regions**

class toil.lib.ec2nodes.**InstanceType**(*name*, *cores*, *memory*, *disks*, *disk_capacity*, *architecture*)

Parameters

- **name** (*str*) –
- **cores** (*int*) –
- **memory** (*float*) –
- **disks** (*float*) –
- **disk_capacity** (*float*) –
- **architecture** (*str*) –

__slots__ = ('name', 'cores', 'memory', 'disks', 'disk_capacity', 'architecture')

__str__()

Return str(self).

Return type
str

__eq__(*other*)

Return self==value.

Parameters
other (*object*) –

Return type
bool

toil.lib.ec2nodes.**isNumber**(*s*)

Determines if a unicode string (that may include commas) is a number.

Parameters
s (*str*) – Any unicode string.

Returns
True if s represents a number, False otherwise.

Return type
bool

toil.lib.ec2nodes.**parseStorage**(*storageData*)

Parses EC2 JSON storage param string into a number.

Examples:
"2 x 160 SSD" "3 x 2000 HDD" "EBS only" "1 x 410" "8 x 1.9 NVMe SSD" "900 GB NVMe SSD"

Parameters
storageData (*str*) – EC2 JSON storage param string.

Returns
Two floats representing: (# of disks), and (disk_capacity in GiB of each disk).

Return type
Union[List[int], Tuple[Union[int, float], float]]

toil.lib.ec2nodes.**parseMemory**(*memAttribute*)

Returns EC2 'memory' string as a float.

Format should always be '#' GiB (example: '244 GiB' or '1,952 GiB'). Amazon loves to put commas in their numbers, so we have to accommodate that. If the syntax ever changes, this will raise.

Parameters
memAttribute (*str*) – EC2 JSON memory param string.

Returns
A float representing memory in GiB.

Return type
float

toil.lib.ec2nodes.**fetchEC2Index**(*filename*)

Downloads and writes the AWS Billing JSON to a file using the AWS pricing API.

See: https://aws.amazon.com/blogs/aws/new-aws-price-list-api/

Returns
A dict of InstanceType objects, where the key is the string: aws instance name (example: 't2.micro'), and the value is an InstanceType object representing that aws instance name.

Parameters
filename (*str*) –

Return type
None

toil.lib.ec2nodes.**fetchEC2InstanceDict**(*awsBillingJson*, *region*)

Takes a JSON and returns a list of InstanceType objects representing EC2 instance params.

Parameters

- **region** (*str*) –
- **awsBillingJson** (*Dict[str, Any]*) –

Returns

Return type
Dict[str, *InstanceType*]

toil.lib.ec2nodes.**updateStaticEC2Instances**()

Generates a new python file of fetchable EC2 Instances by region with current prices and specs.

Takes a few (~3+) minutes to run (you'll need decent internet).

Returns
Nothing. Writes a new 'generatedEC2Lists.py' file.

Return type
None

toil.lib.exceptions

Module Contents

Classes

panic	The Python idiom for reraising a primary exception fails when the except block raises a

Functions

*raise_*(exc_type, exc_value, traceback)

class toil.lib.exceptions.**panic**(*log=None*)

The Python idiom for reraising a primary exception fails when the except block raises a secondary exception, e.g. while trying to cleanup. In that case the original exception is lost and the secondary exception is reraised. The solution seems to be to save the primary exception info as returned from sys.exc_info() and then reraise that.

This is a contextmanager that should be used like this

try:
do something that can fail

except:

with panic(log):
do cleanup that can also fail

If a logging logger is passed to panic(), any secondary Exception raised within the with block will be logged. Otherwise those exceptions are swallowed. At the end of the with block the primary exception will be reraised.

__enter__()

__exit__(**exc_info*)

toil.lib.exceptions.**raise_**(*exc_type*, *exc_value*, *traceback*)

Return type
None

toil.lib.expando

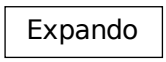
Module Contents

Classes

Expando	Pass inital attributes to the constructor:
MagicExpando	Use MagicExpando for chained attribute access.

class toil.lib.expando.**Expando**(**args*, ***kwargs*)

Bases: dict



Pass inital attributes to the constructor:

```
>>> o = Expando(foo=42)
>>> o.foo
42
```

Dynamically create new attributes:

```
>>> o.bar = 'hi'
>>> o.bar
'hi'
```

Expando is a dictionary:

```
>>> isinstance(o,dict)
True
>>> o['foo']
42
```

Works great with JSON:

```
>>> import json
>>> s='{"foo":42}'
>>> o = json.loads(s,object_hook=Expando)
>>> o.foo
42
>>> o.bar = 'hi'
>>> o.bar
'hi'
```

And since Expando is a dict, it serializes back to JSON just fine:

```
>>> json.dumps(o, sort_keys=True)
'{"bar": "hi", "foo": 42}'
```

Attributes can be deleted, too:

```
>>> o = Expando(foo=42)
>>> o.foo
42
>>> del o.foo
>>> o.foo
Traceback (most recent call last):
...
AttributeError: 'Expando' object has no attribute 'foo'
>>> o['foo']
Traceback (most recent call last):
...
KeyError: 'foo'
```

```
>>> del o.foo
Traceback (most recent call last):
...
AttributeError: foo
```

And copied:

```
>>> o = Expando(foo=42)
>>> p = o.copy()
>>> isinstance(p,Expando)
True
>>> o == p
True
>>> o is p
False
```

Same with MagicExpando ...

```
>>> o = MagicExpando()
>>> o.foo.bar = 42
>>> p = o.copy()
>>> isinstance(p,MagicExpando)
True
>>> o == p
True
>>> o is p
False
```

... but the copy is shallow:

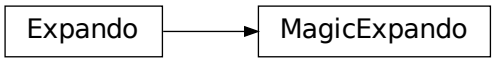
```
>>> o.foo is p.foo
True
```

copy()

D.copy() -> a shallow copy of D

class toil.lib.expando.**MagicExpando**(**args*, ***kwargs*)

Bases: *Expando*



Use MagicExpando for chained attribute access.

The first time a missing attribute is accessed, it will be set to a new child MagicExpando.

```
>>> o=MagicExpando()
>>> o.foo = 42
>>> o
{'foo': 42}
>>> o.bar.hello = 'hi'
>>> o.bar
{'hello': 'hi'}
```

__getattribute__(*name*)

Return getattr(self, name).

Parameters

name (*str*) –

toil.lib.generatedEC2Lists

Module Contents

toil.lib.generatedEC2Lists.**E2Instances**

toil.lib.generatedEC2Lists.**regionDict**

toil.lib.generatedEC2Lists.**ec2InstancesByRegion**

toil.lib.humanize

Module Contents

Functions

bytes2human(n)	Convert n bytes into a human readable string.
human2bytes(s)	Attempts to guess the string format based on default symbols

Attributes

logger	

toil.lib.humanize.**logger**

toil.lib.humanize.**bytes2human**(*n*)

Convert n bytes into a human readable string.

Parameters
n (*SupportsInt*) –

Return type
str

toil.lib.humanize.**human2bytes**(*s*)

Attempts to guess the string format based on default symbols set and return the corresponding bytes as an integer.

When unable to recognize the format ValueError is raised.

Parameters
s (*str*) –

Return type
int

toil.lib.io

Module Contents

Classes

WriteWatchingStream	A stream wrapping class that calls any functions passed to onWrite() with the number of bytes written for every write.

Functions

robust_rmtree(path)	Robustly tries to delete paths.
atomic_tmp_file(final_path)	Return a tmp file name to use with atomic_install. This will be in the
atomic_install(tmp_path, final_path)	atomic install of tmp_path as final_path
AtomicFileCreate(final_path[, keep])	Context manager to create a temporary file. Entering returns path to
atomic_copy(src_path, dest_path[, executable])	Copy a file using posix atomic creations semantics.
atomic_copyobj(src_fh, dest_path[, length, executable])	Copy an open file using posix atomic creations semantics.
make_public_dir([in_directory])	Try to make a random directory name with length 4 that doesn't exist, with the given prefix.
try_path(path)	Try to use the given path. Return it if it exists or can be made,

Attributes

logger

toil.lib.io.**logger**

toil.lib.io.**robust_rmtree**(*path*)

Robustly tries to delete paths.

Continues silently if the path to be removed is already gone, or if it goes away while this function is executing.

May raise an error if a path changes between file and directory while the function is executing, or if a permission error is encountered.

Parameters
path (*Union[str, bytes]*) –

Return type
None

toil.lib.io.**atomic_tmp_file**(*final_path*)

Return a tmp file name to use with atomic_install. This will be in the same directory as final_path. The temporary file will have the same extension as finalPath. It the final path is in /dev (/dev/null, /dev/stdout), it is returned unchanged and atomic_tmp_install will do nothing.

Parameters
final_path (*str*) –

Return type
str

toil.lib.io.**atomic_install**(*tmp_path*, *final_path*)

atomic install of tmp_path as final_path

Return type
None

toil.lib.io.**AtomicFileCreate**(*final_path*, *keep=False*)

Context manager to create a temporary file. Entering returns path to the temporary file in the same directory as finalPath. If the code in context succeeds, the file renamed to its actually name. If an error occurs, the file is not installed and is removed unless keep is specified.

Parameters

- **final_path** (*str*) –
- **keep** (*bool*) –

Return type
Iterator[str]

toil.lib.io.**atomic_copy**(*src_path*, *dest_path*, *executable=None*)

Copy a file using posix atomic creations semantics.

Parameters

- **src_path** (*str*) –
- **dest_path** (*str*) –
- **executable** (*Optional[bool]*) –

Return type
None

toil.lib.io.**atomic_copyobj**(*src_fh*, *dest_path*, *length=16384*, *executable=False*)

Copy an open file using posix atomic creations semantics.

Parameters

- **src_fh** (*io.BytesIO*) –
- **dest_path** (*str*) –
- **length** (*int*) –
- **executable** (*bool*) –

Return type
None

toil.lib.io.**make_public_dir**(*in_directory=None*)

Try to make a random directory name with length 4 that doesn't exist, with the given prefix. Otherwise, try length 5, length 6, etc, up to a max of 32 (len of uuid4 with dashes replaced). This function's purpose is mostly to avoid having long file names when generating directories. If somehow this fails, which should be incredibly unlikely, default to a normal uuid4, which was our old default.

Parameters
in_directory (*Optional[str]*) –

Return type
str

toil.lib.io.**try_path**(*path*)

Try to use the given path. Return it if it exists or can be made, and we can make things within it, or None otherwise.

Parameters
path (*str*) –

Return type
Optional[str]

class toil.lib.io.**WriteWatchingStream**(*backingStream*)

A stream wrapping class that calls any functions passed to onWrite() with the number of bytes written for every write.

Not seekable.

Parameters
backingStream (*IO[Any]*) –

onWrite(*listener*)

Call the given listener with the number of bytes written on every write.

Parameters
listener (*Callable[[int], None]*) –

Return type
None

write(*data*)

Write the given data to the file.

writelines(*datas*)

Write each string from the given iterable, without newlines.

flush()

Flush the backing stream.

close()

Close the backing stream.

toil.lib.iterables

Module Contents

Classes

concat	A literal iterable to combine sequence literals (lists, set) with generators or list comprehensions.

Functions

flatten(iterables)	Flatten an iterable, except for string elements.

Attributes

IT

toil.lib.iterables.**IT**

toil.lib.iterables.**flatten**(*iterables*)

Flatten an iterable, except for string elements.

Parameters
iterables (*Iterable[IT]*) –

Return type
Iterator[IT]

class toil.lib.iterables.**concat**(**args*)

A literal iterable to combine sequence literals (lists, set) with generators or list comprehensions.

Instead of

```
>>> [ -1 ] + [ x * 2 for x in range( 3 ) ] + [ -1 ]
[-1, 0, 2, 4, -1]
```

you can write

```
>>> list( concat( -1, ( x * 2 for x in range( 3 ) ), -1 ) )
[-1, 0, 2, 4, -1]
```

This is slightly shorter (not counting the list constructor) and does not involve array construction or concatenation.

Note that concat() flattens (or chains) all iterable arguments into a single result iterable:

```
>>> list( concat( 1, range( 2, 4 ), 4 ) )
[1, 2, 3, 4]
```

It only does so one level deep. If you need to recursively flatten a data structure, check out crush().

If you want to prevent that flattening for an iterable argument, wrap it in concat():

```
>>> list( concat( 1, concat( range( 2, 4 ) ), 4 ) )
[1, range(2, 4), 4]
```

Some more example.

```
>>> list( concat() ) # empty concat
[]
>>> list( concat( 1 ) ) # non-iterable
[1]
>>> list( concat( concat() ) ) # empty iterable
[]
>>> list( concat( concat( 1 ) ) ) # singleton iterable
[1]
>>> list( concat( 1, concat( 2 ), 3 ) ) # flattened iterable
[1, 2, 3]
>>> list( concat( 1, [2], 3 ) ) # flattened iterable
```

(continues on next page)

(continued from previous page)

```
[1, 2, 3]
>>> list( concat( 1, concat( [2] ), 3 ) ) # protecting an iterable from being␣
↪flattened
[1, [2], 3]
>>> list( concat( 1, concat( [2], 3 ), 4 ) ) # protection only works with a single␣
↪argument
[1, 2, 3, 4]
>>> list( concat( 1, 2, concat( 3, 4 ), 5, 6 ) )
[1, 2, 3, 4, 5, 6]
>>> list( concat( 1, 2, concat( [ 3, 4 ] ), 5, 6 ) )
[1, 2, [3, 4], 5, 6]
```

Note that while strings are technically iterable, concat() does not flatten them.

```
>>> list( concat( 'ab' ) )
['ab']
>>> list( concat( concat( 'ab' ) ) )
['ab']
```

Parameters
args (*Any*) –

__iter__()

Return type
Iterator[Any]

toil.lib.memoize

Module Contents

Functions

sync_memoize(f)	Like memoize, but guarantees that decorated function is only called once, even when multiple
parse_iso_utc(s)	Parses an ISO time with a hard-coded Z for zulu-time (UTC) at the end. Other timezones are
strict_bool(s)	Variant of bool() that only accepts two possible string values.

Attributes

memoize	Memoize a function result based on its parameters using this decorator.
MAT	
MRT	

toil.lib.memoize.**memoize**

Memoize a function result based on its parameters using this decorator.

For example, this can be used in place of lazy initialization. If the decorating function is invoked by multiple threads, the decorated function may be called more than once with the same arguments.

toil.lib.memoize.**MAT**

toil.lib.memoize.**MRT**

toil.lib.memoize.**sync_memoize**(*f*)

Like memoize, but guarantees that decorated function is only called once, even when multiple threads are calling the decorating function with multiple parameters.

Parameters
f (*Callable[[MAT], MRT]*) –

Return type
Callable[[MAT], MRT]

toil.lib.memoize.**parse_iso_utc**(*s*)

Parses an ISO time with a hard-coded Z for zulu-time (UTC) at the end. Other timezones are not supported. Returns a timezone-naive datetime object.

Parameters
s (*str*) – The ISO-formatted time

Returns
A timezone-naive datetime object

Return type
datetime.datetime

```
>>> parse_iso_utc('2016-04-27T00:28:04.000Z')
datetime.datetime(2016, 4, 27, 0, 28, 4)
>>> parse_iso_utc('2016-04-27T00:28:04Z')
datetime.datetime(2016, 4, 27, 0, 28, 4)
>>> parse_iso_utc('2016-04-27T00:28:04X')
Traceback (most recent call last):
...
ValueError: Not a valid ISO datetime in UTC: 2016-04-27T00:28:04X
```

toil.lib.memoize.**strict_bool**(*s*)

Variant of bool() that only accepts two possible string values.

Parameters
s (*str*) –

Return type
bool

toil.lib.misc

Module Contents

Functions

get_public_ip()	Get the IP that this machine uses to contact the internet.
get_user_name()	Get the current user name, or a suitable substitute string if the user name
utc_now()	Return a datetime in the UTC timezone corresponding to right now.
unix_now_ms()	Return the current time in milliseconds since the Unix epoch.
slow_down(seconds)	Toil jobs that have completed are not allowed to have taken 0 seconds, but
printq(msg, quiet)	
truncExpBackoff()	
call_command(cmd, *args[, input, timeout, useCLocale, ...])	Simplified calling of external commands.

Attributes

logger

toil.lib.misc.**logger**

toil.lib.misc.**get_public_ip**()

Get the IP that this machine uses to contact the internet.

If behind a NAT, this will still be this computer's IP, and not the router's.

Return type
str

toil.lib.misc.**get_user_name**()

Get the current user name, or a suitable substitute string if the user name is not available.

Return type
str

toil.lib.misc.**utc_now**()

Return a datetime in the UTC timezone corresponding to right now.

Return type
datetime.datetime

toil.lib.misc.**unix_now_ms**()

Return the current time in milliseconds since the Unix epoch.

Return type
float

toil.lib.misc.**slow_down**(*seconds*)

Toil jobs that have completed are not allowed to have taken 0 seconds, but Kubernetes timestamps round things to the nearest second. It is possible in some batch systems for a pod to have identical start and end timestamps.

This function takes a possibly 0 job length in seconds and enforces a minimum length to satisfy Toil.

Parameters
seconds (*float*) – Timestamp difference

Returns
seconds, or a small positive number if seconds is 0

Return type
float

toil.lib.misc.**printq**(*msg*, *quiet*)

Parameters

- **msg** (*str*) –
- **quiet** (*bool*) –

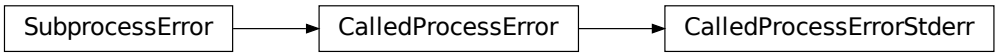
Return type
None

toil.lib.misc.**truncExpBackoff**()

Return type
Iterator[float]

exception toil.lib.misc.**CalledProcessErrorStderr**(*returncode*, *cmd*, *output=None*, *stderr=None*)

Bases: subprocess.CalledProcessError



Version of CalledProcessError that include stderr in the error message if it is set

__str__()

Return str(self).

Return type
str

toil.lib.misc.**call_command**(*cmd*, **args*, *input=None*, *timeout=None*, *useCLocale=True*, *env=None*, *quiet=False*)

Simplified calling of external commands.

If the process fails, CalledProcessErrorStderr is raised.

The captured stderr is always printed, regardless of if an exception occurs, so it can be logged.

Always logs the command at debug log level.

Parameters

- **quiet** (*Optional[bool]*) – If True, do not log the command output. If False (the default), do log the command output at debug log level.
- **useCLocale** (*bool*) – If True, C locale is forced, to prevent failures that can occur in some batch systems when using UTF-8 locale.
- **cmd** (*List[str]*) –
- **args** (*str*) –
- **input** (*Optional[str]*) –
- **timeout** (*Optional[float]*) –
- **env** (*Optional[Dict[str, str]]*) –

Returns

Command standard output, decoded as utf-8.

Return type

str

toil.lib.objects

Module Contents

Classes

InnerClass	Note that this is EXPERIMENTAL code.

class toil.lib.objects.**InnerClass**(*inner_class*)

Note that this is EXPERIMENTAL code.

A nested class (the inner class) decorated with this will have an additional attribute called 'outer' referencing the instance of the nesting class (the outer class) that was used to create the inner class. The outer instance does not need to be passed to the inner class's constructor, it will be set magically. Shamelessly stolen from

http://stackoverflow.com/questions/2278426/inner-classes-how-can-i-get-the-outer-class-object-at-construction-time#answer-2278595.

with names made more descriptive (I hope) and added caching of the BoundInner classes.

Caveat: Within the inner class, self.__class__ will not be the inner class but a dynamically created subclass thereof. It's name will be the same as that of the inner class, but its __module__ will be different. There will be one such dynamic subclass per inner class and instance of outer class, if that outer class instance created any instances of inner the class.

```
>>> class Outer(object):
...     def new_inner(self):
...         # self is an instance of the outer class
...         inner = self.Inner()
...         # the inner instance's 'outer' attribute is set to the outer instance
```

(continues on next page)

(continued from previous page)

```
...         assert inner.outer is self
...         return inner
...     @InnerClass
...     class Inner(object):
...         def get_outer(self):
...             return self.outer
...         @classmethod
...         def new_inner(cls):
...             return cls()
>>> o = Outer()
>>> i = o.new_inner()
>>> i
<toil.lib.objects.Inner...> bound to <toil.lib.objects.Outer object at ...>
```

```
>>> i.get_outer()
<toil.lib.objects.Outer object at ...>
```

Now with inheritance for both inner and outer:

```
>>> class DerivedOuter(Outer):
...     def new_inner(self):
...         return self.DerivedInner()
...     @InnerClass
...     class DerivedInner(Outer.Inner):
...         def get_outer(self):
...             assert super( DerivedOuter.DerivedInner, self ).get_outer() == self.
↪outer
...             return self.outer
>>> derived_outer = DerivedOuter()
>>> derived_inner = derived_outer.new_inner()
>>> derived_inner
<toil.lib.objects...> bound to <toil.lib.objects.DerivedOuter object at ...>
```

```
>>> derived_inner.get_outer()
<toil.lib.objects.DerivedOuter object at ...>
```

Test a static references: >>> Outer.Inner # doctest: +ELLIPSIS <class 'toil.lib.objects...Inner'> >>> DerivedOuter.Inner # doctest: +ELLIPSIS <class 'toil.lib.objects...Inner'> >>> DerivedOuter.DerivedInner #doctest: +ELLIPSIS <class 'toil.lib.objects...DerivedInner'>

Can't decorate top-level classes. Unfortunately, this is detected when the instance is created, not when the class is defined. >>> @InnerClass ... class Foo(object): ... pass >>> Foo() Traceback (most recent call last): ... RuntimeError: Inner classes must be nested in another class.

All inner instances should refer to a single outer instance: >>> o = Outer() >>> o.new_inner().outer == o == o.new_inner().outer True

All inner instances should be of the same class ... >>> o.new_inner().__class__ == o.new_inner().__class__ True

... but that class isn't the inner class ... >>> o.new_inner().__class__ != Outer.Inner True

... but a subclass of the inner class. >>> isinstance(o.new_inner(), Outer.Inner) True

Static and class methods, e.g. should work, too

```
>>> o.Inner.new_inner().outer == o
True
```

__get__(*instance*, *owner*)

__call__(***kwargs*)

toil.lib.resources

Module Contents

Functions

get_total_cpu_time_and_memory_usage()	Gives the total cpu time of itself and all its children, and the maximum RSS memory usage of
get_total_cpu_time()	Gives the total cpu time, including the children.
glob(glob_pattern, directoryname)	Walks through a directory and its subdirectories looking for files matching

toil.lib.resources.**get_total_cpu_time_and_memory_usage**()

Gives the total cpu time of itself and all its children, and the maximum RSS memory usage of itself and its single largest child.

Return type
Tuple[float, int]

toil.lib.resources.**get_total_cpu_time**()

Gives the total cpu time, including the children.

Return type
float

toil.lib.resources.**glob**(*glob_pattern*, *directoryname*)

Walks through a directory and its subdirectories looking for files matching the glob_pattern and returns a list=[].

Parameters

- **directoryname** (*str*) – Any accessible folder name on the filesystem.
- **glob_pattern** (*str*) – A string like "*.txt", which would find all text files.

Returns
A list=[] of absolute filepaths matching the glob pattern.

Return type
List[str]

toil.lib.retry

This file holds the retry() decorator function and RetryCondition object.

retry() can be used to decorate any function based on the list of errors one wishes to retry on.

This list of errors can contain normal Exception objects, and/or RetryCondition objects wrapping Exceptions to include additional conditions.

For example, retrying on a one Exception (HTTPError):

```
from requests import get
from requests.exceptions import HTTPError

@retry(errors=[HTTPError])
def update_my_wallpaper():
    return get('https://www.deviantart.com/')
```

Or:

```
from requests import get
from requests.exceptions import HTTPError

@retry(errors=[HTTPError, ValueError])
def update_my_wallpaper():
    return get('https://www.deviantart.com/')
```

The examples above will retry for the default interval on any errors specified the "errors=" arg list.

To retry on specifically 500/502/503/504 errors, you could specify an ErrorCondition object instead, for example:

```
from requests import get
from requests.exceptions import HTTPError

@retry(errors=[
    ErrorCondition(
                error=HTTPError,
                error_codes=[500, 502, 503, 504]
            )])
def update_my_wallpaper():
    return requests.get('https://www.deviantart.com/')
```

To retry on specifically errors containing the phrase "NotFound":

```
from requests import get
from requests.exceptions import HTTPError

@retry(errors=[
    ErrorCondition(
        error=HTTPError,
        error_message_must_include="NotFound"
    )])
def update_my_wallpaper():
    return requests.get('https://www.deviantart.com/')
```

To retry on all HTTPError errors EXCEPT an HTTPError containing the phrase "NotFound":

```
from requests import get
from requests.exceptions import HTTPError

@retry(errors=[
    HTTPError,
    ErrorCondition(
                error=HTTPError,
                error_message_must_include="NotFound",
                retry_on_this_condition=False
            )])
def update_my_wallpaper():
    return requests.get('https://www.deviantart.com/')
```

To retry on boto3's specific status errors, an example of the implementation is:

```
import boto3
from botocore.exceptions import ClientError

@retry(errors=[
    ErrorCondition(
                error=ClientError,
                boto_error_codes=["BucketNotFound"]
            )])
def boto_bucket(bucket_name):
    boto_session = boto3.session.Session()
    s3_resource = boto_session.resource('s3')
    return s3_resource.Bucket(bucket_name)
```

Any combination of these will also work, provided the codes are matched to the correct exceptions. A ValueError will not return a 404, for example.

The retry function as a decorator should make retrying functions easier and clearer It also encourages smaller independent functions, as opposed to lumping many different things that may need to be retried on different conditions in the same function.

The ErrorCondition object tries to take some of the heavy lifting of writing specific retry conditions and boil it down to an API that covers all common use-cases without the user having to write any new bespoke functions.

Use-cases covered currently:

1. Retrying on a normal error, like a KeyError.
2. Retrying on HTTP error codes (use ErrorCondition).
3. Retrying on boto 3's specific status errors, like "BucketNotFound" (use ErrorCondition).
4. Retrying when an error message contains a certain phrase (use ErrorCondition).
5. Explicitly NOT retrying on a condition (use ErrorCondition).

If new functionality is needed, it's currently best practice in Toil to add functionality to the ErrorCondition itself rather than making a new custom retry method.

Module Contents

Classes

ErrorCondition	A wrapper describing an error condition.

Functions

retry([intervals, infinite_retries, errors, ...])	Retry a function if it fails with any Exception defined in "errors".
return_status_code(e)	
get_error_code(e)	Get the error code name from a Boto 2 or 3 error, or compatible types.
get_error_message(e)	Get the error message string from a Boto 2 or 3 error, or compatible types.
get_error_status(e)	Get the HTTP status code from a compatible source.
get_error_body(e)	Get the body from a Boto 2 or 3 error, or compatible types.
meets_error_message_condition(e, error_message)	
meets_error_code_condition(e, error_codes)	These are expected to be normal HTTP error codes, like 404 or 500.
meets_boto_error_code_condition(e, boto_error_codes)	These are expected to be AWS's custom error aliases, like 'BucketNotFound' or 'AccessDenied'.
error_meets_conditions(e, error_conditions)	
old_retry([delays, timeout, predicate])	Deprecated.

Attributes

SUPPORTED_HTTP_ERRORS
kubernetes
botocore
logger
DEFAULT_DELAYS
DEFAULT_TIMEOUT
retry_flaky_test

toil.lib.retry.**SUPPORTED_HTTP_ERRORS**

toil.lib.retry.**kubernetes**

toil.lib.retry.**botocore**

toil.lib.retry.**logger**

class toil.lib.retry.**ErrorCondition**(*error=None*, *error_codes=None*, *boto_error_codes=None*, *error_message_must_include=None*, *retry_on_this_condition=True*)

A wrapper describing an error condition.

ErrorCondition events may be used to define errors in more detail to determine whether to retry.

Parameters

- **error** (*Optional[Any]*) –
- **error_codes** (*List[int]*) –
- **boto_error_codes** (*List[str]*) –
- **error_message_must_include** (*str*) –
- **retry_on_this_condition** (*bool*) –

toil.lib.retry.**retry**(*intervals=None*, *infinite_retries=False*, *errors=None*, *log_message=None*, *prepare=None*)

Retry a function if it fails with any Exception defined in "errors".

Does so every x seconds, where x is defined by a list of numbers (ints or floats) in "intervals". Also accepts ErrorCondition events for more detailed retry attempts.

Parameters

- **intervals** (*Optional[List]*) – A list of times in seconds we keep retrying until returning failure. Defaults to retrying with the following exponential back-off before failing: 1s, 1s, 2s, 4s, 8s, 16s
- **infinite_retries** (*bool*) – If this is True, reset the intervals when they run out. Defaults to: False.
- **errors** (*Optional[Sequence[Union[ErrorCondition, Type[Exception]]]]*) – A list of exceptions OR ErrorCondition objects to catch and retry on. ErrorCondition objects describe more detailed error event conditions than a plain error. An ErrorCondition specifies: - Exception (required) - Error codes that must match to be retried (optional; defaults to not checking) - A string that must be in the error message to be retried (optional; defaults to not checking) - A bool that can be set to False to always error on this condition.

  If not specified, this will default to a generic Exception.
- **log_message** (*Optional[Tuple[Callable, str]]*) – Optional tuple of ("log/print function()", "message string") that will precede each attempt.
- **prepare** (*Optional[List[Callable]]*) – Optional list of functions to call, with the function's arguments, between retries, to reset state.

Returns
The result of the wrapped function or raise.

Return type
Callable[[Any], Any]

toil.lib.retry.**return_status_code**(*e*)

toil.lib.retry.**get_error_code**(*e*)

Get the error code name from a Boto 2 or 3 error, or compatible types.

Returns empty string for other errors.

Parameters
e (*Exception*) –

Return type
str

toil.lib.retry.**get_error_message**(*e*)

Get the error message string from a Boto 2 or 3 error, or compatible types.

Note that error message conditions also check more than this; this function does not fall back to the traceback for incompatible types.

Parameters
e (*Exception*) –

Return type
str

toil.lib.retry.**get_error_status**(*e*)

Get the HTTP status code from a compatible source.

Such as a Boto 2 or 3 error, kubernetes.client.rest.ApiException, http.client.HTTPException, urllib3.exceptions.HTTPError, requests.exceptions.HTTPError, urllib.error.HTTPError, or compatible type

Returns 0 from other errors.

Parameters
e (*Exception*) –

Return type
int

toil.lib.retry.**get_error_body**(*e*)

Get the body from a Boto 2 or 3 error, or compatible types.

Returns the code and message if the error does not have a body.

Parameters
e (*Exception*) –

Return type
str

toil.lib.retry.**meets_error_message_condition**(*e*, *error_message*)

Parameters

- **e** (*Exception*) –
- **error_message** (*Optional[str]*) –

toil.lib.retry.**meets_error_code_condition**(*e*, *error_codes*)

These are expected to be normal HTTP error codes, like 404 or 500.

Parameters

- **e** (*Exception*) –
- **error_codes** (*Optional[List[int]]*) –

toil.lib.retry.**meets_boto_error_code_condition**(*e*, *boto_error_codes*)

These are expected to be AWS's custom error aliases, like 'BucketNotFound' or 'AccessDenied'.

Parameters

- **e** (*Exception*) –
- **boto_error_codes** (*Optional[List[str]]*) –

toil.lib.retry.**error_meets_conditions**(*e*, *error_conditions*)

toil.lib.retry.**DEFAULT_DELAYS = (0, 1, 1, 4, 16, 64)**

toil.lib.retry.**DEFAULT_TIMEOUT = 300**

toil.lib.retry.**old_retry**(*delays=DEFAULT_DELAYS*, *timeout=DEFAULT_TIMEOUT*, *predicate=lambda e:* ...)

Deprecated.

Retry an operation while the failure matches a given predicate and until a given timeout expires, waiting a given amount of time in between attempts. This function is a generator that yields contextmanagers. See doctests below for example usage.

Parameters

- **delays** (*Iterable[float]*) – an interable yielding the time in seconds to wait before each retried attempt, the last element of the iterable will be repeated.
- **timeout** (*float*) – a overall timeout that should not be exceeded for all attempts together. This is a best-effort mechanism only and it won't abort an ongoing attempt, even if the timeout expires during that attempt.
- **predicate** (*Callable[[Exception],bool]*) – a unary callable returning True if another attempt should be made to recover from the given exception. The default value for this parameter will prevent any retries!

Returns

a generator yielding context managers, one per attempt

Return type

Iterator

Retry for a limited amount of time:

```
>>> true = lambda _:True
>>> false = lambda _:False
>>> i = 0
>>> for attempt in old_retry( delays=[0], timeout=.1, predicate=true ):
...     with attempt:
...         i += 1
...         raise RuntimeError('foo')
Traceback (most recent call last):
...
RuntimeError: foo
>>> i > 1
True
```

If timeout is 0, do exactly one attempt:

```
>>> i = 0
>>> for attempt in old_retry( timeout=0 ):
...     with attempt:
...         i += 1
...         raise RuntimeError( 'foo' )
Traceback (most recent call last):
...
RuntimeError: foo
>>> i
1
```

Don't retry on success:

```
>>> i = 0
>>> for attempt in old_retry( delays=[0], timeout=.1, predicate=true ):
...     with attempt:
...         i += 1
>>> i
1
```

Don't retry on unless predicate returns True:

```
>>> i = 0
>>> for attempt in old_retry( delays=[0], timeout=.1, predicate=false):
...     with attempt:
...         i += 1
...         raise RuntimeError( 'foo' )
Traceback (most recent call last):
...
RuntimeError: foo
>>> i
1
```

toil.lib.retry.**retry_flaky_test**

toil.lib.threading

Module Contents

Classes

ExceptionalThread	A thread whose join() method re-raises exceptions raised during run(). While join() is
LastProcessStandingArena	Class that lets a bunch of processes detect and elect a last process

Functions

cpu_count()	Get the rounded-up integer number of whole CPUs available.
collect_process_name_garbage()	Delete all the process names that point to files that don't exist anymore
destroy_all_process_names()	Delete all our process name files because our process is going away.
get_process_name(base_dir)	Return the name of the current process. Like a PID but visible between
process_name_exists(base_dir, name)	Return true if the process named by the given name (from process_name) exists, and false otherwise.
global_mutex(base_dir, mutex)	Context manager that locks a mutex. The mutex is identified by the given

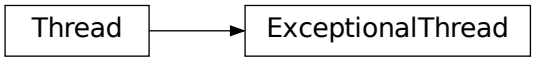
Attributes

logger
current_process_name_lock
current_process_name_for

toil.lib.threading.**logger**

class toil.lib.threading.**ExceptionalThread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs=None*, ***, *daemon=None*)

Bases: threading.Thread



A thread whose join() method re-raises exceptions raised during run(). While join() is idempotent, the exception is only during the first invocation of join() that successfully joined the thread. If join() times out, no exception will be re reraised even though an exception might already have occured in run().

When subclassing this thread, override tryRun() instead of run().

```
>>> def f():
...     assert 0
>>> t = ExceptionalThread(target=f)
>>> t.start()
>>> t.join()
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
AssertionError
```

```
>>> class MyThread(ExceptionalThread):
...     def tryRun( self ):
...         assert 0
>>> t = MyThread()
>>> t.start()
>>> t.join()
Traceback (most recent call last):
...
AssertionError
```

exc_info

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

Return type
None

tryRun()

Return type
None

join(**args*, ***kwargs*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

Parameters

- **args** (*Optional[float]*) –
- **kwargs** (*Optional[float]*) –

Return type
None

toil.lib.threading.**cpu_count**()

Get the rounded-up integer number of whole CPUs available.

Counts hyperthreads as CPUs.

Uses the system's actual CPU count, or the current v1 cgroup's quota per period, if the quota is set.

Ignores the cgroup's cpu shares value, because it's extremely difficult to interpret. See https://github.com/kubernetes/kubernetes/issues/81021.

Caches result for efficiency.

Returns
Integer count of available CPUs, minimum 1.

Return type
int

toil.lib.threading.**current_process_name_lock**

toil.lib.threading.**current_process_name_for: Dict[str, str]**

toil.lib.threading.**collect_process_name_garbage**()

Delete all the process names that point to files that don't exist anymore (because the work directory was temporary and got cleaned up). This is known to happen during the tests, which get their own temp directories.

Caller must hold current_process_name_lock.

Return type
None

toil.lib.threading.**destroy_all_process_names**()

Delete all our process name files because our process is going away.

We let all our FDs get closed by the process death.

We assume there is nobody else using the system during exit to race with.

Return type
None

toil.lib.threading.**get_process_name**(*base_dir*)

Return the name of the current process. Like a PID but visible between containers on what to Toil appears to be a node.

Parameters
base_dir (*str*) – Base directory to work in. Defines the shared namespace.

Returns
Process's assigned name

Return type
str

toil.lib.threading.**process_name_exists**(*base_dir*, *name*)

Return true if the process named by the given name (from process_name) exists, and false otherwise.

Can see across container boundaries using the given node workflow directory.

Parameters

- **base_dir** (*str*) – Base directory to work in. Defines the shared namespace.
- **name** (*str*) – Process's name to poll

Returns
True if the named process is still alive, and False otherwise.

Return type
bool

toil.lib.threading.**global_mutex**(*base_dir*, *mutex*)

Context manager that locks a mutex. The mutex is identified by the given name, and scoped to the given directory. Works across all containers that have access to the given diectory. Mutexes held by dead processes are automatically released.

Only works between processes, NOT between threads.

Parameters

- **base_dir** (*str*) – Base directory to work in. Defines the shared namespace.
- **mutex** (*str*) – Mutex to lock. Must be a permissible path component.

Return type
Iterator[None]

class toil.lib.threading.**LastProcessStandingArena**(*base_dir*, *name*)

Class that lets a bunch of processes detect and elect a last process standing.

Process enter and leave (sometimes due to sudden existence failure). We guarantee that the last process to leave, if it leaves properly, will get a chance to do some cleanup. If new processes try to enter during the cleanup, they will be delayed until after the cleanup has happened and the previous "last" process has finished leaving.

The user is responsible for making sure you always leave if you enter! Consider using a try/finally; this class is not a context manager.

Parameters

- **base_dir** (*str*) –
- **name** (*str*) –

enter()

This process is entering the arena. If cleanup is in progress, blocks until it is finished.

You may not enter the arena again before leaving it.

Return type
None

leave()

This process is leaving the arena. If this process happens to be the last process standing, yields something, with other processes blocked from joining the arena until the loop body completes and the process has finished leaving. Otherwise, does not yield anything.

Should be used in a loop:

for _ in arena.leave():
If we get here, we were the last process. Do the cleanup pass

Return type
Iterator[bool]

toil.lib.throttle

Module Contents

Classes

LocalThrottle	A thread-safe rate limiter that throttles each thread independently. Can be used as a
throttle	A context manager for ensuring that the execution of its body takes at least a given amount

class toil.lib.throttle.**LocalThrottle**(*min_interval*)

A thread-safe rate limiter that throttles each thread independently. Can be used as a function or method decorator or as a simple object, via its .throttle() method.

The use as a decorator is deprecated in favor of throttle().

Parameters
min_interval (*int*) –

throttle(*wait=True*)

If the wait parameter is True, this method returns True after suspending the current thread as necessary to ensure that no less than the configured minimum interval has passed since the last invocation of this method in the current thread returned True.

If the wait parameter is False, this method immediatly returns True (if at least the configured minimum interval has passed since the last time this method returned True in the current thread) or False otherwise.

Parameters
wait (*bool*) –

Return type
bool

__call__(*function*)

class toil.lib.throttle.**throttle**(*min_interval*)

A context manager for ensuring that the execution of its body takes at least a given amount of time, sleeping if necessary. It is a simpler version of LocalThrottle if used as a decorator.

Ensures that body takes at least the given amount of time.

```
>>> start = time.time()
>>> with throttle(1):
...     pass
>>> 1 <= time.time() - start <= 1.1
True
```

Ditto when used as a decorator.

```
>>> @throttle(1)
... def f():
...     pass
>>> start = time.time()
>>> f()
```

(continues on next page)

(continued from previous page)

```
>>> 1 <= time.time() - start <= 1.1
True
```

If the body takes longer by itself, don't throttle.

```
>>> start = time.time()
>>> with throttle(1):
...     time.sleep(2)
>>> 2 <= time.time() - start <= 2.1
True
```

Ditto when used as a decorator.

```
>>> @throttle(1)
... def f():
...     time.sleep(2)
>>> start = time.time()
>>> f()
>>> 2 <= time.time() - start <= 2.1
True
```

If an exception occurs, don't throttle.

```
>>> start = time.time()
>>> try:
...     with throttle(1):
...         raise ValueError('foo')
... except ValueError:
...     end = time.time()
...     raise
Traceback (most recent call last):
...
ValueError: foo
>>> 0 <= end - start <= 0.1
True
```

Ditto when used as a decorator.

```
>>> @throttle(1)
... def f():
...     raise ValueError('foo')
>>> start = time.time()
>>> try:
...     f()
... except ValueError:
...     end = time.time()
...     raise
Traceback (most recent call last):
...
ValueError: foo
>>> 0 <= end - start <= 0.1
True
```

Parameters
min_interval (*Union[int, float]*) –

__enter__()

__exit__(*exc_type*, *exc_val*, *exc_tb*)

__call__(*function*)

toil.provisioners

Subpackages

toil.provisioners.aws

Submodules

toil.provisioners.aws.awsProvisioner

Module Contents

Classes

AWSProvisioner	Interface for provisioning worker nodes to use in a Toil cluster.

Functions

awsRetryPredicate(e)	
expectedShutdownErrors(e)	Matches errors that we expect to occur during shutdown, and which indicate
awsRetry(f)	This decorator retries the wrapped function if aws throws unexpected errors
awsFilterImpairedNodes(nodes, ec2)	

Attributes

logger	

toil.provisioners.aws.awsProvisioner.**logger**

toil.provisioners.aws.awsProvisioner.**awsRetryPredicate**(*e*)

toil.provisioners.aws.awsProvisioner.**expectedShutdownErrors**(*e*)

Matches errors that we expect to occur during shutdown, and which indicate that we need to wait or try again.

Should *not* match any errors which indicate that an operation is impossible or unnecessary (such as errors resulting from a thing not existing to be deleted).

Parameters
e (*Exception*) –

Return type
bool

toil.provisioners.aws.awsProvisioner.**awsRetry**(*f*)

This decorator retries the wrapped function if aws throws unexpected errors errors. It should wrap any function that makes use of boto

toil.provisioners.aws.awsProvisioner.**awsFilterImpairedNodes**(*nodes*, *ec2*)

exception toil.provisioners.aws.awsProvisioner.**InvalidClusterStateException**

Bases: Exception

Common base class for all non-exit exceptions.

class toil.provisioners.aws.awsProvisioner.**AWSProvisioner**(*clusterName*, *clusterType*, *zone*, *nodeStorage*, *nodeStorageOverrides*, *sseKey*)

Bases: *toil.provisioners.abstractProvisioner.AbstractProvisioner*

Interface for provisioning worker nodes to use in a Toil cluster.

supportedClusterTypes()

Get all the cluster types that this provisioner implementation supports.

createClusterSettings()

Create a new set of cluster settings for a cluster to be deployed into AWS.

readClusterSettings()

Reads the cluster settings from the instance metadata, which assumes the instance is the leader.

launchCluster(*leaderNodeType*, *leaderStorage*, *owner*, *keyName*, *botoPath*, *userTags*, *vpcSubnet*, *awsEc2ProfileArn*, *awsEc2ExtraSecurityGroupIds*, ***kwargs*)

Starts a single leader node and populates this class with the leader's metadata.

Parameters

- **leaderNodeType** (*str*) – An AWS instance type, like "t2.medium", for example.
- **leaderStorage** (*int*) – An integer number of gigabytes to provide the leader instance with.
- **owner** (*str*) – Resources will be tagged with this owner string.
- **keyName** (*str*) – The ssh key to use to access the leader node.
- **botoPath** (*str*) – The path to the boto credentials directory.
- **userTags** (*Optional[dict]*) – Optionally provided user tags to put on the cluster.
- **vpcSubnet** (*Optional[str]*) – Optionally specify the VPC subnet for the leader.
- **awsEc2ProfileArn** (*Optional[str]*) – Optionally provide the profile ARN.
- **awsEc2ExtraSecurityGroupIds** (*Optional[list]*) – Optionally provide additional security group IDs.

Returns
None

toil_service_env_options()

Set AWS tags in user docker container

Return type
str

getKubernetesAutoscalerSetupCommands(*values*)

Get the Bash commands necessary to configure the Kubernetes Cluster Autoscaler for AWS.

Parameters
values (*Dict[str, str]*) –

Return type
str

getKubernetesCloudProvider()

Use the "aws" Kubernetes cloud provider when setting up Kubernetes.

Return type
Optional[str]

getNodeShape(*instance_type*, *preemptible=False*)

Get the Shape for the given instance type (e.g. 't2.medium').

Parameters
instance_type (*str*) –

Return type
toil.provisioners.abstractProvisioner.Shape

static retryPredicate(*e*)

Return true if the exception e should be retried by the cluster scaler. For example, should return true if the exception was due to exceeding an API rate limit. The error will be retried with exponential backoff.

Parameters
e – exception raised during execution of setNodeCount

Returns
boolean indicating whether the exception e should be retried

`destroyCluster()`

Terminate instances and delete the profile and security group.

Return type
None

`terminateNodes(nodes)`

Terminate the nodes represented by given Node objects

Parameters
nodes (*List[toil.provisioners.node.Node]*) – list of Node objects

Return type
None

`addNodes(nodeTypes, numNodes, preemptible, spotBid=None)`

Used to add worker nodes to the cluster

Parameters

- **numNodes** – The number of nodes to add
- **preemptible** – whether or not the nodes will be preemptible
- **spotBid** – The bid for preemptible nodes if applicable (this can be set in config, also).
- **nodeTypes** (*Set[str]*) –

Returns
number of nodes successfully added

Return type
int

`addManagedNodes(nodeTypes, minNodes, maxNodes, preemptible, spotBid=None)`

Add a group of managed nodes of the given type, up to the given maximum. The nodes will automatically be launched and terminated depending on cluster load.

Raises ManagedNodesNotSupportedException if the provisioner implementation or cluster configuration can't have managed nodes.

Parameters

- **minNodes** – The minimum number of nodes to scale to
- **maxNodes** – The maximum number of nodes to scale to
- **preemptible** – whether or not the nodes will be preemptible
- **spotBid** – The bid for preemptible nodes if applicable (this can be set in config, also).
- **nodeTypes** (*Set[str]*) –

Return type
None

getProvisionedWorkers(*instance_type=None*, *preemptible=None*)

Gets all nodes, optionally of the given instance type or preemptability, from the provisioner. Includes both static and autoscaled nodes.

Parameters

- **preemptible** (*Optional[bool]*) – Boolean value to restrict to preemptible nodes or non-preemptible nodes
- **instance_type** (*Optional[str]*) –

Returns
list of Node objects

Return type
List[*toil.provisioners.node.Node*]

getLeader(*wait=False*)

Get the leader for the cluster as a Toil Node object.

Return type
toil.provisioners.node.Node

full_policy(*resource*)

Produce a dict describing the JSON form of a full-access-granting AWS IAM policy for the service with the given name (e.g. 's3').

Parameters
resource (*str*) –

Return type
dict

kubernetes_policy()

Get the Kubernetes policy grants not provided by the full grants on EC2 and IAM. See <https://github.com/DataBiosphere/toil/wiki/Manual-Autoscaling-Kubernetes-Setup#leader-policy> and <https://github.com/DataBiosphere/toil/wiki/Manual-Autoscaling-Kubernetes-Setup#worker-policy>.

These are mostly needed to support Kubernetes' AWS CloudProvider, and some are for the Kubernetes Cluster Autoscaler's AWS integration.

Some of these are really only needed on the leader.

Return type
dict

Package Contents

Functions

get_aws_zone_from_boto()	Get the AWS zone from the Boto config file, if it is configured and the
get_aws_zone_from_environment()	Get the AWS zone from TOIL_AWS_ZONE if set.
get_aws_zone_from_environment_region()	Pick an AWS zone in the region defined by TOIL_AWS_REGION, if it is set.
get_aws_zone_from_metadata()	Get the AWS zone from instance metadata, if on EC2 and the boto module is
running_on_ec2()	Return True if we are currently running on EC2, and false otherwise.
zone_to_region(zone)	Get a region (e.g. us-west-2) from a zone (e.g. us-west-1c).
get_aws_zone_from_spot_market(spotBid, nodeType, ...)	If a spot bid, node type, and Boto2 EC2 connection are specified, picks a
get_best_aws_zone([spotBid, nodeType, boto2_ec2, ...])	Get the right AWS zone to use.
choose_spot_zone(zones, bid, spot_history)	Returns the zone to put the spot request based on, in order of priority:
optimize_spot_bid(boto2_ec2, instance_type, spot_bid, ...)	Check whether the bid is in line with history and makes an effort to place

Attributes

logger
ZoneTuple

toil.provisioners.aws.**get_aws_zone_from_boto**()

Get the AWS zone from the Boto config file, if it is configured and the boto module is available.

Return type
Optional[str]

toil.provisioners.aws.**get_aws_zone_from_environment**()

Get the AWS zone from TOIL_AWS_ZONE if set.

Return type
Optional[str]

toil.provisioners.aws.**get_aws_zone_from_environment_region**()

Pick an AWS zone in the region defined by TOIL_AWS_REGION, if it is set.

Return type
Optional[str]

toil.provisioners.aws.**get_aws_zone_from_metadata**()

Get the AWS zone from instance metadata, if on EC2 and the boto module is available. Otherwise, gets the AWS zone from ECS task metadata, if on ECS.

Return type
Optional[str]

toil.provisioners.aws.**running_on_ec2**()

Return True if we are currently running on EC2, and false otherwise.

Return type
bool

toil.provisioners.aws.**zone_to_region**(*zone*)

Get a region (e.g. us-west-2) from a zone (e.g. us-west-1c).

Parameters
zone (*str*) –

Return type
str

toil.provisioners.aws.**logger**

toil.provisioners.aws.**ZoneTuple**

toil.provisioners.aws.**get_aws_zone_from_spot_market**(*spotBid*, *nodeType*, *boto2_ec2*, *zone_options*)

If a spot bid, node type, and Boto2 EC2 connection are specified, picks a zone where instances are easy to buy from the zones in the region of the Boto2 connection. These parameters must always be specified together, or not at all.

In this case, zone_options can be used to restrict to a subset of the zones in the region.

Parameters

- **spotBid** (*Optional[float]*) –
- **nodeType** (*Optional[str]*) –
- **boto2_ec2** (*Optional[boto.connection.AWSAuthConnection]*) –
- **zone_options** (*Optional[List[str]]*) –

Return type
Optional[str]

toil.provisioners.aws.**get_best_aws_zone**(*spotBid=None*, *nodeType=None*, *boto2_ec2=None*, *zone_options=None*)

Get the right AWS zone to use.

Reports the TOIL_AWS_ZONE environment variable if set.

Otherwise, if we are running on EC2 or ECS, reports the zone we are running in.

Otherwise, if a spot bid, node type, and Boto2 EC2 connection are specified, picks a zone where instances are easy to buy from the zones in the region of the Boto2 connection. These parameters must always be specified together, or not at all.

In this case, zone_options can be used to restrict to a subset of the zones in the region.

Otherwise, if we have the TOIL_AWS_REGION variable set, chooses a zone in that region.

Finally, if a default region is configured in Boto 2, chooses a zone in that region.

Returns None if no method can produce a zone to use.

Parameters

- **spotBid** (*Optional[float]*) –
- **nodeType** (*Optional[str]*) –

- **boto2_ec2** (*Optional[boto.connection.AWSAuthConnection]*) –
- **zone_options** (*Optional[List[str]]*) –

Return type
Optional[str]

toil.provisioners.aws.**choose_spot_zone**(*zones*, *bid*, *spot_history*)

Returns the zone to put the spot request based on, in order of priority:

1) zones with prices currently under the bid
2) zones with the most stable price

Returns
the name of the selected zone

Parameters

- **zones** (*List[str]*) –
- **bid** (*float*) –
- **spot_history** (*List[boto.ec2.spotpricehistory.SpotPriceHistory]*) –

Return type
str

```
>>> from collections import namedtuple
>>> FauxHistory = namedtuple('FauxHistory', ['price', 'availability_zone'])
>>> zones = ['us-west-2a', 'us-west-2b']
>>> spot_history = [FauxHistory(0.1, 'us-west-2a'),
↪FauxHistory(0.2, 'us-west-2a'),                    FauxHistory(0.3, 'us-west-
↪2b'),                    FauxHistory(0.6, 'us-west-2b')]
>>> choose_spot_zone(zones, 0.15, spot_history)
'us-west-2a'
```

```
>>> spot_history=[FauxHistory(0.3, 'us-west-2a'),
↪FauxHistory(0.2, 'us-west-2a'),                    FauxHistory(0.1, 'us-west-2b
↪'),                    FauxHistory(0.6, 'us-west-2b')]
>>> choose_spot_zone(zones, 0.15, spot_history)
'us-west-2b'
```

```
>>> spot_history=[FauxHistory(0.1, 'us-west-2a'),
↪FauxHistory(0.7, 'us-west-2a'),                    FauxHistory(0.1, 'us-west-2b
↪'),                    FauxHistory(0.6, 'us-west-2b')]
>>> choose_spot_zone(zones, 0.15, spot_history)
'us-west-2b'
```

toil.provisioners.aws.**optimize_spot_bid**(*boto2_ec2*, *instance_type*, *spot_bid*, *zone_options*)

Check whether the bid is in line with history and makes an effort to place the instance in a sensible zone.

Parameters
zone_options (*List[str]*) – The collection of allowed zones to consider, within

the region associated with the Boto2 connection.

Submodules

toil.provisioners.abstractProvisioner

Module Contents

Classes

Shape	Represents a job or a node's "shape", in terms of the dimensions of memory, cores, disk and
AbstractProvisioner	Interface for provisioning worker nodes to use in a Toil cluster.

Attributes

a_short_time
logger

toil.provisioners.abstractProvisioner.**a_short_time** = 5

toil.provisioners.abstractProvisioner.**logger**

exception toil.provisioners.abstractProvisioner.**ManagedNodesNotSupportedException**

Bases: RuntimeError

Raised when attempting to add managed nodes (which autoscale up and down by themselves, without the provisioner doing the work) to a provisioner that does not support them.

Polling with this and try/except is the Right Way to check if managed nodes are available from a provisioner.

class toil.provisioners.abstractProvisioner.**Shape**(*wallTime*, *memory*, *cores*, *disk*, *preemptible*)

Represents a job or a node's "shape", in terms of the dimensions of memory, cores, disk and wall-time allocation.

The wallTime attribute stores the number of seconds of a node allocation, e.g. 3600 for AWS. FIXME: and for jobs?

The memory and disk attributes store the number of bytes required by a job (or provided by a node) in RAM or on disk (SSD or HDD), respectively.

Parameters

- **wallTime** (*Union[int, float]*) –

- **memory** (*int*) –
- **cores** (*Union[int, float]*) –
- **disk** (*int*) –
- **preemptible** (*bool*) –

__eq__(*other*)

Return self==value.

Parameters
other (*Any*) –

Return type
bool

greater_than(*other*)

Parameters
other (*Any*) –

Return type
bool

__gt__(*other*)

Return self>value.

Parameters
other (*Any*) –

Return type
bool

__repr__()

Return repr(self).

Return type
str

__str__()

Return str(self).

Return type
str

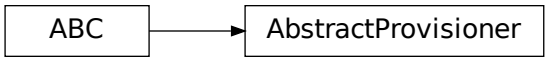
__hash__()

Return hash(self).

Return type
int

class toil.provisioners.abstractProvisioner.**AbstractProvisioner**(*clusterName=None*, *clusterType='mesos'*, *zone=None*, *nodeStorage=50*, *nodeStorageOverrides=None*)

Bases: abc.ABC



Interface for provisioning worker nodes to use in a Toil cluster.

Parameters

- **clusterName** (*Optional[str]*) –
- **clusterType** (*Optional[str]*) –
- **zone** (*Optional[str]*) –
- **nodeStorage** (*int*) –
- **nodeStorageOverrides** (*Optional[List[str]]*) –

`class InstanceConfiguration`

Allows defining the initial setup for an instance and then turning it into an Ignition configuration for instance user data.

`addFile`(*path*, *filesystem='root'*, *mode='0755'*, *contents=''*, *append=False*)

Make a file on the instance with the given filesystem, mode, and contents.

See the storage.files section: https://github.com/kinvolk/ignition/blob/flatcar-master/doc/configuration-v2_2.md

Parameters

- **path** (*str*) –
- **filesystem** (*str*) –
- **mode** (*Union[str, int]*) –
- **contents** (*str*) –
- **append** (*bool*) –

`addUnit`(*name*, *enabled=True*, *contents=''*)

Make a systemd unit on the instance with the given name (including .service), and content. Units will be enabled by default.

Unit logs can be investigated with:

systemctl status whatever.service

or:

journalctl -xe

Parameters

- **name** (*str*) –
- **enabled** (*bool*) –
- **contents** (*str*) –

`addSSHRSAKey`(*keyData*)

Authorize the given bare, encoded RSA key (without "ssh-rsa").

Parameters

keyData (*str*) –

`toIgnitionConfig`()

Return an Ignition configuration describing the desired config.

Return type
str

LEADER_HOME_DIR = '/root/'

cloud: str

abstract supportedClusterTypes()

Get all the cluster types that this provisioner implementation supports.

Return type
Set[str]

abstract createClusterSettings()

Initialize class for a new cluster, to be deployed, when running outside the cloud.

abstract readClusterSettings()

Initialize class from an existing cluster. This method assumes that the instance we are running on is the leader.

Implementations must call _setLeaderWorkerAuthentication().

setAutoscaledNodeTypes(*nodeTypes*)

Set node types, shapes and spot bids for Toil-managed autoscaling. :param nodeTypes: A list of node types, as parsed with parse_node_types.

Parameters
nodeTypes (*List[Tuple[Set[str], Optional[float]]]*) –

hasAutoscaledNodeTypes()

Check if node types have been configured on the provisioner (via setAutoscaledNodeTypes).

Returns
True if node types are configured for autoscaling, and false otherwise.

Return type
bool

getAutoscaledInstanceShapes()

Get all the node shapes and their named instance types that the Toil autoscaler should manage.

Return type
Dict[*Shape*, str]

static retryPredicate(*e*)

Return true if the exception e should be retried by the cluster scaler. For example, should return true if the exception was due to exceeding an API rate limit. The error will be retried with exponential backoff.

Parameters
e – exception raised during execution of setNodeCount

Returns
boolean indicating whether the exception e should be retried

abstract launchCluster(**args*, ***kwargs*)

Initialize a cluster and create a leader node.

Implementations must call _setLeaderWorkerAuthentication() with the leader so that workers can be launched.

Parameters

- **leaderNodeType** – The leader instance.
- **leaderStorage** – The amount of disk to allocate to the leader in gigabytes.
- **owner** – Tag identifying the owner of the instances.

abstract addNodes(*nodeTypes*, *numNodes*, *preemptible*, *spotBid=None*)

Used to add worker nodes to the cluster

Parameters

- **numNodes** (*int*) – The number of nodes to add
- **preemptible** (*bool*) – whether or not the nodes will be preemptible
- **spotBid** (*Optional[float]*) – The bid for preemptible nodes if applicable (this can be set in config, also).
- **nodeTypes** (*Set[str]*) –

Returns
number of nodes successfully added

Return type
int

addManagedNodes(*nodeTypes*, *minNodes*, *maxNodes*, *preemptible*, *spotBid=None*)

Add a group of managed nodes of the given type, up to the given maximum. The nodes will automatically be launched and terminated depending on cluster load.

Raises ManagedNodesNotSupportedException if the provisioner implementation or cluster configuration can't have managed nodes.

Parameters

- **minNodes** – The minimum number of nodes to scale to
- **maxNodes** – The maximum number of nodes to scale to
- **preemptible** – whether or not the nodes will be preemptible
- **spotBid** – The bid for preemptible nodes if applicable (this can be set in config, also).
- **nodeTypes** (*Set[str]*) –

Return type
None

abstract terminateNodes(*nodes*)

Terminate the nodes represented by given Node objects

Parameters
nodes (*List[toil.provisioners.node.Node]*) – list of Node objects

Return type
None

abstract getLeader()

Returns
The leader node.

abstract getProvisionedWorkers(*instance_type=None*, *preemptible=None*)

Gets all nodes, optionally of the given instance type or preemptability, from the provisioner. Includes both static and autoscaled nodes.

Parameters

- **preemptible** (*Optional[bool]*) – Boolean value to restrict to preemptible nodes or non-preemptible nodes
- **instance_type** (*Optional[str]*) –

Returns
list of Node objects

Return type
List[*toil.provisioners.node.Node*]

abstract getNodeShape(*instance_type*, *preemptible=False*)

The shape of a preemptible or non-preemptible node managed by this provisioner. The node shape defines key properties of a machine, such as its number of cores or the time between billing intervals.

Parameters
instance_type (*str*) – Instance type name to return the shape of.

Return type
Shape

abstract destroyCluster()

Terminates all nodes in the specified cluster and cleans up all resources associated with the cluster. :param clusterName: identifier of the cluster to terminate.

Return type
None

getBaseInstanceConfiguration()

Get the base configuration for both leader and worker instances for all cluster types.

Return type
InstanceConfiguration

addVolumesService(*config*)

Add a service to prepare and mount local scratch volumes.

Parameters
config (*InstanceConfiguration*) –

addNodeExporterService(*config*)

Add the node exporter service for Prometheus to an instance configuration.

Parameters
config (*InstanceConfiguration*) –

toil_service_env_options()

Return type
str

add_toil_service(*config*, *role*, *keyPath=None*, *preemptible=False*)

Add the Toil leader or worker service to an instance configuration.

Will run Mesos master or agent as appropriate in Mesos clusters. For Kubernetes clusters, will just sleep to provide a place to shell into on the leader, and shouldn't run on the worker.

Parameters

- **role** (*str*) – Should be 'leader' or 'worker'. Will not work for 'worker' until leader credentials have been collected.

- **keyPath** (*str*) – path on the node to a server-side encryption key that will be added to the node after it starts. The service will wait until the key is present before starting.
- **preemptible** (*bool*) – Whether a worker should identify itself as preemptible or not to the scheduler.
- **config** (InstanceConfiguration) –

getKubernetesValues(*architecture='amd64'*)

Returns a dict of Kubernetes component versions and paths for formatting into Kubernetes-related templates.

Parameters
architecture (*str*) –

addKubernetesServices(*config*, *architecture='amd64'*)

Add installing Kubernetes and Kubeadm and setting up the Kubelet to run when configured to an instance configuration. The same process applies to leaders and workers.

Parameters

- **config** (InstanceConfiguration) –
- **architecture** (*str*) –

abstract getKubernetesAutoscalerSetupCommands(*values*)

Return Bash commands that set up the Kubernetes cluster autoscaler for provisioning from the environment supported by this provisioner.

Should only be implemented if Kubernetes clusters are supported.

Parameters
values (*Dict[str, str]*) – Contains definitions of cluster variables, like AUTOSCALER_VERSION and CLUSTER_NAME.

Returns
Bash snippet

Return type
str

getKubernetesCloudProvider()

Return the Kubernetes cloud provider (for example, 'aws'), to pass to the kubelets in a Kubernetes cluster provisioned using this provisioner.

Defaults to None if not overridden, in which case no cloud provider integration will be used.

Returns
Cloud provider name, or None

Return type
Optional[str]

addKubernetesLeader(*config*)

Add services to configure as a Kubernetes leader, if Kubernetes is already set to be installed.

Parameters
config (InstanceConfiguration) –

addKubernetesWorker(*config*, *authVars*, *preemptible=False*)

Add services to configure as a Kubernetes worker, if Kubernetes is already set to be installed.

Authenticate back to the leader using the JOIN_TOKEN, JOIN_CERT_HASH, and JOIN_ENDPOINT set in the given authentication data dict.

Parameters

- **config** (InstanceConfiguration) – The configuration to add services to
- **authVars** (*Dict[str, str]*) – Dict with authentication info
- **preemptible** (*bool*) – Whether the worker should be labeled as preemptible or not

toil.provisioners.clusterScaler

Module Contents

Classes

BinPackedFit	If jobShapes is a set of tasks with run requirements (mem/disk/cpu), and nodeShapes is a sorted
NodeReservation	The amount of resources that we expect to be available on a given node at each point in time.
ClusterScaler	
ScalerThread	A thread that automatically scales the number of either preemptible or non-preemptible worker
ClusterStats	

Functions

adjustEndingReservationForJob(reservation, jobShape, ...)	Add a job to an ending reservation that ends at wallTime.
split(nodeShape, jobShape, wallTime)	Partition a node allocation into two to fit the job.
binPacking(nodeShapes, jobShapes, goalTime)	Using the given node shape bins, pack the given job shapes into nodes to

Attributes

logger
EVICTION_THRESHOLD
RESERVE_SMALL_LIMIT
RESERVE_SMALL_AMOUNT
RESERVE_BREAKPOINTS
RESERVE_FRACTIONS
OS_SIZE
FailedConstraint

toil.provisioners.clusterScaler.**logger**

toil.provisioners.clusterScaler.**EVICTION_THRESHOLD**

toil.provisioners.clusterScaler.**RESERVE_SMALL_LIMIT**

toil.provisioners.clusterScaler.**RESERVE_SMALL_AMOUNT**

toil.provisioners.clusterScaler.**RESERVE_BREAKPOINTS: List[Union[int, float]]**

toil.provisioners.clusterScaler.**RESERVE_FRACTIONS = [0.25, 0.2, 0.1, 0.06, 0.02]**

toil.provisioners.clusterScaler.**OS_SIZE**

toil.provisioners.clusterScaler.**FailedConstraint**

class toil.provisioners.clusterScaler.**BinPackedFit**(*nodeShapes*, *targetTime=defaultTargetTime*)

If jobShapes is a set of tasks with run requirements (mem/disk/cpu), and nodeShapes is a sorted list of available computers to run these jobs on, this function attempts to return a dictionary representing the minimum set of computerNode computers needed to run the tasks in jobShapes.

Uses a first fit decreasing (FFD) bin packing like algorithm to calculate an approximate minimum number of nodes that will fit the given list of jobs. BinPackingFit assumes the ordered list, nodeShapes, is ordered for "node preference" outside of BinPackingFit beforehand. So when virtually "creating" nodes, the first node within nodeShapes that fits the job is the one that's added.

Parameters

- **nodeShapes** (*list*) – The properties of an atomic node allocation, in terms of wall-time, memory, cores, disk, and whether it is preemptible or not.
- **targetTime** (*float*) – The time before which all jobs should at least be started.

Returns

The minimum number of minimal node allocations estimated to be required to run all the jobs in jobShapes.

nodeReservations: Dict[*toil.provisioners.abstractProvisioner.Shape*, List[*NodeReservation*]]

binPack(*jobShapes*)

Pack a list of jobShapes into the fewest nodes reasonable.

Can be run multiple times.

Returns any distinct Shapes that did not fit, mapping to reasons they did not fit.

Parameters
jobShapes (*List[*toil.provisioners.abstractProvisioner.Shape*]*) –

Return type
Dict[*toil.provisioners.abstractProvisioner.Shape*, List[FailedConstraint]]

addJobShape(*jobShape*)

Add the job to the first node reservation in which it will fit. (This is the bin-packing aspect).

Returns the job shape again, and a list of failed constraints, if it did not fit.

Parameters
jobShape (toil.provisioners.abstractProvisioner.Shape) –

Return type
Optional[Tuple[*toil.provisioners.abstractProvisioner.Shape*, List[FailedConstraint]]]

getRequiredNodes()

Return a dict from node shape to number of nodes required to run the packed jobs.

Return type
Dict[*toil.provisioners.abstractProvisioner.Shape*, int]

class toil.provisioners.clusterScaler.**NodeReservation**(*shape*)

The amount of resources that we expect to be available on a given node at each point in time.

To represent the resources available in a reservation, we represent a reservation as a linked list of NodeReservations, each giving the resources free within a single timeslice.

Parameters
shape (toil.provisioners.abstractProvisioner.Shape) –

__str__()

Return str(self).

Return type
str

get_failed_constraints(*job_shape*)

Check if a job shape's resource requirements will fit within this allocation.

If the job does *not* fit, returns the failing constraints: the resources that can't be accomodated, and the limits that were hit.

If the job *does* fit, returns an empty list.

Must always agree with fits()! This codepath is slower and used for diagnosis.

Parameters
job_shape (toil.provisioners.abstractProvisioner.Shape) –

Return type
List[FailedConstraint]

fits(*jobShape*)

Check if a job shape's resource requirements will fit within this allocation.

Parameters
jobShape (toil.provisioners.abstractProvisioner.Shape) –

Return type
bool

shapes()

Get all time-slice shapes, in order, from this reservation on.

Return type
List[*toil.provisioners.abstractProvisioner.Shape*]

subtract(*jobShape*)

Subtract the resources necessary to run a jobShape from the reservation.

Parameters
jobShape (toil.provisioners.abstractProvisioner.Shape) –

Return type
None

attemptToAddJob(*jobShape*, *nodeShape*, *targetTime*)

Attempt to pack a job into this reservation timeslice and/or the reservations after it.

jobShape is the Shape of the job requirements, nodeShape is the Shape of the node this is a reservation for, and targetTime is the maximum time to wait before starting this job.

Parameters

- **jobShape** (toil.provisioners.abstractProvisioner.Shape) –
- **nodeShape** (toil.provisioners.abstractProvisioner.Shape) –
- **targetTime** (*float*) –

Return type
bool

toil.provisioners.clusterScaler.**adjustEndingReservationForJob**(*reservation*, *jobShape*, *wallTime*)

Add a job to an ending reservation that ends at wallTime.

(splitting the reservation if the job doesn't fill the entire timeslice)

Parameters

- **reservation** (NodeReservation) –
- **jobShape** (toil.provisioners.abstractProvisioner.Shape) –
- **wallTime** (*float*) –

Return type
None

toil.provisioners.clusterScaler.**split**(*nodeShape*, *jobShape*, *wallTime*)

Partition a node allocation into two to fit the job.

Returning the modified shape of the node and a new node reservation for the extra time that the job didn't fill.

Parameters

- **nodeShape** (toil.provisioners.abstractProvisioner.Shape) –

- **jobShape** (toil.provisioners.abstractProvisioner.Shape) –
- **wallTime** (*float*) –

Return type
Tuple[*toil.provisioners.abstractProvisioner.Shape*, *NodeReservation*]

toil.provisioners.clusterScaler.**binPacking**(*nodeShapes*, *jobShapes*, *goalTime*)

Using the given node shape bins, pack the given job shapes into nodes to get them done in the given amount of time.

Returns a dict saying how many of each node will be needed, a dict from job shapes that could not fit to reasons why.

Parameters

- **nodeShapes** (*List[*toil.provisioners.abstractProvisioner.Shape*]*) –
- **jobShapes** (*List[*toil.provisioners.abstractProvisioner.Shape*]*) –
- **goalTime** (*float*) –

Return type
Tuple[Dict[*toil.provisioners.abstractProvisioner.Shape*, int], Dict[*toil.provisioners.abstractProvisioner.Shape*, List[FailedConstraint]]]

class toil.provisioners.clusterScaler.**ClusterScaler**(*provisioner*, *leader*, *config*)

Parameters

- **provisioner** (toil.provisioners.abstractProvisioner.AbstractProvisioner) –
- **leader** (toil.leader.Leader) –
- **config** (toil.common.Config) –

getAverageRuntime(*jobName*, *service=False*)

Parameters

- **jobName** (*str*) –
- **service** (*bool*) –

Return type
float

addCompletedJob(*job*, *wallTime*)

Adds the shape of a completed job to the queue, allowing the scalar to use the last N completed jobs in factoring how many nodes are required in the cluster. :param toil.job.JobDescription job: The description of the completed job :param int wallTime: The wall-time taken to complete the job in seconds.

Parameters

- **job** (toil.job.JobDescription) –
- **wallTime** (*int*) –

Return type
None

setStaticNodes(*nodes*, *preemptible*)

Used to track statically provisioned nodes. This method must be called before any auto-scaled nodes are provisioned.

These nodes are treated differently than auto-scaled nodes in that they should not be automatically terminated.

Parameters

- **nodes** (*List[toil.provisioners.node.Node]*) – list of Node objects
- **preemptible** (*bool*) –

Return type
None

getStaticNodes(*preemptible*)

Returns nodes set in setStaticNodes().

Parameters
preemptible (*bool*) –

Returns
Statically provisioned nodes.

Return type
Dict[str, *toil.provisioners.node.Node*]

smoothEstimate(*nodeShape*, *estimatedNodeCount*)

Smooth out fluctuations in the estimate for this node compared to previous runs.

Returns an integer.

Parameters

- **nodeShape** (*toil.provisioners.abstractProvisioner.Shape*) –
- **estimatedNodeCount** (*int*) –

Return type
int

getEstimatedNodeCounts(*queuedJobShapes*, *currentNodeCounts*)

Given the resource requirements of queued jobs and the current size of the cluster.

Returns a dict mapping from nodeShape to the number of nodes we want in the cluster right now, and a dict from job shapes that are too big to run on any node to reasons why.

Parameters

- **queuedJobShapes** (*List[toil.provisioners.abstractProvisioner.Shape]*) –
- **currentNodeCounts** (*Dict[toil.provisioners.abstractProvisioner.Shape, int]*) –

Return type
Tuple[Dict[*toil.provisioners.abstractProvisioner.Shape*, int], Dict[*toil.provisioners.abstractProvisioner.Shape*, List[FailedConstraint]]]

updateClusterSize(*estimatedNodeCounts*)

Given the desired and current size of the cluster, attempts to launch/remove instances to get to the desired size.

Also attempts to remove ignored nodes that were marked for graceful removal.

Returns the new size of the cluster.

Parameters

estimatedNodeCounts (*Dict[toil.provisioners.abstractProvisioner.Shape, int]*) –

Return type

Dict[*toil.provisioners.abstractProvisioner.Shape*, int]

setNodeCount(*instance_type*, *numNodes*, *preemptible=False*, *force=False*)

Attempt to grow or shrink the number of preemptible or non-preemptible worker nodes in the cluster to the given value, or as close a value as possible, and, after performing the necessary additions or removals of worker nodes, return the resulting number of preemptible or non-preemptible nodes currently in the cluster.

Parameters

- **instance_type** (*str*) – The instance type to add or remove.
- **numNodes** (*int*) – Desired size of the cluster
- **preemptible** (*bool*) – whether the added nodes will be preemptible, i.e. whether they may be removed spontaneously by the underlying platform at any time.
- **force** (*bool*) – If False, the provisioner is allowed to deviate from the given number of nodes. For example, when downsizing a cluster, a provisioner might leave nodes running if they have active jobs running on them.

Returns

the number of worker nodes in the cluster after making the necessary adjustments. This value should be, but is not guaranteed to be, close or equal to the *numNodes* argument. It represents the closest possible approximation of the actual cluster size at the time this method returns.

Return type

int

filter_out_static_nodes(*nodes*, *preemptible=False*)

Parameters

- **nodes** (*Dict[toil.provisioners.node.Node, toil.batchSystems.abstractBatchSystem.NodeInfo]*) –
- **preemptible** (*bool*) –

Return type

List[Tuple[*toil.provisioners.node.Node*, *toil.batchSystems.abstractBatchSystem.NodeInfo*]]

getNodes(*preemptible=None*)

Returns a dictionary mapping node identifiers of preemptible or non-preemptible nodes to NodeInfo objects, one for each node.

This method is the definitive source on nodes in cluster, & is responsible for consolidating cluster state between the provisioner & batch system.

Parameters

preemptible (*bool*) – If True (False) only (non-)preemptible nodes will be returned. If None, all nodes will be returned.

Return type

Dict[*toil.provisioners.node.Node*, *toil.batchSystems.abstractBatchSystem.NodeInfo*]

shutDown()

Return type
None

exception toil.provisioners.clusterScaler.**JobTooBigError**(*job=None*, *shape=None*, *constraints=None*)

Bases: Exception

Raised in the scaler thread when a job cannot fit in any available node type and is likely to lock up the workflow.

Parameters

- **job** (*Optional[toil.job.JobDescription]*) –
- **shape** (*Optional[toil.provisioners.abstractProvisioner.Shape]*) –
- **constraints** (*Optional[List[FailedConstraint]]*) –

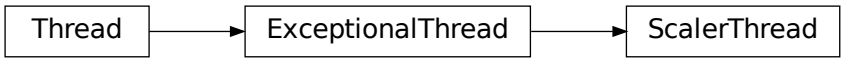
__str__()

Stringify the exception, including the message.

Return type
str

class toil.provisioners.clusterScaler.**ScalerThread**(*provisioner*, *leader*, *config*, *stop_on_exception=False*)

Bases: *toil.lib.threading.ExceptionalThread*



A thread that automatically scales the number of either preemptible or non-preemptible worker nodes according to the resource requirements of the queued jobs.

The scaling calculation is essentially as follows: start with 0 estimated worker nodes. For each queued job, check if we expect it can be scheduled into a worker node before a certain time (currently one hour). Otherwise, attempt to add a single new node of the smallest type that can fit that job.

At each scaling decision point a comparison between the current, C, and newly estimated number of nodes is made. If the absolute difference is less than beta * C then no change is made, else the size of the cluster is adapted. The beta factor is an inertia parameter that prevents continual fluctuations in the number of nodes.

Parameters

- **provisioner** (toil.provisioners.abstractProvisioner.AbstractProvisioner) –
- **leader** (toil.leader.Leader) –
- **config** (toil.common.Config) –
- **stop_on_exception** (*bool*) –

check()

Attempt to join any existing scaler threads that may have died or finished.

This insures any exceptions raised in the threads are propagated in a timely fashion.

Return type
None

shutdown()

Shutdown the cluster.

Return type
None

addCompletedJob(*job*, *wallTime*)

Parameters

- **job** (toil.job.JobDescription) –
- **wallTime** (*int*) –

Return type
None

tryRun()

Return type
None

class toil.provisioners.clusterScaler.**ClusterStats**(*path*, *batchSystem*, *clusterName*)

Parameters

- **path** (*str*) –
- **batchSystem** (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem) –
- **clusterName** (*Optional[str]*) –

shutDownStats()

Return type
None

startStats(*preemptible*)

Parameters
preemptible (*bool*) –

Return type
None

checkStats()

Return type
None

toil.provisioners.gceProvisioner

Module Contents

Classes

GCEProvisioner	Implements a Google Compute Engine Provisioner using libcloud.

Attributes

logger

toil.provisioners.gceProvisioner.**logger**

class toil.provisioners.gceProvisioner.**GCEProvisioner**(*clusterName*, *clusterType*, *zone*, *nodeStorage*, *nodeStorageOverrides*, *sseKey*)

Bases: *toil.provisioners.abstractProvisioner.AbstractProvisioner*

Implements a Google Compute Engine Provisioner using libcloud.

NODE_BOTO_PATH = '/root/.boto'

SOURCE_IMAGE = b'projects/kinvolk-public/global/images/family/flatcar-stable'

DEFAULT_TASK_COMPLETION_TIMEOUT = 180

supportedClusterTypes()

Get all the cluster types that this provisioner implementation supports.

createClusterSettings()

Initialize class for a new cluster, to be deployed, when running outside the cloud.

readClusterSettings()

Read the cluster settings from the instance, which should be the leader. See https://cloud.google.com/compute/docs/storing-retrieving-metadata for details about reading the metadata.

launchCluster(*leaderNodeType*, *leaderStorage*, *owner*, ***kwargs*)

In addition to the parameters inherited from the abstractProvisioner, the Google launchCluster takes the following parameters: keyName: The key used to communicate with instances botoPath: Boto credentials for reading an AWS jobStore (optional). network: a network (optional) vpcSubnet: A subnet (optional). use_private_ip: even though a public ip exists, ignore it (optional)

getNodeShape(*instance_type*, *preemptible=False*)

The shape of a preemptible or non-preemptible node managed by this provisioner. The node shape defines key properties of a machine, such as its number of cores or the time between billing intervals.

Parameters
instance_type (*str*) – Instance type name to return the shape of.

Return type
Shape

static retryPredicate(*e*)

Not used by GCE

destroyCluster()

Try a few times to terminate all of the instances in the group.

Return type
None

terminateNodes(*nodes*)

Terminate the nodes represented by given Node objects

Parameters
nodes – list of Node objects

addNodes(*nodeTypes*, *numNodes*, *preemptible*, *spotBid=None*)

Used to add worker nodes to the cluster

Parameters

- **numNodes** – The number of nodes to add
- **preemptible** – whether or not the nodes will be preemptible
- **spotBid** – The bid for preemptible nodes if applicable (this can be set in config, also).
- **nodeTypes** (*Set[str]*) –

Returns
number of nodes successfully added

Return type
int

getProvisionedWorkers(*instance_type=None*, *preemptible=None*)

Gets all nodes, optionally of the given instance type or preemptability, from the provisioner. Includes both static and autoscaled nodes.

Parameters

- **preemptible** (*Optional[bool]*) – Boolean value to restrict to preemptible nodes or non-preemptible nodes
- **instance_type** (*Optional[str]*) –

Returns
list of Node objects

getLeader()

Returns
The leader node.

ex_create_multiple_nodes(*base_name*, *size*, *image*, *number*, *location=None*, *ex_network='default'*, *ex_subnetwork=None*, *ex_tags=None*, *ex_metadata=None*, *ignore_errors=True*, *use_existing_disk=True*, *poll_interval=2*, *external_ip='ephemeral'*, *ex_disk_type='pd-standard'*, *ex_disk_auto_delete=True*, *ex_service_accounts=None*, *timeout=DEFAULT_TASK_COMPLETION_TIMEOUT*, *description=None*, *ex_can_ip_forward=None*, *ex_disks_gce_struct=None*, *ex_nic_gce_struct=None*, *ex_on_host_maintenance=None*, *ex_automatic_restart=None*, *ex_image_family=None*, *ex_preemptible=None*)

Monkey patch to gce.py in libcloud to allow disk and images to be specified. Also changed name to a uuid below. The prefix 'wp' identifies preemptible nodes and 'wn' non-preemptible nodes.

toil.provisioners.node

Module Contents

Classes

Node

Attributes

a_short_time
logger

toil.provisioners.node.**a_short_time** = 5

toil.provisioners.node.**logger**

class toil.provisioners.node.**Node**(*publicIP*, *privateIP*, *name*, *launchTime*, *nodeType*, *preemptible*, *tags=None*, *use_private_ip=None*)

maxWaitTime

__str__()
Return str(self).

__repr__()
Return repr(self).

__hash__()
Return hash(self).

remainingBillingInterval()

If the node has a launch time, this function returns a floating point value between 0 and 1.0 representing how far we are into the current billing cycle for the given instance. If the return value is .25, we are one quarter into the billing cycle, with three quarters remaining before we will be charged again for that instance.

Assumes a billing cycle of one hour.

Returns
Float from 0 -> 1.0 representing percentage of pre-paid time left in cycle.

Return type
float

waitForNode(*role*, *keyName='core'*)

copySshKeys(*keyName*)

Copy authorized_keys file to the core user from the keyName user.

injectFile(*fromFile*, *toFile*, *role*)

rysnc a file to the container with the given role

extractFile(*fromFile*, *toFile*, *role*)

rysnc a file from the container with the given role

sshAppliance(**args*, ***kwargs*)

Parameters

- **args** – arguments to execute in the appliance
- **kwargs** – tty=bool tells docker whether or not to create a TTY shell for interactive SSHing. The default value is False. Input=string is passed as input to the Popen call.

sshInstance(**args*, ***kwargs*)

Run a command on the instance. Returns the binary output of the command.

coreSSH(**args*, ***kwargs*)

If strict=False, strict host key checking will be temporarily disabled. This is provided as a convenience for internal/automated functions and ought to be set to True whenever feasible, or whenever the user is directly interacting with a resource (e.g. rsync-cluster or ssh-cluster). Assumed to be False by default.

kwargs: input, tty, appliance, collectStdout, sshOptions, strict

Parameters
input (*bytes*) – UTF-8 encoded input bytes to send to the command

coreRsync(*args*, *applianceName='toil_leader'*, ***kwargs*)

Package Contents

Functions

cluster_factory(provisioner[, clusterName, ...])	Find and instantiate the appropriate provisioner instance to make clusters in the given cloud.
add_provisioner_options(parser)	
parse_node_types(node_type_specs)	Parse a specification for zero or more node types.
check_valid_node_types(provisioner, node_types)	Raises if an invalid nodeType is specified for aws or gce.

Attributes

logger

toil.provisioners.**logger**

toil.provisioners.**cluster_factory**(*provisioner*, *clusterName=None*, *clusterType='mesos'*, *zone=None*, *nodeStorage=50*, *nodeStorageOverrides=None*, *sseKey=None*)

Find and instantiate the appropriate provisioner instance to make clusters in the given cloud.

Raises ClusterTypeNotSupportedException if the given provisioner does not implement clusters of the given type.

Parameters

- **provisioner** (*str*) – The cloud type of the cluster.
- **clusterName** (*Optional[str]*) – The name of the cluster.
- **clusterType** (*str*) – The type of cluster: 'mesos' or 'kubernetes'.
- **zone** (*Optional[str]*) – The cloud zone
- **nodeStorage** (*int*) –
- **nodeStorageOverrides** (*Optional[List[str]]*) –
- **sseKey** (*Optional[str]*) –

Returns
A cluster object for the the cloud type.

Return type
Union[*aws.awsProvisioner.AWSProvisioner*, *gceProvisioner.GCEProvisioner*]

toil.provisioners.**add_provisioner_options**(*parser*)

Parameters
parser (*argparse.ArgumentParser*) –

Return type
None

toil.provisioners.**parse_node_types**(*node_type_specs*)

Parse a specification for zero or more node types.

Takes a comma-separated list of node types. Each node type is a slash-separated list of at least one instance type name (like 'm5a.large' for AWS), and an optional bid in dollars after a colon.

Raises ValueError if a node type cannot be parsed.

Inputs should look something like this:

```
>>> parse_node_types('c5.4xlarge/c5a.4xlarge:0.42,t2.large')
[({'c5.4xlarge', 'c5a.4xlarge'}, 0.42), ({'t2.large'}, None)]
```

Parameters
node_type_specs (*Optional[str]*) – A string defining node types

Returns
a list of node types, where each type is the set of instance types, and the float bid, or None.

Return type
List[Tuple[Set[str], Optional[float]]]

toil.provisioners.**check_valid_node_types**(*provisioner*, *node_types*)

Raises if an invalid nodeType is specified for aws or gce.

Parameters

- **provisioner** (*str*) – 'aws' or 'gce' to specify which cloud provisioner used.
- **node_types** (*List[Tuple[Set[str], Optional[float]]]*) – A list of node types. Example: [({'t2.micro'}, None), ({'t2.medium'}, 0.5)]

Returns
Nothing. Raises if any instance type in the node type isn't real.

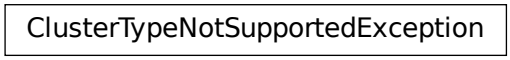
exception toil.provisioners.**NoSuchClusterException**(*cluster_name*)

Bases: Exception

Indicates that the specified cluster does not exist.

exception toil.provisioners.**ClusterTypeNotSupportedException**(*provisioner_class*, *cluster_type*)

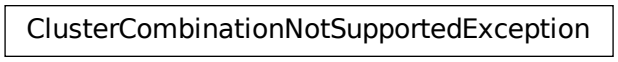
Bases: Exception



Indicates that a provisioner does not support a given cluster type.

exception toil.provisioners.**ClusterCombinationNotSupportedException**(*provisioner_class*, *cluster_type*, *architecture*, *reason=None*)

Bases: Exception



Indicates that a provisioner does not support making a given type of cluster with a given architecture.

Parameters

- **provisioner_class** (*Type*) –
- **cluster_type** (*str*) –
- **architecture** (*str*) –
- **reason** (*Optional[str]*) –

toil.server

Subpackages

toil.server.api_spec

toil.server.cli

Submodules

toil.server.cli.wes_cwl_runner

Module Contents

Classes

WESClientWithWorkflowEngineParameters	A modified version of the WESClient from the wes-service package that

Functions

generate_attachment_path_names(paths)	Take in a list of path names and return a list of names with the common path
get_deps_from_cwltool(cwl_file[, input_file])	Return a list of dependencies of the given workflow from cwltool.
submit_run(client, cwl_file[, input_file, engine_options])	Given a CWL file, its input files, and an optional list of engine options,
poll_run(client, run_id)	Return True if the given workflow run is in a finished state.
print_logs_and_exit(client, run_id)	Fetch the workflow logs from the WES server, print the results, then exit
main()	

Attributes

logger

toil.server.cli.wes_cwl_runner.**logger**

toil.server.cli.wes_cwl_runner.**generate_attachment_path_names**(*paths*)

Take in a list of path names and return a list of names with the common path name stripped out, while preserving the input order. This guarantees that there are no relative paths that traverse up.

For example, for the following CWL workflow where "hello.yaml" references a file "message.txt",

~/toil/workflows/hello.cwl ~/toil/input_files/hello.yaml ~/toil/input_files/message.txt

This may be run with the command:

toil-wes-cwl-runner hello.cwl ../input_files/hello.yaml

Where "message.txt" is resolved to "../input_files/message.txt".

We'd send the workflow file as "workflows/hello.cwl", and send the inputs as "input_files/hello.yaml" and "input_files/message.txt".

Parameters

paths (*List[str]*) – A list of absolute or relative path names. Relative paths are interpreted as relative to the current working directory.

Returns

The common path name and a list of minimal path names.

Return type

Tuple[str, List[str]]

class toil.server.cli.wes_cwl_runner.**WESClientWithWorkflowEngineParameters**(*endpoint*, *auth=None*)

Bases: wes_client.util.WESClient

WESClientWithWorkflowEngineParameters

A modified version of the WESClient from the wes-service package that includes workflow_engine_parameters support.

TODO: Propose a PR in wes-service to include workflow_engine_params.

Parameters

- **endpoint** (*str*) –
- **auth** (*Optional[Tuple[str, str]]*) –

get_version(*extension*, *workflow_file*)

Determines the version of a .py, .wdl, or .cwl file.

Parameters

- **extension** (*str*) –
- **workflow_file** (*str*) –

Return type
str

parse_params(*workflow_params_file*)

Parse the CWL input file into a dictionary to be attached to the body of the WES run request.

Parameters
workflow_params_file (*str*) – The URL or path to the CWL input file.

Return type
Dict[str, Any]

modify_param_paths(*base_dir*, *workflow_params*)

Modify the file paths in the input workflow parameters to be relative to base_dir.

Parameters

- **base_dir** (*str*) – The base directory to make the file paths relative to. This should be the common ancestor of all attached files, which will become the root of the execution folder.
- **workflow_params** (*Dict[str, Any]*) – A dict containing the workflow parameters.

Return type
None

build_wes_request(*workflow_file*, *workflow_params_file*, *attachments*, *workflow_engine_parameters=None*)

Build the workflow run request to submit to WES.

Parameters

- **workflow_file** (*str*) – The path or URL to the CWL workflow document. Only file:// URL supported at the moment.

- **workflow_params_file** (*Optional[str]*) – The path or URL to the CWL input file.
- **attachments** (*Optional[List[str]]*) – A list of local paths to files that will be uploaded to the server.
- **workflow_engine_parameters** (*Optional[List[str]]*) – A list of engine parameters to set along with this workflow run.

Returns
A dictionary of parameters as the body of the request, and an iterable for the pairs of filename and file contents to upload to the server.

Return type
Tuple[Dict[str, str], Iterable[Tuple[str, Tuple[str, io.BytesIO]]]]

run_with_engine_options(*workflow_file*, *workflow_params_file*, *attachments*, *workflow_engine_parameters*)

Composes and sends a post request that signals the WES server to run a workflow.

Parameters

- **workflow_file** (*str*) – The path to the CWL workflow document.
- **workflow_params_file** (*Optional[str]*) – The path to the CWL input file.
- **attachments** (*Optional[List[str]]*) – A list of local paths to files that will be uploaded to the server.
- **workflow_engine_parameters** (*Optional[List[str]]*) – A list of engine parameters to set along with this workflow run.

Returns
The body of the post result as a dictionary.

Return type
Dict[str, Any]

toil.server.cli.wes_cwl_runner.**get_deps_from_cwltool**(*cwl_file*, *input_file=None*)

Return a list of dependencies of the given workflow from cwltool.

Parameters

- **cwl_file** (*str*) – The CWL file.
- **input_file** (*Optional[str]*) – Omit to get the dependencies from the CWL file. If set, this returns the dependencies from the input file.

Return type
List[str]

toil.server.cli.wes_cwl_runner.**submit_run**(*client*, *cwl_file*, *input_file=None*, *engine_options=None*)

Given a CWL file, its input files, and an optional list of engine options, submit the CWL workflow to the WES server via the WES client.

This function also attempts to find the attachments from the CWL workflow and its input file, and attach them to the WES run request.

Parameters

- **client** (*WESClientWithWorkflowEngineParameters*) – The WES client.
- **cwl_file** (*str*) – The path to the CWL workflow document.
- **input_file** (*Optional[str]*) – The path to the CWL input file.

- **engine_options** (*Optional[List[str]]*) – A list of engine parameters to set along with this workflow run.

Return type
str

toil.server.cli.wes_cwl_runner.**poll_run**(*client*, *run_id*)

Return True if the given workflow run is in a finished state.

Parameters

- **client** (WESClientWithWorkflowEngineParameters) –
- **run_id** (*str*) –

Return type
bool

toil.server.cli.wes_cwl_runner.**print_logs_and_exit**(*client*, *run_id*)

Fetch the workflow logs from the WES server, print the results, then exit the program with the same exit code as the workflow run.

Parameters

- **client** (WESClientWithWorkflowEngineParameters) – The WES client.
- **run_id** (*str*) – The run_id of the target workflow.

Return type
None

toil.server.cli.wes_cwl_runner.**main**()

Return type
None

toil.server.wes

Submodules

toil.server.wes.abstract_backend

Module Contents

Classes

WESBackend	A class to represent a GA4GH Workflow Execution Service (WES) API backend.

Functions

handle_errors(func)	This decorator catches errors from the wrapped function and returns a JSON

Attributes

logger
TaskLog

toil.server.wes.abstract_backend.**logger**

toil.server.wes.abstract_backend.**TaskLog**

exception toil.server.wes.abstract_backend.**VersionNotImplementedException**(*wf_type*, *version=None*, *supported_versions=None*)

Bases: Exception

Raised when the requested workflow version is not implemented.

Parameters

- **wf_type** (*str*) –
- **version** (*Optional[str]*) –
- **supported_versions** (*Optional[List[str]]*) –

exception toil.server.wes.abstract_backend.**MalformedRequestException**(*message*)

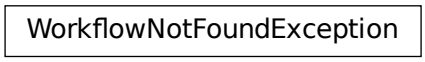
Bases: Exception

Raised when the request is malformed.

Parameters
message (*str*) –

exception toil.server.wes.abstract_backend.**WorkflowNotFoundException**

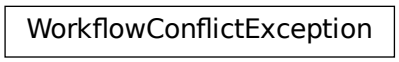
Bases: Exception



Raised when the requested run ID is not found.

exception toil.server.wes.abstract_backend.**WorkflowConflictException**(*run_id*)

Bases: Exception

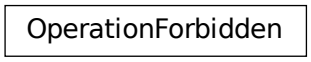


Raised when the requested workflow is not in the expected state.

Parameters
run_id (*str*) –

exception toil.server.wes.abstract_backend.**OperationForbidden**(*message*)

Bases: Exception

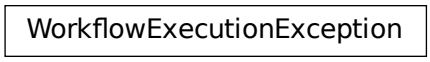


Raised when the request is forbidden.

Parameters
message (*str*) –

exception toil.server.wes.abstract_backend.**WorkflowExecutionException**(*message*)

Bases: Exception



Raised when an internal error occurred during the execution of the workflow.

Parameters
message (*str*) –

toil.server.wes.abstract_backend.**handle_errors**(*func*)

This decorator catches errors from the wrapped function and returns a JSON formatted error message with the appropriate status code defined by the GA4GH WES spec.

Parameters
func (*Callable[Ellipsis, Any]*) –

Return type
Callable[Ellipsis, Any]

class toil.server.wes.abstract_backend.**WESBackend**(*options*)

A class to represent a GA4GH Workflow Execution Service (WES) API backend. Intended to be inherited. Subclasses should implement all abstract methods to handle user requests when they hit different endpoints.

Parameters
options (*List[str]*) –

resolve_operation_id(*operation_id*)

Map an operationId defined in the OpenAPI or swagger yaml file to a function.

Parameters
operation_id (*str*) – The operation ID defined in the specification.

Returns
A function that should be called when the given endpoint is reached.

Return type
Any

abstract get_service_info()

Get information about the Workflow Execution Service.

GET /service-info

Return type
Dict[str, Any]

abstract list_runs(*page_size=None*, *page_token=None*)

List the workflow runs.

GET /runs

Parameters

- **page_size** (*Optional[int]*) –
- **page_token** (*Optional[str]*) –

Return type
Dict[str, Any]

abstract run_workflow()

Run a workflow. This endpoint creates a new workflow run and returns a *RunId* to monitor its progress.

POST /runs

Return type
Dict[str, str]

abstract get_run_log(*run_id*)

Get detailed info about a workflow run.

GET /runs/{run_id}

Parameters
run_id (*str*) –

Return type
Dict[str, Any]

abstract cancel_run(*run_id*)

Cancel a running workflow.

POST /runs/{run_id}/cancel

Parameters
run_id (*str*) –

Return type
Dict[str, str]

abstract get_run_status(*run_id*)

Get quick status info about a workflow run, returning a simple result with the overall state of the workflow run.

GET /runs/{run_id}/status

Parameters
run_id (*str*) –

Return type
Dict[str, str]

static log_for_run(*run_id*, *message*)

Parameters

- **run_id** (*Optional[str]*) –
- **message** (*str*) –

Return type
None

static secure_path(*path*)

Parameters
path (*str*) –

Return type
str

collect_attachments(*run_id*, *temp_dir*)

Collect attachments from the current request by staging uploaded files to temp_dir, and return the temp_dir and parsed body of the request.

Parameters

- **run_id** (*Optional[str]*) – The run ID for logging.
- **temp_dir** (*Optional[str]*) – The directory where uploaded files should be staged. If None, a temporary directory is created.

Return type

Tuple[str, Dict[str, Any]]

toil.server.wes.amazon_wes_utils

Module Contents

Classes

WorkflowPlan	These functions pass around dicts of a certain type, with *data* and *files* keys.
DataDict	Under *data*, there can be:
FilesDict	Under *files*, there can be:

Functions

parse_workflow_zip_file(file, workflow_type)	Processes a workflow zip bundle
parse_workflow_manifest_file(manifest_file)	Reads a MANIFEST.json file for a workflow zip bundle
workflow_manifest_url_to_path(url[, parent_dir])	Interpret a possibly-relative parsed URL, relative to the given parent directory.
task_filter(task, job_status)	AGC requires task names to be annotated with an AWS Batch job ID that they

Attributes

logger
NOTICE

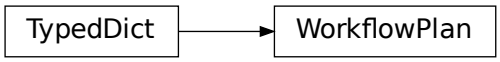
toil.server.wes.amazon_wes_utils.**logger**

toil.server.wes.amazon_wes_utils.**NOTICE = Multiline-String**

```
"""
Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
"""
```

class toil.server.wes.amazon_wes_utils.**WorkflowPlan**

Bases: TypedDict



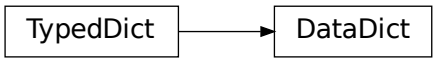
These functions pass around dicts of a certain type, with *data* and *files* keys.

data: *DataDict*

files: *FilesDict*

class toil.server.wes.amazon_wes_utils.**DataDict**

Bases: TypedDict



Under *data*, there can be: * *workflowUrl* (required if no *workflowSource*): URL to main workflow code.

workflowUrl: str

class toil.server.wes.amazon_wes_utils.**FilesDict**

Bases: TypedDict

Under *files*, there can be: * *workflowSource* (required if no *workflowUrl*): Open binary-mode file for the main workflow code. * *workflowInputFiles*: List of open binary-mode file for input files. Expected to be JSONs. * *workflowOptions*: Open binary-mode file for a JSON of options sent along with the workflow. * *workflowDependencies*: Open binary-mode file for the zip the workflow came in, if any.

workflowSource: IO[bytes]

workflowInputFiles: List[IO[bytes]]

workflowOptions: IO[bytes]

workflowDependencies: IO[bytes]

toil.server.wes.amazon_wes_utils.**parse_workflow_zip_file**(*file*, *workflow_type*)

Processes a workflow zip bundle

Parameters

- **file** (*str*) – String or Path-like path to a workflow.zip file
- **workflow_type** (*str*) – String, extension of workflow to expect (e.g. "wdl")

Return type
dict of *data* and *files*

If the zip only contains a single file, that file is set as *workflowSource*

If the zip contains multiple files with a MANIFEST.json file, the MANIFEST is used to determine appropriate *data* and *file* arguments. (See: parse_workflow_manifest_file())

If the zip contains multiple files without a MANIFEST.json file:

- a *main* workflow file with an extension matching the workflow_type is expected and will be set as *workflowSource*
- optionally, if *inputs*.json* files are found in the root level of the zip, they will be set as *workflowInputs(_d)** in the order they are found
- optionally, if an *options.json* file is found in the root level of the zip, it will be set as *workflowOptions*

If the zip contains multiple files, the original zip is set as *workflowDependencies*

toil.server.wes.amazon_wes_utils.**parse_workflow_manifest_file**(*manifest_file*)

Reads a MANIFEST.json file for a workflow zip bundle

Parameters
manifest_file (*str*) – String or Path-like path to a MANIFEST.json file

Return type
dict of *data* and *files*

MANIFEST.json is expected to be formatted like: .. code-block:: json

{
"mainWorkflowURL": "relpath/to/workflow", "inputFileURLs": [

"relpath/to/input-file-1", "relpath/to/input-file-2", ...

], "optionsFileURL" "relpath/to/option-file

}

The *mainWorkflowURL* property that provides a relative file path in the zip to a workflow file, which will be set as *workflowSource*

The inputFileURLs property is optional and provides a list of relative file paths in the zip to input.json files. The list is assumed to be in the order the inputs should be applied - e.g. higher list index is higher priority. If present, it will be used to set *workflowInputs(_d)* arguments.

The optionsFileURL property is optional and provides a relative file path in the zip to an options.json file. If present, it will be used to set *workflowOptions*.

toil.server.wes.amazon_wes_utils.**workflow_manifest_url_to_path**(*url*, *parent_dir=None*)

Interpret a possibly-relative parsed URL, relative to the given parent directory.

Parameters

- **url** (*urllib.parse.ParseResult*) –
- **parent_dir** (*Optional[str]*) –

Return type
str

toil.server.wes.amazon_wes_utils.**task_filter**(*task*, *job_status*)

AGC requires task names to be annotated with an AWS Batch job ID that they were run under. If it encounters an un-annotated task name, it will crash. See <https://github.com/aws/amazon-genomics-cli/issues/494>.

This encodes the AWSBatchJobID annotation, from the AmazonBatchBatchSystem, into the task name of the given task, and returns the modified task. If no such annotation is available, the task is censored and None is returned.

Parameters

- **task** (*toil.server.wes.abstract_backend.TaskLog*) –
- **job_status** (toil.bus.JobStatus) –

Return type
Optional[toil.server.wes.abstract_backend.TaskLog]

toil.server.wes.tasks

Module Contents

Classes

ToilWorkflowRunner	A class to represent a workflow runner to run the requested workflow.
TaskRunner	Abstraction over the Celery API. Runs our run_wes task and allows canceling it.
MultiprocessingTaskRunner	Version of TaskRunner that just runs tasks with Multiprocessing.

Functions

run_wes_task(base_scratch_dir, state_store_url, ...)	Run a requested workflow.
cancel_run(task_id)	Send a SIGTERM signal to the process that is running task_id.

Attributes

logger
WAIT_FOR_DEATH_TIMEOUT
run_wes

toil.server.wes.tasks.**logger**

toil.server.wes.tasks.**WAIT_FOR_DEATH_TIMEOUT = 20**

class toil.server.wes.tasks.**ToilWorkflowRunner**(*base_scratch_dir*, *state_store_url*, *workflow_id*, *request*, *engine_options*)

A class to represent a workflow runner to run the requested workflow.

Responsible for parsing the user request into a shell command, executing that command, and collecting the outputs of the resulting workflow run.

Parameters

- **base_scratch_dir** (*str*) –
- **state_store_url** (*str*) –
- **workflow_id** (*str*) –
- **request** (*Dict[str, Any]*) –
- **engine_options** (*List[str]*) –

write_scratch_file(*filename*, *contents*)

Write a file to the scratch directory.

Parameters

- **filename** (*str*) –
- **contents** (*str*) –

Return type
None

get_state()

Return type
str

write_workflow(*src_url*)

Fetch the workflow file from its source and write it to a destination file.

Parameters
src_url (*str*) –

Return type
str

sort_options(*workflow_engine_parameters=None*)

Sort the command line arguments in the order that can be recognized by the workflow execution engine.

Parameters
workflow_engine_parameters (*Optional[Dict[str, Optional[str]]]*) – User-specified parameters for this

Return type
List[str]

particular workflow. Keys are command-line options, and values are option arguments, or None for options that are flags.

initialize_run()

Write workflow and input files and construct a list of shell commands to be executed. Return that list of shell commands that should be executed in order to complete this workflow run.

Return type
List[str]

call_cmd(*cmd*, *cwd*)

Calls a command with Popen. Writes stdout, stderr, and the command to separate files.

Parameters

- **cmd** (*Union[List[str], str]*) –
- **cwd** (*str*) –

Return type
subprocess.Popen[bytes]

run()

Construct a command to run a the requested workflow with the options, run it, and deposit the outputs in the output directory.

Return type
None

write_output_files()

Fetch all the files that this workflow generated and output information about them to *outputs.json*.

Return type
None

toil.server.wes.tasks.**run_wes_task**(*base_scratch_dir*, *state_store_url*, *workflow_id*, *request*, *engine_options*)

Run a requested workflow.

Parameters

- **base_scratch_dir** (*str*) – Directory where the workflow's scratch dir will live, under the workflow's ID.
- **state_store_url** (*str*) – URL/path at which the server and Celery task communicate about workflow state.
- **workflow_id** (*str*) – ID of the workflow run.
- **request** (*Dict[str, Any]*) –
- **engine_options** (*List[str]*) –

Returns
the state of the workflow run.

Return type
str

toil.server.wes.tasks.**run_wes**

toil.server.wes.tasks.**cancel_run**(*task_id*)

Send a SIGTERM signal to the process that is running task_id.

Parameters
task_id (*str*) –

Return type
None

class toil.server.wes.tasks.**TaskRunner**

Abstraction over the Celery API. Runs our run_wes task and allows canceling it.

We can swap this out in the server to allow testing without Celery.

static run(*args*, *task_id*)

Run the given task args with the given ID on Celery.

Parameters

- **args** (*Tuple[str, str, str, Dict[str, Any], List[str]]*) –
- **task_id** (*str*) –

Return type
None

static cancel(*task_id*)

Cancel the task with the given ID on Celery.

Parameters
task_id (*str*) –

Return type
None

static is_ok(*task_id*)

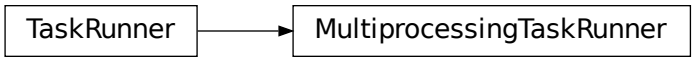
Make sure that the task running system is working for the given task. If the task system has detected an internal failure, return False.

Parameters
task_id (*str*) –

Return type
bool

class toil.server.wes.tasks.**MultiprocessingTaskRunner**

Bases: *TaskRunner*



Version of TaskRunner that just runs tasks with Multiprocessing.

Can't use threading because there's no way to send a cancel signal or exception to a Python thread, if loops in the task (i.e. ToilWorkflowRunner) don't poll for it.

static set_up_and_run_task(*output_path*, *args*)

Set up logging for the process into the given file and then call run_wes_task with the given arguments.

If the process finishes successfully, it will clean up the log, but if the process crashes, the caller must clean up the log.

Parameters

- **output_path** (*str*) –
- **args** (*Tuple[str, str, str, Dict[str, Any], List[str]]*) –

Return type
None

classmethod run(*args*, *task_id*)

Run the given task args with the given ID.

Parameters

- **args** (*Tuple[str, str, str, Dict[str, Any], List[str]]*) –
- **task_id** (*str*) –

Return type
None

classmethod cancel(*task_id*)

Cancel the task with the given ID.

Parameters
task_id (*str*) –

Return type
None

classmethod is_ok(*task_id*)

Make sure that the task running system is working for the given task. If the task system has detected an internal failure, return False.

Parameters
task_id (*str*) –

Return type
bool

toil.server.wes.toil_backend

Module Contents

Classes

ToilWorkflow	
ToilBackend	WES backend implemented for Toil to run CWL, WDL, or Toil workflows. This

Attributes

logger

toil.server.wes.toil_backend.**logger**

class toil.server.wes.toil_backend.**ToilWorkflow**(*base_work_dir*, *state_store_url*, *run_id*)

Parameters

- **base_work_dir** (*str*) –
- **state_store_url** (*str*) –
- **run_id** (*str*) –

fetch_state(*key: str*, *default: str*) → str

fetch_state(*key: str*, *default: None = None*) → Optional[str]

Return the contents of the given key in the workflow's state store. If the key does not exist, the default value is returned.

fetch_scratch(*filename*)

Get a context manager for either a stream for the given file from the workflow's scratch directory, or None if it isn't there.

Parameters
filename (*str*) –

Return type
Generator[Optional[TextIO], None, None]

exists()

Return True if the workflow run exists.

Return type
bool

get_state()

Return the state of the current run.

Return type
str

check_on_run(*task_runner*)

Check to make sure nothing has gone wrong in the task runner for this workflow. If something has, log, and fail the workflow with an error.

Parameters
task_runner (*Type[toil.server.wes.tasks.TaskRunner]*) –

Return type
None

set_up_run()

Set up necessary directories for the run.

Return type
None

clean_up()

Clean directory and files related to the run.

Return type
None

queue_run(*task_runner*, *request*, *options*)

This workflow should be ready to run. Hand this to the task system.

Parameters

- **task_runner** (*Type[toil.server.wes.tasks.TaskRunner]*) –
- **request** (*Dict[str, Any]*) –
- **options** (*List[str]*) –

Return type
None

get_output_files()

Return a collection of output files that this workflow generated.

Return type
Any

get_stdout_path()

Return the path to the standard output log, relative to the run's scratch_dir, or None if it doesn't exist.

Return type
Optional[str]

get_stderr_path()

Return the path to the standard output log, relative to the run's scratch_dir, or None if it doesn't exist.

Return type
Optional[str]

get_messages_path()

Return the path to the bus message log, relative to the run's scratch_dir, or None if it doesn't exist.

Return type
Optional[str]

get_task_logs(*filter_function=None*)

Return all the task log objects for the individual tasks in the workflow.

Task names will be the job_type values from issued/completed/failed messages, with annotations from JobAnnotationMessage messages if available.

Parameters

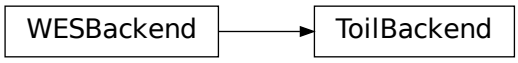
filter_function (*Optional[Callable[[toil.server.wes.abstract_backend.TaskLog, toil.bus.JobStatus], Optional[toil.server.wes.abstract_backend.TaskLog]]]*) – If set, will be called with each task log and its job annotations. Returns a modified copy of the task log to actually report, or None if the task log should be omitted.

Return type

List[Dict[str, Union[str, int, None]]]

class toil.server.wes.toil_backend.**ToilBackend**(*work_dir*, *state_store*, *options*, *dest_bucket_base*, *bypass_celery=False*, *wes_dialect='standard'*)

Bases: *toil.server.wes.abstract_backend.WESBackend*



WES backend implemented for Toil to run CWL, WDL, or Toil workflows. This class is responsible for validating and executing submitted workflows.

Parameters

- **work_dir** (*str*) –
- **state_store** (*Optional[str]*) –
- **options** (*List[str]*) –
- **dest_bucket_base** (*Optional[str]*) –
- **bypass_celery** (*bool*) –
- **wes_dialect** (*str*) –

get_runs()

A generator of a list of run ids and their state.

Return type

Generator[Tuple[str, str], None, None]

get_state(*run_id*)

Return the state of the workflow run with the given run ID. May raise an error if the workflow does not exist.

Parameters

run_id (*str*) –

Return type

str

`get_service_info()`

Get information about the Workflow Execution Service.

Return type
Dict[str, Any]

`list_runs(page_size=None, page_token=None)`

List the workflow runs.

Parameters

- **page_size** (*Optional[int]*) –
- **page_token** (*Optional[str]*) –

Return type
Dict[str, Any]

`run_workflow()`

Run a workflow.

Return type
Dict[str, str]

`get_run_log(run_id)`

Get detailed info about a workflow run.

Parameters
run_id (*str*) –

Return type
Dict[str, Any]

`cancel_run(run_id)`

Cancel a running workflow.

Parameters
run_id (*str*) –

Return type
Dict[str, str]

`get_run_status(run_id)`

Get quick status info about a workflow run, returning a simple result with the overall state of the workflow run.

Parameters
run_id (*str*) –

Return type
Dict[str, str]

`get_stdout(run_id)`

Get the stdout of a workflow run as a static file.

Parameters
run_id (*str*) –

Return type
Any

get_stderr(*run_id*)

Get the stderr of a workflow run as a static file.

Parameters
run_id (*str*) –

Return type
Any

get_health()

Return successfully if the server is healthy.

Return type
werkzeug.wrappers.response.Response

get_homepage()

Provide a sensible result for / other than 404.

Return type
werkzeug.wrappers.response.Response

Submodules

toil.server.app

Module Contents

Functions

parser_with_server_options()	
create_app(args)	Create a "connexion.FlaskApp" instance with Toil server configurations.
start_server(args)	Start a Toil server.

Attributes

logger	

toil.server.app.**logger**

toil.server.app.**parser_with_server_options**()

Return type
argparse.ArgumentParser

toil.server.app.**create_app**(*args*)

Create a “connexion.FlaskApp” instance with Toil server configurations.

Parameters
args (*argparse.Namespace*) –

Return type
connexion.FlaskApp

toil.server.app.**start_server**(*args*)
Start a Toil server.

Parameters
args (*argparse.Namespace*) –

Return type
None

toil.server.celery_app

Module Contents

Functions

create_celery_app()	

Attributes

celery	

toil.server.celery_app.**create_celery_app**()

Return type
celery.Celery

toil.server.celery_app.**celery**

toil.server.utils

Module Contents

Classes

MemoryStateCache	An in-memory place to store workflow state.
AbstractStateStore	A place for the WES server to keep its state: the set of workflows that
MemoryStateStore	An in-memory place to store workflow state, for testing.
FileStateStore	A place to store workflow state that uses a POSIX-compatible file system.
S3StateStore	A place to store workflow state that uses an S3-compatible object store.
WorkflowStateStore	Slice of a state store for the state of a particular workflow.
WorkflowStateMachine	Class for managing the WES workflow state machine.

Functions

get_iso_time()	Return the current time in ISO 8601 format.
link_file(src, dest)	Create a link to a file from src to dest.
download_file_from_internet(src, dest[, content_type])	Download a file from the Internet and write it to dest.
download_file_from_s3(src, dest[, content_type])	Download a file from Amazon S3 and write it to dest.
get_file_class(path)	Return the type of the file as a human readable string.
safe_read_file(file)	Safely read a file by acquiring a shared lock to prevent other processes
safe_write_file(file, s)	Safely write to a file by acquiring an exclusive lock to prevent other
connect_to_state_store(url)	Connect to a place to store state for workflows, defined by a URL.
connect_to_workflow_state_store(url, workflow_id)	Connect to a place to store state for the given workflow, in the state

Attributes

HAVE_S3
logger
state_store_cache
TERMINAL_STATES
MAX_CANCELING_SECONDS

toil.server.utils.**HAVE_S3 = True**

toil.server.utils.**logger**

toil.server.utils.**get_iso_time**()

Return the current time in ISO 8601 format.

Return type
str

toil.server.utils.**link_file**(*src*, *dest*)

Create a link to a file from src to dest.

Parameters

- **src** (*str*) –
- **dest** (*str*) –

Return type
None

toil.server.utils.**download_file_from_internet**(*src*, *dest*, *content_type=None*)

Download a file from the Internet and write it to dest.

Parameters

- **src** (*str*) –
- **dest** (*str*) –
- **content_type** (*Optional[str]*) –

Return type
None

toil.server.utils.**download_file_from_s3**(*src*, *dest*, *content_type=None*)

Download a file from Amazon S3 and write it to dest.

Parameters

- **src** (*str*) –
- **dest** (*str*) –
- **content_type** (*Optional[str]*) –

Return type
None

toil.server.utils.**get_file_class**(*path*)

Return the type of the file as a human readable string.

Parameters
path (*str*) –

Return type
str

toil.server.utils.**safe_read_file**(*file*)

Safely read a file by acquiring a shared lock to prevent other processes from writing to it while reading.

Parameters
file (*str*) –

Return type
Optional[str]

toil.server.utils.**safe_write_file**(*file*, *s*)

Safely write to a file by acquiring an exclusive lock to prevent other processes from reading and writing to it while writing.

Parameters

- **file** (*str*) –
- **s** (*str*) –

Return type
None

class toil.server.utils.**MemoryStateCache**

An in-memory place to store workflow state.

get(*workflow_id*, *key*)

Get a key value from memory.

Parameters

- **workflow_id** (*str*) –
- **key** (*str*) –

Return type
Optional[str]

set(*workflow_id*, *key*, *value*)

Set or clear a key value in memory.

Parameters

- **workflow_id** (*str*) –
- **key** (*str*) –
- **value** (*Optional[str]*) –

Return type
None

class toil.server.utils.**AbstractStateStore**

A place for the WES server to keep its state: the set of workflows that exist and whether they are done or not.

This is a key-value store, with keys namespaced by workflow ID. Concurrent access from multiple threads or processes is safe and globally consistent.

Keys and workflow IDs are restricted to [-a-zA-Z0-9_], because backends may use them as path or URL components.

Key values are either a string, or None if the key is not set.

Workflow existence isn't a thing; nonexistent workflows just have None for all keys.

Note that we don't yet have a cleanup operation: things are stored permanently. Even clearing all the keys may leave data behind.

Also handles storage for a local cache, with a separate key namespace (not a read/write-through cache).

TODO: Can we replace this with just using a JobStore eventually, when AWSJobStore no longer needs SimpleDB?

abstract get(*workflow_id*, *key*)

Get the value of the given key for the given workflow, or None if the key is not set for the workflow.

Parameters

- **workflow_id** (*str*) –
- **key** (*str*) –

Return type
Optional[str]

abstract set(*workflow_id*, *key*, *value*)

Set the value of the given key for the given workflow. If the value is None, clear the key.

Parameters

- **workflow_id** (*str*) –

- **key** (*str*) –
- **value** (*Optional[str]*) –

Return type
None

read_cache(*workflow_id*, *key*)

Read a value from a local cache, without checking the actual backend.

Parameters

- **workflow_id** (*str*) –
- **key** (*str*) –

Return type
Optional[str]

write_cache(*workflow_id*, *key*, *value*)

Write a value to a local cache, without modifying the actual backend.

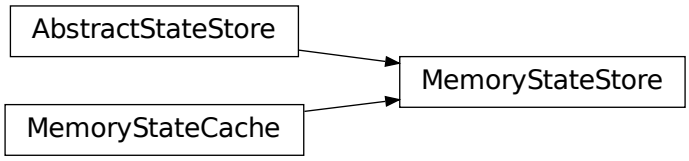
Parameters

- **workflow_id** (*str*) –
- **key** (*str*) –
- **value** (*Optional[str]*) –

Return type
None

class toil.server.utils.**MemoryStateStore**

Bases: *MemoryStateCache*, *AbstractStateStore*

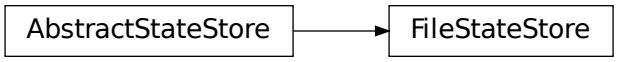


An in-memory place to store workflow state, for testing.

Inherits from MemoryStateCache first to provide implementations for AbstractStateStore.

class toil.server.utils.**FileStateStore**(*url*)

Bases: *AbstractStateStore*



A place to store workflow state that uses a POSIX-compatible file system.

Parameters
url (*str*) –

get(*workflow_id*, *key*)

Get a key value from the filesystem.

Parameters

- **workflow_id** (*str*) –
- **key** (*str*) –

Return type
Optional[str]

set(*workflow_id*, *key*, *value*)

Set or clear a key value on the filesystem.

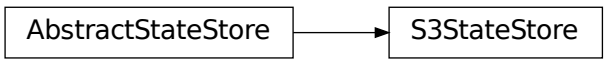
Parameters

- **workflow_id** (*str*) –
- **key** (*str*) –
- **value** (*Optional[str]*) –

Return type
None

class toil.server.utils.**S3StateStore**(*url*)

Bases: *AbstractStateStore*



A place to store workflow state that uses an S3-compatible object store.

Parameters
url (*str*) –

get(*workflow_id*, *key*)

Get a key value from S3.

Parameters

- **workflow_id** (*str*) –
- **key** (*str*) –

Return type
Optional[str]

set(*workflow_id*, *key*, *value*)

Set or clear a key value on S3.

Parameters

- **workflow_id** (*str*) –
- **key** (*str*) –
- **value** (*Optional[str]*) –

Return type
None

toil.server.utils.**state_store_cache: Dict[str, *AbstractStateStore*]**

toil.server.utils.**connect_to_state_store**(*url*)

Connect to a place to store state for workflows, defined by a URL.

URL may be a local file path or URL or an S3 URL.

Parameters
url (*str*) –

Return type
AbstractStateStore

class toil.server.utils.**WorkflowStateStore**(*state_store*, *workflow_id*)

Slice of a state store for the state of a particular workflow.

Parameters

- **state_store** (AbstractStateStore) –
- **workflow_id** (*str*) –

get(*key*)

Get the given item of workflow state.

Parameters
key (*str*) –

Return type
Optional[str]

set(*key*, *value*)

Set the given item of workflow state.

Parameters

- **key** (*str*) –
- **value** (*Optional[str]*) –

Return type
None

read_cache(*key*)

Read a value from a local cache, without checking the actual backend.

Parameters
key (*str*) –

Return type
Optional[str]

write_cache(*key*, *value*)

Write a value to a local cache, without modifying the actual backend.

Parameters

- **key** (*str*) –
- **value** (*Optional[str]*) –

Return type
None

toil.server.utils.**connect_to_workflow_state_store**(*url*, *workflow_id*)

Connect to a place to store state for the given workflow, in the state store defined by the given URL.

Parameters

- **url** (*str*) – A URL that can be used for connect_to_state_store()
- **workflow_id** (*str*) –

Return type
WorkflowStateStore

toil.server.utils.**TERMINAL_STATES**

toil.server.utils.**MAX_CANCELING_SECONDS = 30**

class toil.server.utils.**WorkflowStateMachine**(*store*)

Class for managing the WES workflow state machine.

This is the authority on the WES "state" of a workflow. You need one to read or change the state.

Guaranteeing that only certain transitions can be observed is possible but not worth it. Instead, we just let updates clobber each other and grab and cache the first terminal state we see forever. If it becomes important that clients never see e.g. CANCELED -> COMPLETE or COMPLETE -> SYSTEM_ERROR, we can implement a real distributed state machine here.

We do handle making sure that tasks don't get stuck in CANCELING.

State can be:

"UNKNOWN" "QUEUED" "INITIALIZING" "RUNNING" "PAUSED" "COMPLETE" "EXECUTOR_ERROR" "SYSTEM_ERROR" "CANCELED" "CANCELING"

Uses the state store's local cache to prevent needing to read things we've seen already.

Parameters
store (WorkflowStateStore) –

send_enqueue()

Send an enqueue message that would move from UNKNOWN to QUEUED.

Return type
None

`send_initialize()`

Send an initialize message that would move from QUEUED to INITIALIZING.

Return type
None

`send_run()`

Send a run message that would move from INITIALIZING to RUNNING.

Return type
None

`send_cancel()`

Send a cancel message that would move to CANCELING from any non-terminal state.

Return type
None

`send_canceled()`

Send a canceled message that would move to CANCELED from CANCELLING.

Return type
None

`send_complete()`

Send a complete message that would move from RUNNING to COMPLETE.

Return type
None

`send_executor_error()`

Send an executor_error message that would move from QUEUED, INITIALIZING, or RUNNING to EXECUTOR_ERROR.

Return type
None

`send_system_error()`

Send a system_error message that would move from QUEUED, INITIALIZING,
or RUNNING to SYSTEM_ERROR.

Return type
None

`get_current_state()`

Get the current state of the workflow.

Return type
str

toil.server.wsgi_app

Module Contents

Classes

GunicornApplication	An entry point to integrate a Gunicorn WSGI server in Python. To start a

Functions

run_app(app[, options])	Run a Gunicorn WSGI server.

class toil.server.wsgi_app.**GunicornApplication**(*app*, *options=None*)

Bases: gunicorn.app.base.BaseApplication

An entry point to integrate a Gunicorn WSGI server in Python. To start a WSGI application with callable *app*, run the following code:

WSGIApplication(app, options={

…

}).run()

For more details, see: https://docs.gunicorn.org/en/latest/custom.html

Parameters

- **app** (*object*) –
- **options** (*Optional[Dict[str, Any]]*) –

init(**args*)

Parameters
args (*Any*) –

Return type
None

load_config()

Return type
None

load()

Return type
object

toil.server.wsgi_app.**run_app**(*app*, *options=None*)

Run a Gunicorn WSGI server.

Parameters

- **app** (*object*) –
- **options** (*Optional[Dict[str, Any]]*) –

Return type
None

toil.test

Base testing class for Toil.

Subpackages

toil.test.batchSystems

Submodules

toil.test.batchSystems.batchSystemTest

Module Contents

Classes

BatchSystemPluginTest	Class for testing batch system plugin functionality.
hidden	Hide abstract base class from unittest's test case loader
KubernetesBatchSystemTest	Tests against the Kubernetes batch system
KubernetesBatchSystemBenchTest	Kubernetes batch system unit tests that don't need to actually talk to a cluster.
TESBatchSystemTest	Tests against the TES batch system
AWSBatchBatchSystemTest	Tests against the AWS Batch batch system
MesosBatchSystemTest	Tests against the Mesos batch system
SingleMachineBatchSystemTest	Tests against the single-machine batch system
MaxCoresSingleMachineBatchSystemTest	This test ensures that single machine batch system doesn't exceed the configured number
Service	Abstract class used to define the interface to a service.
ParasolBatchSystemTest	Tests the Parasol batch system
GridEngineBatchSystemTest	Tests against the GridEngine batch system
SlurmBatchSystemTest	Tests against the Slurm batch system
LSFBatchSystemTest	Tests against the LSF batch system
TorqueBatchSystemTest	Tests against the Torque batch system
HTCondorBatchSystemTest	Tests against the HTCondor batch system
SingleMachineBatchSystemJobTest	Tests Toil workflow against the SingleMachine batch system
MesosBatchSystemJobTest	Tests Toil workflow against the Mesos batch system

Functions

write_temp_file(s, temp_dir)	Dump a string into a temp file and return its path.
parentJob(job, cmd)	
childJob(job, cmd)	
grandChildJob(job, cmd)	
greatGrandChild(cmd)	
measureConcurrency(filepath[, sleep_time])	Run in parallel to determine the number of concurrent tasks.
count(delta, file_path)	Increments counter file and returns the max number of times the file
getCounters(path)	
resetCounters(path)	
get_omp_threads()	

Attributes

logger
numCores
preemptible
defaultRequirements

toil.test.batchSystems.batchSystemTest.**logger**

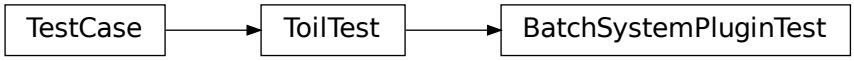
toil.test.batchSystems.batchSystemTest.**numCores = 2**

toil.test.batchSystems.batchSystemTest.**preemptible = False**

toil.test.batchSystems.batchSystemTest.**defaultRequirements**

class toil.test.batchSystems.batchSystemTest.**BatchSystemPluginTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Class for testing batch system plugin functionality.

setUp()

Hook method for setting up the test fixture before exercising it.

tearDown()

Hook method for deconstructing the test fixture after testing it.

testAddBatchSystemFactory()

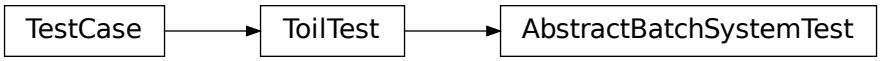
class toil.test.batchSystems.batchSystemTest.**hidden**

Hide abstract base class from unittest's test case loader

http://stackoverflow.com/questions/1323455/python-unit-test-with-base-and-sub-class#answer-25695512

class AbstractBatchSystemTest(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



A base test case with generic tests that every batch system should pass.

Cannot assume that the batch system actually executes commands on the local machine/filesystem.

abstract createBatchSystem()

Return type
toil.batchSystems.abstractBatchSystem.AbstractBatchSystem

supportsWallTime()

classmethod createConfig()

Returns a dummy config for the batch system tests. We need a workflowID to be set up since we are running tests without setting up a jobstore. This is the class version to be used when an instance is not available.

Return type
toil.common.Config

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

setUp()

Hook method for setting up the test fixture before exercising it.

tearDown()

Hook method for deconstructing the test fixture after testing it.

get_max_startup_seconds()

Get the number of seconds this test ought to wait for the first job to run. Some batch systems may need time to scale up.

Return type
int

test_available_cores()

test_run_jobs()

test_set_env()

test_set_job_env()

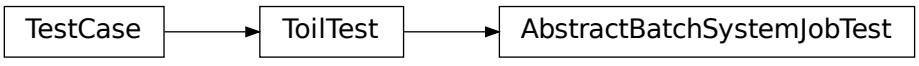
Test the mechanism for setting per-job environment variables to batch system jobs.

testCheckResourceRequest()

testScalableBatchSystem()

class AbstractBatchSystemJobTest(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



An abstract base class for batch system tests that use a full Toil workflow rather than using the batch system directly.

cpuCount

allocatedCores

sleepTime = 5

abstract getBatchSystemName()

Return type
(str, *AbstractBatchSystem*)

getOptions(*tempDir*)

Configures options for Toil workflow and makes job store. :param str tempDir: path to test directory :return: Toil options object

setUp()

Hook method for setting up the test fixture before exercising it.

tearDown()

Hook method for deconstructing the test fixture after testing it.

testJobConcurrency()

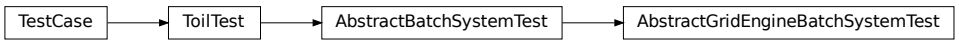
Tests that the batch system is allocating core resources properly for concurrent tasks.

test_omp_threads()

Test if the OMP_NUM_THREADS env var is set correctly based on jobs.cores.

class AbstractGridEngineBatchSystemTest(*methodName='runTest'*)

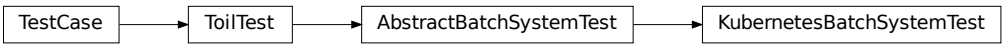
Bases: *hidden.AbstractBatchSystemTest*



An abstract class to reduce redundancy between Grid Engine, Slurm, and other similar batch systems

class toil.test.batchSystems.batchSystemTest.**KubernetesBatchSystemTest**(*methodName='runTest'*)

Bases: *hidden*



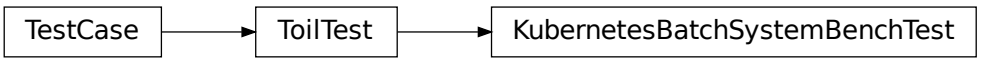
Tests against the Kubernetes batch system

supportsWallTime()

createBatchSystem()

class toil.test.batchSystems.batchSystemTest.**KubernetesBatchSystemBenchTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Kubernetes batch system unit tests that don't need to actually talk to a cluster.

test_preemptability_constraints()

Make sure we generate the right preemptability constraints.

test_label_constraints()

Make sure we generate the right preemptability constraints.

class toil.test.batchSystems.batchSystemTest.**TESBatchSystemTest**(*methodName='runTest'*)

Bases: *hidden*

Tests against the TES batch system

supportsWallTime()

createBatchSystem()

class toil.test.batchSystems.batchSystemTest.**AWSBatchBatchSystemTest**(*methodName='runTest'*)

Bases: *hidden*

Tests against the AWS Batch batch system

supportsWallTime()

createBatchSystem()

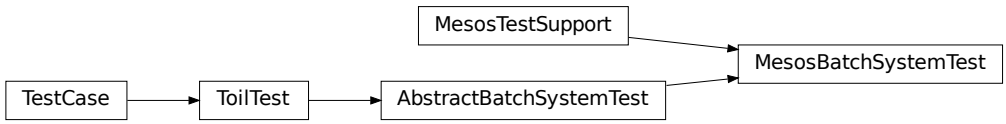
get_max_startup_seconds()

Get the number of seconds this test ought to wait for the first job to run. Some batch systems may need time to scale up.

Return type
int

class toil.test.batchSystems.batchSystemTest.**MesosBatchSystemTest**(*methodName='runTest'*)

Bases: *hidden*, *toil.batchSystems.mesos.test.MesosTestSupport*



Tests against the Mesos batch system

classmethod createConfig()

needs to set mesos_endpoint to localhost for testing since the default is now the private IP address

supportsWallTime()

createBatchSystem()

tearDown()

Hook method for deconstructing the test fixture after testing it.

testIgnoreNode()

toil.test.batchSystems.batchSystemTest.**write_temp_file**(*s*, *temp_dir*)

Dump a string into a temp file and return its path.

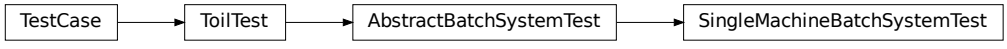
Parameters

- **s** (*str*) –
- **temp_dir** (*str*) –

Return type
str

class toil.test.batchSystems.batchSystemTest.**SingleMachineBatchSystemTest**(*methodName='runTest'*)

Bases: *hidden*



Tests against the single-machine batch system

supportsWallTime()

Return type
bool

createBatchSystem()

Return type
toil.batchSystems.abstractBatchSystem.AbstractBatchSystem

testProcessEscape(*hide=False*)

Test to make sure that child processes and their descendants go away when the Toil workflow stops.

If hide is true, will try and hide the child processes to make them hard to stop.

Parameters
hide (*bool*) –

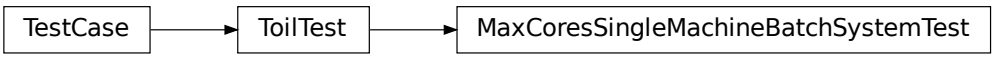
Return type
None

testHidingProcessEscape()

Test to make sure that child processes and their descendants go away when the Toil workflow stops, even if the job process stops and leaves children.

class toil.test.batchSystems.batchSystemTest.**MaxCoresSingleMachineBatchSystemTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



This test ensures that single machine batch system doesn't exceed the configured number cores

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

Return type
None

setUp()

Hook method for setting up the test fixture before exercising it.

Return type
None

tearDown()

Hook method for deconstructing the test fixture after testing it.

Return type
None

scriptCommand()

Return type
str

test()

testServices()

toil.test.batchSystems.batchSystemTest.**parentJob**(*job*, *cmd*)

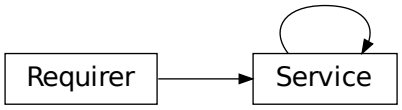
toil.test.batchSystems.batchSystemTest.**childJob**(*job*, *cmd*)

toil.test.batchSystems.batchSystemTest.**grandChildJob**(*job*, *cmd*)

toil.test.batchSystems.batchSystemTest.**greatGrandChild**(*cmd*)

class toil.test.batchSystems.batchSystemTest.**Service**(*cmd*)

Bases: *toil.job.Job.Service*



Abstract class used to define the interface to a service.

Should be subclassed by the user to define services.

Is not executed as a job; runs within a ServiceHostJob.

start(*fileStore*)

Start the service.

Parameters

job – The underlying host job that the service is being run in. Can be used to register deferred functions, or to access the fileStore for creating temporary files.

Returns

An object describing how to access the service. The object must be pickleable and will be used by jobs to access the service (see *toil.job.Job.addService()*).

check()

Checks the service is still running.

Raises

exceptions.RuntimeError – If the service failed, this will cause the service job to be labeled failed.

Returns

True if the service is still running, else False. If False then the service job will be terminated, and considered a success. Important point: if the service job exits due to a failure, it should raise a RuntimeError, not return False!

stop(*fileStore*)

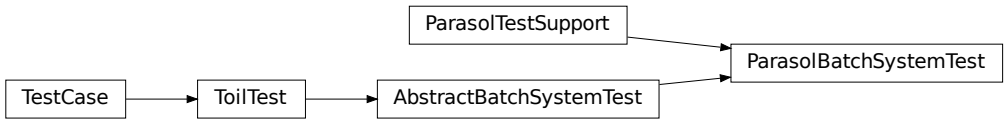
Stops the service. Function can block until complete.

Parameters

job – The underlying host job that the service is being run in. Can be used to register deferred functions, or to access the fileStore for creating temporary files.

class toil.test.batchSystems.batchSystemTest.**ParasolBatchSystemTest**(*methodName='runTest'*)

Bases: *hidden*, *toil.test.batchSystems.parasolTestSupport.ParasolTestSupport*



Tests the Parasol batch system

supportsWallTime()

createBatchSystem()

Return type
toil.batchSystems.abstractBatchSystem.AbstractBatchSystem

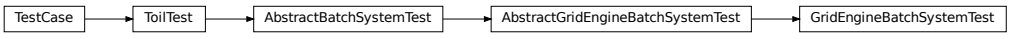
tearDown()

Hook method for deconstructing the test fixture after testing it.

testBatchResourceLimits()

class toil.test.batchSystems.batchSystemTest.**GridEngineBatchSystemTest**(*methodName='runTest'*)

Bases: *hidden*



Tests against the GridEngine batch system

createBatchSystem()

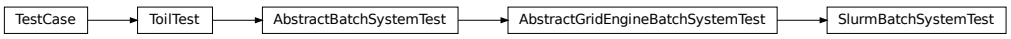
Return type
toil.batchSystems.abstractBatchSystem.AbstractBatchSystem

tearDown()

Hook method for deconstructing the test fixture after testing it.

class toil.test.batchSystems.batchSystemTest.**SlurmBatchSystemTest**(*methodName='runTest'*)

Bases: *hidden*



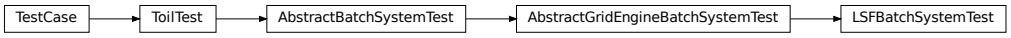
Tests against the Slurm batch system

createBatchSystem()

Return type
toil.batchSystems.abstractBatchSystem.AbstractBatchSystem

tearDown()
Hook method for deconstructing the test fixture after testing it.

class toil.test.batchSystems.batchSystemTest.**LSFBatchSystemTest**(*methodName='runTest'*)
Bases: *hidden*

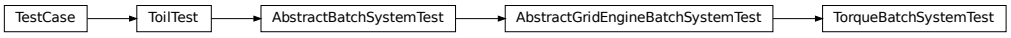


Tests against the LSF batch system

createBatchSystem()

Return type
toil.batchSystems.abstractBatchSystem.AbstractBatchSystem

class toil.test.batchSystems.batchSystemTest.**TorqueBatchSystemTest**(*methodName='runTest'*)
Bases: *hidden*



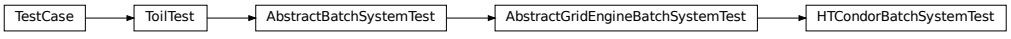
Tests against the Torque batch system

createBatchSystem()

Return type
toil.batchSystems.abstractBatchSystem.AbstractBatchSystem

tearDown()
Hook method for deconstructing the test fixture after testing it.

class toil.test.batchSystems.batchSystemTest.**HTCondorBatchSystemTest**(*methodName='runTest'*)
Bases: *hidden*



Tests against the HTCondor batch system

createBatchSystem()

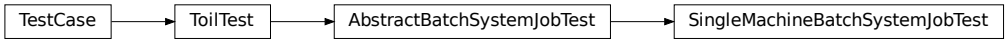
Return type
toil.batchSystems.abstractBatchSystem.AbstractBatchSystem

tearDown()

Hook method for deconstructing the test fixture after testing it.

class toil.test.batchSystems.batchSystemTest.**SingleMachineBatchSystemJobTest**(*methodName='runTest'*)

Bases: *hidden*



Tests Toil workflow against the SingleMachine batch system

getBatchSystemName()

Return type
(str, *AbstractBatchSystem*)

testConcurrencyWithDisk()

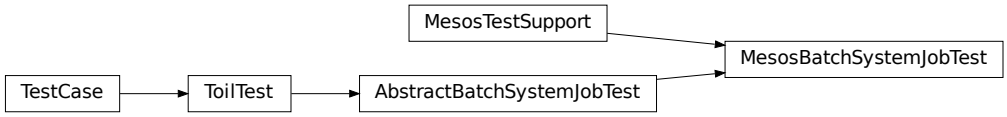
Tests that the batch system is allocating disk resources properly

testNestedResourcesDoNotBlock()

Resources are requested in the order Memory > Cpu > Disk. Test that unavailability of cpus for one job that is scheduled does not block another job that can run.

class toil.test.batchSystems.batchSystemTest.**MesosBatchSystemJobTest**(*methodName='runTest'*)

Bases: *hidden*, *toil.batchSystems.mesos.test.MesosTestSupport*



Tests Toil workflow against the Mesos batch system

getOptions(*tempDir*)

Configures options for Toil workflow and makes job store. :param str tempDir: path to test directory :return: Toil options object

getBatchSystemName()

Return type
(str, *AbstractBatchSystem*)

tearDown()

Hook method for deconstructing the test fixture after testing it.

toil.test.batchSystems.batchSystemTest.**measureConcurrency**(*filepath*, *sleep_time=10*)

Run in parallel to determine the number of concurrent tasks. This code was copied from toil.batchSystemTestMaxCoresSingleMachineBatchSystemTest :param str filepath: path to counter file :param int sleep_time: number of seconds to sleep before counting down :return int max concurrency value:

toil.test.batchSystems.batchSystemTest.**count**(*delta*, *file_path*)

Increments counter file and returns the max number of times the file has been modified. Counter data must be in the form: concurrent tasks, max concurrent tasks (counter should be initialized to 0,0)

Parameters

- **delta** (*int*) – increment value
- **file_path** (*str*) – path to shared counter file

Return int max concurrent tasks

toil.test.batchSystems.batchSystemTest.**getCounters**(*path*)

toil.test.batchSystems.batchSystemTest.**resetCounters**(*path*)

toil.test.batchSystems.batchSystemTest.**get_omp_threads**()

Return type
str

toil.test.batchSystems.parasolTestSupport

Module Contents

Classes

ParasolTestSupport	For test cases that need a running Parasol leader and worker on the local host

Attributes

log

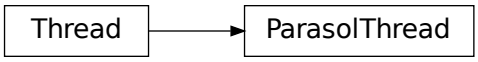
toil.test.batchSystems.parasolTestSupport.**log**

class toil.test.batchSystems.parasolTestSupport.**ParasolTestSupport**

For test cases that need a running Parasol leader and worker on the local host

class ParasolThread

Bases: threading.Thread



A class that represents a thread of control.

This class can be safely subclassed in a limited fashion. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the run() method in a subclass.

lock

abstract parasolCommand()

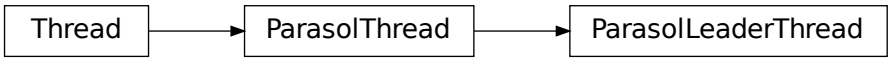
run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class ParasolLeaderThread

Bases: *ParasolTestSupport.ParasolThread*



A class that represents a thread of control.

This class can be safely subclassed in a limited fashion. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the run() method in a subclass.

run()

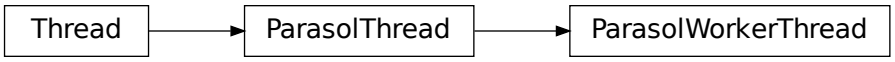
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

parasolCommand()

class ParasolWorkerThread

Bases: *ParasolTestSupport.ParasolThread*



A class that represents a thread of control.

This class can be safely subclassed in a limited fashion. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the run() method in a subclass.

parasolCommand()

toil.test.batchSystems.test_lsf_helper

lsfHelper.py shouldn't need a batch system and so the unit tests here should aim to run on any system.

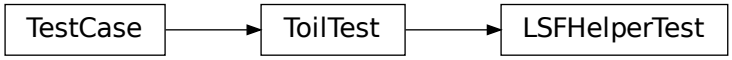
Module Contents

Classes

LSFHelperTest	A common base class for Toil tests.

class toil.test.batchSystems.test_lsf_helper.**LSFHelperTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

test_parse_mem_and_cmd_from_output()

toil.test.batchSystems.test_slurm

Module Contents

Classes

FakeBatchSystem	Class that implements a minimal Batch System, needed to create a Worker (see below).
SlurmTest	Class for unit-testing SlurmBatchSystem

Functions

call_sacct(args, **_)	The arguments passed to *call_command* when executing *sacct* are:
call_scontrol(args, **_)	The arguments passed to *call_command* when executing *scontrol* are:
call_sacct_raises(*_)	Fake that the *sacct* command fails by raising a *CalledProcessErrorStderr*

toil.test.batchSystems.test_slurm.**call_sacct**(*args*, **_)

The arguments passed to *call_command* when executing *sacct* are: [‘sacct’, ‘-n’, ‘-j’, ‘<comma-separated list of job-ids>’, ‘–format’, ‘JobIDRaw,State,ExitCode’, ‘-P’, ‘-S’, ‘1970-01-01’] The multi-line output is something like:

1234|COMPLETED|0:0 1234.batch|COMPLETED|0:0 1235|PENDING|0:0 1236|FAILED|0:2 1236.extern|COMPLETED|0:0

Return type
str

toil.test.batchSystems.test_slurm.**call_scontrol**(*args*, **_)

The arguments passed to *call_command* when executing *scontrol* are: [‘scontrol’, ‘show’, ‘job’] or [‘scontrol’, ‘show’, ‘job’, ‘<job-id>’]

Return type
str

toil.test.batchSystems.test_slurm.**call_sacct_raises**(*_)

Fake that the *sacct* command fails by raising a *CalledProcessErrorStderr*

class toil.test.batchSystems.test_slurm.**FakeBatchSystem**

Class that implements a minimal Batch System, needed to create a Worker (see below).

getWaitDuration()

class toil.test.batchSystems.test_slurm.**SlurmTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

Class for unit-testing SlurmBatchSystem

setUp()

Hook method for setting up the test fixture before exercising it.

test_getJobDetailsFromSacct_one_exists()

test_getJobDetailsFromSacct_one_not_exists()

test_getJobDetailsFromSacct_many_all_exist()

test_getJobDetailsFromSacct_many_some_exist()

test_getJobDetailsFromSacct_many_none_exist()

test_getJobDetailsFromScontrol_one_exists()

test_getJobDetailsFromScontrol_one_not_exists()

Asking for the job details of a single job that *scontrol* doesn't know about should raise an exception.

test_getJobDetailsFromScontrol_many_all_exist()

test_getJobDetailsFromScontrol_many_some_exist()

test_getJobDetailsFromScontrol_many_none_exist()

test_getJobExitCode_job_exists()

test_getJobExitCode_job_not_exists()

test_getJobExitCode_sacct_raises_job_exists()

This test forces the use of *scontrol* to get job information, by letting *sacct* raise an exception.

test_getJobExitCode_sacct_raises_job_not_exists()

This test forces the use of *scontrol* to get job information, by letting *sacct* raise an exception. Next, *scontrol* should also raise because it doesn't know the job.

test_coalesce_job_exit_codes_one_exists()

test_coalesce_job_exit_codes_one_not_exists()

test_coalesce_job_exit_codes_many_all_exist()

test_coalesce_job_exit_codes_some_exists()

test_coalesce_job_exit_codes_sacct_raises_job_exists()

This test forces the use of *scontrol* to get job information, by letting *sacct* raise an exception.

test_coalesce_job_exit_codes_sacct_raises_job_not_exists()

This test forces the use of *scontrol* to get job information, by letting *sacct* raise an exception. Next, *scontrol* should also raise because it doesn't know the job.

toil.test.cwl

Submodules

toil.test.cwl.conftest

Module Contents

toil.test.cwl.conftest.**collect_ignore** = ['spec']

toil.test.cwl.cwlTest

Module Contents

Classes

CWLWorkflowTest	CWL tests included in Toil that don't involve the whole CWL conformance
CWLv10Test	Run the CWL 1.0 conformance tests in various environments.
CWLv11Test	Run the CWL 1.1 conformance tests in various environments.
CWLv12Test	Run the CWL 1.2 conformance tests in various environments.
CWLOnARMTest	Run the CWL 1.2 conformance tests on ARM specifically.

Functions

run_conformance_tests(workDir, yml[, runner, caching, ...])	Run the CWL conformance tests.
test_workflow_echo_string_scatter_stderr_log_dir(tmp_path)	
test_log_dir_echo_no_output(tmp_path)	
test_log_dir_echo_stderr(tmp_path)	
test_filename_conflict_resolution(tmp_path)	
test_filename_conflict_detection(tmp_path)	Make sure we don't just stage files over each other when using a container.
test_filename_conflict_detection_at_root(tmp_path)	Make sure we don't just stage files over each other.
test_pick_value_with_one_null_value(caplog)	Make sure toil-cwl-runner does not false log a warning when pickValue is
test_usage_message()	This is purely to ensure a (more) helpful error message is printed if a user does
test_workflow_echo_string()	
test_workflow_echo_string_scatter_capture_stdout()	
test_visit_top_cwl_class()	
test_visit_cwl_class_and_reduce()	
test_download_structure(tmp_path)	Make sure that download_structure makes the right calls to what it thinks is the file store.

Attributes

pkg_root	
log	
CONFORMANCE_TEST_TIMEOUT	

toil.test.cwl.cwlTest.**pkg_root**

toil.test.cwl.cwlTest.**log**

toil.test.cwl.cwlTest.**CONFORMANCE_TEST_TIMEOUT = 3600**

toil.test.cwl.cwlTest.**run_conformance_tests**(*workDir*, *yml*, *runner=None*, *caching=False*, *batchSystem=None*, *selected_tests=None*, *selected_tags=None*, *skipped_tests=None*, *extra_args=None*, *must_support_all_features=False*, *junit_file=None*)

Run the CWL conformance tests.

Parameters

- **workDir** (*str*) – Directory to run tests in.
- **yml** (*str*) – CWL test list YML to run tests from.
- **runner** (*Optional[str]*) – If set, use this cwl runner instead of the default toil-cwl-runner.
- **caching** (*bool*) – If True, use Toil file store caching.
- **batchSystem** (*str*) – If set, use this batch system instead of the default single_machine.
- **selected_tests** (*str*) – If set, use this description of test numbers to run (comma-separated numbers or ranges)
- **selected_tags** (*str*) – As an alternative to selected_tests, run tests with the given tags.
- **skipped_tests** (*str*) – Comma-separated string labels of tests to skip.
- **extra_args** (*Optional[List[str]]*) – Provide these extra arguments to runner for each test.
- **must_support_all_features** (*bool*) – If set, fail if some CWL optional features are unsupported.
- **junit_file** (*Optional[str]*) – JUnit XML file to write test info to.

class toil.test.cwl.cwlTest.**CWLWorkflowTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

CWL tests included in Toil that don't involve the whole CWL conformance test suite. Tests Toil-specific functions like URL types supported for inputs.

setUp()

Runs anew before each test to create farm fresh temp dirs.

tearDown()

Clean up outputs.

revsort(*cwl_filename*, *tester_fn*)

revsort_no_checksum(*cwl_filename*, *tester_fn*)

download(*inputs*, *tester_fn*)

load_contents(*inputs*, *tester_fn*)

download_directory(*inputs*, *tester_fn*)

download_subdirectory(*inputs*, *tester_fn*)

test_mpi()

test_s3_as_secondary_file()

test_run_revsort()

test_run_revsort_nochecksum()

test_run_revsort2()

test_run_revsort_debug_worker()

test_run_colon_output()

test_download_s3()

test_download_http()

test_download_https()

test_download_file()

test_download_directory_s3()

test_download_directory_file()

test_download_subdirectory_s3()

test_download_subdirectory_file()

test_load_contents_s3()

test_load_contents_http()

test_load_contents_https()

test_load_contents_file()

test_bioconda()

test_biocontainers()

test_cuda()

test_restart()

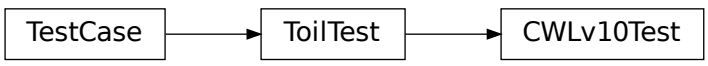
Enable restarts with toil-cwl-runner – run failing test, re-run correct test. Only implemented for single machine.

test_streamable()

Test that a file with 'streamable'=True is a named pipe. This is a CWL1.2 feature.

class toil.test.cwl.cwlTest.**CWLv10Test**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Run the CWL 1.0 conformance tests in various environments.

setUp()

Runs anew before each test to create farm fresh temp dirs.

tearDown()

Clean up outputs.

test_run_conformance_with_caching()

test_run_conformance(*batchSystem=None*, *caching=False*, *selected_tests=None*)

test_lsf_cwl_conformance(***kwargs*)

test_slurm_cwl_conformance(***kwargs*)

test_torque_cwl_conformance(***kwargs*)

test_gridengine_cwl_conformance(***kwargs*)

test_mesos_cwl_conformance(***kwargs*)

test_parasol_cwl_conformance(***kwargs*)

test_kubernetes_cwl_conformance(***kwargs*)

test_lsf_cwl_conformance_with_caching()

test_slurm_cwl_conformance_with_caching()

test_torque_cwl_conformance_with_caching()

test_gridengine_cwl_conformance_with_caching()

test_mesos_cwl_conformance_with_caching()

test_parasol_cwl_conformance_with_caching()

test_kubernetes_cwl_conformance_with_caching()

class toil.test.cwl.cwlTest.**CWLv11Test**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

Run the CWL 1.1 conformance tests in various environments.

classmethod setUpClass()

Runs anew before each test.

tearDown()

Clean up outputs.

test_run_conformance(***kwargs*)

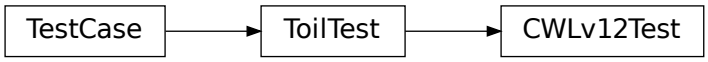
test_run_conformance_with_caching()

test_kubernetes_cwl_conformance(***kwargs*)

test_kubernetes_cwl_conformance_with_caching()

class toil.test.cwl.cwlTest.**CWLv12Test**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Run the CWL 1.2 conformance tests in various environments.

classmethod setUpClass()

Runs anew before each test.

tearDown()

Clean up outputs.

test_run_conformance(***kwargs*)

test_run_conformance_with_caching()

test_run_conformance_with_in_place_update()

Make sure that with –bypass-file-store we properly support in place update on a single node, and that this doesn't break any other features.

test_kubernetes_cwl_conformance(***kwargs*)

test_kubernetes_cwl_conformance_with_caching()

test_wes_server_cwl_conformance()

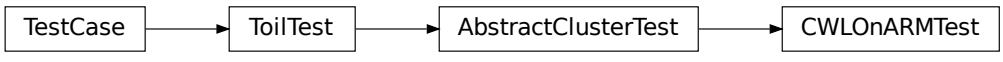
Run the CWL conformance tests via WES. TOIL_WES_ENDPOINT must be specified. If the WES server requires authentication, set TOIL_WES_USER and TOIL_WES_PASSWORD.

To run manually:

TOIL_WES_ENDPOINT=http://localhost:8080 TOIL_WES_USER=test TOIL_WES_PASSWORD=password python -m pytest src/toil/test/cwl/cwlTest.py::CWLv12Test::test_wes_server_cwl_conf -vv –log-level INFO –log-cli-level INFO

class toil.test.cwl.cwlTest.**CWLOnARMTest**(*methodName*)

Bases: *toil.test.provisioners.clusterTest.AbstractClusterTest*



Run the CWL 1.2 conformance tests on ARM specifically.

setUp()

Set up for the test. Must be overridden to call this method and set self.jobStore.

test_cwl_on_arm()

toil.test.cwl.cwlTest.**test_workflow_echo_string_scatter_stderr_log_dir**(*tmp_path*)

Parameters
tmp_path (*pathlib.Path*) –

toil.test.cwl.cwlTest.**test_log_dir_echo_no_output**(*tmp_path*)

Parameters
tmp_path (*pathlib.Path*) –

Return type
None

toil.test.cwl.cwlTest.**test_log_dir_echo_stderr**(*tmp_path*)

Parameters
tmp_path (*pathlib.Path*) –

Return type
None

toil.test.cwl.cwlTest.**test_filename_conflict_resolution**(*tmp_path*)

Parameters
tmp_path (*pathlib.Path*) –

toil.test.cwl.cwlTest.**test_filename_conflict_detection**(*tmp_path*)

Make sure we don't just stage files over each other when using a container.

Parameters
tmp_path (*pathlib.Path*) –

toil.test.cwl.cwlTest.**test_filename_conflict_detection_at_root**(*tmp_path*)

Make sure we don't just stage files over each other.

Specifically, when using a container and the files are at the root of the work dir.

Parameters
tmp_path (*pathlib.Path*) –

toil.test.cwl.cwlTest.**test_pick_value_with_one_null_value**(*caplog*)

Make sure toil-cwl-runner does not false log a warning when pickValue is used but outputSource only contains one null value. See: #3991.

toil.test.cwl.cwlTest.**test_usage_message**()

This is purely to ensure a (more) helpful error message is printed if a user does not order their positional args correctly [cwl, cwl-job (json/yml/yaml), jobstore].

toil.test.cwl.cwlTest.**test_workflow_echo_string**()

toil.test.cwl.cwlTest.**test_workflow_echo_string_scatter_capture_stdout**()

toil.test.cwl.cwlTest.**test_visit_top_cwl_class**()

toil.test.cwl.cwlTest.**test_visit_cwl_class_and_reduce**()

toil.test.cwl.cwlTest.**test_download_structure**(*tmp_path*)

Make sure that download_structure makes the right calls to what it thinks is the file store.

Return type
None

toil.test.docs

Submodules

toil.test.docs.scriptsTest

Module Contents

Classes

ToilDocumentationTest	Tests for scripts in the toil tutorials.

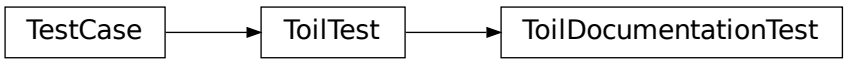
Attributes

pkg_root

toil.test.docs.scriptsTest.**pkg_root**

class toil.test.docs.scriptsTest.**ToilDocumentationTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Tests for scripts in the toil tutorials.

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

tearDown()

Hook method for deconstructing the test fixture after testing it.

Return type
None

checkExitCode(*script*)

checkExpectedOut(*script*, *expectedOutput*)

checkExpectedPattern(*script*, *expectedPattern*)

testCwlexample()

testDiscoverfiles()

testDynamic()

testEncapsulation()

testEncapsulation2()

testHelloworld()

testInvokeworkflow()

testInvokeworkflow2()

testJobFunctions()

testManaging()

testManaging2()

testMultiplejobs()

testMultiplejobs2()

testMultiplejobs3()

testPromises2()

testQuickstart()

testRequirements()

testArguments()

testDocker()

testPromises()

testServices()

testStaging()

toil.test.jobStores

Submodules

toil.test.jobStores.jobStoreTest

Module Contents

Classes

AbstractJobStoreTest	Hide abstract base class from unittest's test case loader
AbstractEncryptedJobStoreTest	
FileJobStoreTest	A common base class for Toil tests.
GoogleJobStoreTest	A common base class for Toil tests.
AWSJobStoreTest	A common base class for Toil tests.
InvalidAWSJobStoreTest	A common base class for Toil tests.
EncryptedAWSJobStoreTest	A common base class for Toil tests.
StubHttpRequestHandler	Simple HTTP request handler with GET and HEAD commands.

Functions

google_retry(x)	
tearDownModule()	

Attributes

logger	

toil.test.jobStores.jobStoreTest.**google_retry**(*x*)

toil.test.jobStores.jobStoreTest.**logger**

toil.test.jobStores.jobStoreTest.**tearDownModule**()

class toil.test.jobStores.jobStoreTest.**AbstractJobStoreTest**

Hide abstract base class from unittest's test case loader

http://stackoverflow.com/questions/1323455/python-unit-test-with-base-and-sub-class#answer-25695512

class Test(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

externalStoreCache

mpTestPartSize

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

setUp()

Hook method for setting up the test fixture before exercising it.

tearDown()

Hook method for deconstructing the test fixture after testing it.

testInitialState()

Ensure proper handling of nonexistant files.

testJobCreation()

Test creation of a job.

Does the job exist in the jobstore it is supposed to be in? Are its attributes what is expected?

testConfigEquality()

Ensure that the command line configurations are successfully loaded and stored.

In setUp() self.jobstore1 is created and initialized. In this test, after creating newJobStore, .resume() will look for a previously instantiated job store and load its config options. This is expected to be equal but not the same object.

testJobLoadEquality()

Tests that a job created via one JobStore instance can be loaded from another.

testChildLoadingEquality()

Test that loading a child job operates as expected.

testPersistantFilesToDelete()

Make sure that updating a job carries over filesToDelete.

The following demonstrates the job update pattern, where files to be deleted are referenced in "filesToDelete" array, which is persisted to disk first. If things go wrong during the update, this list of files to delete is used to remove the unneeded files.

testUpdateBehavior()

Tests the proper behavior during updating jobs.

testJobDeletions()

Tests the consequences of deleting jobs.

testSharedFiles()

Tests the sharing of files.

testReadWriteSharedFilesTextMode()

Checks if text mode is compatible for shared file streams.

testReadWriteFileStreamTextMode()

Checks if text mode is compatible for file streams.

testPerJobFiles()

Tests the behavior of files on jobs.

testStatsAndLogging()

Tests behavior of reading and writting stats and logging.

testWriteLogFiles()

Test writing log files.

testBatchCreate()

Test creation of many jobs.

testGrowingAndShrinkingJob()

Make sure jobs update correctly if they grow/shrink.

classmethod cleanUpExternalStores()

classmethod makeImportExportTests()

testImportHttpFile()

Test importing a file over HTTP.

testImportFtpFile()

Test importing a file over FTP

testFileDeletion()

Intended to cover the batch deletion of items in the AWSJobStore, but it doesn't hurt running it on the other job stores.

testMultipartUploads()

This test is meant to cover multi-part uploads in the AWSJobStore but it doesn't hurt running it against the other job stores as well.

testZeroLengthFiles()

Test reading and writing of empty files.

testLargeFile()

Test the reading and writing of large files.

fetch_url(*url*)

Fetch the given URL. Throw an error if it cannot be fetched in a reasonable number of attempts.

Parameters

url (*str*) –

Return type

None

assertUrl(*url*)

testCleanCache()

testPartialReadFromStream()

Test whether readFileStream will deadlock on a partial read.

testDestructionOfCorruptedJobStore()

testDestructionIdempotence()

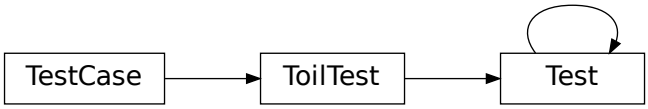
testEmptyFileStoreIDIsReadable()

Simply creates an empty fileStoreID and attempts to read from it.

class toil.test.jobStores.jobStoreTest.**AbstractEncryptedJobStoreTest**

class **Test**(*methodName='runTest'*)

Bases: *AbstractJobStoreTest*



A test of job stores that use encryption

setUp()

Hook method for setting up the test fixture before exercising it.

tearDown()

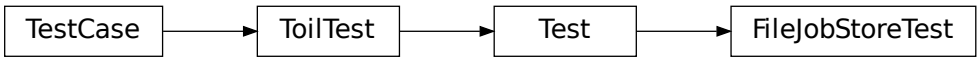
Hook method for deconstructing the test fixture after testing it.

testEncrypted()

Create an encrypted file. Read it in encrypted mode then try with encryption off to ensure that it fails.

class toil.test.jobStores.jobStoreTest.**FileJobStoreTest**(*methodName='runTest'*)

Bases: *AbstractJobStoreTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

testPreserveFileName()

Check that the fileID ends with the given file name.

test_jobstore_init_preserves_symlink_path()

Test that if we provide a fileJobStore with a symlink to a directory, it doesn't de-reference it.

test_jobstore_does_not_leak_symlinks()

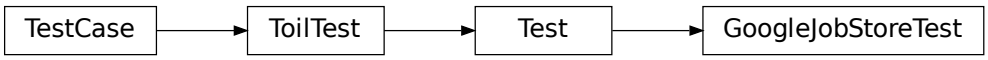
Test that if we link imports into the FileJobStore, we can't get hardlinks to symlinks.

test_file_link_imports()

Test that imported files are symlinked when when expected

class toil.test.jobStores.jobStoreTest.**GoogleJobStoreTest**(*methodName='runTest'*)

Bases: *AbstractJobStoreTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

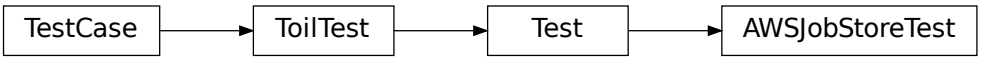
When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

projectID

headers

class toil.test.jobStores.jobStoreTest.**AWSJobStoreTest**(*methodName='runTest'*)

Bases: *AbstractJobStoreTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

testSDBDomainsDeletedOnFailedJobstoreBucketCreation()

This test ensures that SDB domains bound to a jobstore are deleted if the jobstore bucket failed to be created. We simulate a failed jobstore bucket creation by using a bucket in a different region with the same name.

testInlinedFiles()

testOverlargeJob()

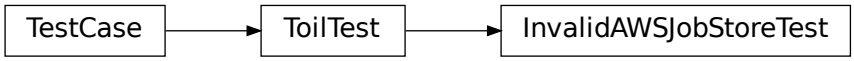
testMultiThreadImportFile()

Tests that importFile is thread-safe.

Return type
None

class toil.test.jobStores.jobStoreTest.**InvalidAWSJobStoreTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



A common base class for Toil tests.

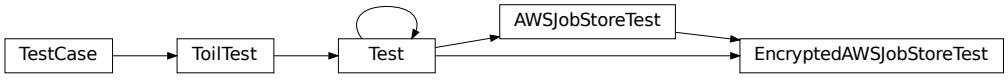
Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

testInvalidJobStoreName()

class toil.test.jobStores.jobStoreTest.**EncryptedAWSJobStoreTest**(*methodName='runTest'*)

Bases: *AWSJobStoreTest*, *AbstractEncryptedJobStoreTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and

any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

class toil.test.jobStores.jobStoreTest.**StubHttpRequestHandler**(**args*, *directory=None*, ***kwargs*)

Bases: http.server.SimpleHTTPRequestHandler

Simple HTTP request handler with GET and HEAD commands.

This serves files from the current directory and any of its subdirectories. The MIME type for files is determined by calling the .guess_type() method.

The GET and HEAD requests are identical except that the HEAD request omits the actual contents of the file.

fileContents = 'A good programmer looks both ways before crossing a one-way street'

do_GET()

Serve a GET request.

toil.test.lib

Subpackages

toil.test.lib.aws

Submodules

toil.test.lib.aws.test_iam

Module Contents

Classes

IAMTest	Check that given permissions and associated functions perform correctly

Attributes

logger	

toil.test.lib.aws.test_iam.**logger**

class toil.test.lib.aws.test_iam.**IAMTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

Check that given permissions and associated functions perform correctly

test_permissions_iam()

test_negative_permissions_iam()

test_wildcard_handling()

toil.test.lib.aws.test_s3

Module Contents

Classes

S3Test	Confirm the workarounds for us-east-1.

Attributes

logger	

toil.test.lib.aws.test_s3.**logger**

class toil.test.lib.aws.test_s3.**S3Test**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

Confirm the workarounds for us-east-1.

s3_resource: Optional[mypy_boto3_s3.S3ServiceResource]

bucket: Optional[mypy_boto3_s3.service_resource.Bucket]

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

Return type
None

test_create_bucket()

Test bucket creation for us-east-1.

Return type
None

test_get_bucket_location_public_bucket()

Test getting buket location for a bucket we don't own.

Return type
None

classmethod tearDownClass()

Hook method for deconstructing the class fixture after running all tests in the class.

Return type
None

toil.test.lib.aws.test_utils

Module Contents

Classes

TagGenerationTest	Test for tag generation from environment variables

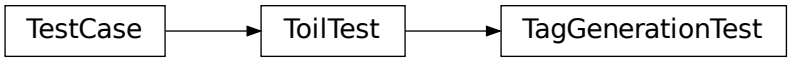
Attributes

logger	

toil.test.lib.aws.test_utils.**logger**

class toil.test.lib.aws.test_utils.**TagGenerationTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Test for tag generation from environment variables

test_build_tag()

test_empty_aws_tags()

test_incorrect_json_object()

test_incorrect_json_emoji()

test_build_tag_with_tags()

Submodules

toil.test.lib.dockerTest

Module Contents

Classes

DockerTest	Tests dockerCall and ensures no containers are left around.

Attributes

logger

toil.test.lib.dockerTest.**logger**

class toil.test.lib.dockerTest.**DockerTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

Tests dockerCall and ensures no containers are left around. When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

setUp()

Hook method for setting up the test fixture before exercising it.

testDockerClean(*caching=False*, *detached=True*, *rm=True*, *deferParam=None*)

Run the test container that creates a file in the work dir, and sleeps for 5 minutes. Ensure that the calling job gets SIGKILLed after a minute, leaving behind the spooky/ghost/zombie container. Ensure that the container is killed on batch system shutdown (through the deferParam mechanism).

testDockerClean_CRx_FORGO()

testDockerClean_CRx_STOP()

testDockerClean_CRx_RM()

testDockerClean_CRx_None()

testDockerClean_CxD_FORGO()

testDockerClean_CxD_STOP()

testDockerClean_CxD_RM()

testDockerClean_CxD_None()

testDockerClean_Cxx_FORGO()

testDockerClean_Cxx_STOP()

testDockerClean_Cxx_RM()

testDockerClean_Cxx_None()

testDockerClean_xRx_FORGO()

testDockerClean_xRx_STOP()

testDockerClean_xRx_RM()

testDockerClean_xRx_None()

testDockerClean_xxD_FORGO()

testDockerClean_xxD_STOP()

testDockerClean_xxD_RM()

testDockerClean_xxD_None()

testDockerClean_xxx_FORGO()

testDockerClean_xxx_STOP()

testDockerClean_xxx_RM()

testDockerClean_xxx_None()

testDockerPipeChain(*caching=False*)

Test for piping API for dockerCall(). Using this API (activated when list of argument lists is given as parameters), commands a piped together into a chain. ex: parameters=[[‘printf’, ‘x

y

‘], [‘wc’, ‘-l’]] should execute:

printf ‘x

y

‘ | wc -l

testDockerPipeChainErrorDetection(*caching=False*)

By default, executing cmd1 | cmd2 | ... | cmdN, will only return an error if cmdN fails. This can lead to all manor of errors being silently missed. This tests to make sure that the piping API for dockerCall() throws an exception if non-last commands in the chain fail.

testNonCachingDockerChain()

testNonCachingDockerChainErrorDetection()

testDockerLogs(*stream=False*, *demux=False*)

Test for the different log outputs when deatch=False.

testDockerLogs_Stream()

testDockerLogs_Demux()

testDockerLogs_Demux_Stream()

toil.test.lib.test_conversions

Module Contents

Classes

ConversionTest	A common base class for Toil tests.

Attributes

logger

toil.test.lib.test_conversions.**logger**

class toil.test.lib.test_conversions.**ConversionTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

test_convert()

test_human2bytes()

test_hms_duration_to_seconds()

toil.test.lib.test_ec2

Module Contents

Classes

FlatcarFeedTest	Test accessing the FLatcar AMI release feed, independent of the AWS API
AMITest	A common base class for Toil tests.

Attributes

logger

toil.test.lib.test_ec2.**logger**

class toil.test.lib.test_ec2.**FlatcarFeedTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

Test accessing the FLatcar AMI release feed, independent of the AWS API

test_parse_archive_feed()

Make sure we can get a Flatcar release from the Internet Archive.

test_parse_beta_feed()

Make sure we can get a Flatcar release from the beta channel.

test_parse_stable_feed()

Make sure we can get a Flatcar release from the stable channel.

test_bypass_stable_feed()

Make sure we can either get or safely not get a Flatcar release from the stable channel.

class toil.test.lib.test_ec2.**AMITest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

test_fetch_flatcar()

test_fetch_arm_flatcar()

Test flatcar AMI finder architecture parameter.

toil.test.lib.test_misc

Module Contents

Classes

UserNameAvailableTest	Make sure we can get user names when they are available.
UserNameUnvailableTest	Make sure we can get something for a user name when user names are not
UserNameVeryBrokenTest	Make sure we can get something for a user name when user name fetching is

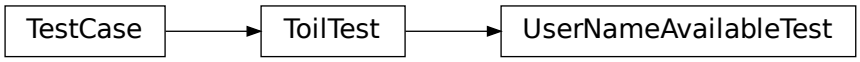
Attributes

logger

toil.test.lib.test_misc.**logger**

class toil.test.lib.test_misc.**UserNameAvailableTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

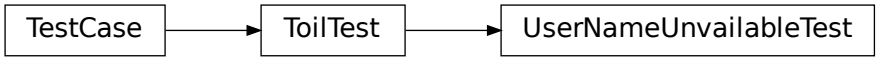


Make sure we can get user names when they are available.

test_get_user_name()

class toil.test.lib.test_misc.**UserNameUnvailableTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Make sure we can get something for a user name when user names are not available.

setUp()

Hook method for setting up the test fixture before exercising it.

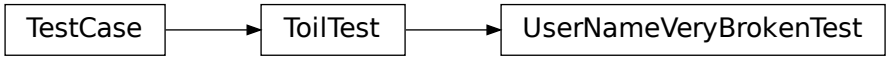
tearDown()

Hook method for deconstructing the test fixture after testing it.

test_get_user_name()

class toil.test.lib.test_misc.**UserNameVeryBrokenTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Make sure we can get something for a user name when user name fetching is broken in ways we did not expect.

setUp()

Hook method for setting up the test fixture before exercising it.

tearDown()

Hook method for deconstructing the test fixture after testing it.

test_get_user_name()

toil.test.mesos

Submodules

toil.test.mesos.MesosDataStructuresTest

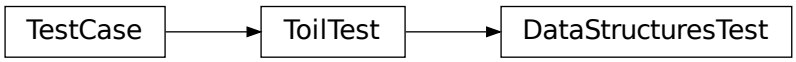
Module Contents

Classes

DataStructuresTest	A common base class for Toil tests.

class toil.test.mesos.MesosDataStructuresTest.**DataStructuresTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

testJobQueue(*testJobs=1000*)

The mesos JobQueue sorts MesosShape objects by requirement and this test ensures that that sorting is what is expected: non-preemptible jobs groups first, with priority given to large jobs.

toil.test.mesos.helloWorld

A simple user script for Toil

Module Contents

Functions

hello_world(job)	
hello_world_child(job, hw)	
main()	

Attributes

childMessage	
parentMessage	

toil.test.mesos.helloWorld.**childMessage = 'The child job is now running!'**

toil.test.mesos.helloWorld.**parentMessage = 'The parent job is now running!'**

toil.test.mesos.helloWorld.**hello_world**(*job*)

toil.test.mesos.helloWorld.**hello_world_child**(*job*, *hw*)

toil.test.mesos.helloWorld.**main**()

toil.test.mesos.stress

Module Contents

Classes

LongTestJob	Class represents a unit of work in toil.
LongTestFollowOn	Class represents a unit of work in toil.
HelloWorldJob	Class represents a unit of work in toil.
HelloWorldFollowOn	Class represents a unit of work in toil.

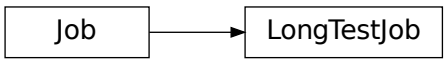
Functions

touchFile(fileStore)	
main(numJobs)	

toil.test.mesos.stress.**touchFile**(*fileStore*)

class toil.test.mesos.stress.**LongTestJob**(*numJobs*)

Bases: *toil.job.Job*



Class represents a unit of work in toil.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

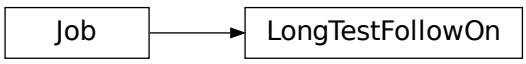
fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

class toil.test.mesos.stress.**LongTestFollowOn**

Bases: *toil.job.Job*



Class represents a unit of work in toil.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

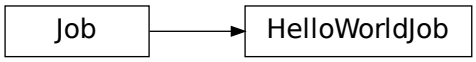
Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns
The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

class toil.test.mesos.stress.**HelloWorldJob**(*i*)

Bases: *toil.job.Job*



Class represents a unit of work in toil.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters
fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns
The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

class toil.test.mesos.stress.**HelloWorldFollowOn**(*i*)

Bases: *toil.job.Job*

Class represents a unit of work in toil.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters
fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns
The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

toil.test.mesos.stress.**main**(*numJobs*)

toil.test.provisioners

Subpackages

toil.test.provisioners.aws

Submodules

toil.test.provisioners.aws.awsProvisionerTest

Module Contents

Classes

AWSProvisionerBenchTest	Tests for the AWS provisioner that don't actually provision anything.
AbstractAWSAutoscaleTest	A common base class for Toil tests.
AWSAutoscaleTest	A common base class for Toil tests.
AWSStaticAutoscaleTest	Runs the tests on a statically provisioned cluster with autoscaling enabled.
AWSManagedAutoscaleTest	Runs the tests on a self-scaling Kubernetes cluster.
AWSAutoscaleTestMultipleNodeTypes	A common base class for Toil tests.
AWSRestartTest	This test insures autoscaling works on a restarted Toil run.
PreemptibleDeficitCompensationTest	A common base class for Toil tests.

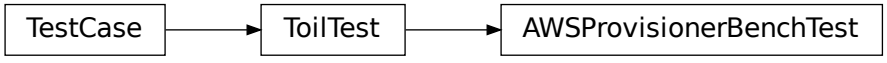
Attributes

log

toil.test.provisioners.aws.awsProvisionerTest.**log**

class toil.test.provisioners.aws.awsProvisionerTest.**AWSProvisionerBenchTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Tests for the AWS provisioner that don't actually provision anything.

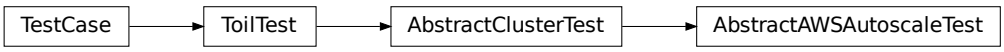
test_AMI_finding()

test_read_write_global_files()

Make sure the *_write_file_to_cloud()* and *_read_file_from_cloud()* functions of the AWS provisioner work as intended.

class toil.test.provisioners.aws.awsProvisionerTest.**AbstractAWSAutoscaleTest**(*methodName*)

Bases: *toil.test.provisioners.clusterTest.AbstractClusterTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

script()

Return the full path to the user script on the leader.

data(*filename*)

Return the full path to the data file with the given name on the leader.

rsyncUtil(*src*, *dest*)

getRootVolID()

putScript(*content*)

Helper method for _getScript to inject a script file at the configured script path, from text.

Parameters
content (*str*) –

class toil.test.provisioners.aws.awsProvisionerTest.**AWSAutoscaleTest**(*name*)

Bases: *AbstractAWSAutoscaleTest*

A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP

is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

setUp()

Set up for the test. Must be overridden to call this method and set self.jobStore.

launchCluster()

getRootVolID()

Adds in test to check that EBS volume is build with adequate size. Otherwise is functionally equivalent to parent. :return: volumeID

testAutoScale()

testSpotAutoScale()

testSpotAutoScaleBalancingTypes()

class toil.test.provisioners.aws.awsProvisionerTest.**AWSStaticAutoscaleTest**(*name*)

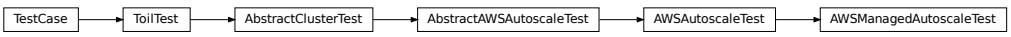
Bases: *AWSAutoscaleTest*

Runs the tests on a statically provisioned cluster with autoscaling enabled.

launchCluster()

class toil.test.provisioners.aws.awsProvisionerTest.**AWSManagedAutoscaleTest**(*name*)

Bases: *AWSAutoscaleTest*

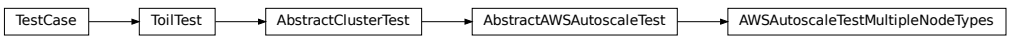


Runs the tests on a self-scaling Kubernetes cluster.

launchCluster()

class toil.test.provisioners.aws.awsProvisionerTest.**AWSAutoscaleTestMultipleNodeTypes**(*name*)

Bases: *AbstractAWSAutoscaleTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

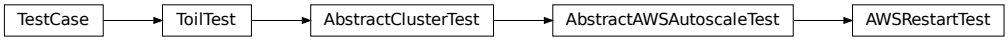
setUp()

Set up for the test. Must be overridden to call this method and set self.jobStore.

testAutoScale()

class toil.test.provisioners.aws.awsProvisionerTest.**AWSRestartTest**(*name*)

Bases: *AbstractAWSAutoscaleTest*



This test insures autoscaling works on a restarted Toil run.

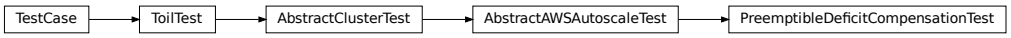
setUp()

Set up for the test. Must be overridden to call this method and set self.jobStore.

testAutoScaledCluster()

class toil.test.provisioners.aws.awsProvisionerTest.**PreemptibleDeficitCompensationTest**(*name*)

Bases: *AbstractAWSAutoscaleTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

setUp()

Set up for the test. Must be overridden to call this method and set self.jobStore.

test()

Submodules

toil.test.provisioners.clusterScalerTest

Module Contents

Classes

BinPackingTest	A common base class for Toil tests.
ClusterScalerTest	A common base class for Toil tests.
ScalerThreadTest	A common base class for Toil tests.
MockBatchSystemAndProvisioner	Mimics a leader, job batcher, provisioner and scalable batch system.

Attributes

logger
c4_8xlarge_preemptible
c4_8xlarge
r3_8xlarge
r5_2xlarge
r5_4xlarge
t2_micro

toil.test.provisioners.clusterScalerTest.**logger**

toil.test.provisioners.clusterScalerTest.**c4_8xlarge_preemptible**

toil.test.provisioners.clusterScalerTest.**c4_8xlarge**

toil.test.provisioners.clusterScalerTest.**r3_8xlarge**

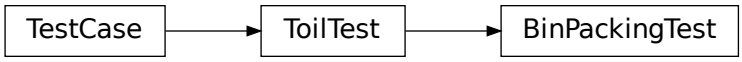
toil.test.provisioners.clusterScalerTest.**r5_2xlarge**

toil.test.provisioners.clusterScalerTest.**r5_4xlarge**

toil.test.provisioners.clusterScalerTest.**t2_micro**

class toil.test.provisioners.clusterScalerTest.**BinPackingTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

setUp()

Hook method for setting up the test fixture before exercising it.

testPackingOneShape()

Pack one shape and check that the resulting reservations look sane.

testSorting()

Test that sorting is correct: preemptible, then memory, then cores, then disk, then wallTime.

testAddingInitialNode()

Pack one shape when no nodes are available and confirm that we fit one node properly.

testLowTargetTime()

Test that a low targetTime (0) parallelizes jobs aggressively (1000 queued jobs require 1000 nodes).

Ideally, low targetTime means: Start quickly and maximize parallelization after the cpu/disk/mem have been packed.

Disk/cpu/mem packing is prioritized first, so we set job resource reqs so that each t2.micro (1 cpu/8G disk/1G RAM) can only run one job at a time with its resources.

Each job is parametrized to take 300 seconds, so (the minimum of) 1 of them should fit into each node's 0 second window, so we expect 1000 nodes.

testHighTargetTime()

Test that a high targetTime (3600 seconds) maximizes packing within the targetTime.

Ideally, high targetTime means: Maximize packing within the targetTime after the cpu/disk/mem have been packed.

Disk/cpu/mem packing is prioritized first, so we set job resource reqs so that each t2.micro (1 cpu/8G disk/1G RAM) can only run one job at a time with its resources.

Each job is parametrized to take 300 seconds, so 12 of them should fit into each node's 3600 second window. 1000/12 = 83.33, so we expect 84 nodes.

testZeroResourceJobs()

Test that jobs requiring zero cpu/disk/mem pack first, regardless of targetTime.

Disk/cpu/mem packing is prioritized first, so we set job resource reqs so that each t2.micro (1 cpu/8G disk/1G RAM) can run a seemingly infinite number of jobs with its resources.

Since all jobs should pack cpu/disk/mem-wise on a t2.micro, we expect only one t2.micro to be provisioned. If we raise this, as in testLowTargetTime, it will launch 1000 t2.micros.

testLongRunningJobs()

Test that jobs with long run times (especially service jobs) are aggressively parallelized.

This is important, because services are one case where the degree of parallelization really, really matters. If you have multiple services, they may all need to be running simultaneously before any real work can be done.

Despite setting globalTargetTime=3600, this should launch 1000 t2.micros because each job's estimated runtime (30000 seconds) extends well beyond 3600 seconds.

run1000JobsOnMicros(*jobCores*, *jobMem*, *jobDisk*, *jobTime*, *globalTargetTime*)

Test packing 1000 jobs on t2.micros. Depending on the targetTime and resources, these should pack differently.

testPathologicalCase()

Test a pathological case where only one node can be requested to fit months' worth of jobs.

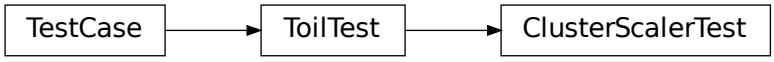
If the reservation is extended to fit a long job, and the bin-packer naively searches through all the reservation slices to find the first slice that fits, it will happily assign the first slot that fits the job, even if that slot occurs days in the future.

testJobTooLargeForAllNodes()

If a job is too large for all node types, the scaler should print a warning, but definitely not crash.

class toil.test.provisioners.clusterScalerTest.**ClusterScalerTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

setUp()

Hook method for setting up the test fixture before exercising it.

testRounding()

Test to make sure the ClusterScaler's rounding rounds properly.

testMaxNodes()

Set the scaler to be very aggressive, give it a ton of jobs, and make sure it doesn't go over maxNodes.

testMinNodes()

Without any jobs queued, the scaler should still estimate "minNodes" nodes.

testPreemptibleDeficitResponse()

When a preemptible deficit was detected by a previous run of the loop, the scaler should add non-preemptible nodes to compensate in proportion to preemptibleCompensation.

testPreemptibleDeficitIsSet()

Make sure that updateClusterSize sets the preemptible deficit if it can't launch preemptible nodes properly. That way, the deficit can be communicated to the next run of estimateNodeCount.

testNoLaunchingIfDeltaAlreadyMet()

Check that the scaler doesn't try to launch "0" more instances if the delta was able to be met by unignoring nodes.

testBetaInertia()

test_overhead_accounting_large()

If a node has a certain raw memory or disk capacity, that won't all be available when it actually comes up; some disk and memory will be used by the OS, and the backing scheduler (Mesos, Kubernetes, etc.).

Make sure this overhead is accounted for for large nodes.

test_overhead_accounting_small()

If a node has a certain raw memory or disk capacity, that won't all be available when it actually comes up; some disk and memory will be used by the OS, and the backing scheduler (Mesos, Kubernetes, etc.).

Make sure this overhead is accounted for for small nodes.

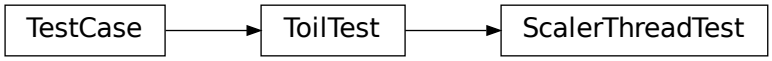
test_overhead_accounting_observed()

If a node has a certain raw memory or disk capacity, that won't all be available when it actually comes up; some disk and memory will be used by the OS, and the backing scheduler (Mesos, Kubernetes, etc.).

Make sure this overhead is accounted for so that real-world observed failures cannot happen again.

class toil.test.provisioners.clusterScalerTest.**ScalerThreadTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

testClusterScaling()

Test scaling for a batch of non-preemptible jobs and no preemptible jobs (makes debugging easier).

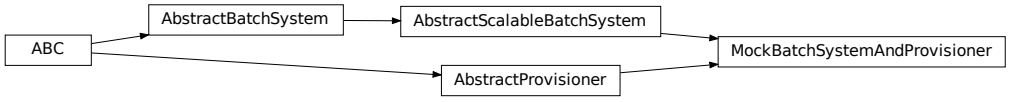
testClusterScalingMultipleNodeTypes()

testClusterScalingWithPreemptibleJobs()

Test scaling simultaneously for a batch of preemptible and non-preemptible jobs.

class toil.test.provisioners.clusterScalerTest.**MockBatchSystemAndProvisioner**(*config*, *secondsPerJob*)

Bases: *toil.batchSystems.abstractBatchSystem.AbstractScalableBatchSystem*, *toil.provisioners.abstractProvisioner.AbstractProvisioner*



Mimics a leader, job batcher, provisioner and scalable batch system.

start()

shutDown()

nodeInUse(*nodeIP*)

Can be used to determine if a worker node is running any tasks. If the node is doesn't exist, this function should simply return False.

Parameters
nodeIP – The worker nodes private IP address

Returns
True if the worker node has been issued any tasks, else False

ignoreNode(*nodeAddress*)

Stop sending jobs to this node. Used in autoscaling when the autoscaler is ready to terminate a node, but jobs are still running. This allows the node to be terminated after the current jobs have finished.

Parameters
nodeAddress – IP address of node to ignore.

unignoreNode(*nodeAddress*)

Stop ignoring this address, presumably after a node with this address has been terminated. This allows for the possibility of a new node having the same address as a terminated one.

supportedClusterTypes()

Get all the cluster types that this provisioner implementation supports.

createClusterSettings()

Initialize class for a new cluster, to be deployed, when running outside the cloud.

readClusterSettings()

Initialize class from an existing cluster. This method assumes that the instance we are running on is the leader.

Implementations must call _setLeaderWorkerAuthentication().

setAutoscaledNodeTypes(*node_types*)

Set node types, shapes and spot bids for Toil-managed autoscaling. :param nodeTypes: A list of node types, as parsed with parse_node_types.

Parameters

node_types (*List[Tuple[Set[toil.provisioners.abstractProvisioner.Shape], Optional[float]]]*) –

getProvisionedWorkers(*instance_type=None*, *preemptible=None*)

Returns a list of Node objects, each representing a worker node in the cluster

Parameters

preemptible – If True only return preemptible nodes else return non-preemptible nodes

Returns

list of Node

terminateNodes(*nodes*)

Terminate the nodes represented by given Node objects

Parameters

nodes – list of Node objects

remainingBillingInterval(*node*)

addJob(*jobShape*, *preemptible=False*)

Add a job to the job queue

getNumberOfJobsIssued(*preemptible=None*)

getJobs()

getNodes(*preemptible=False*, *timeout=600*)

Returns a dictionary mapping node identifiers of preemptible or non-preemptible nodes to NodeInfo objects, one for each node.

Parameters

- **preemptible** (*Optional[bool]*) – If True (False) only (non-)preemptible nodes will be returned. If None, all nodes will be returned.
- **timeout** (*int*) –

addNodes(*nodeTypes*, *numNodes*, *preemptible*)

Used to add worker nodes to the cluster

Parameters

- **numNodes** – The number of nodes to add
- **preemptible** – whether or not the nodes will be preemptible
- **spotBid** – The bid for preemptible nodes if applicable (this can be set in config, also).
- **nodeTypes** (*Set[str]*) –

Returns

number of nodes successfully added

Return type

int

getNodeShape(*nodeType*, *preemptible=False*)

The shape of a preemptible or non-preemptible node managed by this provisioner. The node shape defines key properties of a machine, such as its number of cores or the time between billing intervals.

Parameters
instance_type (*str*) – Instance type name to return the shape of.

Return type
Shape

getWorkersInCluster(*nodeShape*)

launchCluster(*leaderNodeType*, *keyName*, *userTags=None*, *vpcSubnet=None*, *leaderStorage=50*, *nodeStorage=50*, *botoPath=None*, ***kwargs*)

Initialize a cluster and create a leader node.

Implementations must call _setLeaderWorkerAuthentication() with the leader so that workers can be launched.

Parameters

- **leaderNodeType** – The leader instance.
- **leaderStorage** – The amount of disk to allocate to the leader in gigabytes.
- **owner** – Tag identifying the owner of the instances.

destroyCluster()

Terminates all nodes in the specified cluster and cleans up all resources associated with the cluster. :param clusterName: identifier of the cluster to terminate.

Return type
None

getLeader()

Returns
The leader node.

getNumberOfNodes(*nodeType=None*, *preemptible=None*)

toil.test.provisioners.clusterTest

Module Contents

Classes

AbstractClusterTest	A common base class for Toil tests.

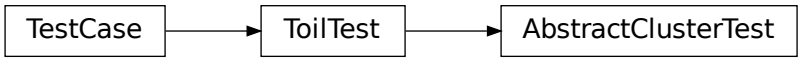
Attributes

log

toil.test.provisioners.clusterTest.**log**

class toil.test.provisioners.clusterTest.**AbstractClusterTest**(*methodName*)

Bases: *toil.test.ToilTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

python()

Return the full path to the venv Python on the leader.

pip()

Return the full path to the venv pip on the leader.

destroyCluster()

Destroy the cluster we built, if it exists.

Succeeds if the cluster does not currently exist.

Return type
None

setUp()

Set up for the test. Must be overridden to call this method and set self.jobStore.

tearDown()

Hook method for deconstructing the test fixture after testing it.

sshUtil(*command*)

Run the given command on the cluster. Raise subprocess.CalledProcessError if it fails.

createClusterUtil(*args=None*)

launchCluster()

toil.test.provisioners.gceProvisionerTest

Module Contents

Classes

AbstractGCEAutoscaleTest	A common base class for Toil tests.
GCEAutoscaleTest	A common base class for Toil tests.
GCEStaticAutoscaleTest	Runs the tests on a statically provisioned cluster with autoscaling enabled.
GCEAutoscaleTestMultipleNodeTypes	A common base class for Toil tests.
GCERestartTest	This test insures autoscaling works on a restarted Toil run

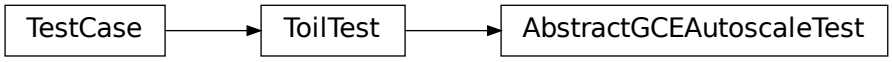
Attributes

log

toil.test.provisioners.gceProvisionerTest.**log**

class toil.test.provisioners.gceProvisionerTest.**AbstractGCEAutoscaleTest**(*methodName*)

Bases: *toil.test.ToilTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

projectID

sshUtil(*command*)

rsyncUtil(*src*, *dest*)

destroyClusterUtil()

createClusterUtil(*args=None*)

cleanJobStoreUtil()

setUp()

Hook method for setting up the test fixture before exercising it.

tearDown()

Hook method for deconstructing the test fixture after testing it.

launchCluster()

class toil.test.provisioners.gceProvisionerTest.**GCEAutoscaleTest**(*name*)

Bases: *AbstractGCEAutoscaleTest*

A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

setUp()

Hook method for setting up the test fixture before exercising it.

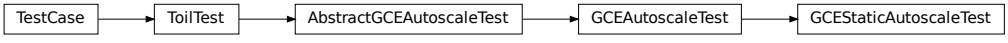
launchCluster()

testAutoScale()

testSpotAutoScale()

class toil.test.provisioners.gceProvisionerTest.**GCEStaticAutoscaleTest**(*name*)

Bases: *GCEAutoscaleTest*



Runs the tests on a statically provisioned cluster with autoscaling enabled.

launchCluster()

class toil.test.provisioners.gceProvisionerTest.**GCEAutoscaleTestMultipleNodeTypes**(*name*)

Bases: *AbstractGCEAutoscaleTest*

A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

setUp()

Hook method for setting up the test fixture before exercising it.

testAutoScale()

class toil.test.provisioners.gceProvisionerTest.**GCERestartTest**(*name*)

Bases: *AbstractGCEAutoscaleTest*

This test insures autoscaling works on a restarted Toil run

setUp()

Hook method for setting up the test fixture before exercising it.

testAutoScaledCluster()

toil.test.provisioners.provisionerTest

Module Contents

Classes

ProvisionerTest	A common base class for Toil tests.

Attributes

log

toil.test.provisioners.provisionerTest.**log**

class toil.test.provisioners.provisionerTest.**ProvisionerTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

test_node_type_parsing()

Return type
None

toil.test.provisioners.restartScript

Module Contents

Functions

f0(job)

Attributes

parser

toil.test.provisioners.restartScript.**f0**(*job*)

toil.test.provisioners.restartScript.**parser**

toil.test.server

Submodules

toil.test.server.serverTest

Module Contents

Classes

ToilServerUtilsTest	Tests for the utility functions used by the Toil server.
hidden	
FileStateStoreTest	Test file-based state storage.
FileStateStoreURLTest	Test file-based state storage using URLs instead of local paths.
BucketUsingTest	Base class for tests that need a bucket.
AWSStateStoreTest	Test AWS-based state storage.
AbstractToilWESServerTest	Class for server tests that provides a self.app in testing mode.
ToilWESServerBenchTest	Tests for Toil's Workflow Execution Service API that don't run workflows.
ToilWESServerWorkflowTest	Tests of the WES server running workflows.
ToilWESServerCeleryWorkflowTest	End-to-end workflow-running tests against Celery.
ToilWESServerCeleryS3StateWorkflowTest	Test the server with Celery and state stored in S3.

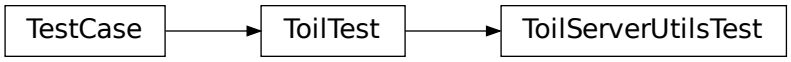
Attributes

logger

toil.test.server.serverTest.**logger**

class toil.test.server.serverTest.**ToilServerUtilsTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Tests for the utility functions used by the Toil server.

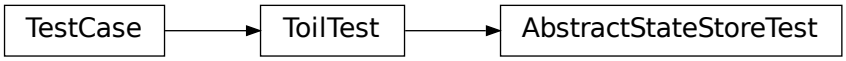
test_workflow_canceling_recovery()

Make sure that a workflow in CANCELING state will be recovered to a terminal state eventually even if the workflow runner Celery task goes away without flipping the state.

class toil.test.server.serverTest.**hidden**

class AbstractStateStoreTest(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Basic tests for state stores.

abstract get_state_store()

Make a state store to test, on a single fixed URL.

Return type

AbstractStateStore

test_state_store()

Make sure that the state store under test can store and load keys.

Return type

None

class toil.test.server.serverTest.**FileStateStoreTest**(*methodName='runTest'*)

Bases: *hidden*

Test file-based state storage.

setUp()

Hook method for setting up the test fixture before exercising it.

Return type
None

`get_state_store()`

Make a state store to test, on a single fixed local path.

Return type
AbstractStateStore

`class toil.test.server.serverTest.FileStateStoreURLTest(methodName='runTest')`

Bases: *hidden*

Test file-based state storage using URLs instead of local paths.

`setUp()`

Hook method for setting up the test fixture before exercising it.

Return type
None

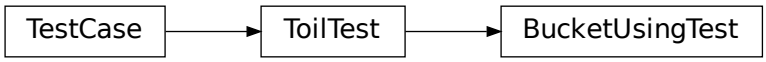
`get_state_store()`

Make a state store to test, on a single fixed URL.

Return type
AbstractStateStore

`class toil.test.server.serverTest.BucketUsingTest(methodName='runTest')`

Bases: *toil.test.ToilTest*



Base class for tests that need a bucket.

`region: Optional[str]`

`s3_resource: Optional[mypy_boto3_s3.S3ServiceResource]`

`bucket: Optional[mypy_boto3_s3.service_resource.Bucket]`

`bucket_name: Optional[str]`

`classmethod setUpClass()`

Set up the class with a single pre-existing AWS bucket for all tests.

Return type
None

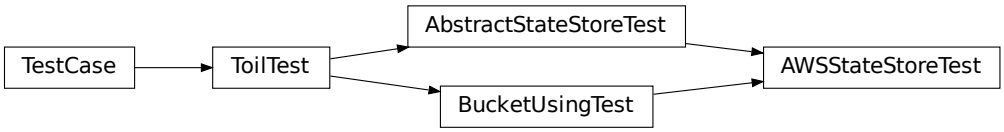
classmethod tearDownClass()

Hook method for deconstructing the class fixture after running all tests in the class.

Return type
None

class toil.test.server.serverTest.**AWSStateStoreTest**(*methodName='runTest'*)

Bases: *hidden*, *BucketUsingTest*



Test AWS-based state storage.

bucket_path = 'prefix/of/keys'

get_state_store()

Make a state store to test, on a single fixed URL.

Return type
AbstractStateStore

test_state_store_paths()

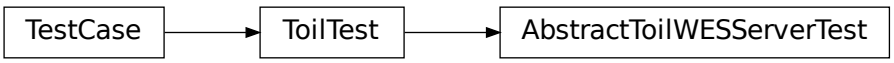
Make sure that the S3 state store puts things in the right places.

We don't *really* care about the exact internal structure, but we do care about actually being under the path we are supposed to use.

Return type
None

class toil.test.server.serverTest.**AbstractToilWESServerTest**(**args*, ***kwargs*)

Bases: *toil.test.ToilTest*



Class for server tests that provides a self.app in testing mode.

setUp()

Hook method for setting up the test fixture before exercising it.

Return type
None

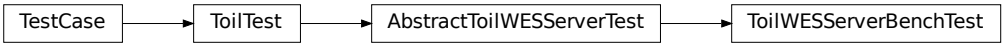
`tearDown()`

Hook method for deconstructing the test fixture after testing it.

Return type
None

`class toil.test.server.serverTest.ToilWESServerBenchTest(*args, **kwargs)`

Bases: *AbstractToilWESServerTest*



Tests for Toil's Workflow Execution Service API that don't run workflows.

`test_home()`

Test the homepage endpoint.

Return type
None

`test_health()`

Test the health check endpoint.

Return type
None

`test_get_service_info()`

Test the GET /service-info endpoint.

Return type
None

`class toil.test.server.serverTest.ToilWESServerWorkflowTest(*args, **kwargs)`

Bases: *AbstractToilWESServerTest*

Tests of the WES server running workflows.

`run_zip_workflow(zip_path, include_message=True, include_params=True)`

We have several zip file tests; this submits a zip file and makes sure it ran OK.

If include_message is set to False, don't send a "message" argument in workflow_params. If include_params is also set to False, don't send workflow_params at all.

Parameters

- **zip_path** (*str*) –
- **include_message** (*bool*) –
- **include_params** (*bool*) –

Return type
None

test_run_workflow_relative_url_no_attachments_fails()

Test run example CWL workflow from relative workflow URL but with no attachments.

Return type
None

test_run_workflow_relative_url()

Test run example CWL workflow from relative workflow URL.

Return type
None

test_run_workflow_https_url()

Test run example CWL workflow from the Internet.

Return type
None

test_run_workflow_single_file_zip()

Test run example CWL workflow from single-file ZIP.

Return type
None

test_run_workflow_multi_file_zip()

Test run example CWL workflow from multi-file ZIP.

Return type
None

test_run_workflow_manifest_zip()

Test run example CWL workflow from ZIP with manifest.

Return type
None

test_run_workflow_inputs_zip()

Test run example CWL workflow from ZIP without manifest but with inputs.

Return type
None

test_run_workflow_manifest_and_inputs_zip()

Test run example CWL workflow from ZIP with manifest and inputs.

Return type
None

test_run_workflow_no_params_zip()

Test run example CWL workflow from ZIP without workflow_params.

Return type
None

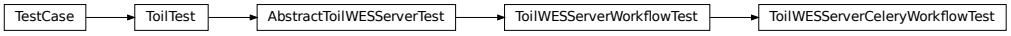
test_run_and_cancel_workflows()

Run two workflows, cancel one of them, and make sure they all exist.

Return type
None

class toil.test.server.serverTest.**ToilWESServerCeleryWorkflowTest**(**args*, ***kwargs*)

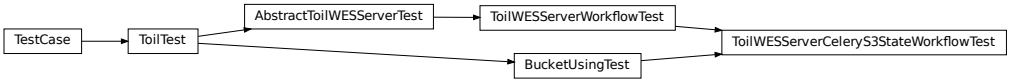
Bases: *ToilWESServerWorkflowTest*



End-to-end workflow-running tests against Celery.

class toil.test.server.serverTest.**ToilWESServerCeleryS3StateWorkflowTest**(**args*, ***kwargs*)

Bases: *ToilWESServerWorkflowTest*, *BucketUsingTest*



Test the server with Celery and state stored in S3.

setUp()

Hook method for setting up the test fixture before exercising it.

Return type
None

toil.test.sort

Submodules

toil.test.sort.restart_sort

A demonstration of toil. Sorts the lines of a file into ascending order by doing a parallel merge sort. This is an intentionally buggy version that doesn't include restart() for testing purposes.

Module Contents

Functions

setup(job, inputFile, N, downCheckpoints, options)	Sets up the sort.
down(job, inputFileStoreID, N, path, downCheckpoints, ...)	Input is a file, a subdivision size N, and a path in the hierarchy of jobs.
up(job, inputFileID1, inputFileID2, path, options[, ...])	Merges the two files and places them in the output.
sort(file)	Sorts the given file.
merge(fileHandle1, fileHandle2, outputFileHandle)	Merges together two files maintaining sorted order.
copySubRangeOfFile(inputFile, fileStart, fileEnd)	Copies the range (in bytes) between fileStart and fileEnd to the given
getMidPoint(file, fileStart, fileEnd)	Finds the point in the file to split.
makeFileToSort(fileName[, lines, lineLen])	
main([options])	

Attributes

defaultLines
defaultLineLen
sortMemory

toil.test.sort.restart_sort.**defaultLines = 1000**

toil.test.sort.restart_sort.**defaultLineLen = 50**

toil.test.sort.restart_sort.**sortMemory = '600M'**

toil.test.sort.restart_sort.**setup**(*job*, *inputFile*, *N*, *downCheckpoints*, *options*)

Sets up the sort. Returns the FileID of the sorted file

toil.test.sort.restart_sort.**down**(*job*, *inputFileStoreID*, *N*, *path*, *downCheckpoints*, *options*, *memory=sortMemory*)

Input is a file, a subdivision size N, and a path in the hierarchy of jobs. If the range is larger than a threshold N the range is divided recursively and a follow on job is then created which merges back the results else the file is sorted and placed in the output.

toil.test.sort.restart_sort.**up**(*job*, *inputFileID1*, *inputFileID2*, *path*, *options*, *memory=sortMemory*)

Merges the two files and places them in the output.

toil.test.sort.restart_sort.**sort**(*file*)

Sorts the given file.

toil.test.sort.restart_sort.**merge**(*fileHandle1*, *fileHandle2*, *outputFileHandle*)

Merges together two files maintaining sorted order.

All handles must be text-mode streams.

toil.test.sort.restart_sort.**copySubRangeOfFile**(*inputFile*, *fileStart*, *fileEnd*)

Copies the range (in bytes) between fileStart and fileEnd to the given output file handle.

toil.test.sort.restart_sort.**getMidPoint**(*file*, *fileStart*, *fileEnd*)

Finds the point in the file to split. Returns an int i such that fileStart <= i < fileEnd

toil.test.sort.restart_sort.**makeFileToSort**(*fileName*, *lines=defaultLines*, *lineLen=defaultLineLen*)

toil.test.sort.restart_sort.**main**(*options=None*)

toil.test.sort.sort

A demonstration of toil. Sorts the lines of a file into ascending order by doing a parallel merge sort.

Module Contents

Functions

setup(job, inputFile, N, downCheckpoints, options)	Sets up the sort.
down(job, inputFileStoreID, N, path, downCheckpoints, ...)	Input is a file, a subdivision size N, and a path in the hierarchy of jobs.
up(job, inputFileID1, inputFileID2, path, options[, ...])	Merges the two files and places them in the output.
sort(file)	Sorts the given file.
merge(fileHandle1, fileHandle2, outputFileHandle)	Merges together two files maintaining sorted order.
copySubRangeOfFile(inputFile, fileStart, fileEnd)	Copies the range (in bytes) between fileStart and fileEnd to the given
getMidPoint(file, fileStart, fileEnd)	Finds the point in the file to split.
makeFileToSort(fileName[, lines, lineLen])	
main([options])	

Attributes

defaultLines
defaultLineLen
sortMemory

toil.test.sort.sort.**defaultLines = 1000**

toil.test.sort.sort.**defaultLineLen = 50**

toil.test.sort.sort.**sortMemory = '600M'**

toil.test.sort.sort.**setup**(*job*, *inputFile*, *N*, *downCheckpoints*, *options*)

Sets up the sort. Returns the FileID of the sorted file

toil.test.sort.sort.**down**(*job*, *inputFileStoreID*, *N*, *path*, *downCheckpoints*, *options*, *memory=sortMemory*)

Input is a file, a subdivision size N, and a path in the hierarchy of jobs. If the range is larger than a threshold N the range is divided recursively and a follow on job is then created which merges back the results else the file is sorted and placed in the output.

toil.test.sort.sort.**up**(*job*, *inputFileID1*, *inputFileID2*, *path*, *options*, *memory=sortMemory*)

Merges the two files and places them in the output.

toil.test.sort.sort.**sort**(*file*)

Sorts the given file.

toil.test.sort.sort.**merge**(*fileHandle1*, *fileHandle2*, *outputFileHandle*)

Merges together two files maintaining sorted order.

All handles must be text-mode streams.

toil.test.sort.sort.**copySubRangeOfFile**(*inputFile*, *fileStart*, *fileEnd*)

Copies the range (in bytes) between fileStart and fileEnd to the given output file handle.

toil.test.sort.sort.**getMidPoint**(*file*, *fileStart*, *fileEnd*)

Finds the point in the file to split. Returns an int i such that fileStart <= i < fileEnd

toil.test.sort.sort.**makeFileToSort**(*fileName*, *lines=defaultLines*, *lineLen=defaultLineLen*)

toil.test.sort.sort.**main**(*options=None*)

toil.test.sort.sortTest

Module Contents

Classes

SortTest	Tests Toil by sorting a file in parallel on various combinations of job stores and batch

Functions

runMain(options)	make sure the output file is deleted every time main is run

Attributes

logger
defaultLineLen
defaultLines
defaultN

toil.test.sort.sortTest.**logger**

toil.test.sort.sortTest.**defaultLineLen**

toil.test.sort.sortTest.**defaultLines**

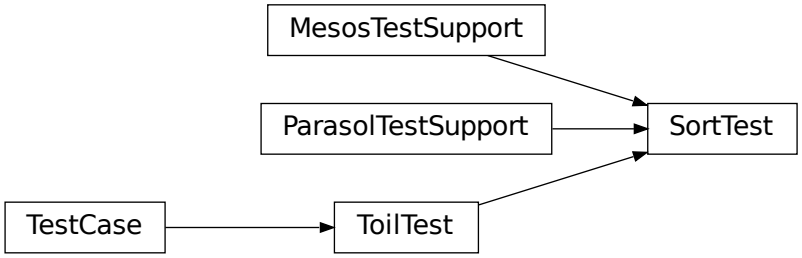
toil.test.sort.sortTest.**defaultN**

toil.test.sort.sortTest.**runMain**(*options*)

make sure the output file is deleted every time main is run

class toil.test.sort.sortTest.**SortTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*, *toil.batchSystems.mesos.test.MesosTestSupport*, *toil.test.batchSystems.parasolTestSupport.ParasolTestSupport*



Tests Toil by sorting a file in parallel on various combinations of job stores and batch systems.

testNo = 5

setUp()

Hook method for setting up the test fixture before exercising it.

tearDown()

Hook method for deconstructing the test fixture after testing it.

testAwsSingle()

testAwsMesos()

testFileMesos()

testGoogleSingle()

testGoogleMesos()

testFileSingle()

testFileSingleNonCaching()

testFileSingleCheckpoints()

testFileSingle10000()

testFileGridEngine()

testFileTorqueEngine()

testFileParasol()

testSort()

testMerge()

testCopySubRangeOfFile()

testGetMidPoint()

toil.test.src

Submodules

toil.test.src.autoDeploymentTest

Module Contents

Classes

AutoDeploymentTest	Tests various auto-deployment scenarios. Using the appliance, i.e. a docker container,

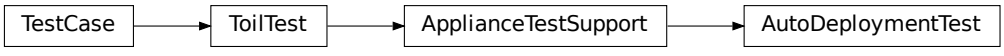
Attributes

logger

toil.test.src.autoDeploymentTest.**logger**

class toil.test.src.autoDeploymentTest.**AutoDeploymentTest**(*methodName='runTest'*)

Bases: *toil.test.ApplianceTestSupport*



Tests various auto-deployment scenarios. Using the appliance, i.e. a docker container, for these tests allows for running worker processes on the same node as the leader process while keeping their file systems separate from each other and the leader process. Separate file systems are crucial to prove that auto-deployment does its job.

sitePackages

setUp()

Hook method for setting up the test fixture before exercising it.

testRestart()

Test whether auto-deployment works on restart.

testSplitRootPackages()

Test whether auto-deployment works with a virtualenv in which jobs are defined in completely separate branches of the package hierarchy. Initially, auto-deployment did deploy the entire virtualenv but jobs could only be defined in one branch of the package hierarchy. We define a branch as the maximum set of fully qualified package paths that share the same first component. IOW, a.b and a.c are in the same branch, while a.b and d.c are not.

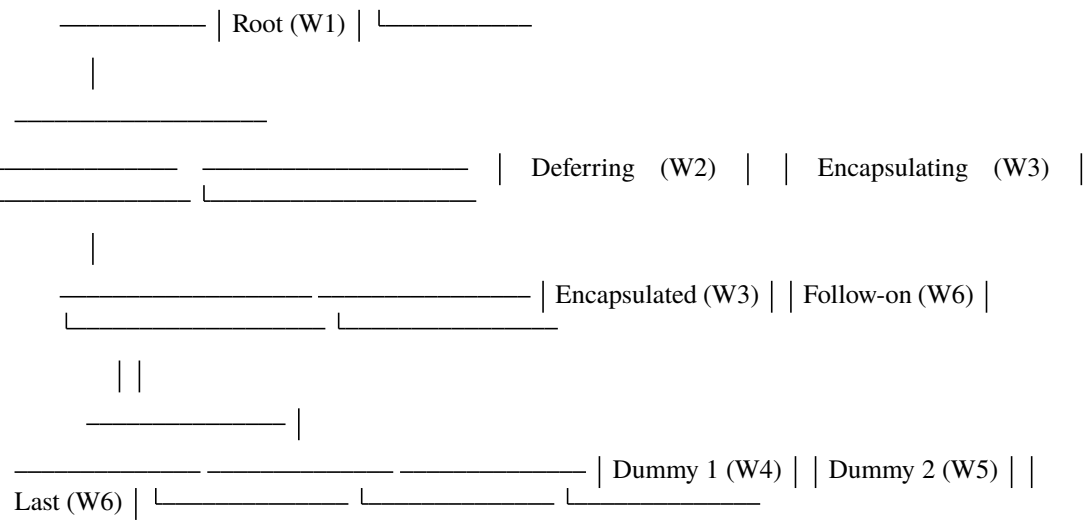
testUserTypesInJobFunctionArgs()

Test encapsulated, function-wrapping jobs where the function arguments reference user-defined types.

Mainly written to cover https://github.com/BD2KGenomics/toil/issues/1259 but then also revealed https://github.com/BD2KGenomics/toil/issues/1278.

testDeferralWithConcurrentEncapsulation()

Ensure that the following DAG succeeds:



The Wn numbers denote the worker processes that a particular job is run in. *Deferring* adds a deferred function and then runs for a long time. The deferred function will be present in the cache state for the duration of *Deferred*. *Follow-on* is the generic Job instance that's added by encapsulating a job. It runs on the same worker node but in a separate worker process, as the first job in that worker. Because ...

1) it is the first job in its worker process (the user script has not been made available on the sys.path by a previous job in that worker) and

2) it shares the cache state with the *Deferring* job and

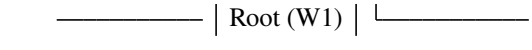
3) it is an instance of Job (and so does not introduce the user script to sys.path itself),

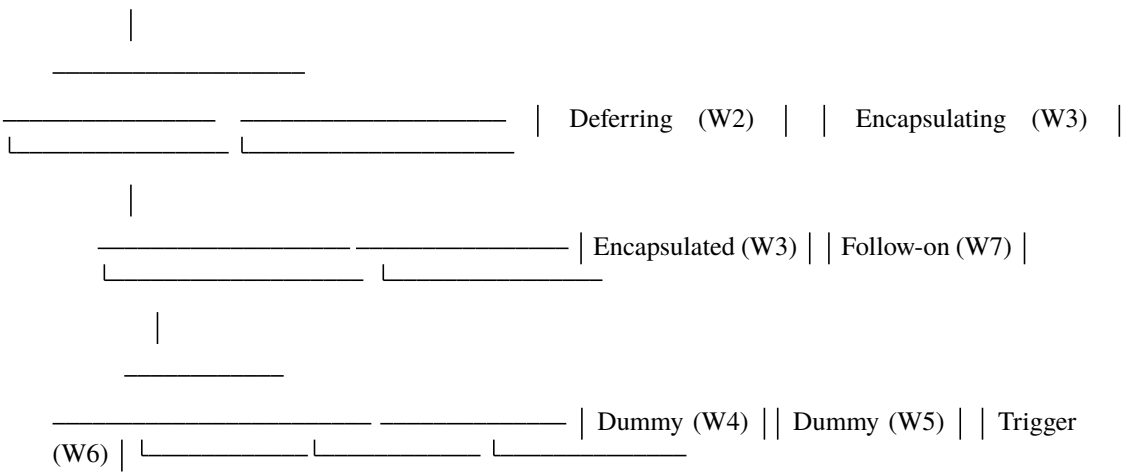
... it might cause problems with deserializing a defered function defined in the user script.

Encapsulated has two children to ensure that *Follow-on* is run in a separate worker.

testDeferralWithFailureAndEncapsulation()

Ensure that the following DAG succeeds:





Trigger causes *Deferring* to crash. *Follow-on* runs next, detects *Deferring*'s left-overs and runs the deferred function. *Follow-on* is an instance of *Job* and the first job in its worker process. This test ensures that despite these circumstances, the user script is loaded before the deferred functions defined in it are being run.

Encapsulated has two children to ensure that *Follow-on* is run in a new worker. That's the only way to guarantee that the user script has not been loaded yet, which would cause the test to succeed coincidentally. We want to test that auto-deploying and loading of the user script are done properly *before* deferred functions are being run and before any jobs have been executed by that worker.

toil.test.src.busTest

Module Contents

Classes

MessageBusTest	A common base class for Toil tests.

Functions

failing_job_fn(job)	This function is guaranteed to fail.

Attributes

logger

toil.test.src.busTest.**logger**

class toil.test.src.busTest.**MessageBusTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

test_enum_ints_in_file()

Make sure writing bus messages to files works with enums.

Return type
None

test_cross_thread_messaging()

Make sure message bus works across threads.

Return type
None

test_restart_without_bus_path()

Test the ability to restart a workflow when the message bus path used by the previous attempt is gone.

Return type
None

toil.test.src.busTest.**failing_job_fn**(*job*)

This function is guaranteed to fail.

Parameters
job (toil.job.Job) –

Return type
None

toil.test.src.checkpointTest

Module Contents

Classes

CheckpointTest	A common base class for Toil tests.
CheckRetryCount	Fail N times, succeed on the next try.
AlwaysFail	Class represents a unit of work in toil.
CheckpointFailsFirstTime	Class represents a unit of work in toil.
FailOnce	Fail the first time the workflow is run, but succeed thereafter.

class toil.test.src.checkpointTest.**CheckpointTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

testCheckpointNotRetried()

A checkpoint job should not be retried if the workflow has a retryCount of 0.

testCheckpointRetriedOnce()

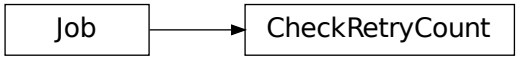
A checkpoint job should be retried exactly once if the workflow has a retryCount of 1.

testCheckpointedRestartSucceeds()

A checkpointed job should succeed on restart of a failed run if its child job succeeds.

class toil.test.src.checkpointTest.**CheckRetryCount**(*numFailuresBeforeSuccess*)

Bases: *toil.job.Job*



Fail N times, succeed on the next try.

getNumRetries(*fileStore*)

Mark a retry in the fileStore, and return the number of retries so far.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

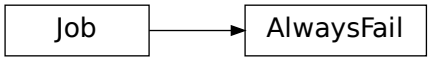
fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

class toil.test.src.checkpointTest.**AlwaysFail**(*memory=None*, *cores=None*, *disk=None*, *accelerators=None*, *preemptible=None*, *preemptable=None*, *unitName=''*, *checkpoint=False*, *displayName=''*, *descriptionClass=None*, *local=None*)

Bases: *toil.job.Job*



Class represents a unit of work in toil.

Parameters

- **memory** (*Optional[ParseableIndivisibleResource]*) –
- **cores** (*Optional[ParseableDivisibleResource]*) –
- **disk** (*Optional[ParseableIndivisibleResource]*) –
- **accelerators** (*Optional[ParseableAcceleratorRequirement]*) –
- **preemptible** (*Optional[ParseableFlag]*) –
- **preemptable** (*Optional[ParseableFlag]*) –
- **unitName** (*Optional[str]*) –
- **checkpoint** (*Optional[bool]*) –
- **displayName** (*Optional[str]*) –
- **descriptionClass** (*Optional[type]*) –
- **local** (*Optional[bool]*) –

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns
The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

class toil.test.src.checkpointTest.**CheckpointFailsFirstTime**

Bases: *toil.job.Job*

Class represents a unit of work in toil.

run(*fileStore*)

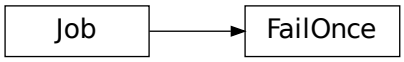
Override this function to perform work and dynamically create successor jobs.

Parameters
fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns
The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

class toil.test.src.checkpointTest.**FailOnce**(*memory=None*, *cores=None*, *disk=None*, *accelerators=None*, *preemptible=None*, *preemptable=None*, *unitName=''*, *checkpoint=False*, *displayName=''*, *descriptionClass=None*, *local=None*)

Bases: *toil.job.Job*



Fail the first time the workflow is run, but succeed thereafter.

Parameters

- **memory** (*Optional[ParseableIndivisibleResource]*) –
- **cores** (*Optional[ParseableDivisibleResource]*) –
- **disk** (*Optional[ParseableIndivisibleResource]*) –
- **accelerators** (*Optional[ParseableAcceleratorRequirement]*) –
- **preemptible** (*Optional[ParseableFlag]*) –
- **preemptable** (*Optional[ParseableFlag]*) –
- **unitName** (*Optional[str]*) –

- **checkpoint** (*Optional[bool]*) –
- **displayName** (*Optional[str]*) –
- **descriptionClass** (*Optional[type]*) –
- **local** (*Optional[bool]*) –

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

toil.test.src.deferredFunctionTest

Module Contents

Classes

DeferredFunctionTest	Test the deferred function system.

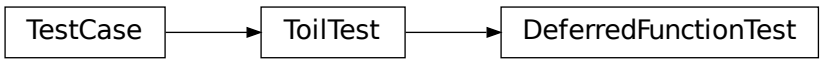
Attributes

logger

toil.test.src.deferredFunctionTest.**logger**

class toil.test.src.deferredFunctionTest.**DeferredFunctionTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Test the deferred function system.

jobStoreType = 'file'

setUp()

Hook method for setting up the test fixture before exercising it.

testDeferredFunctionRunsWithMethod()

Refer docstring in _testDeferredFunctionRuns. Test with Method

testDeferredFunctionRunsWithClassMethod()

Refer docstring in _testDeferredFunctionRuns. Test with Class Method

testDeferredFunctionRunsWithLambda()

Refer docstring in _testDeferredFunctionRuns. Test with Lambda

testDeferredFunctionRunsWithFailures()

Create 2 non local filesto use as flags. Create a job that registers a function that deletes one non local file. If that file exists, the job SIGKILLs itself. If it doesn't exist, the job registers a second deferred function to delete the second non local file and exits normally.

Initially the first file exists, so the job should SIGKILL itself and neither deferred function will run (in fact, the second should not even be registered). On the restart, the first deferred function should run and the first file should not exist, but the second one should. We assert the presence of the second, then register the second deferred function and exit normally. At the end of the test, neither file should exist.

Incidentally, this also tests for multiple registered deferred functions, and the case where a deferred function fails (since the first file doesn't exist on the retry).

testNewJobsCanHandleOtherJobDeaths()

Create 2 non-local files and then create 2 jobs. The first job registers a deferred job to delete the second non-local file, deletes the first non-local file and then kills itself. The second job waits for the first file to be deleted, then sleeps for a few seconds and then spawns a child. the child of the second does nothing. However starting it should handle the untimely demise of the first job and run the registered deferred function that deletes the first file. We assert the absence of the two files at the end of the run.

testBatchSystemCleanupCanHandleWorkerDeaths()

Create some non-local files. Create a job that registers a deferred function to delete the file and then kills its worker.

Assert that the file is missing after the pipeline fails, because we're using a single-machine batch system and the leader's batch system cleanup will find and run the deferred function.

toil.test.src.dockerCheckTest

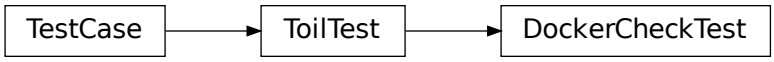
Module Contents

Classes

DockerCheckTest	Tests checking whether a docker image exists or not.

class toil.test.src.dockerCheckTest.**DockerCheckTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Tests checking whether a docker image exists or not.

testOfficialUbuntuRepo()

Image exists. This should pass.

testBroadDockerRepo()

Image exists. This should pass.

testBroadDockerRepoBadTag()

Bad tag. This should raise.

testNonexistentRepo()

Bad image. This should raise.

testToilQuayRepo()

Image exists. Should pass.

testBadQuayRepoNTag()

Bad repo and tag. This should raise.

testBadQuayRepo()

Bad repo. This should raise.

testBadQuayTag()

Bad tag. This should raise.

testGoogleRepo()

Image exists. Should pass.

testBadGoogleRepo()

Bad repo and tag. This should raise.

testApplianceParser()

Test that a specified appliance is parsed correctly.

toil.test.src.fileStoreTest

Module Contents

Classes

hidden	Hiding the abstract test classes from the Unittest loader so it can be inherited in different
NonCachingFileStoreTestWithFileJobStore	Abstract tests for the the various functions in
CachingFileStoreTestWithFileJobStore	Abstract tests for the the various cache-related functions in
NonCachingFileStoreTestWithAwsJobStore	Abstract tests for the the various functions in
CachingFileStoreTestWithAwsJobStore	Abstract tests for the the various cache-related functions in
NonCachingFileStoreTestWithGoogleJobStore	Abstract tests for the the various functions in
CachingFileStoreTestWithGoogleJobStore	Abstract tests for the the various cache-related functions in

Attributes

testingIsAutomatic
logger

toil.test.src.fileStoreTest.**testingIsAutomatic = True**

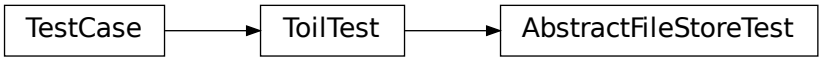
toil.test.src.fileStoreTest.**logger**

class toil.test.src.fileStoreTest.**hidden**

Hiding the abstract test classes from the Unittest loader so it can be inherited in different test suites for the different job stores.

class AbstractFileStoreTest(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



An abstract base class for testing the various general functions described in :class:toil.fileStores.abstractFileStore.AbstractFileStore

jobStoreType

setUp()

Hook method for setting up the test fixture before exercising it.

create_file(*content*, *executable=False*)

testToilIsNotBroken()

Runs a simple DAG to test if if any features other that caching were broken.

testFileStoreLogging()

Write a couple of files to the jobstore. Delete a couple of them. Read back written and locally deleted files.

testFileStoreOperations()

Write a couple of files to the jobstore. Delete a couple of them. Read back written and locally deleted files.

testWriteReadGlobalFilePermissions()

Ensures that uploaded files preserve their file permissions when they are downloaded again. This function checks that a written executable file maintains its executability after being read.

testWriteExportFileCompatibility()

Ensures that files created in a job preserve their executable permissions when they are exported from the leader.

testImportReadFileCompatibility()

Ensures that files imported to the leader preserve their executable permissions when they are read by the fileStore.

testReadWriteFileStreamTextMode()

Checks if text mode is compatible with file streams.

class AbstractNonCachingFileStoreTest(*methodName='runTest'*)

Bases: *hidden.AbstractFileStoreTest*

Abstract tests for the the various functions in :class:toil.fileStores.nonCachingFileStore.NonCachingFileStore. These tests are general enough that they can also be used for :class:toil.fileStores.CachingFileStore.

setUp()

Hook method for setting up the test fixture before exercising it.

class AbstractCachingFileStoreTest(*methodName='runTest'*)

Bases: *hidden.AbstractFileStoreTest*

Abstract tests for the the various cache-related functions in :class:toil.fileStores.cachingFileStore.CachingFileStore.

setUp()

Hook method for setting up the test fixture before exercising it.

testExtremeCacheSetup()

Try to create the cache with bad worker active and then have 10 child jobs try to run in the chain. This tests whether the cache is created properly even when the job crashes randomly.

testCacheEvictionPartialEvict()

Ensure the cache eviction happens as expected. Two files (20MB and 30MB) are written sequentially into the job store in separate jobs. The cache max is force set to 50MB. A Third Job requests 10MB of disk requiring eviction of the 1st file. Ensure that the behavior is as expected.

testCacheEvictionTotalEvict()

Ensure the cache eviction happens as expected. Two files (20MB and 30MB) are written sequentially into the job store in separate jobs. The cache max is force set to 50MB. A Third Job requests 10MB of disk requiring eviction of the 1st file. Ensure that the behavior is as expected.

testCacheEvictionFailCase()

Ensure the cache eviction happens as expected. Two files (20MB and 30MB) are written sequentially into the job store in separate jobs. The cache max is force set to 50MB. A Third Job requests 10MB of disk requiring eviction of the 1st file. Ensure that the behavior is as expected.

testAsyncWriteWithCaching()

Ensure the Async Writing of files happens as expected. The first Job forcefully modifies the cache size to 1GB. The second asks for 1GB of disk and writes a 900MB file into cache then rewrites it to the job store triggering an async write since the two unique jobstore IDs point to the same local file. Also, the second write is not cached since the first was written to cache, and there "isn't enough space" to cache the second. Imediately assert that the second write isn't cached, and is being asynchronously written to the job store.

Attempting to get the file from the jobstore should not fail.

testWriteNonLocalFileToJobStore()

Write a file not in localTempDir to the job store. Such a file should not be cached. Ensure the file is not cached.

testWriteLocalFileToJobStore()

Write a file from the localTempDir to the job store. Such a file will be cached by default. Ensure the file is cached.

testReadCacheMissFileFromJobStoreWithoutCachingReadFile()

Read a file from the file store that does not have a corresponding cached copy. Do not cache the read file. Ensure the number of links on the file are appropriate.

testReadCacheMissFileFromJobStoreWithCachingReadFile()

Read a file from the file store that does not have a corresponding cached copy. Cache the read file. Ensure the number of links on the file are appropriate.

testReadCachHitFileFromJobStore()

Read a file from the file store that has a corresponding cached copy. Ensure the number of links on the file are appropriate.

testMultipleJobsReadSameCacheHitGlobalFile()

Write a local file to the job store (hence adding a copy to cache), then have 10 jobs read it. Assert cached file size never goes up, assert unused job required disk space is always:

(a multiple of job reqs) - (number of current file readers * filesize).

At the end, assert the cache shows unused job-required space = 0.

testMultipleJobsReadSameCacheMissGlobalFile()

Write a non-local file to the job store(hence no cached copy), then have 10 jobs read it. Assert cached file size never goes up, assert unused job required disk space is always:
(a multiple of job reqs) - (number of current file readers * filesize).
At the end, assert the cache shows unused job-required space = 0.

testFileStoreExportFile()

testReturnFileSizes()

Write a couple of files to the jobstore. Delete a couple of them. Read back written and locally deleted files. Ensure that after every step that the cache is in a valid state.

testReturnFileSizesWithBadWorker()

Write a couple of files to the jobstore. Delete a couple of them. Read back written and locally deleted files. Ensure that after every step that the cache is in a valid state.

testControlledFailedWorkerRetry()

Conduct a couple of job store operations. Then die. Ensure that the restarted job is tracking values in the cache state file appropriately.

testRemoveLocalMutablyReadFile()

If a mutably read file is deleted by the user, it is ok.

testRemoveLocalImmutablyReadFile()

If an immutably read file is deleted by the user, it is not ok.

testDeleteLocalFile()

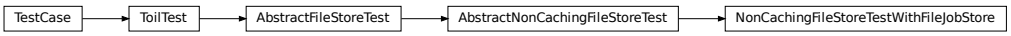
Test the deletion capabilities of deleteLocalFile

testSimultaneousReadsUncachedStream()

Test many simultaneous read attempts on a file created via a stream directly to the job store.

class toil.test.src.fileStoreTest.**NonCachingFileStoreTestWithFileJobStore**(*methodName='runTest'*)

Bases: *hidden*

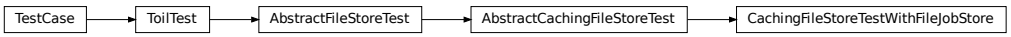


Abstract tests for the the various functions in :class:toil.fileStores.nonCachingFileStore.NonCachingFileStore. These tests are general enough that they can also be used for :class:toil.fileStores.CachingFileStore.

jobStoreType = 'file'

class toil.test.src.fileStoreTest.**CachingFileStoreTestWithFileJobStore**(*methodName='runTest'*)

Bases: *hidden*

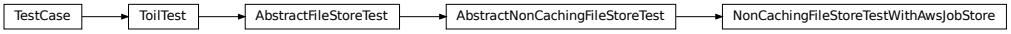


Abstract tests for the the various cache-related functions in :class:toil.fileStores.cachingFileStore.CachingFileStore.

jobStoreType = 'file'

class toil.test.src.fileStoreTest.**NonCachingFileStoreTestWithAwsJobStore**(*methodName='runTest'*)

Bases: *hidden*

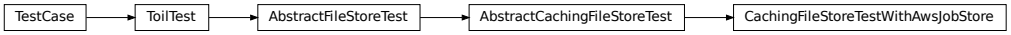


Abstract tests for the the various functions in :class:toil.fileStores.nonCachingFileStore.NonCachingFileStore. These tests are general enough that they can also be used for :class:toil.fileStores.CachingFileStore.

jobStoreType = 'aws'

class toil.test.src.fileStoreTest.**CachingFileStoreTestWithAwsJobStore**(*methodName='runTest'*)

Bases: *hidden*

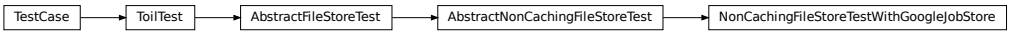


Abstract tests for the the various cache-related functions in :class:toil.fileStores.cachingFileStore.CachingFileStore.

jobStoreType = 'aws'

class toil.test.src.fileStoreTest.**NonCachingFileStoreTestWithGoogleJobStore**(*methodName='runTest'*)

Bases: *hidden*



Abstract tests for the the various functions in :class:toil.fileStores.nonCachingFileStore.NonCachingFileStore. These tests are general enough that they can also be used for :class:toil.fileStores.CachingFileStore.

jobStoreType = 'google'

class toil.test.src.fileStoreTest.**CachingFileStoreTestWithGoogleJobStore**(*methodName='runTest'*)

Bases: *hidden*

Abstract tests for the the various cache-related functions in :class:toil.fileStores.cachingFileStore.CachingFileStore.

jobStoreType = 'google'

toil.test.src.helloWorldTest

Module Contents

Classes

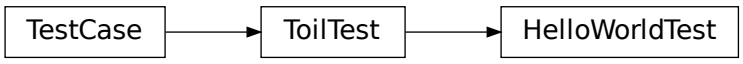
HelloWorldTest	A common base class for Toil tests.
HelloWorld	Class represents a unit of work in toil.
FollowOn	Class represents a unit of work in toil.

Functions

childFn(job)

class toil.test.src.helloWorldTest.**HelloWorldTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



A common base class for Toil tests.

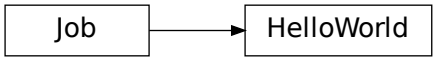
Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

testHelloWorld()

class toil.test.src.helloWorldTest.**HelloWorld**

Bases: *toil.job.Job*



Class represents a unit of work in toil.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

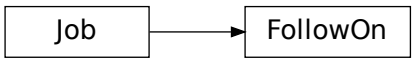
Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

toil.test.src.helloWorldTest.**childFn**(*job*)

class toil.test.src.helloWorldTest.**FollowOn**(*fileId*)

Bases: *toil.job.Job*



Class represents a unit of work in toil.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

toil.test.src.importExportFileTest

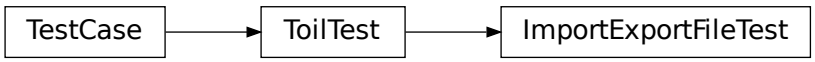
Module Contents

Classes

ImportExportFileTest	A common base class for Toil tests.
RestartingJob	Class represents a unit of work in toil.

class toil.test.src.importExportFileTest.**ImportExportFileTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

setUp()

Hook method for setting up the test fixture before exercising it.

create_file(*content*, *executable=False*)

test_import_export_restart_true()

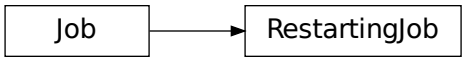
test_import_export_restart_false()

test_basic_import_export()

Ensures that uploaded files preserve their file permissions when they are downloaded again. This function checks that an imported executable file maintains its executability after being exported.

class toil.test.src.importExportFileTest.**RestartingJob**(*msg_portion_file_id*, *trigger_file_id*, *message_portion_2*)

Bases: *toil.job.Job*



Class represents a unit of work in toil.

run(*file_store*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

toil.test.src.jobDescriptionTest

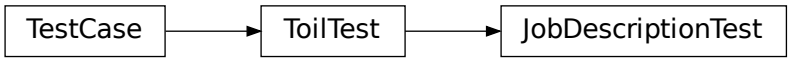
Module Contents

Classes

JobDescriptionTest	A common base class for Toil tests.

class toil.test.src.jobDescriptionTest.**JobDescriptionTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

setUp()

Hook method for setting up the test fixture before exercising it.

tearDown()

Hook method for deconstructing the test fixture after testing it.

testJobDescription()

Tests the public interface of a JobDescription.

testJobDescriptionSequencing()

toil.test.src.jobEncapsulationTest

Module Contents

Classes

JobEncapsulationTest	Tests testing the EncapsulationJob class.

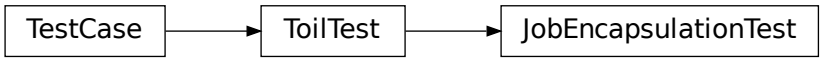
Functions

noOp()

encapsulatedJobFn(job, string, outFile)

class toil.test.src.jobEncapsulationTest.**JobEncapsulationTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Tests testing the EncapsulationJob class.

testEncapsulation()

Tests the Job.encapsulation method, which uses the EncapsulationJob class.

testAddChildEncapsulate()

Make sure that the encapsulate child does not have two parents with unique roots.

toil.test.src.jobEncapsulationTest.**noOp**()

toil.test.src.jobEncapsulationTest.**encapsulatedJobFn**(*job*, *string*, *outFile*)

toil.test.src.jobFileStoreTest

Module Contents

Classes

JobFileStoreTest	Tests testing the methods defined in :class:toil.fileStores.abstractFileStore.AbstractFileStore.

Functions

fileTestJob(job, inputFileStoreIDs, testStrings, ...)	Test job exercises toil.fileStores.abstractFileStore.AbstractFileStore functions
simpleFileStoreJob(job)	
fileStoreChild(job, testID1, testID2)	

Attributes

logger	
PREFIX_LENGTH	
fileStoreString	
streamingFileStoreString	

toil.test.src.jobFileStoreTest.**logger**

toil.test.src.jobFileStoreTest.**PREFIX_LENGTH = 200**

class toil.test.src.jobFileStoreTest.**JobFileStoreTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

Tests testing the methods defined in :class:toil.fileStores.abstractFileStore.AbstractFileStore.

testCachingFileStore()

testNonCachingFileStore()

testJobFileStore()

Tests case that about half the files are cached

testJobFileStoreWithBadWorker()

Tests case that about half the files are cached and the worker is randomly failing.

toil.test.src.jobFileStoreTest.**fileTestJob**(*job*, *inputFileStoreIDs*, *testStrings*, *chainLength*)

Test job exercises toil.fileStores.abstractFileStore.AbstractFileStore functions

toil.test.src.jobFileStoreTest.**fileStoreString = 'Testing writeGlobalFile'**

toil.test.src.jobFileStoreTest.**streamingFileStoreString = 'Testing writeGlobalFileStream'**

toil.test.src.jobFileStoreTest.**simpleFileStoreJob**(*job*)

toil.test.src.jobFileStoreTest.**fileStoreChild**(*job*, *testID1*, *testID2*)

toil.test.src.jobServiceTest

Module Contents

Classes

JobServiceTest	Tests testing the Job.Service class
PerfectServiceTest	Tests testing the Job.Service class
ToyService	Abstract class used to define the interface to a service.
ToySerializableService	Abstract class used to define the interface to a service.

Functions

serviceTest(job, outFile, messageInt)	Creates one service and one accessing job, which communicate with two files to establish
serviceTestRecursive(job, outFile, messages)	Creates a chain of services and accessing jobs, each paired together.
serviceTestParallelRecursive(job, outFiles, messageBundles)	Creates multiple chains of services and accessing jobs.
serviceAccessor(job, communicationFiles, outFile, randInt)	Writes a random integer iinto the inJobStoreFileID file, then tries 10 times reading
fnTest(strings, outputFile)	Function concatenates the strings together and writes them to the output file

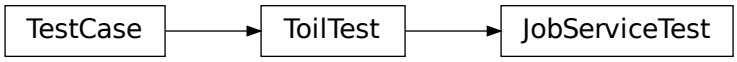
Attributes

logger

toil.test.src.jobServiceTest.**logger**

class toil.test.src.jobServiceTest.**JobServiceTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Tests testing the Job.Service class

testServiceSerialization()

Tests that a service can receive a promise without producing a serialization error.

testService(*checkpoint=False*)

Tests the creation of a Job.Service with random failures of the worker.

testServiceDeadlock()

Creates a job with more services than maxServices, checks that deadlock is detected.

testServiceWithCheckpoints()

Tests the creation of a Job.Service with random failures of the worker, making the root job use checkpointing to restart the subtree.

testServiceRecursive(*checkpoint=True*)

Tests the creation of a Job.Service, creating a chain of services and accessing jobs. Randomly fails the worker.

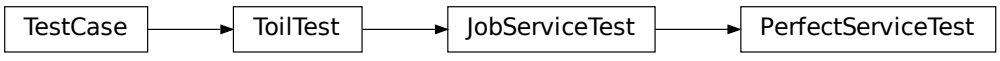
testServiceParallelRecursive(*checkpoint=True*)

Tests the creation of a Job.Service, creating parallel chains of services and accessing jobs. Randomly fails the worker.

runToil(*rootJob*, *retryCount=1*, *badWorker=0.5*, *badWorkedFailInterval=0.1*, *maxServiceJobs=sys.maxsize*, *deadlockWait=60*)

class toil.test.src.jobServiceTest.**PerfectServiceTest**(*methodName='runTest'*)

Bases: *JobServiceTest*



Tests testing the Job.Service class

runToil(*rootJob*, *retryCount=1*, *badWorker=0*, *badWorkedFailInterval=1000*, *maxServiceJobs=sys.maxsize*, *deadlockWait=60*)

Let us run all the tests in the other service test class, but without worker failures.

toil.test.src.jobServiceTest.**serviceTest**(*job*, *outFile*, *messageInt*)

Creates one service and one accessing job, which communicate with two files to establish that both run concurrently.

toil.test.src.jobServiceTest.**serviceTestRecursive**(*job*, *outFile*, *messages*)

Creates a chain of services and accessing jobs, each paired together.

toil.test.src.jobServiceTest.**serviceTestParallelRecursive**(*job*, *outFiles*, *messageBundles*)

Creates multiple chains of services and accessing jobs.

class toil.test.src.jobServiceTest.**ToyService**(*messageInt*, **args*, ***kwargs*)

Bases: *toil.job.Job.Service*

Abstract class used to define the interface to a service.

Should be subclassed by the user to define services.

Is not executed as a job; runs within a ServiceHostJob.

start(*job*)

Start the service.

Parameters

job – The underlying host job that the service is being run in. Can be used to register deferred functions, or to access the fileStore for creating temporary files.

Returns

An object describing how to access the service. The object must be pickleable and will be used by jobs to access the service (see *toil.job.Job.addService()*).

stop(*job*)

Stops the service. Function can block until complete.

Parameters

job – The underlying host job that the service is being run in. Can be used to register deferred functions, or to access the fileStore for creating temporary files.

check()

Checks the service is still running.

Raises

exceptions.RuntimeError – If the service failed, this will cause the service job to be labeled failed.

Returns
True if the service is still running, else False. If False then the service job will be terminated, and considered a success. Important point: if the service job exits due to a failure, it should raise a RuntimeError, not return False!

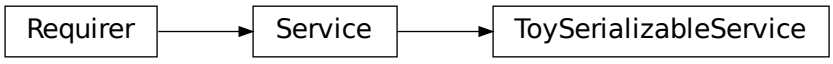
static serviceWorker(*jobStore*, *terminate*, *error*, *inJobStoreID*, *outJobStoreID*, *messageInt*)

toil.test.src.jobServiceTest.**serviceAccessor**(*job*, *communicationFiles*, *outFile*, *randInt*)

Writes a random integer iinto the inJobStoreFileID file, then tries 10 times reading from outJobStoreFileID to get a pair of integers, the first equal to i the second written into the outputFile.

class toil.test.src.jobServiceTest.**ToySerializableService**(*messageInt*, **args*, ***kwargs*)

Bases: *toil.job.Job.Service*



Abstract class used to define the interface to a service.

Should be subclassed by the user to define services.

Is not executed as a job; runs within a ServiceHostJob.

start(*job*)

Start the service.

Parameters
job – The underlying host job that the service is being run in. Can be used to register deferred functions, or to access the fileStore for creating temporary files.

Returns
An object describing how to access the service. The object must be pickleable and will be used by jobs to access the service (see *toil.job.Job.addService()*).

stop(*job*)

Stops the service. Function can block until complete.

Parameters
job – The underlying host job that the service is being run in. Can be used to register deferred functions, or to access the fileStore for creating temporary files.

check()

Checks the service is still running.

Raises
exceptions.RuntimeError – If the service failed, this will cause the service job to be labeled failed.

Returns
True if the service is still running, else False. If False then the service job will be terminated, and considered a success. Important point: if the service job exits due to a failure, it should raise a RuntimeError, not return False!

toil.test.src.jobServiceTest.**fnTest**(*strings*, *outputFile*)

Function concatenates the strings together and writes them to the output file

toil.test.src.jobTest

Module Contents

Classes

JobTest	Tests the job class.
TrivialService	Abstract class used to define the interface to a service.

Functions

simpleJobFn(job, value)	
fn1Test(string, outputFile)	Function appends the next character after the last character in the given
fn2Test(pStrings, s, outputFile)	Function concatenates the strings in pStrings and s, in that order, and writes the result to
trivialParent(job)	
parent(job)	
diamond(job)	
child(job)	
errorChild(job)	

Attributes

logger

toil.test.src.jobTest.**logger**

class toil.test.src.jobTest.**JobTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

Tests the job class.

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

testStatic()

Create a DAG of jobs non-dynamically and run it. DAG is:

A -> F -—— B -> D

——- C -> E

Follow on is marked by ->

testStatic2()

Create a DAG of jobs non-dynamically and run it. DAG is:

A -> F -—— B -> D

——- C -> E

Follow on is marked by ->

testTrivialDAGConsistency()

testDAGConsistency()

testSiblingDAGConsistency()

Slightly more complex case. The stranded job's predecessors are siblings instead of parent/child.

testDeadlockDetection()

Randomly generate job graphs with various types of cycle in them and check they cause an exception properly. Also check that multiple roots causes a deadlock exception.

testNewCheckpointIsLeafVertexNonRootCase()

Test for issue #1465: Detection of checkpoint jobs that are not leaf vertices identifies leaf vertices incorrectly

Test verification of new checkpoint jobs being leaf verticies, starting with the following baseline workflow:

Parent

Child # Checkpoint=True

testNewCheckpointIsLeafVertexRootCase()

Test for issue #1466: Detection of checkpoint jobs that are not leaf vertices
omits the workflow root job

Test verification of a new checkpoint job being leaf vertex, starting with a baseline workflow of a single, root job:

Root # Checkpoint=True

runNewCheckpointIsLeafVertexTest(*createWorkflowFn*)

Test verification that a checkpoint job is a leaf vertex using both valid and invalid cases.

Parameters

createWorkflowFn – function to create and new workflow and return a tuple of:

0) the workflow root job

1) a checkpoint job to test within the workflow

runCheckpointVertexTest(*workflowRootJob*, *checkpointJob*, *checkpointJobService=None*, *checkpointJobChild=None*, *checkpointJobFollowOn=None*, *expectedException=None*)

Modifies the checkpoint job according to the given parameters then runs the workflow, checking for the expected exception, if any.

testEvaluatingRandomDAG()

Randomly generate test input then check that the job graph can be run successfully, using the existence of promises to validate the run.

static getRandomEdge(*nodeNumber*)

static makeRandomDAG(*nodeNumber*)

Makes a random dag with "nodeNumber" nodes in which all nodes are connected. Return value is list of edges, each of form (a, b), where a and b are integers >= 0 < nodeNumber referring to nodes and the edge is from a to b.

static getAdjacencyList(*nodeNumber*, *edges*)

Make adjacency list representation of edges

reachable(*node*, *adjacencyList*, *followOnAdjacencyList=None*)

Find the set of nodes reachable from this node (including the node). Return is a set of integers.

addRandomFollowOnEdges(*childAdjacencyList*)

Adds random follow on edges to the graph, represented as an adjacency list. The follow on edges are returned as a set and their augmented edges are added to the adjacency list.

makeJobGraph(*nodeNumber*, *childEdges*, *followOnEdges*, *outPath*, *addServices=True*)

Converts a DAG into a job graph. childEdges and followOnEdges are the lists of child and followOn edges.

isAcyclic(*adjacencyList*)

Returns true if there are any cycles in the graph, which is represented as an adjacency list.

toil.test.src.jobTest.**simpleJobFn**(*job*, *value*)

toil.test.src.jobTest.**fn1Test**(*string*, *outputFile*)

Function appends the next character after the last character in the given string to the string, writes the string to a file, and returns it. For example, if string is "AB", we will write and return "ABC".

toil.test.src.jobTest.**fn2Test**(*pStrings*, *s*, *outputFile*)

Function concatenates the strings in pStrings and s, in that order, and writes the result to the output file. Returns s.

toil.test.src.jobTest.**trivialParent**(*job*)

toil.test.src.jobTest.**parent**(*job*)

toil.test.src.jobTest.**diamond**(*job*)

toil.test.src.jobTest.**child**(*job*)

toil.test.src.jobTest.**errorChild**(*job*)

class toil.test.src.jobTest.**TrivialService**(*message*, **args*, ***kwargs*)

Bases: *toil.job.Job.Service*

Abstract class used to define the interface to a service.

Should be subclassed by the user to define services.

Is not executed as a job; runs within a ServiceHostJob.

start(*job*)

Start the service.

Parameters

job – The underlying host job that the service is being run in. Can be used to register deferred functions, or to access the fileStore for creating temporary files.

Returns

An object describing how to access the service. The object must be pickleable and will be used by jobs to access the service (see *toil.job.Job.addService()*).

stop(*job*)

Stops the service. Function can block until complete.

Parameters

job – The underlying host job that the service is being run in. Can be used to register deferred functions, or to access the fileStore for creating temporary files.

check()

Checks the service is still running.

Raises

exceptions.RuntimeError – If the service failed, this will cause the service job to be labeled failed.

Returns

True if the service is still running, else False. If False then the service job will be terminated, and considered a success. Important point: if the service job exits due to a failure, it should raise a RuntimeError, not return False!

toil.test.src.miscTests

Module Contents

Classes

MiscTests	This class contains miscellaneous tests that don't have enough content to be their own test
TestPanic	A common base class for Toil tests.

Attributes

log

toil.test.src.miscTests.**log**

class toil.test.src.miscTests.**MiscTests**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

This class contains miscellaneous tests that don't have enough content to be their own test file, and that don't logically fit in with any of the other test suites.

setUp()

Hook method for setting up the test fixture before exercising it.

testIDStability()

testGetSizeOfDirectoryWorks()

A test to make sure toil.common.getDirSizeRecursively does not underestimate the amount of disk space needed.

Disk space allocation varies from system to system. The computed value should always be equal to or slightly greater than the creation value. This test generates a number of random directories and randomly sized files to test this using getDirSizeRecursively.

test_atomic_install()

test_atomic_install_dev()

test_atomic_context_ok()

test_atomic_context_error()

test_call_command_ok()

test_call_command_err()

class toil.test.src.miscTests.**TestPanic**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

test_panic_by_hand()

test_panic()

test_panic_with_secondary()

test_nested_panic()

try_and_panic_by_hand()

try_and_panic()

try_and_panic_with_secondary()

try_and_nested_panic_with_secondary()

toil.test.src.promisedRequirementTest

Module Contents

Classes

hidden	Hide abstract base class from unittest's test case loader.
SingleMachinePromisedRequirementsTest	Tests against the SingleMachine batch system
MesosPromisedRequirementsTest	Tests against the Mesos batch system

Functions

maxConcurrency(job, cpuCount, filename, coresPerJob)	Returns the max number of concurrent tasks when using a PromisedRequirement instance
getOne()	
getThirtyTwoMb()	
logDiskUsage(job, funcName[, sleep])	Logs the job's disk usage to master and sleeps for specified amount of time.

Attributes

log

toil.test.src.promisedRequirementTest.**log**

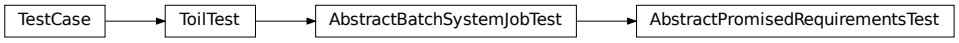
class toil.test.src.promisedRequirementTest.**hidden**

Hide abstract base class from unittest's test case loader.

http://stackoverflow.com/questions/1323455/python-unit-test-with-base-and-sub-class#answer-25695512

class AbstractPromisedRequirementsTest(*methodName='runTest'*)

Bases: *toil.test.batchSystems.batchSystemTest.hidden.AbstractBatchSystemJobTest*



An abstract base class for testing Toil workflows with promised requirements.

testConcurrencyDynamic()

Asserts that promised core resources are allocated properly using a dynamic Toil workflow

testConcurrencyStatic()

Asserts that promised core resources are allocated properly using a static DAG

getOptions(*tempDir*, *caching=True*)

Configures options for Toil workflow and makes job store. :param str tempDir: path to test directory :return: Toil options object

getCounterPath(*tempDir*)

Returns path to a counter file :param str tempDir: path to test directory :return: path to counter file

testPromisesWithJobStoreFileObjects(*caching=True*)

Check whether FileID objects are being pickled properly when used as return values of functions. Then ensure that lambdas of promised FileID objects can be used to describe the requirements of a subsequent job. This type of operation will be used commonly in Toil scripts. :return: None

testPromisesWithNonCachingFileStore()

testPromiseRequirementRaceStatic()

Checks for a race condition when using promised requirements and child job functions.

toil.test.src.promisedRequirementTest.**maxConcurrency**(*job*, *cpuCount*, *filename*, *coresPerJob*)

Returns the max number of concurrent tasks when using a PromisedRequirement instance to allocate the number of cores per job.

Parameters

- **cpuCount** (*int*) – number of available cpus
- **filename** (*str*) – path to counter file
- **coresPerJob** (*int*) – number of cores assigned to each job

Return int max concurrency value

toil.test.src.promisedRequirementTest.**getOne**()

toil.test.src.promisedRequirementTest.**getThirtyTwoMb**()

toil.test.src.promisedRequirementTest.**logDiskUsage**(*job*, *funcName*, *sleep=0*)

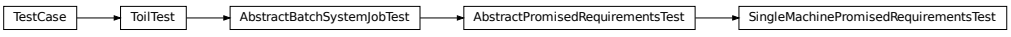
Logs the job's disk usage to master and sleeps for specified amount of time.

Returns

job function's disk usage

class toil.test.src.promisedRequirementTest.**SingleMachinePromisedRequirementsTest**(*methodName='runTest'*)

Bases: *hidden*



Tests against the SingleMachine batch system

getBatchSystemName()

Return type

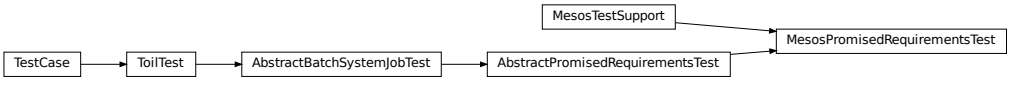
(str, *AbstractBatchSystem*)

tearDown()

Hook method for deconstructing the test fixture after testing it.

class toil.test.src.promisedRequirementTest.**MesosPromisedRequirementsTest**(*methodName='runTest'*)

Bases: *hidden*, *toil.batchSystems.mesos.test.MesosTestSupport*



Tests against the Mesos batch system

getOptions(*tempDir*, *caching=True*)

Configures options for Toil workflow and makes job store. :param str tempDir: path to test directory :return: Toil options object

getBatchSystemName()

Return type
(str, *AbstractBatchSystem*)

tearDown()

Hook method for deconstructing the test fixture after testing it.

toil.test.src.promisesTest

Module Contents

Classes

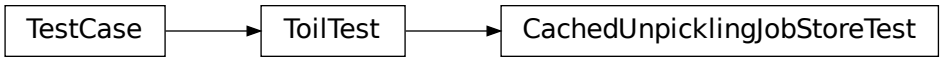
CachedUnpicklingJobStoreTest	A common base class for Toil tests.
ChainedIndexedPromisesTest	A common base class for Toil tests.
PathIndexingPromiseTest	Test support for indexing promises of arbitrarily nested data structures of lists, dicts and

Functions

parent(job)	
child()	
a(job)	
b(job)	
c()	
d(job)	
e()	

class toil.test.src.promisesTest.**CachedUnpicklingJobStoreTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

test()

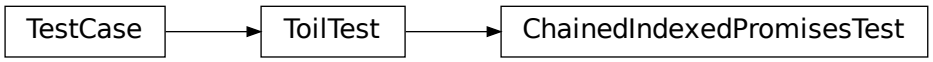
Runs two identical Toil workflows with different job store paths

toil.test.src.promisesTest.**parent**(*job*)

toil.test.src.promisesTest.**child**()

class toil.test.src.promisesTest.**ChainedIndexedPromisesTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

test()

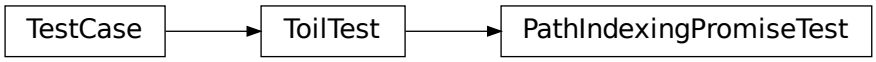
toil.test.src.promisesTest.**a**(*job*)

toil.test.src.promisesTest.**b**(*job*)

toil.test.src.promisesTest.**c**()

class toil.test.src.promisesTest.**PathIndexingPromiseTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Test support for indexing promises of arbitrarily nested data structures of lists, dicts and tuples, or any other object supporting the __getitem__() protocol.

test()

toil.test.src.promisesTest.**d**(*job*)

toil.test.src.promisesTest.**e**()

toil.test.src.realtimeLoggerTest

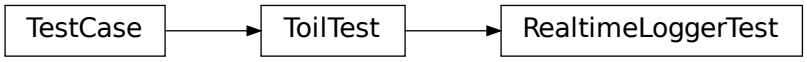
Module Contents

Classes

RealtimeLoggerTest	A common base class for Toil tests.
MessageDetector	Detect the secret message and set a flag.
LogTest	Class represents a unit of work in toil.

class toil.test.src.realtimeLoggerTest.**RealtimeLoggerTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

testRealtimeLogger()

class toil.test.src.realtimeLoggerTest.**MessageDetector**

Bases: logging.StreamHandler

Detect the secret message and set a flag.

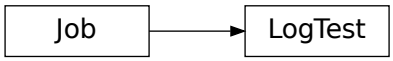
emit(*record*)

Emit a record.

If a formatter is specified, it is used to format the record. The record is then written to the stream with a trailing newline. If exception information is present, it is formatted using traceback.print_exception and appended to the stream. If the stream has an 'encoding' attribute, it is used to determine how to do the output to the stream.

class toil.test.src.realtimeLoggerTest.**LogTest**

Bases: *toil.job.Job*



Class represents a unit of work in toil.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

toil.test.src.regularLogTest

Module Contents

Classes

RegularLogTest	A common base class for Toil tests.

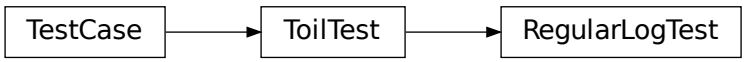
Attributes

logger

toil.test.src.regularLogTest.**logger**

class toil.test.src.regularLogTest.**RegularLogTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

setUp()

Hook method for setting up the test fixture before exercising it.

Return type
None

testLogToMaster()

testWriteLogs()

testWriteGzipLogs()

testMultipleLogToMaster()

testRegularLog()

toil.test.src.resourceTest

Module Contents

Classes

ResourceTest	Test module descriptors and resources derived from them.

Functions

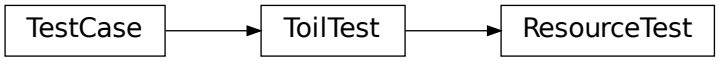
tempFileContaining(content[, suffix])	Write a file with the given contents, and keep it on disk as long as the context is active.

toil.test.src.resourceTest.**tempFileContaining**(*content*, *suffix=''*)

Write a file with the given contents, and keep it on disk as long as the context is active. :param str content: The contents of the file. :param str suffix: The extension to use for the temporary file.

class toil.test.src.resourceTest.**ResourceTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Test module descriptors and resources derived from them.

testStandAlone()

testPackage()

testVirtualEnv()

testStandAloneInPackage()

testBuiltIn()

testNonPyStandAlone()

Asserts that Toil enforces the user script to have a .py or .pyc extension because that's the only way auto-deployment can re-import the module on a worker. See

https://github.com/BD2KGenomics/toil/issues/631 and https://github.com/BD2KGenomics/toil/issues/858

toil.test.src.restartDAGTest

Module Contents

Classes

RestartDAGTest	Tests that restarted job DAGs don't run children of jobs that failed in the first run till the

Functions

passingFn(job[, fileName])	This function is guaranteed to pass as it does nothing out of the ordinary. If fileName is
failingFn(job, failType, fileName)	This function is guaranteed to fail via a raised assertion, or an os.kill

Attributes

logger

toil.test.src.restartDAGTest.**logger**

class toil.test.src.restartDAGTest.**RestartDAGTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

Tests that restarted job DAGs don't run children of jobs that failed in the first run till the parent completes successfully in the restart.

setUp()

Hook method for setting up the test fixture before exercising it.

tearDown()

Hook method for deconstructing the test fixture after testing it.

testRestartedWorkflowSchedulesCorrectJobsOnFailedParent()

testRestartedWorkflowSchedulesCorrectJobsOnKilledParent()

toil.test.src.restartDAGTest.**passingFn**(*job*, *fileName=None*)

This function is guaranteed to pass as it does nothing out of the ordinary. If fileName is provided, it will be created.

Parameters

fileName (*str*) – The name of a file that must be created if provided.

toil.test.src.restartDAGTest.**failingFn**(*job*, *failType*, *fileName*)

This function is guaranteed to fail via a raised assertion, or an os.kill

Parameters

- **job** – Job
- **failType** (*str*) – 'raise' or 'kill
- **fileName** (*str*) – The name of a file that must be created.

toil.test.src.resumabilityTest

Module Contents

Classes

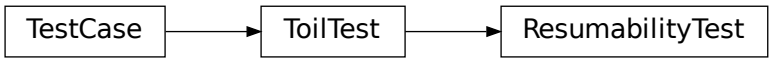
ResumabilityTest	https://github.com/BD2KGenomics/toil/issues/808

Functions

parent(job)	Set up a bunch of dummy child jobs, and a bad job that needs to be
goodChild(job)	Does nothing.
badChild(job)	Fails the first time it's run, succeeds the second time.

class toil.test.src.resumabilityTest.**ResumabilityTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



https://github.com/BD2KGenomics/toil/issues/808

test()

Tests that a toil workflow that fails once can be resumed without a NoSuchJobException.

toil.test.src.resumabilityTest.**parent**(*job*)

Set up a bunch of dummy child jobs, and a bad job that needs to be restarted as the follow on.

toil.test.src.resumabilityTest.**goodChild**(*job*)

Does nothing.

toil.test.src.resumabilityTest.**badChild**(*job*)

Fails the first time it's run, succeeds the second time.

toil.test.src.retainTempDirTest

Module Contents

Classes

CleanWorkDirTest	Tests testing :class:toil.fileStores.abstractFileStore.AbstractFileStore

Functions

tempFileTestJob(job)
tempFileTestErrorJob(job)

class toil.test.src.retainTempDirTest.**CleanWorkDirTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

Tests testing :class:toil.fileStores.abstractFileStore.AbstractFileStore

setUp()

Hook method for setting up the test fixture before exercising it.

tearDown()

Hook method for deconstructing the test fixture after testing it.

testNever()

testAlways()

testOnErrorWithError()

testOnErrorWithNoError()

testOnSuccessWithError()

testOnSuccessWithSuccess()

toil.test.src.retainTempDirTest.**tempFileTestJob**(*job*)

toil.test.src.retainTempDirTest.**tempFileTestErrorJob**(*job*)

toil.test.src.systemTest

Module Contents

Classes

SystemTest	Test various assumptions about the operating system's behavior.

class toil.test.src.systemTest.**SystemTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

Test various assumptions about the operating system's behavior.

testAtomicityOfNonEmptyDirectoryRenames()

toil.test.src.threadingTest

Module Contents

Classes

ThreadingTest	Test Toil threading/synchronization tools.

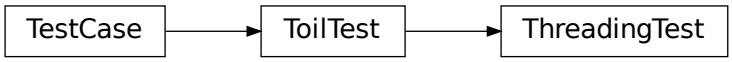
Attributes

log

toil.test.src.threadingTest.**log**

class toil.test.src.threadingTest.**ThreadingTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Test Toil threading/synchronization tools.

testGlobalMutexOrdering()

testLastProcessStanding()

toil.test.src.toilContextManagerTest

Module Contents

Classes

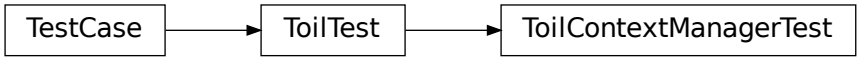
ToilContextManagerTest	A common base class for Toil tests.
HelloWorld	Class represents a unit of work in toil.
FollowOn	Class represents a unit of work in toil.

Functions

childFn(job)

class toil.test.src.toilContextManagerTest.**ToilContextManagerTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The

path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

setUp()

Hook method for setting up the test fixture before exercising it.

tearDown()

Hook method for deconstructing the test fixture after testing it.

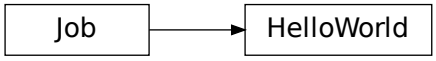
testContextManger()

testNoContextManger()

testExportAfterFailedExport()

class toil.test.src.toilContextManagerTest.**HelloWorld**

Bases: *toil.job.Job*



Class represents a unit of work in toil.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

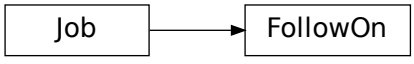
Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

toil.test.src.toilContextManagerTest.**childFn**(*job*)

class toil.test.src.toilContextManagerTest.**FollowOn**(*fileId*)

Bases: *toil.job.Job*



Class represents a unit of work in toil.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

toil.test.src.userDefinedJobArgTypeTest

Module Contents

Classes

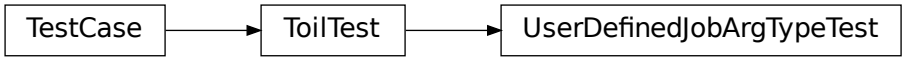
UserDefinedJobArgTypeTest	Test for issue #423 (Toil can't unpickle classes defined in user scripts) and variants
JobClass	Class represents a unit of work in toil.
Foo	

Functions

jobFunction(job, level, foo)	
main()	

class toil.test.src.userDefinedJobArgTypeTest.**UserDefinedJobArgTypeTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Test for issue #423 (Toil can't unpickle classes defined in user scripts) and variants thereof.

https://github.com/BD2KGenomics/toil/issues/423

setUp()

Hook method for setting up the test fixture before exercising it.

testJobFunction()

Test with first job being a function

testJobClass()

Test with first job being an instance of a class

testJobFunctionFromMain()

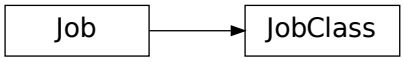
Test with first job being a function defined in __main__

testJobClassFromMain()

Test with first job being an instance of a class defined in __main__

class toil.test.src.userDefinedJobArgTypeTest.**JobClass**(*level*, *foo*)

Bases: *toil.job.Job*



Class represents a unit of work in toil.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

toil.test.src.userDefinedJobArgTypeTest.**jobFunction**(*job*, *level*, *foo*)

class toil.test.src.userDefinedJobArgTypeTest.**Foo**

assertIsCopy()

toil.test.src.userDefinedJobArgTypeTest.**main**()

toil.test.src.workerTest

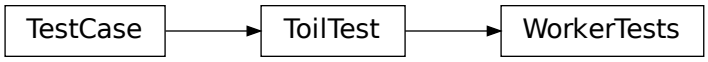
Module Contents

Classes

WorkerTests	Test miscellaneous units of the worker.

class toil.test.src.workerTest.**WorkerTests**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



Test miscellaneous units of the worker.

setUp()

Hook method for setting up the test fixture before exercising it.

testNextChainable()

Make sure chainable/non-chainable jobs are identified correctly.

toil.test.utils

Submodules

toil.test.utils.toilDebugTest

A set of test cases for toilwdl.py

Module Contents

Functions

workflow_debug_jobstore(tmp_path)	
testJobStoreContents(workflow_debug_jobstore)	Test toilDebugFile.printContentsOfJobStore().
fetchFiles(symLink, jobStoreDir, outputDir)	Fn for testFetchJobStoreFiles() and testFetchJobStoreFilesWSymlinks().
testFetchJobStoreFiles(tmp_path, workflow_debug_jobstore)	Test toilDebugFile.fetchJobStoreFiles() without using symlinks.
testFetchJobStoreFilesWSymlinks(tmp_path, ...)	Test toilDebugFile.fetchJobStoreFiles() using symlinks.

Attributes

logger	

toil.test.utils.toilDebugTest.**logger**

toil.test.utils.toilDebugTest.**workflow_debug_jobstore**(*tmp_path*)

Parameters

tmp_path (*pathlib.Path*) –

Return type
str

toil.test.utils.toilDebugTest.**testJobStoreContents**(*workflow_debug_jobstore*)

Test toilDebugFile.printContentsOfJobStore().

Runs a workflow that imports 'B.txt' and 'mkFile.py' into the jobStore. 'A.txt', 'C.txt', 'ABC.txt' are then created. This checks to make sure these contents are found in the jobStore and printed.

Parameters
workflow_debug_jobstore (*str*) –

toil.test.utils.toilDebugTest.**fetchFiles**(*symLink*, *jobStoreDir*, *outputDir*)

Fn for testFetchJobStoreFiles() and testFetchJobStoreFilesWSymlinks().

Runs a workflow that imports 'B.txt' and 'mkFile.py' into the jobStore. 'A.txt', 'C.txt', 'ABC.txt' are then created. This test then attempts to get a list of these files and copy them over into our output diectory from the jobStore, confirm that they are present, and then delete them.

Parameters
jobStoreDir (*str*) –

toil.test.utils.toilDebugTest.**testFetchJobStoreFiles**(*tmp_path*, *workflow_debug_jobstore*)

Test toilDebugFile.fetchJobStoreFiles() without using symlinks.

Parameters

- **tmp_path** (*pathlib.Path*) –
- **workflow_debug_jobstore** (*str*) –

Return type
None

toil.test.utils.toilDebugTest.**testFetchJobStoreFilesWSymlinks**(*tmp_path*, *workflow_debug_jobstore*)

Test toilDebugFile.fetchJobStoreFiles() using symlinks.

Parameters

- **tmp_path** (*pathlib.Path*) –
- **workflow_debug_jobstore** (*str*) –

Return type
None

toil.test.utils.toilKillTest

Module Contents

Classes

ToilKillTest	A set of test cases for "toil kill".
ToilKillTestWithAWSJobStore	A set of test cases for "toil kill" using the AWS job store.

Attributes

logger

pkg_root

toil.test.utils.toilKillTest.**logger**

toil.test.utils.toilKillTest.**pkg_root**

class toil.test.utils.toilKillTest.**ToilKillTest**(**args*, ***kwargs*)

Bases: *toil.test.ToilTest*

A set of test cases for "toil kill".

setUp()

Shared test variables.

tearDown()

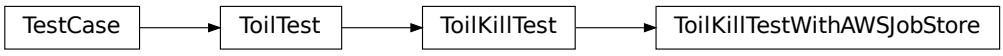
Default tearDown for unittest.

test_cwl_toil_kill()

Test "toil kill" on a CWL workflow with a 100 second sleep.

class toil.test.utils.toilKillTest.**ToilKillTestWithAWSJobStore**(**args*, ***kwargs*)

Bases: *ToilKillTest*



A set of test cases for "toil kill" using the AWS job store.

toil.test.utils.utilsTest

Module Contents

Classes

UtilsTest	Tests the utilities that toil ships with, e.g. stats and status, in conjunction with restart
RunTwoJobsPerWorker	Runs child job with same resources as self in an attempt to chain the jobs on the same worker

Functions

printUnicodeCharacter()

Attributes

pkg_root
logger

toil.test.utils.utilsTest.**pkg_root**

toil.test.utils.utilsTest.**logger**

class toil.test.utils.utilsTest.**UtilsTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

Tests the utilities that toil ships with, e.g. stats and status, in conjunction with restart functionality.

property toilMain

property cleanCommand

property statsCommand

setUp()

Hook method for setting up the test fixture before exercising it.

tearDown()

Hook method for deconstructing the test fixture after testing it.

statusCommand(*failIfNotComplete=False*)

testAWSProvisionerUtils()

Runs a number of the cluster utilities in sequence.

Launches a cluster with custom tags. Verifies the tags exist. ssh's into the cluster. Does some weird string comparisons. Makes certain that TOIL_WORKDIR is set as expected in the ssh'ed cluster. Rsyncs a file and verifies it exists on the leader. Destroys the cluster.

Returns

testUtilsSort()

Tests the status and stats commands of the toil command line utility using the sort example with the –restart flag.

testUtilsStatsSort()

Tests the stats commands on a complete run of the stats test.

testUnicodeSupport()

testMultipleJobsPerWorkerStats()

Tests case where multiple jobs are run on 1 worker to ensure that all jobs report back their data

check_status(*status*, *status_fn*, *seconds=20*)

testGetPIDStatus()

Test that ToilStatus.getPIDStatus() behaves as expected.

testGetStatusFailedToilWF()

Test that ToilStatus.getStatus() behaves as expected with a failing Toil workflow. While this workflow could be called by importing and evoking its main function, doing so would remove the opportunity to test the 'RUNNING' functionality of getStatus().

testGetStatusFailedCWLWF()

Test that ToilStatus.getStatus() behaves as expected with a failing CWL workflow.

testGetStatusSuccessfulCWLWF()

Test that ToilStatus.getStatus() behaves as expected with a successful CWL workflow.

testPrintJobLog(*mock_print*)

Test that ToilStatus.printJobLog() reads the log from a failed command without error.

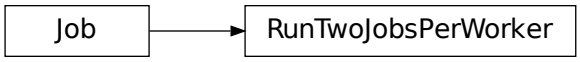
testRestartAttribute()

Test that the job store is only destroyed when we observe a succcessful workflow run. The following simulates a failing workflow that attempts to resume without restart(). In this case, the job store should not be destroyed until restart() is called.

toil.test.utils.utilsTest.**printUnicodeCharacter**()

class toil.test.utils.utilsTest.**RunTwoJobsPerWorker**

Bases: *toil.job.Job*



Runs child job with same resources as self in an attempt to chain the jobs on the same worker

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

toil.test.wdl

Submodules

toil.test.wdl.builtinTest

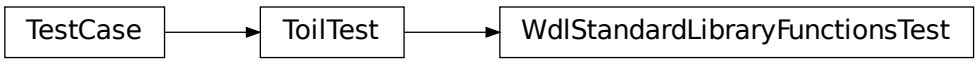
Module Contents

Classes

WdlStandardLibraryFunctionsTest	A set of test cases for toil's wdl functions.
WdlWorkflowsTest	A set of test cases for toil's conformance with WDL.
WdlLanguageSpecWorkflowsTest	A set of test cases for toil's conformance with the WDL language specification:
WdlStandardLibraryWorkflowsTest	A set of test cases for toil's conformance with the WDL built-in standard library:

class toil.test.wdl.builtinTest.**WdlStandardLibraryFunctionsTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*



A set of test cases for toil's wdl functions.

setUp()

Runs anew before each test to create farm fresh temp dirs.

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

tearDown()

Clean up outputs.

testFn_Sub()

Test the wdl built-in functional equivalent of 'sub()'.

testFn_Ceil()

Test the wdl built-in functional equivalent of 'ceil()', which converts a Float value into an Int by rounding up to the next higher integer

testFn_Floor()

Test the wdl built-in functional equivalent of 'floor()', which converts a Float value into an Int by rounding down to the next lower integer

testFn_ReadLines()

Test the wdl built-in functional equivalent of 'read_lines()'.

testFn_ReadTsv()

Test the wdl built-in functional equivalent of 'read_tsv()'.

testFn_ReadJson()

Test the wdl built-in functional equivalent of 'read_json()'.

testFn_ReadMap()

Test the wdl built-in functional equivalent of 'read_map()'.

testFn_ReadInt()

Test the wdl built-in functional equivalent of 'read_int()'.

testFn_ReadString()

Test the wdl built-in functional equivalent of 'read_string()'.

testFn_ReadFloat()

Test the wdl built-in functional equivalent of 'read_float()'.

testFn_ReadBoolean()

Test the wdl built-in functional equivalent of 'read_boolean()'.

testFn_WriteLines()

Test the wdl built-in functional equivalent of 'write_lines()'.

testFn_WriteTsv()

Test the wdl built-in functional equivalent of 'write_tsv()'.

testFn_WriteJson()

Test the wdl built-in functional equivalent of 'write_json()'.

testFn_WriteMap()

Test the wdl built-in functional equivalent of 'write_map()'.

testFn_Transpose()

Test the wdl built-in functional equivalent of 'transpose()'.

testFn_Length()

Test the WDL 'length()' built-in.

testFn_Zip()

Test the wdl built-in functional equivalent of 'zip()'.

testFn_Cross()

Test the wdl built-in functional equivalent of 'cross()'.

class toil.test.wdl.builtinTest.**WdlWorkflowsTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

A set of test cases for toil's conformance with WDL.

All tests should include a simple wdl and json file for toil to run that checks the output.

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

check_function(*function_name*, *cases*, *json_file_name=None*, *expected_result=None*, *expected_exception=None*)

Run the given WDL workflow and check its output. The WDL workflow should store its output inside a 'output.txt' file that can be compared to *expected_result*.

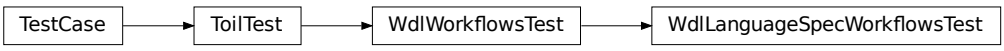
If *expected_exception* is set, this test passes only when both the workflow fails and that the given *expected_exception* string is present in standard error.

Parameters

- **function_name** (*str*) –
- **cases** (*List[str]*) –
- **json_file_name** (*Optional[str]*) –
- **expected_result** (*Optional[str]*) –
- **expected_exception** (*Optional[str]*) –

class toil.test.wdl.builtinTest.**WdlLanguageSpecWorkflowsTest**(*methodName='runTest'*)

Bases: *WdlWorkflowsTest*



A set of test cases for toil's conformance with the WDL language specification:

https://github.com/openwdl/wdl/blob/main/versions/development/SPEC.md#language-specification

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

test_type_pair()

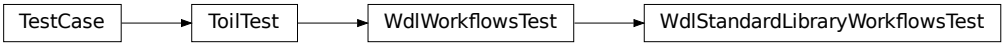
test_v1_declaration()

Basic declaration example modified from the WDL 1.0 spec:

https://github.com/openwdl/wdl/blob/main/versions/1.0/SPEC.md#declarations

class toil.test.wdl.builtinTest.**WdlStandardLibraryWorkflowsTest**(*methodName='runTest'*)

Bases: *WdlWorkflowsTest*



A set of test cases for toil's conformance with the WDL built-in standard library:

https://github.com/openwdl/wdl/blob/main/versions/development/SPEC.md#standard-library

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

test_sub()

test_size()

test_ceil()

test_floor()

test_round()

test_stdout()

test_read()

Test the set of WDL read functions.

test_write()

Test the set of WDL write functions.

test_range()

test_transpose()

test_length()

test_zip()

test_cross()

test_as_pairs()

test_as_map()

test_keys()

test_collect_by_key()

test_flatten()

toil.test.wdl.conftest

Module Contents

toil.test.wdl.conftest.**collect_ignore** = []

toil.test.wdl.toilwdlTest

Module Contents

Classes

BaseToilWdlTest	Base test class for WDL tests
ToilWdlTest	General tests for Toil WDL
ToilWDLLibraryTest	Test class for WDL standard functions.
ToilWdlIntegrationTest	Test class for WDL tests that need extra workflows and data downloaded

Functions

compare_runs(output_dir, ref_dir)	Takes two directories and compares all of the files between those two
compare_vcf_files(filepath1, filepath2)	Asserts that two .vcf files contain the same variant findings.

class toil.test.wdl.toilwdlTest.**BaseToilWdlTest**(*methodName='runTest'*)

Bases: *toil.test.ToilTest*

Base test class for WDL tests

setUp()

Runs anew before each test to create farm fresh temp dirs.

Return type
None

`tearDown()`

Hook method for deconstructing the test fixture after testing it.

Return type
None

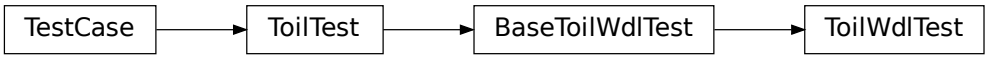
`classmethod setUpClass()`

Runs once for all tests.

Return type
None

`class toil.test.wdl.toilwdlTest.ToilWdlTest(methodName='runTest')`

Bases: *BaseToilWdlTest*



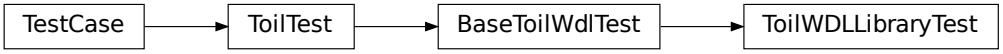
General tests for Toil WDL

`testMD5sum()`

Test if toilwdl produces the same outputs as known good outputs for WDL's GATK tutorial #1.

`class toil.test.wdl.toilwdlTest.ToilWDLLibraryTest(methodName='runTest')`

Bases: *BaseToilWdlTest*



Test class for WDL standard functions.

`testFn_SelectFirst()`

Test the wdl built-in functional equivalent of 'select_first()', which returns the first value in a list that is not None.

`testFn_Size()`

Test the wdl built-in functional equivalent of 'size()', which returns a file's size based on the path.

Return type
None

`testFn_Basename()`

`testFn_Glob()`

Test the wdl built-in functional equivalent of 'glob()', which finds all files with a pattern in a directory.

testFn_ParseMemory()

Test the wdl built-in functional equivalent of 'parse_memory()', which parses a specified memory input to an int output.

The input can be a string or an int or a float and may include units such as 'Gb' or 'mib' as a separate argument.

testFn_ParseCores()

Test the wdl built-in functional equivalent of 'parse_cores()', which parses a specified disk input to an int output.

The input can be a string or an int.

testFn_ParseDisk()

Test the wdl built-in functional equivalent of 'parse_disk()', which parses a specified disk input to an int output.

The input can be a string or an int or a float and may include units such as 'Gb' or 'mib' as a separate argument.

The minimum returned value is 2147483648 bytes.

testPrimitives()

Test if toilwdl correctly interprets some basic declarations.

testCSV()

testTSV()

class toil.test.wdl.toilwdlTest.**ToilWdlIntegrationTest**(*methodName='runTest'*)

Bases: *BaseToilWdlTest*

Test class for WDL tests that need extra workflows and data downloaded

gatk_data: str

gatk_data_dir: str

encode_data: str

encode_data_dir: str

wdl_data: str

wdl_data_dir: str

classmethod setUpClass()

Runs once for all tests.

Return type
None

classmethod tearDownClass()

We generate a lot of cruft.

Return type
None

testTut01()

Test if toilwdl produces the same outputs as known good outputs for WDL's GATK tutorial #1.

testTut02()

Test if toilwdl produces the same outputs as known good outputs for WDL's GATK tutorial #2.

testTut03()

Test if toilwdl produces the same outputs as known good outputs for WDL's GATK tutorial #3.

testTut04()

Test if toilwdl produces the same outputs as known good outputs for WDL's GATK tutorial #4.

testENCODE()

Test if toilwdl produces the same outputs as known good outputs for a short ENCODE run.

testPipe()

Test basic bash input functionality with a pipe.

testJSON()

test_size_large()

Test the wdl built-in functional equivalent of 'size()', which returns a file's size based on the path, on a large file.

Return type
None

classmethod fetch_and_unzip_from_s3(*filename*, *data*, *data_dir*)

toil.test.wdl.toilwdlTest.**compare_runs**(*output_dir*, *ref_dir*)

Takes two directories and compares all of the files between those two directories, asserting that they match.

- Ignores outputs.txt, which contains a list of the outputs in the folder.
- Compares line by line, unless the file is a .vcf file.
- Ignores potentially date-stamped comments (lines starting with '#').
- Ignores quality scores in .vcf files and only checks that they found the same variants. This is due to assumed small observed rounding differences between systems.

Parameters

- **ref_dir** – The first directory to compare (with output_dir).
- **output_dir** – The second directory to compare (with ref_dir).

toil.test.wdl.toilwdlTest.**compare_vcf_files**(*filepath1*, *filepath2*)

Asserts that two .vcf files contain the same variant findings.

- Ignores potentially date-stamped comments (lines starting with '#').
- Ignores quality scores in .vcf files and only checks that they found the same variants. This is due to assumed small observed rounding differences between systems.

VCF File Column Contents: 1: #CHROM 2: POS 3: ID 4: REF 5: ALT 6: QUAL 7: FILTER 8: INFO

Parameters

- **filepath1** – First .vcf file to compare.
- **filepath2** – Second .vcf file to compare.

toil.test.wdl.wdltoil_test

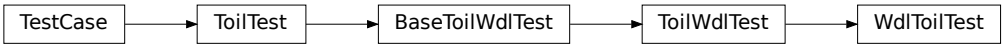
Module Contents

Classes

WdlToilTest	Version of the old Toil WDL tests that tests the new MiniWDL-based implementation.

class toil.test.wdl.wdltoil_test.**WdlToilTest**(*methodName='runTest'*)

Bases: *toil.test.wdl.toilwdlTest.ToilWdlTest*



Version of the old Toil WDL tests that tests the new MiniWDL-based implementation.

classmethod setUpClass()

Runs once for all tests.

Return type
None

testMD5sum()

Test if toilwdl produces the same outputs as known good outputs for WDL's GATK tutorial #1.

test_empty_file_path()

Test if empty File type inputs are protected against

test_miniwdl_self_test()

Test if the MiniWDL self test runs and produces the expected output.

test_giraffe_deepvariant()

Test if Giraffe and CPU DeepVariant run. This could take 25 minutes.

test_giraffe()

Test if Giraffe runs. This could take 12 minutes. Also we scale it down.

Package Contents

Classes

concat	A literal iterable to combine sequence literals (lists, set) with generators or list comprehensions.
ExceptionalThread	A thread whose join() method re-raises exceptions raised during run(). While join() is
ToilTest	A common base class for Toil tests.
ApplianceTestSupport	A Toil test that runs a user script on a minimal cluster of appliance containers.

Functions

applianceSelf([forceDockerAppliance])	Return the fully qualified name of the Docker image to start Toil appliance containers from.
toilPackageDirPath()	Return the absolute path of the directory that corresponds to the top-level toil package.
have_working_nvidia_docker_runtime()	Return True if Docker exists and can handle an "nvidia" runtime and the "--gpus" option.
have_working_nvidia_smi()	Return True if the nvidia-smi binary, from nvidia's CUDA userspace
running_on_ec2()	Return True if we are currently running on EC2, and false otherwise.
cpu_count()	Get the rounded-up integer number of whole CPUs available.
get_temp_file([suffix, rootDir])	Return a string representing a temporary file, that must be manually deleted.
needs_env_var(var_name[, comment])	Use as a decorator before test classes or methods to run only if the given
needs_rsync3(test_item)	Decorate classes or methods that depend on any features from rsync version 3.0.0+.
needs_aws_s3(test_item)	Use as a decorator before test classes or methods to run only if AWS S3 is usable.
needs_aws_ec2(test_item)	Use as a decorator before test classes or methods to run only if AWS EC2 is usable.
needs_aws_batch(test_item)	Use as a decorator before test classes or methods to run only if AWS Batch
needs_google(test_item)	Use as a decorator before test classes or methods to run only if Google
needs_gridengine(test_item)	Use as a decorator before test classes or methods to run only if GridEngine is installed.
needs_torque(test_item)	Use as a decorator before test classes or methods to run only if PBS/Torque is installed.
needs_tes(test_item)	Use as a decorator before test classes or methods to run only if TES is available.
needs_kubernetes_installed(test_item)	Use as a decorator before test classes or methods to run only if Kubernetes is installed.

continues on next page

Table 1 – continued from previous page

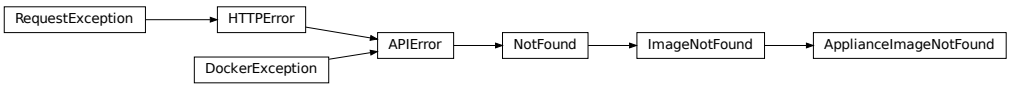
needs_kubernetes(test_item)	Use as a decorator before test classes or methods to run only if Kubernetes is installed and configured.
needs_mesos(test_item)	Use as a decorator before test classes or methods to run only if Mesos is installed.
needs_parasol(test_item)	Use as decorator so tests are only run if Parasol is installed.
needs_slurm(test_item)	Use as a decorator before test classes or methods to run only if Slurm is installed.
needs_htcondor(test_item)	Use a decorator before test classes or methods to run only if the HTCondor is installed.
needs_lsf(test_item)	Use as a decorator before test classes or methods to only run them if LSF is installed.
needs_java(test_item)	Use as a test decorator to run only if java is installed.
needs_docker(test_item)	Use as a decorator before test classes or methods to only run them if
needs_singularity(test_item)	Use as a decorator before test classes or methods to only run them if
needs_local_cuda(test_item)	Use as a decorator before test classes or methods to only run them if
needs_docker_cuda(test_item)	Use as a decorator before test classes or methods to only run them if
needs_encryption(test_item)	Use as a decorator before test classes or methods to only run them if PyNaCl is installed
needs_cwl(test_item)	Use as a decorator before test classes or methods to only run them if CWLTool is installed
needs_server(test_item)	Use as a decorator before test classes or methods to only run them if Connexion is installed.
needs_celery_broker(test_item)	Use as a decorator before test classes or methods to run only if RabbitMQ is set up to take Celery jobs.
needs_wes_server(test_item)	Use as a decorator before test classes or methods to run only if a WES
needs_local_appliance(test_item)	Use as a decorator before test classes or methods to only run them if
needs_fetchable_appliance(test_item)	Use as a decorator before test classes or methods to only run them if
integrative(test_item)	Use this to decorate integration tests so as to skip them during regular builds.
slow(test_item)	Use this decorator to identify tests that are slow and not critical.
timeLimit(seconds)	http://stackoverflow.com/a/601168
make_tests(generalMethod, targetClass, **kwargs)	This method dynamically generates test methods using the generalMethod as a template. Each

Attributes

memoize	Memoize a function result based on its parameters using this decorator.
distVersion	
logger	
MT	
methodNamePartRegex	

exception toil.test.**ApplianceImageNotFound**(*origAppliance*, *url*, *statusCode*)

Bases: docker.errors.ImageNotFound



Error raised when using TOIL_APPLIANCE_SELF results in an HTTP error.

Parameters

- **origAppliance** (*str*) – The full url of the docker image originally specified by the user (or the default). e.g. "quay.io/ucsc_cgl/toil:latest"
- **url** (*str*) – The URL at which the image's manifest is supposed to appear
- **statusCode** (*int*) – the failing HTTP status code returned by the URL

toil.test.**applianceSelf**(*forceDockerAppliance=False*)

Return the fully qualified name of the Docker image to start Toil appliance containers from.

The result is determined by the current version of Toil and three environment variables: TOIL_DOCKER_REGISTRY, TOIL_DOCKER_NAME and TOIL_APPLIANCE_SELF.

TOIL_DOCKER_REGISTRY specifies an account on a publicly hosted docker registry like Quay or Docker Hub. The default is UCSC's CGL account on Quay.io where the Toil team publishes the official appliance images. TOIL_DOCKER_NAME specifies the base name of the image. The default of *toil* will be adequate in most cases. TOIL_APPLIANCE_SELF fully qualifies the appliance image, complete with registry, image name and version tag, overriding both TOIL_DOCKER_NAME and *TOIL_DOCKER_REGISTRY*` as well as the version tag of the image. Setting TOIL_APPLIANCE_SELF will not be necessary in most cases.

Parameters

forceDockerAppliance (*bool*) –

Return type

str

toil.test.**toilPackageDirPath**()

Return the absolute path of the directory that corresponds to the top-level toil package.

The return value is guaranteed to end in '/toil'.

Return type
str

toil.test.**have_working_nvidia_docker_runtime**()

Return True if Docker exists and can handle an "nvidia" runtime and the "–gpus" option.

Return type
bool

toil.test.**have_working_nvidia_smi**()

Return True if the nvidia-smi binary, from nvidia's CUDA userspace utilities, is installed and can be run successfully.

TODO: This isn't quite the same as the check that cwltool uses to decide if it can fulfill a CUDARequirement.

Return type
bool

toil.test.**running_on_ec2**()

Return True if we are currently running on EC2, and false otherwise.

Return type
bool

class toil.test.**concat**(**args*)

A literal iterable to combine sequence literals (lists, set) with generators or list comprehensions.

Instead of

```
>>> [ -1 ] + [ x * 2 for x in range( 3 ) ] + [ -1 ]
[-1, 0, 2, 4, -1]
```

you can write

```
>>> list( concat( -1, ( x * 2 for x in range( 3 ) ), -1 ) )
[-1, 0, 2, 4, -1]
```

This is slightly shorter (not counting the list constructor) and does not involve array construction or concatenation.

Note that concat() flattens (or chains) all iterable arguments into a single result iterable:

```
>>> list( concat( 1, range( 2, 4 ), 4 ) )
[1, 2, 3, 4]
```

It only does so one level deep. If you need to recursively flatten a data structure, check out crush().

If you want to prevent that flattening for an iterable argument, wrap it in concat():

```
>>> list( concat( 1, concat( range( 2, 4 ) ), 4 ) )
[1, range(2, 4), 4]
```

Some more example.

```
>>> list( concat() ) # empty concat
[]
>>> list( concat( 1 ) ) # non-iterable
[1]
```

(continues on next page)

(continued from previous page)

```
>>> list( concat( concat() ) ) # empty iterable
[]
>>> list( concat( concat( 1 ) ) ) # singleton iterable
[1]
>>> list( concat( 1, concat( 2 ), 3 ) ) # flattened iterable
[1, 2, 3]
>>> list( concat( 1, [2], 3 ) ) # flattened iterable
[1, 2, 3]
>>> list( concat( 1, concat( [2] ), 3 ) ) # protecting an iterable from being␣
↪flattened
[1, [2], 3]
>>> list( concat( 1, concat( [2], 3 ), 4 ) ) # protection only works with a single␣
↪argument
[1, 2, 3, 4]
>>> list( concat( 1, 2, concat( 3, 4 ), 5, 6 ) )
[1, 2, 3, 4, 5, 6]
>>> list( concat( 1, 2, concat( [ 3, 4 ] ), 5, 6 ) )
[1, 2, [3, 4], 5, 6]
```

Note that while strings are technically iterable, concat() does not flatten them.

```
>>> list( concat( 'ab' ) )
['ab']
>>> list( concat( concat( 'ab' ) ) )
['ab']
```

Parameters

args (*Any*) –

__iter__()

Return type

Iterator[Any]

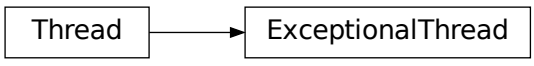
toil.test.**memoize**

Memoize a function result based on its parameters using this decorator.

For example, this can be used in place of lazy initialization. If the decorating function is invoked by multiple threads, the decorated function may be called more than once with the same arguments.

class toil.test.**ExceptionalThread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs=None*, ***, *daemon=None*)

Bases: threading.Thread



A thread whose join() method re-raises exceptions raised during run(). While join() is idempotent, the exception is only during the first invocation of join() that successfully joined the thread. If join() times out, no exception will be re reraised even though an exception might already have occured in run().

When subclassing this thread, override tryRun() instead of run().

```
>>> def f():
...     assert 0
>>> t = ExceptionalThread(target=f)
>>> t.start()
>>> t.join()
Traceback (most recent call last):
...
AssertionError
```

```
>>> class MyThread(ExceptionalThread):
...     def tryRun( self ):
...         assert 0
>>> t = MyThread()
>>> t.start()
>>> t.join()
Traceback (most recent call last):
...
AssertionError
```

exc_info

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

Return type
None

tryRun()

Return type
None

join(**args*, ***kwargs*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

Parameters

- **args** (*Optional[float]*) –
- **kwargs** (*Optional[float]*) –

Return type
None

toil.test.**cpu_count**()

Get the rounded-up integer number of whole CPUs available.

Counts hyperthreads as CPUs.

Uses the system's actual CPU count, or the current v1 cgroup's quota per period, if the quota is set.

Ignores the cgroup's cpu shares value, because it's extremely difficult to interpret. See https://github.com/kubernetes/kubernetes/issues/81021.

Caches result for efficiency.

Returns
Integer count of available CPUs, minimum 1.

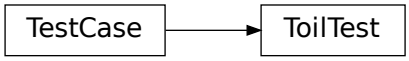
Return type
int

toil.test.**distVersion = '5.11.0'**

toil.test.**logger**

class toil.test.**ToilTest**(*methodName='runTest'*)

Bases: unittest.TestCase



A common base class for Toil tests.

Please have every test case directly or indirectly inherit this one.

When running tests you may optionally set the TOIL_TEST_TEMP environment variable to the path of a directory where you want temporary test files be placed. The directory will be created if it doesn't exist. The path may be relative in which case it will be assumed to be relative to the project root. If TOIL_TEST_TEMP is not defined, temporary files and directories will be created in the system's default location for such files and any temporary files or directories left over from tests will be removed automatically removed during tear down. Otherwise, left-over files will not be removed.

setup_method(*method*)

Parameters
method (*Any*) –

Return type
None

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

Return type
None

classmethod tearDownClass()

Hook method for deconstructing the class fixture after running all tests in the class.

Return type
None

setUp()

Hook method for setting up the test fixture before exercising it.

Return type
None

tearDown()

Hook method for deconstructing the test fixture after testing it.

Return type
None

classmethod awsRegion()

Pick an appropriate AWS region.

Use us-west-2 unless running on EC2, in which case use the region in which the instance is located

Return type
str

toil.test.**MT**

toil.test.**get_temp_file**(*suffix=''*, *rootDir=None*)

Return a string representing a temporary file, that must be manually deleted.

Parameters

- **suffix** (*str*) –
- **rootDir** (*Optional[str]*) –

Return type
str

toil.test.**needs_env_var**(*var_name*, *comment=None*)

Use as a decorator before test classes or methods to run only if the given environment variable is set. Can include a comment saying what the variable should be set to.

Parameters

- **var_name** (*str*) –
- **comment** (*Optional[str]*) –

Return type
Callable[[MT], MT]

toil.test.**needs_rsync3**(*test_item*)

Decorate classes or methods that depend on any features from rsync version 3.0.0+.

Necessary because `utilsTest.testAWSProvisionerUtils()` uses option *–protect-args* which is only available in rsync 3

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_aws_s3**(*test_item*)

Use as a decorator before test classes or methods to run only if AWS S3 is usable.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_aws_ec2**(*test_item*)

Use as a decorator before test classes or methods to run only if AWS EC2 is usable.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_aws_batch**(*test_item*)

Use as a decorator before test classes or methods to run only if AWS Batch is usable.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_google**(*test_item*)

Use as a decorator before test classes or methods to run only if Google Cloud is usable.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_gridengine**(*test_item*)

Use as a decorator before test classes or methods to run only if GridEngine is installed.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_torque**(*test_item*)

Use as a decorator before test classes or methods to run only if PBS/Torque is installed.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_tes**(*test_item*)

Use as a decorator before test classes or methods to run only if TES is available.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_kubernetes_installed**(*test_item*)

Use as a decorator before test classes or methods to run only if Kubernetes is installed.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_kubernetes**(*test_item*)

Use as a decorator before test classes or methods to run only if Kubernetes is installed and configured.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_mesos**(*test_item*)

Use as a decorator before test classes or methods to run only if Mesos is installed.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_parasol**(*test_item*)

Use as decorator so tests are only run if Parasol is installed.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_slurm**(*test_item*)

Use as a decorator before test classes or methods to run only if Slurm is installed.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_htcondor**(*test_item*)

Use a decorator before test classes or methods to run only if the HTCondor is installed.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_lsf**(*test_item*)

Use as a decorator before test classes or methods to only run them if LSF is installed.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_java**(*test_item*)

Use as a test decorator to run only if java is installed.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_docker**(*test_item*)

Use as a decorator before test classes or methods to only run them if docker is installed and docker-based tests are enabled.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_singularity**(*test_item*)

Use as a decorator before test classes or methods to only run them if singularity is installed.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_local_cuda**(*test_item*)

Use as a decorator before test classes or methods to only run them if a CUDA setup legible to cwltool (i.e. providing userspace nvidia-smi) is present.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_docker_cuda**(*test_item*)

Use as a decorator before test classes or methods to only run them if a CUDA setup is available through Docker.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_encryption**(*test_item*)

Use as a decorator before test classes or methods to only run them if PyNaCl is installed and configured.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_cwl**(*test_item*)

Use as a decorator before test classes or methods to only run them if CWLTool is installed and configured.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_server**(*test_item*)

Use as a decorator before test classes or methods to only run them if Connexion is installed.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_celery_broker**(*test_item*)

Use as a decorator before test classes or methods to run only if RabbitMQ is set up to take Celery jobs.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_wes_server**(*test_item*)

Use as a decorator before test classes or methods to run only if a WES server is available to run against.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_local_appliance**(*test_item*)

Use as a decorator before test classes or methods to only run them if the Toil appliance Docker image is downloaded.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**needs_fetchable_appliance**(*test_item*)

Use as a decorator before test classes or methods to only run them if the Toil appliance Docker image is able to be downloaded from the Internet.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**integrative**(*test_item*)

Use this to decorate integration tests so as to skip them during regular builds.

We define integration tests as A) involving other, non-Toil software components that we develop and/or B) having a higher cost (time or money). Note that brittleness does not qualify a test for being integrative. Neither does involvement of external services such as AWS, since that would cover most of Toil's test.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**slow**(*test_item*)

Use this decorator to identify tests that are slow and not critical. Skip if TOIL_TEST_QUICK is true.

Parameters
test_item (*MT*) –

Return type
MT

toil.test.**methodNamePartRegex**

toil.test.**timeLimit**(*seconds*)

http://stackoverflow.com/a/601168 Use to limit the execution time of a function. Raises an exception if the execution of the function takes more than the specified amount of time.

Parameters
seconds (*int*) – maximum allowable time, in seconds

Return type
Generator[None, None, None]

```
>>> import time
>>> with timeLimit(2):
...     time.sleep(1)
>>> import time
>>> with timeLimit(1):
...     time.sleep(2)
Traceback (most recent call last):
    ...
RuntimeError: Timed out
```

toil.test.**make_tests**(*generalMethod*, *targetClass*, ***kwargs*)

This method dynamically generates test methods using the generalMethod as a template. Each generated function is the result of a unique combination of parameters applied to the generalMethod. Each of the parameters has a corresponding string that will be used to name the method. These generated functions are named in the scheme: test_[generalMethodName]___[firstParamaterName]_[someValueName]__[secondParamaterName]_...

The arguments following the generalMethodName should be a series of one or more dictionaries of the form {str : type, ... } where the key represents the name of the value. The names will be used to represent the permutation of values passed for each parameter in the generalMethod.

The generated method names will list the parameters in lexicographic order by parameter name.

Parameters

- **generalMethod** – A method that will be parameterized with values passed as kwargs. Note that the generalMethod must be a regular method.

- **targetClass** – This represents the class to which the generated test methods will be bound. If no targetClass is specified the class of the generalMethod is assumed the target.
- **kwargs** – a series of dictionaries defining values, and their respective names where each keyword is the name of a parameter in generalMethod.

```
>>> class Foo:
...     def has(self, num, letter):
...         return num, letter
...
...     def hasOne(self, num):
...         return num
```

```
>>> class Bar(Foo):
...     pass
```

```
>>> make_tests(Foo.has, Bar, num={'one':1, 'two':2}, letter={'a':'a', 'b':'b'})
```

```
>>> b = Bar()
```

Note that num comes lexicographically before letter and so appears first in the generated method names.

```
>>> assert b.test_has__letter_a__num_one() == b.has(1, 'a')
```

```
>>> assert b.test_has__letter_b__num_one() == b.has(1, 'b')
```

```
>>> assert b.test_has__letter_a__num_two() == b.has(2, 'a')
```

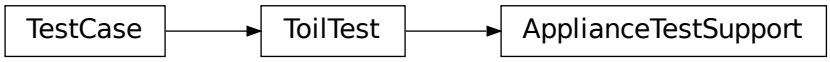
```
>>> assert b.test_has__letter_b__num_two() == b.has(2, 'b')
```

```
>>> f = Foo()
```

```
>>> hasattr(f, 'test_has__num_one__letter_a')  # should be false because Foo has no␣
↪test methods
False
```

class toil.test.**ApplianceTestSupport**(*methodName='runTest'*)

Bases: *ToilTest*

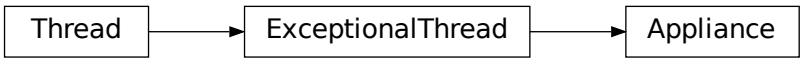


A Toil test that runs a user script on a minimal cluster of appliance containers.

i.e. one leader container and one worker container.

class Appliance(*outer*, *mounts*, *cleanMounts=False*)

Bases: *toil.lib.threading.ExceptionalThread*



A thread whose join() method re-raises exceptions raised during run(). While join() is idempotent, the exception is only during the first invocation of join() that successfully joined the thread. If join() times out, no exception will be re reraised even though an exception might already have occured in run().

When subclassing this thread, override tryRun() instead of run().

```
>>> def f():
...     assert 0
>>> t = ExceptionalThread(target=f)
>>> t.start()
>>> t.join()
Traceback (most recent call last):
...
AssertionError
```

```
>>> class MyThread(ExceptionalThread):
...     def tryRun( self ):
...         assert 0
>>> t = MyThread()
>>> t.start()
>>> t.join()
Traceback (most recent call last):
...
AssertionError
```

Parameters

- **outer** (*ApplianceTestSupport*) –
- **mounts** (*Dict[str, str]*) –
- **cleanMounts** (*bool*) –

lock

__enter__()

Return type

Appliance

__exit__(*exc_type*, *exc_val*, *exc_tb*)

Parameters

- **exc_type** (*Type[BaseException]*) –
- **exc_val** (*Exception*) –
- **exc_tb** (*Any*) –

Return type
Literal[False]

`tryRun`()

Return type
None

`runOnAppliance`(**args*, ***kwargs*)

Parameters
- **args** (*str*) –
- **kwargs** (*Any*) –

Return type
None

`writeToAppliance`(*path*, *contents*)

Parameters
- **path** (*str*) –
- **contents** (*Any*) –

Return type
None

`deployScript`(*path*, *packagePath*, *script*)

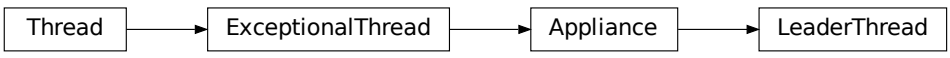
Deploy a Python module on the appliance.

Parameters
- **path** (*str*) – the path (absolute or relative to the WORDIR of the appliance container) to the root of the package hierarchy where the given module should be placed. The given directory should be on the Python path.
- **packagePath** (*str*) – the desired fully qualified module name (dotted form) of the module
- **script** (*str|callable*) – the contents of the Python module. If a callable is given, its source code will be extracted. This is a convenience that lets you embed user scripts into test code as nested function.

Return type
None

`class LeaderThread`(*outer*, *mounts*, *cleanMounts=False*)

Bases: *ApplianceTestSupport.Appliance*



A thread whose join() method re-raises exceptions raised during run(). While join() is idempotent, the exception is only during the first invocation of join() that successfully joined the thread. If join() times out, no exception will be re reraised even though an exception might already have occured in run().

When subclassing this thread, override tryRun() instead of run().

```
>>> def f():
...     assert 0
>>> t = ExceptionalThread(target=f)
>>> t.start()
```

(continues on next page)

(continued from previous page)

```
>>> t.join()
Traceback (most recent call last):
...
AssertionError
```

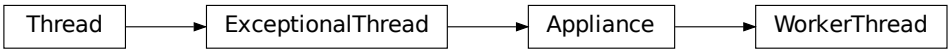
```
>>> class MyThread(ExceptionalThread):
...     def tryRun( self ):
...         assert 0
>>> t = MyThread()
>>> t.start()
>>> t.join()
Traceback (most recent call last):
...
AssertionError
```

Parameters

- **outer** (*ApplianceTestSupport*) –
- **mounts** (*Dict[str, str]*) –
- **cleanMounts** (*bool*) –

class WorkerThread(*outer*, *mounts*, *numCores*)

Bases: *ApplianceTestSupport.Appliance*



A thread whose join() method re-raises exceptions raised during run(). While join() is idempotent, the exception is only during the first invocation of join() that successfully joined the thread. If join() times out, no exception will be re reraised even though an exception might already have occured in run().

When subclassing this thread, override tryRun() instead of run().

```
>>> def f():
...     assert 0
>>> t = ExceptionalThread(target=f)
>>> t.start()
>>> t.join()
Traceback (most recent call last):
...
AssertionError
```

```
>>> class MyThread(ExceptionalThread):
...     def tryRun( self ):
...         assert 0
>>> t = MyThread()
>>> t.start()
```

(continues on next page)

(continued from previous page)

```
>>> t.join()
Traceback (most recent call last):
...
AssertionError
```

Parameters

- **outer** (ApplianceTestSupport) –
- **mounts** (*Dict[str, str]*) –
- **numCores** (*int*) –

toil.utils

Submodules

toil.utils.toilClean

Delete a job store used by a previous Toil workflow invocation.

Module Contents

Functions

main()

Attributes

logger

toil.utils.toilClean.**logger**

toil.utils.toilClean.**main**()

Return type
None

toil.utils.toilDebugFile

Debug tool for copying files contained in a toil jobStore.

Module Contents

Functions

fetchJobStoreFiles(jobStore, options)	Takes a list of file names as glob patterns, searches for these within a
printContentsOfJobStore(jobStorePath[, nameOfJob])	Fetch a list of all files contained in the jobStore directory input if
main()	

Attributes

logger

toil.utils.toilDebugFile.**logger**

toil.utils.toilDebugFile.**fetchJobStoreFiles**(*jobStore*, *options*)

Takes a list of file names as glob patterns, searches for these within a given directory, and attempts to take all of the files found and copy them into options.localFilePath.

Parameters

- **jobStore** (toil.jobStores.abstractJobStore.AbstractJobStore) – A fileJobStore object.
- **options.fetch** – List of file glob patterns to search for in the jobStore and copy into options.localFilePath.
- **options.localFilePath** – Local directory to copy files into.
- **options.jobStore** – The path to the jobStore directory.
- **options** (*argparse.Namespace*) –

Return type

None

toil.utils.toilDebugFile.**printContentsOfJobStore**(*jobStorePath*, *nameOfJob=None*)

Fetch a list of all files contained in the jobStore directory input if nameOfJob is not declared, otherwise it only prints out the names of files for that specific job for which it can find a match. Also creates a logFile containing this same record of job files in the working directory.

Parameters

- **jobStorePath** (*str*) – Directory path to recursively look for files.
- **nameOfJob** (*Optional[str]*) – Default is None, which prints out all files in the jobStore.

Return type
None

If specified, it will print all jobStore files that have been written to the jobStore by that job.

toil.utils.toilDebugFile.**main**()

Return type
None

toil.utils.toilDebugJob

Debug tool for running a toil job locally.

Module Contents

Functions

main()

Attributes

logger

toil.utils.toilDebugJob.**logger**

toil.utils.toilDebugJob.**main**()

Return type
None

toil.utils.toilDestroyCluster

Terminates the specified cluster and associated resources.

Module Contents

Functions

main()

Attributes

logger

toil.utils.toilDestroyCluster.**logger**

toil.utils.toilDestroyCluster.**main**()

Return type
None

toil.utils.toilKill

Kills rogue toil processes.

Module Contents

Functions

main()

Attributes

logger

toil.utils.toilKill.**logger**

toil.utils.toilKill.**main**()

Return type
None

toil.utils.toilLaunchCluster

Launches a toil leader instance with the specified provisioner.

Module Contents

Functions

create_tags_dict(tags)	
main()	

Attributes

logger	

toil.utils.toilLaunchCluster.**logger**

toil.utils.toilLaunchCluster.**create_tags_dict**(*tags*)

Parameters
tags (*List[str]*) –

Return type
Dict[str, str]

toil.utils.toilLaunchCluster.**main**()

Return type
None

toil.utils.toilMain

Module Contents

Functions

main()	
get_or_die(module, name)	Get an object from a module or complain that it is missing.
loadModules()	
printHelp(modules)	
printVersion()	

toil.utils.toilMain.**main**()

Return type
None

toil.utils.toilMain.**get_or_die**(*module*, *name*)

Get an object from a module or complain that it is missing.

Parameters

- **module** (*types.ModuleType*) –
- **name** (*str*) –

Return type
Any

toil.utils.toilMain.**loadModules**()

Return type
Dict[str, types.ModuleType]

toil.utils.toilMain.**printHelp**(*modules*)

Parameters
modules (*Dict[str, types.ModuleType]*) –

Return type
None

toil.utils.toilMain.**printVersion**()

Return type
None

toil.utils.toilRsyncCluster

Rsyncs into the toil appliance container running on the leader of the cluster.

Module Contents

Functions

main()

Attributes

logger

toil.utils.toilRsyncCluster.**logger**

toil.utils.toilRsyncCluster.**main**()

Return type
None

toil.utils.toilServer

CLI entry for the Toil servers.

Module Contents

Functions

main()

Attributes

logger

toil.utils.toilServer.**logger**

toil.utils.toilServer.**main**()

Return type
None

toil.utils.toilSshCluster

SSH into the toil appliance container running on the leader of the cluster.

Module Contents

Functions

main()

Attributes

logger

toil.utils.toilSshCluster.**logger**

toil.utils.toilSshCluster.**main**()

Return type
None

toil.utils.toilStats

Reports statistical data about a given Toil workflow.

Module Contents

Classes

ColumnWidths	Convenience object that stores the width of columns for printing. Helps make things pretty.

Functions

padStr(s[, field])	Pad the beginning of a string with spaces, if necessary.
prettyMemory(k[, field, isBytes])	Given input k as kilobytes, return a nicely formatted string.
prettyTime(t[, field])	Given input t as seconds, return a nicely formatted string.
reportTime(t, options[, field])	Given t seconds, report back the correct format as string.
reportMemory(k, options[, field, isBytes])	Given k kilobytes, report back the correct format as string.
reportNumber(n[, field])	Given n an integer, report back the correct format as string.
sprintTag(key, tag, options[, columnWidths])	Generate a pretty-print ready string from a JTTag().
decorateTitle(title, options)	Add a marker to TITLE if the TITLE is sorted on.
decorateSubHeader(title, columnWidths, options)	Add a marker to the correct field if the TITLE is sorted on.
get(tree, name)	Return a float value attribute NAME from TREE.
sortJobs(jobTypes, options)	Return a jobTypes all sorted.
reportPrettyData(root, worker, job, job_types, options)	Print the important bits out.
computeColumnWidths(job_types, worker, job, options)	Return a ColumnWidths() object with the correct max widths.
updateColumnWidths(tag, cw, options)	Update the column width attributes for this tag's fields.
buildElement(element, items, itemName)	Create an element for output.
createSummary(element, containingItems, ...)	
getStats(jobStore)	Collect and return the stats and config data.
processData(config, stats)	Collate the stats and report
reportData(tree, options)	
add_stats_options(parser)	
main()	Reports stats on the workflow, use with --stats option to toil.

Attributes

logger
category_choices
sort_category_choices
sort_field_choices

toil.utils.toilStats.**logger**

class toil.utils.toilStats.**ColumnWidths**

Convenience object that stores the width of columns for printing. Helps make things pretty.

title(*category*)

Return the total printed length of this category item.

Parameters
category (*str*) –

Return type
int

getWidth(*category*, *field*)

Parameters

- **category** (*str*) –
- **field** (*str*) –

Return type
int

setWidth(*category*, *field*, *width*)

Parameters

- **category** (*str*) –
- **field** (*str*) –
- **width** (*int*) –

Return type
None

report()

Return type
None

toil.utils.toilStats.**padStr**(*s*, *field=None*)

Pad the beginning of a string with spaces, if necessary.

Parameters

- **s** (*str*) –

- **field** (*Optional[int]*) –

Return type
str

toil.utils.toilStats.**prettyMemory**(*k*, *field=None*, *isBytes=False*)

Given input k as kilobytes, return a nicely formatted string.

Parameters

- **k** (*float*) –
- **field** (*Optional[int]*) –
- **isBytes** (*bool*) –

Return type
str

toil.utils.toilStats.**prettyTime**(*t*, *field=None*)

Given input t as seconds, return a nicely formatted string.

Parameters

- **t** (*float*) –
- **field** (*Optional[int]*) –

Return type
str

toil.utils.toilStats.**reportTime**(*t*, *options*, *field=None*)

Given t seconds, report back the correct format as string.

Parameters

- **t** (*float*) –
- **options** (*argparse.Namespace*) –
- **field** (*Optional[int]*) –

Return type
str

toil.utils.toilStats.**reportMemory**(*k*, *options*, *field=None*, *isBytes=False*)

Given k kilobytes, report back the correct format as string.

Parameters

- **k** (*float*) –
- **options** (*argparse.Namespace*) –
- **field** (*Optional[int]*) –
- **isBytes** (*bool*) –

Return type
str

toil.utils.toilStats.**reportNumber**(*n*, *field=None*)

Given n an integer, report back the correct format as string.

Parameters

- **n** (*float*) –

- **field** (*Optional[int]*) –

Return type
str

toil.utils.toilStats.**sprintTag**(*key*, *tag*, *options*, *columnWidths=None*)

Generate a pretty-print ready string from a JTTag().

Parameters

- **key** (*str*) –
- **tag** (toil.lib.expando.Expando) –
- **options** (*argparse.Namespace*) –
- **columnWidths** (*Optional[ColumnWidths]*) –

Return type
str

toil.utils.toilStats.**decorateTitle**(*title*, *options*)

Add a marker to TITLE if the TITLE is sorted on.

Parameters

- **title** (*str*) –
- **options** (*argparse.Namespace*) –

Return type
str

toil.utils.toilStats.**decorateSubHeader**(*title*, *columnWidths*, *options*)

Add a marker to the correct field if the TITLE is sorted on.

Parameters

- **title** (*str*) –
- **columnWidths** (ColumnWidths) –
- **options** (*argparse.Namespace*) –

Return type
str

toil.utils.toilStats.**get**(*tree*, *name*)

Return a float value attribute NAME from TREE.

Parameters

- **tree** (toil.lib.expando.Expando) –
- **name** (*str*) –

Return type
float

toil.utils.toilStats.**sortJobs**(*jobTypes*, *options*)

Return a jobTypes all sorted.

Parameters

- **jobTypes** (*List[Any]*) –
- **options** (*argparse.Namespace*) –

Return type
List[Any]

toil.utils.toilStats.**reportPrettyData**(*root*, *worker*, *job*, *job_types*, *options*)

Print the important bits out.

Parameters

- **root** (toil.lib.expando.Expando) –
- **worker** (*List[*toil.job.Job*]*) –
- **job** (*List[*toil.job.Job*]*) –
- **job_types** (*List[Any]*) –
- **options** (*argparse.Namespace*) –

Return type
str

toil.utils.toilStats.**computeColumnWidths**(*job_types*, *worker*, *job*, *options*)

Return a ColumnWidths() object with the correct max widths.

Parameters

- **job_types** (*List[Any]*) –
- **worker** (*List[*toil.job.Job*]*) –
- **job** (*List[*toil.job.Job*]*) –
- **options** (toil.lib.expando.Expando) –

Return type
ColumnWidths

toil.utils.toilStats.**updateColumnWidths**(*tag*, *cw*, *options*)

Update the column width attributes for this tag's fields.

Parameters

- **tag** (toil.lib.expando.Expando) –
- **cw** (ColumnWidths) –
- **options** (toil.lib.expando.Expando) –

Return type
None

toil.utils.toilStats.**buildElement**(*element*, *items*, *itemName*)

Create an element for output.

Parameters

- **element** (toil.lib.expando.Expando) –
- **items** (*List[*toil.job.Job*]*) –
- **itemName** (*str*) –

Return type
toil.lib.expando.Expando

toil.utils.toilStats.**createSummary**(*element*, *containingItems*, *containingItemName*, *getFn*)

Parameters

- **element** (toil.lib.expando.Expando) –
- **containingItems** (*List[toil.job.Job]*) –
- **containingItemName** (*str*) –
- **getFn** (*Callable[[toil.job.Job], List[Optional[toil.job.Job]]]*) –

Return type
None

toil.utils.toilStats.**getStats**(*jobStore*)

Collect and return the stats and config data.

Parameters
jobStore (toil.jobStores.abstractJobStore.AbstractJobStore) –

Return type
toil.lib.expando.Expando

toil.utils.toilStats.**processData**(*config*, *stats*)

Collate the stats and report

Parameters

- **config** (toil.common.Config) –
- **stats** (toil.lib.expando.Expando) –

Return type
toil.lib.expando.Expando

toil.utils.toilStats.**reportData**(*tree*, *options*)

Parameters

- **tree** (toil.lib.expando.Expando) –
- **options** (*argparse.Namespace*) –

Return type
None

toil.utils.toilStats.**category_choices = ['time', 'clock', 'wait', 'memory']**

toil.utils.toilStats.**sort_category_choices = ['time', 'clock', 'wait', 'memory', 'alpha', 'count']**

toil.utils.toilStats.**sort_field_choices = ['min', 'med', 'ave', 'max', 'total']**

toil.utils.toilStats.**add_stats_options**(*parser*)

Parameters
parser (*argparse.ArgumentParser*) –

Return type
None

toil.utils.toilStats.**main**()

Reports stats on the workflow, use with –stats option to toil.

Return type
None

toil.utils.toilStatus

Tool for reporting on job status.

Module Contents

Classes

ToilStatus	Tool for reporting on job status.

Functions

main()	Reports the state of a Toil workflow.

Attributes

logger

toil.utils.toilStatus.**logger**

class toil.utils.toilStatus.**ToilStatus**(*jobStoreName*, *specifiedJobs=None*)

Tool for reporting on job status.

Parameters

- **jobStoreName** (*str*) –
- **specifiedJobs** (*Optional[List[str]]*) –

print_dot_chart()

Print a dot output graph representing the workflow.

Return type
None

printJobLog()

Takes a list of jobs, finds their log files, and prints them to the terminal.

Return type
None

printJobChildren()

Takes a list of jobs, and prints their successors.

Return type
None

printAggregateJobStats(*properties*, *childNumber*)

Prints a job's ID, log file, remaining tries, and other properties.

Parameters

- **properties** (*List[str]*) –
- **childNumber** (*int*) –

Return type
None

report_on_jobs()

Gathers information about jobs such as its child jobs and status.

Returns jobStats
Pairings of a useful category and a list of jobs which fall into it.

Rtype dict

static getPIDStatus(*jobStoreName*)

Determine the status of a process with a particular local pid.

Checks to see if a process exists or not.

Returns
A string indicating the status of the PID of the workflow as stored in the jobstore.

Return type
str

Parameters
jobStoreName (*str*) –

static getStatus(*jobStoreName*)

Determine the status of a workflow.

If the jobstore does not exist, this returns 'QUEUED', assuming it has not been created yet.

Checks for the existence of files created in the toil.Leader.run(). In toil.Leader.run(), if a workflow completes with failed jobs, 'failed.log' is created, otherwise 'succeeded.log' is written. If neither of these exist, the leader is still running jobs.

Returns
A string indicating the status of the workflow. ['COMPLETED', 'RUNNING', 'ERROR', 'QUEUED']

Return type
str

Parameters
jobStoreName (*str*) –

print_bus_messages()

Goes through bus messages, returns a list of tuples which have correspondence between PID on assigned batch system and

Prints a list of the currently running jobs

Return type
None

`fetchRootJob()`

Fetches the root job from the jobStore that provides context for all other jobs.

Exactly the same as the jobStore.loadRootJob() function, but with a different exit message if the root job is not found (indicating the workflow ran successfully to completion and certain stats cannot be gathered from it meaningfully such as which jobs are left to run).

Raises
JobException – if the root job does not exist.

Return type
toil.job.JobDescription

`fetchUserJobs(jobs)`

Takes a user input array of jobs, verifies that they are in the jobStore and returns the array of jobsToReport.

Parameters
jobs (*list*) – A list of jobs to be verified.

Returns jobsToReport
A list of jobs which are verified to be in the jobStore.

Return type
List[*toil.job.JobDescription*]

`traverseJobGraph(rootJob, jobsToReport=None, foundJobStoreIDs=None)`

Find all current jobs in the jobStore and return them as an Array.

Parameters

- **rootJob** (*toil.job.JobDescription*) – The root job of the workflow.
- **jobsToReport** (*list*) – A list of jobNodes to be added to and returned.
- **foundJobStoreIDs** (*set*) – A set of jobStoreIDs used to keep track of jobStoreIDs encountered in traversal.

Returns jobsToReport
The list of jobs currently in the job graph.

Return type
List[*toil.job.JobDescription*]

`toil.utils.toilStatus.main()`

Reports the state of a Toil workflow.

Return type
None

toil.utils.toilUpdateEC2Instances

Updates Toil's internal list of EC2 instance types.

Module Contents

Functions

internet_connection()	Returns True if there is an internet connection present, and False otherwise.
main()	

Attributes

logger

toil.utils.toilUpdateEC2Instances.**logger**

toil.utils.toilUpdateEC2Instances.**internet_connection**()

Returns True if there is an internet connection present, and False otherwise.

Return type
bool

toil.utils.toilUpdateEC2Instances.**main**()

Return type
None

toil.wdl

Subpackages

toil.wdl.versions

Submodules

toil.wdl.versions.dev

Module Contents

Classes

AnalyzeDevelopmentWDL	AnalyzeWDL implementation for the development version using ANTLR4.

Attributes

logger

toil.wdl.versions.dev.**logger**

class toil.wdl.versions.dev.**AnalyzeDevelopmentWDL**(*wdl_file*)

Bases: *toil.wdl.versions.v1.AnalyzeV1WDL*

AnalyzeWDL implementation for the development version using ANTLR4.

See: **https://github.com/openwdl/wdl/blob/main/versions/development/SPEC.md**
https://github.com/openwdl/wdl/blob/main/versions/development/parsers/antlr4/WdlParser.g4

Parameters
wdl_file (*str*) –

property version: str

Returns the version of the WDL document as a string.

Return type
str

analyze()

Analyzes the WDL file passed into the constructor and generates the two intermediate data structures: *self.workflows_dictionary* and *self.tasks_dictionary*.

visit_document(*ctx*)

Similar to version 1.0, except the 'workflow' element is included in *ctx.document_element()*.

Parameters
ctx (*wdlparse.dev.WdlParser.WdlParser.DocumentContext*) –

Return type
None

visit_document_element(*ctx*)

Similar to version 1.0, except this also contains 'workflow'.

Parameters
ctx (*wdlparse.dev.WdlParser.WdlParser.Document_elementContext*) –

Return type
None

visit_call(*ctx*)

Similar to version 1.0, except *ctx.call_afters()* is added.

Parameters
ctx (*wdlparse.dev.WdlParser.WdlParser.CallContext*) –

Return type
dict

visit_string_expr_part(*ctx*)

Similar to version 1.0, except *ctx.expression_placeholder_option()* is removed.

Parameters
ctx (*wdlparse.dev.WdlParser.WdlParser.String_expr_partContext*) –

Return type
str

visit_wdl_type(*ctx*)

Similar to version 1.0, except Directory type is added.

Parameters
ctx (*wdlparse.dev.WdlParser.WdlParser.Wdl_typeContext*) –

Return type
toil.wdl.wdl_types.WDLType

visit_expr_core(*expr*)

Similar to version 1.0, except struct literal is added.

Parameters
expr (*wdlparse.dev.WdlParser.WdlParser.Expr_coreContext*) –

Return type
str

toil.wdl.versions.draft2

Module Contents

Classes

AnalyzeDraft2WDL	AnalyzeWDL implementation for the draft-2 version.

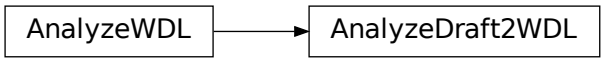
Attributes

logger

toil.wdl.versions.draft2.**logger**

class toil.wdl.versions.draft2.**AnalyzeDraft2WDL**(*wdl_file*)

Bases: *toil.wdl.wdl_analysis.AnalyzeWDL*



AnalyzeWDL implementation for the draft-2 version.

Parameters
wdl_file (*str*) –

property **version**: **str**

Returns the version of the WDL document as a string.

Return type
str

analyze()

Analyzes the WDL file passed into the constructor and generates the two intermediate data structures: *self.workflows_dictionary* and *self.tasks_dictionary*.

Returns
Returns nothing.

write_AST(*out_dir=None*)

Writes a file with the AST for a wdl file in the out_dir.

find_asts(*ast_root*, *name*)

Finds an AST node with the given name and the entire subtree under it. A function borrowed from scottfrazer. Thank you Scott Frazer!

Parameters

- **ast_root** – The WDL AST. The whole thing generally, but really any portion that you wish to search.
- **name** – The name of the subtree you're looking for, like "Task".

Returns
nodes representing the AST subtrees matching the "name" given.

create_tasks_dict(*ast*)

Parse each "Task" in the AST. This will create self.tasks_dictionary, where each task name is a key.

Returns
Creates the self.tasks_dictionary necessary for much of the

parser. Returning it is only necessary for unittests.

parse_task(*task*)

Parses a WDL task AST subtree.

Currently looks at and parses 4 sections: 1. Declarations (e.g. string x = 'helloworld') 2. Commandline (a bash command with dynamic variables inserted) 3. Runtime (docker image; disk; CPU; RAM; etc.) 4. Outputs (expected return values/files)

Parameters
task – An AST subtree of a WDL "Task".

Returns
Returns nothing but adds a task to the self.tasks_dictionary

necessary for much of the parser.

parse_task_rawcommand_attributes(*code_snippet*)

Parameters
code_snippet –

Returns

parse_task_rawcommand(*rawcommand_subAST*)

Parses the rawcommand section of the WDL task AST subtree.

Task "rawcommands" are divided into many parts. There are 2 types of parts: normal strings, & variables that can serve as changeable inputs.

The following example command:
'echo ${variable1} ${variable2} > output_file.txt'

Has 5 parts:
Normal String: 'echo ' Variable Input: variable1 Normal String: ' ' Variable Input: variable2 Normal String: ' > output_file.txt'

Variables can also have additional conditions, like 'sep', which is like the python ''.join() function and in WDL looks like: ${sep=" -V " GVCFs} and would be translated as: ' -V '.join(GVCFs).

Parameters
rawcommand_subAST – A subAST representing some bash command.

Returns

A list=[] of tuples=() representing the parts of the command: e.g. [(command_var, command_type, additional_conditions_list), ...]

Where: command_var = 'GVCFs'
command_type = 'variable' command_actions = {'sep': ' -V '}

modify_cmd_expr_w_attributes(*code_expr*, *code_attr*)

Parameters

- **code_expr** –
- **code_attr** –

Returns

parse_task_runtime_key(*i*)

Parameters
runtime_subAST –

Returns

parse_task_runtime(*runtime_subAST*)

Parses the runtime section of the WDL task AST subtree.

The task "runtime" section currently supports context fields for a docker container, CPU resources, RAM resources, and disk resources.

Parameters
runtime_subAST – A subAST representing runtime parameters.

Returns

A list=[] of runtime attributes, for example: runtime_attributes = [('docker','quay.io/encode-dcc/map:v1.0'),

('cpu','2'), ('memory','17.1 GB'), ('disks','local-disk 420 HDD')]

parse_task_outputs(*i*)

Parse the WDL output section.

Outputs are like declarations, with a type, name, and value. Examples:

Simple Cases

'Int num = 7'
var_name: 'num' var_type: 'Int' var_value: 7

String idea = 'Lab grown golden eagle burgers.'
var_name: 'idea' var_type: 'String' var_value: 'Lab grown golden eagle burgers.'

File ideaFile = 'goldenEagleStemCellStartUpDisrupt.txt'
var_name: 'ideaFile' var_type: 'File' var_value: 'goldenEagleStemCellStartUpDisrupt.txt'

More Abstract Cases

Array[File] allOfMyTerribleIdeas = glob(*.txt)[0]
var_name: 'allOfMyTerribleIdeas' var_type**: 'File' var_value: [*.txt] var_actions: {'index_lookup': '0', 'glob': 'None'}

**toilwdl.py converts 'Array[File]' to 'ArrayFile'

return
output_array representing outputs generated by the job/task: e.g. x = [(var_name, var_type, var_value, var_actions), ...]

translate_wdl_string_to_python_string(*some_string*)

Parses a string representing a given job's output filename into something python can read. Replaces ${string}'s with normal variables and the rest with normal strings all concatenated with ' + '.

Will not work with additional parameters, such as: ${default="foo" bar} or ${true="foo" false="bar" Boolean baz}

This method expects to be passed only strings with some combination of "${abc}" and "abc" blocks.

Parameters

- **job** – A list such that: (job priority #, job ID #, Job Skeleton Name, Job Alias)
- **some_string** – e.g. '${sampleName}.vcf'

Returns
output_string, e.g. 'sampleName + ".vcf"'

create_workflows_dict(*ast*)

Parse each "Workflow" in the AST. This will create self.workflows_dictionary, where each called job is a tuple key of the form: (priority#, job#, name, alias).

Returns
Creates the self.workflows_dictionary necessary for much of the

parser. Returning it is only necessary for unittests.

`parse_workflow`(*workflow*)

Parses a WDL workflow AST subtree.

Returns nothing but creates the self.workflows_dictionary necessary for much of the parser.

Parameters
`workflow` – An AST subtree of a WDL "Workflow".

Returns
Returns nothing but adds a workflow to the self.workflows_dictionary necessary for much of the parser.

`parse_workflow_body`(*i*)

Currently looks at and parses 3 sections: 1. Declarations (e.g. String x = 'helloworld') 2. Calls (similar to a python def) 3. Scatter (which expects to map to a Call or multiple Calls) 4. Conditionals

`parse_workflow_if`(*ifAST*)

`parse_workflow_if_expression`(*i*)

`parse_workflow_scatter`(*scatterAST*)

`parse_workflow_scatter_item`(*i*)

`parse_workflow_scatter_collection`(*i*)

`parse_declaration`(*ast*)

Parses a WDL declaration AST subtree into a Python tuple.

Examples:

String my_name String your_name Int two_chains_i_mean_names = 0

Parameters
`ast` – Some subAST representing a task declaration like: 'String file_name'

Returns

var_name, var_type, var_value Example:

Input subAST representing: 'String file_name' Output: var_name='file_name', var_type='String', var_value=None

`parse_declaration_name`(*nameAST*)

Required.

Nothing fancy here. Just the name of the workflow function. For example: "rnaseqexample" would be the following wdl workflow's name:

workflow rnaseqexample {File y; call a {inputs: y}; call b;} task a {File y} task b {command{"echo 'ATCG'"}}

Parameters
`nameAST` –

Returns

parse_declaration_type(*typeAST*)

Required.

Currently supported: Types are: Boolean, Float, Int, File, String, Array[subtype],

Pair[subtype, subtype], and Map[subtype, subtype].

OptionalTypes are: Boolean?, Float?, Int?, File?, String?, Array[subtype]?,
Pair[subtype, subtype]?, and Map[subtype, subtype]?.

Python is not typed, so we don't need typing except to identify type: "File", which Toil needs to import, so we recursively travel down to the innermost type which will tell us if the variables are files that need importing.

For Pair and Map compound types, we recursively travel down the subtypes and store them as attributes of a *WDLType* string. This way, the type structure is preserved, which will allow us to import files appropriately.

Parameters
typeAST –

Returns
a WDLType instance

parse_declaration_expressn(*expressionAST*, *es*)

Expressions are optional. Workflow declaration valid examples:

File x

or

File x = '/x/x.tmp'

Parameters
expressionAST –

Returns

parse_declaration_expressn_logicalnot(*exprssn*, *es*)

parse_declaration_expressn_arraymaplookup(*lhsAST*, *rhsAST*, *es*)

Parameters

- **lhsAST** –
- **rhsAST** –
- **es** –

Returns

parse_declaration_expressn_memberaccess(*lhsAST*, *rhsAST*, *es*)

Instead of "Class.variablename", use "Class.rv('variablename')".

Parameters

- **lhsAST** –
- **rhsAST** –
- **es** –

Returns

parse_declaration_expressn_ternaryif(*cond*, *iftrue*, *iffalse*, *es*)

Classic if statement. This needs to be rearranged.

In wdl, this looks like: if <condition> then <iftrue> else <iffalse>

In python, this needs to be: <iftrue> if <condition> else <iffalse>

Parameters

- **cond** –
- **iftrue** –
- **iffalse** –
- **es** –

Returns

parse_declaration_expressn_tupleliteral(*values*, *es*)

Same in python. Just a parenthesis enclosed tuple.

Parameters

- **values** –
- **es** –

Returns

parse_declaration_expressn_arrayliteral(*values*, *es*)

Same in python. Just a square bracket enclosed array.

Parameters

- **values** –
- **es** –

Returns

parse_declaration_expressn_operator(*lhsAST*, *rhsAST*, *es*, *operator*)

Simply joins the left and right hand arguments lhs and rhs with an operator.

Parameters

- **lhsAST** –
- **rhsAST** –
- **es** –
- **operator** –

Returns

parse_declaration_expressn_fncall(*name*, *params*, *es*)

Parses out cromwell's built-in function calls.

Some of these are special and need minor adjustments, for example size() requires a fileStore.

Parameters

- **name** –
- **params** –
- **es** –

Returns

parse_declaration_expressn_fncall_normalparams(*params*)

parse_workflow_call_taskname(*i*)

Required.

Parameters
i –

Returns

parse_workflow_call_taskalias(*i*)

Required.

Parameters
i –

Returns

parse_workflow_call_body_declarations(*i*)

Have not seen this used, so expects to return "[]".

Parameters
i –

Returns

parse_workflow_call_body_io(*i*)

Required.

Parameters
i –

Returns

parse_workflow_call_body_io_map(*i*)

Required.

Parameters
i –

Returns

parse_workflow_call_body(*i*)

Required.

Parameters
i –

Returns

parse_workflow_call(*i*)

Parses a WDL workflow call AST subtree to give the variable mappings for that particular job/task "call".

Parameters
i – WDL workflow job object

Returns
python dictionary of io mappings for that job call

toil.wdl.versions.v1

Module Contents

Classes

AnalyzeV1WDL	AnalyzeWDL implementation for the 1.0 version using ANTLR4.

Functions

is_context(ctx, classname)	Returns whether an ANTLR4 context object is of the precise type *classname*.

Attributes

logger

toil.wdl.versions.v1.**logger**

toil.wdl.versions.v1.**is_context**(*ctx*, *classname*)

Returns whether an ANTLR4 context object is of the precise type *classname*.

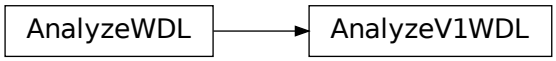
Parameters

- **ctx** – An ANTLR4 context object.
- **classname** (*Union[str, tuple]*) – The class name(s) as a string or a tuple of strings. If a tuple is provided, this returns True if the context object matches one of the class names.

Return type
bool

class toil.wdl.versions.v1.**AnalyzeV1WDL**(*wdl_file*)

Bases: *toil.wdl.wdl_analysis.AnalyzeWDL*



AnalyzeWDL implementation for the 1.0 version using ANTLR4.

See: **https://github.com/openwdl/wdl/blob/main/versions/1.0/SPEC.md**
https://github.com/openwdl/wdl/blob/main/versions/1.0/parsers/antlr4/WdlV1Parser.g4

Parameters
wdl_file (*str*) –

property version: str

Returns the version of the WDL document as a string.

Return type
str

analyze()

Analyzes the WDL file passed into the constructor and generates the two intermediate data structures: *self.workflows_dictionary* and *self.tasks_dictionary*.

visit_document(*ctx*)

Root of tree. Contains *version* followed by an optional workflow and any number of `document_element`s.

visit_document_element(*ctx*)

Contains one of the following: 'import_doc', 'struct', or 'task'.

visit_workflow(*ctx*)

Contains an 'identifier' and an array of `workflow_element`s.

visit_workflow_input(*ctx*)

Contains an array of 'any_decls', which can be unbound or bound declarations. Example:

input {
String in_str = "twenty" Int in_int

}

Returns a list of tuple=(name, type, expr).

visit_workflow_output(*ctx*)

Contains an array of 'bound_decls' (unbound_decls not allowed). Example:

output {
String out_str = "output"

}

Returns a list of tuple=(name, type, expr).

visit_inner_workflow_element(*ctx*)

Returns a tuple=(unique_key, dict), where dict contains the contents of the given inner workflow element.

visit_call(*ctx*)

Pattern: CALL call_name call_alias? call_body? Example WDL syntax: call task_1 {input: arr=arr}

Returns a dict={task, alias, io}.

visit_scatter(*ctx*)

Pattern: SCATTER LPAREN Identifier In expr RPAREN LBRACE inner_workflow_element* RBRACE Example WDL syntax: scatter (i in items) { … }

Returns a dict={item, collection, body}.

visit_conditional(*ctx*)

Pattern: IF LPAREN expr RPAREN LBRACE inner_workflow_element* RBRACE Example WDL syntax: if (condition) { … }

Returns a dict={expression, body}.

`visit_task`(*ctx*)

Root of a task definition. Contains an *identifier* and an array of `task_element`s.

`visit_task_input`(*ctx*)

Contains an array of 'any_decls', which can be unbound or bound declarations. Example:

input {
String in_str = "twenty" Int in_int

}

Returns a list of tuple=(name, type, expr)

`visit_task_output`(*ctx*)

Contains an array of 'bound_decls' (unbound_decls not allowed). Example:

output {
String out_str = read_string(stdout())

}

Returns a list of tuple=(name, type, expr)

`visit_task_command`(*ctx*)

Parses the command section of the WDL task.

Contains a *string_part* plus any number of `expr_with_string`s. The following example command:

'echo ${var1} ${var2} > output_file.txt'

Has 3 parts:

1. string_part: 'echo '
2. **expr_with_string, which has two parts:**
   - expr_part: 'var1'
   - string_part: ' '

1. **expr_with_string, which has two parts:**
   - expr_part: 'var2'
   - string_part: ' > output_file.txt'

Returns a list=[] of strings representing the parts of the command.
e.g. [string_part, expr_part, string_part, ...]

`visit_task_command_string_part`(*ctx*)

Returns a string representing the string_part.

`visit_task_command_expr_with_string`(*ctx*)

Returns a tuple=(expr_part, string_part).

`visit_task_command_expr_part`(*ctx*)

Contains the expression inside ${expr}. Same function as *self.visit_string_expr_part()*.

Returns the expression.

visit_task_runtime(*ctx*)

Contains an array of `task_runtime_kv`s.

Returns a dict={key: value} where key can be 'docker', 'cpu', 'memory', 'cores', or 'disks'.

visit_any_decls(*ctx*)

Contains a bound or unbound declaration.

visit_unbound_decls(*ctx*)

Contains an unbound declaration. E.g.: *String in_str*.

Returns a tuple=(name, type, expr), where *expr* is None.

visit_bound_decls(*ctx*)

Contains a bound declaration. E.g.: *String in_str = "some string"*.

Returns a tuple=(name, type, expr).

visit_wdl_type(*ctx*)

Returns a WDLType instance.

visit_primitive_literal(*ctx*)

Returns the primitive literal as a string.

visit_number(*ctx*)

Contains an *IntLiteral* or a *FloatLiteral*.

visit_string(*ctx*)

Contains a *string_part* followed by an array of `string_expr_with_string_part`s.

visit_string_expr_with_string_part(*ctx*)

Contains a *string_expr_part* and a *string_part*.

visit_string_expr_part(*ctx*)

Contains an array of *expression_placeholder_option`s and an `expr*.

visit_string_part(*ctx*)

Returns a string representing the string_part.

visit_expression_placeholder_option(*ctx*)

Expression placeholder options.

Can match one of the following:

BoolLiteral EQUAL (string | number) DEFAULT EQUAL (string | number) SEP EQUAL (string | number)

See https://github.com/openwdl/wdl/blob/main/versions/1.0/SPEC.md#expression-placeholder-options

e.g.: ${sep=", " array_value} e.g.: ${true="–yes" false="–no" boolean_value} e.g.: ${default="foo" optional_value}

Returns a tuple=(key, value)

visit_expr(*ctx*)

Expression root.

visit_infix0(*ctx*)

Expression infix0 (LOR).

visit_lor(*ctx*)

Logical OR expression.

visit_infix1(*ctx*)

Expression infix1 (LAND).

visit_land(*ctx*)

Logical AND expresion.

visit_infix2(*ctx*)

Expression infix2 (comparisons).

visit_infix3(*ctx*)

Expression infix3 (add/subtract).

visit_infix4(*ctx*)

Expression infix4 (multiply/divide/modulo).

visit_infix5(*ctx*)

Expression infix5.

visit_expr_core(*expr*)

Expression core.

visit_apply(*ctx*)

A function call. Pattern: Identifier LPAREN (expr (COMMA expr)*)? RPAREN

visit_array_literal(*ctx*)

Pattern: LBRACK (expr (COMMA expr)*)* RBRACK

visit_pair_literal(*ctx*)

Pattern: LPAREN expr COMMA expr RPAREN

visit_ifthenelse(*ctx*)

Ternary expression. Pattern: IF expr THEN expr ELSE expr

visit_expression_group(*ctx*)

Pattern: LPAREN expr RPAREN

visit_at(*ctx*)

Array or map lookup. Pattern: expr_core LBRACK expr RBRACK

visit_get_name(*ctx*)

Member access. Pattern: expr_core DOT Identifier

visit_negate(*ctx*)

Pattern: NOT expr

visit_unarysigned(*ctx*)

Pattern: (PLUS | MINUS) expr

visit_primitives(*ctx*)

Expression alias for primitive literal.

Submodules

toil.wdl.toilwdl

Module Contents

Functions

main()	A program to run WDL input files using native Toil scripts.

Attributes

logger

toil.wdl.toilwdl.**logger**

toil.wdl.toilwdl.**main**()

A program to run WDL input files using native Toil scripts.

Calls two files, described below, wdl_analysis.py and wdl_synthesis.py:

wdl_analysis reads the wdl and restructures them into 2 intermediate data structures before writing (python dictionaries):

"wf_dictionary": containing the parsed workflow information. "tasks_dictionary": containing the parsed task information.

wdl_synthesis takes the "wf_dictionary", "tasks_dictionary", and the JSON file and uses them to write a native python script for use with Toil.

Requires a WDL file, and a JSON file. The WDL file contains ordered commands, and the JSON file contains input values for those commands. To run in Toil, these two files must be parsed, restructured into python dictionaries, and then compiled into a Toil formatted python script. This compiled Toil script is deleted unless the user specifies: "–dev_mode" as an option.

The WDL parser was auto-generated from the Broad's current WDL grammar file: https://github.com/openwdl/wdl/blob/master/parsers/grammar.hgr using Scott Frazer's Hermes: https://github.com/scottfrazer/hermes Thank you Scott Frazer!

Currently in alpha testing, and known to work with the Broad's GATK tutorial set for WDL on their main wdl site: software.broadinstitute.org/wdl/documentation/topic?name=wdl-tutorials

And ENCODE's WDL workflow: github.com/ENCODE-DCC/pipeline-container/blob/master/local-workflows/encode_mapping_workflow.wdl

Additional support to be broadened to include more features soon.

toil.wdl.utils

Module Contents

Functions

get_version(iterable)	Get the version of the WDL document.
get_analyzer(wdl_file)	Creates an instance of an AnalyzeWDL implementation based on the version.
dict_from_JSON(JSON_file)	Takes a WDL-mapped json file and creates a dict containing the bindings.
write_mappings(parser[, filename])	Takes an AnalyzeWDL instance and writes the final task dict and workflow

toil.wdl.utils.**get_version**(*iterable*)

Get the version of the WDL document.

Parameters
iterable – An iterable that contains the lines of a WDL document.

Returns
The WDL version used in the workflow.

Return type
str

toil.wdl.utils.**get_analyzer**(*wdl_file*)

Creates an instance of an AnalyzeWDL implementation based on the version.

Parameters
wdl_file (*str*) – The path to the WDL file.

Return type
toil.wdl.wdl_analysis.AnalyzeWDL

toil.wdl.utils.**dict_from_JSON**(*JSON_file*)

Takes a WDL-mapped json file and creates a dict containing the bindings.

Parameters
JSON_file (*str*) – A required JSON file containing WDL variable bindings.

Return type
dict

toil.wdl.utils.**write_mappings**(*parser*, *filename='mappings.out'*)

Takes an AnalyzeWDL instance and writes the final task dict and workflow dict to the given file.

Parameters

- **parser** (toil.wdl.wdl_analysis.AnalyzeWDL) – An AnalyzeWDL instance.
- **filename** (*str*) – The name of a file to write to.

Return type
None

toil.wdl.wdl_analysis

Module Contents

Classes

AnalyzeWDL	An interface to analyze a WDL file. Each version corresponds to a subclass that

Attributes

logger

toil.wdl.wdl_analysis.**logger**

class toil.wdl.wdl_analysis.**AnalyzeWDL**(*wdl_file*)

An interface to analyze a WDL file. Each version corresponds to a subclass that restructures the WDL document into 2 intermediate data structures (python dictionaries):

"workflows_dictionary": containing the parsed workflow information. "tasks_dictionary": containing the parsed task information.

These are then fed into wdl_synthesis.py which uses them to write a native python script for use with Toil.

Requires a WDL file. The WDL file contains ordered commands.

Parameters
wdl_file (*str*) –

abstract property version: str

Returns the version of the WDL document as a string.

Return type
str

primitive_types

compound_types

analyze()

Analyzes the WDL file passed into the constructor and generates the two intermediate data structures: *self.workflows_dictionary* and *self.tasks_dictionary*.

Returns
Returns nothing.

write_AST(*out_dir*)

Writes a file with the AST for a wdl file in the out_dir.

create_wdl_primitive_type(*key*, *optional=False*)

Returns an instance of WDLType.

Parameters

- **key** (*str*) –

- **optional** (*bool*) –

create_wdl_compound_type(*key*, *elements*, *optional=False*)

Returns an instance of WDLCompoundType.

Parameters

- **key** (*str*) –
- **elements** (*list*) –
- **optional** (*bool*) –

toil.wdl.wdl_functions

Module Contents

Classes

WDLJSONEncoder	Extended JSONEncoder to support WDL-specific JSON encoding.

Functions

generate_docker_bashscript_file(temp_dir, docker_dir, ...)	Creates a bashscript to inject into a docker container for the job.
process_single_infile(wdl_file, fileStore)	
process_infile(f, fileStore)	Takes any input and imports the WDLFile into the fileStore.
sub(input_str, pattern, replace)	Given 3 String parameters *input*, *pattern*, *replace*, this function will
defined(i)	
process_single_outfile(wdl_file, fileStore, workDir, ...)	
process_outfile(f, fileStore, workDir, outDir)	
abspath_single_file(f, cwd)	
abspath_file(f, cwd)	
read_single_file(f, tempDir, fileStore[, docker])	
read_file(f, tempDir, fileStore[, docker])	
process_and_read_file(f, tempDir, fileStore[, docker])	
generate_stdout_file(output, tempDir, fileStore[, stderr])	Create a stdout (or stderr) file from a string or bytes object.

continues on next page

Table 2 – continued from previous page

parse_memory(memory)	Parses a string representing memory and returns
parse_cores(cores)	
parse_disk(disk)	
is_number(s)	
size([f, unit, fileStore])	Given a *File* and a *String* (optional), returns the size of the file in Bytes
select_first(values)	
combine_dicts(dict1, dict2)	
basename(path[, suffix])	https://software.broadinstitute.org/wdl/documentation/article?id=10554
heredoc_wdl(template[, dictionary, indent])	
floor(i)	Converts a Float value into an Int by rounding down to the next lower integer.
ceil(i)	Converts a Float value into an Int by rounding up to the next higher integer.
read_lines(path)	Given a file-like object (*String*, *File*) as a parameter, this will read each
read_tsv(path[, delimiter])	Take a tsv filepath and return an array; e.g. [[],[],[]].
read_csv(path)	Take a csv filepath and return an array; e.g. [[],[],[]].
read_json(path)	The *read_json()* function takes one parameter, which is a file-like object
read_map(path)	Given a file-like object (*String*, *File*) as a parameter, this will read each
read_int(path)	The *read_int()* function takes a file path which is expected to contain 1
read_string(path)	The *read_string()* function takes a file path which is expected to contain 1
read_float(path)	The *read_float()* function takes a file path which is expected to contain 1
read_boolean(path)	The *read_boolean()* function takes a file path which is expected to contain 1
write_lines(in_lines[, temp_dir, file_store])	Given something that's compatible with *Array[String]*, this writes each element
write_tsv(in_tsv[, delimiter, temp_dir, file_store])	Given something that's compatible with *Array[Array[String]]*, this writes a TSV
write_json(in_json[, indent, separators, temp_dir, ...])	Given something with any type, this writes the JSON equivalent to a file. See
write_map(in_map[, temp_dir, file_store])	Given something that's compatible with *Map[String, String]*, this writes a TSV
wdl_range(num)	Given an integer argument, the range function creates an array of integers of
transpose(in_array)	Given a two dimensional array argument, the transpose function transposes the
length(in_array)	Given an Array, the *length* function returns the number of elements in the Array

continues on next page

Table 2 – continued from previous page

wdl_zip(left, right)	Return the dot product of the two arrays. If the arrays have different lengths
cross(left, right)	Return the cross product of the two arrays. Array[Y][1] appears before
as_pairs(in_map)	Given a Map, the *as_pairs* function returns an Array containing each element
as_map(in_array)	Given an Array consisting of Pairs, the *as_map* function returns a Map in
keys(in_map)	Given a Map, the *keys* function returns an Array consisting of the keys in
collect_by_key(in_array)	Given an Array consisting of Pairs, the *collect_by_key* function returns a Map
flatten(in_array)	Given an array of arrays, the *flatten* function concatenates all the member

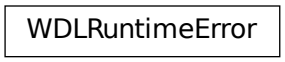
Attributes

logger

toil.wdl.wdl_functions.**logger**

exception toil.wdl.wdl_functions.**WDLRuntimeError**(*message*)

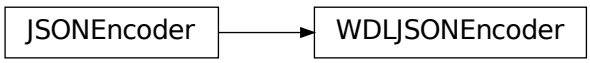
Bases: Exception



WDL-related run-time error.

class toil.wdl.wdl_functions.**WDLJSONEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: json.JSONEncoder



Extended JSONEncoder to support WDL-specific JSON encoding.

default(*obj*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

toil.wdl.wdl_functions.**generate_docker_bashscript_file**(*temp_dir*, *docker_dir*, *globs*, *cmd*, *job_name*)

Creates a bashscript to inject into a docker container for the job.

This script wraps the job command(s) given in a bash script, hard links the outputs and returns an "rc" file containing the exit code. All of this is done in an effort to parallel the Broad's cromwell engine, which is the native WDL runner. As they've chosen to write and then run a bashscript for every command, so shall we.

Parameters

- **temp_dir** – The current directory outside of docker to deposit the bashscript into, which will be the bind mount that docker loads files from into its own containerized filesystem. This is usually the tempDir created by this individual job using 'tempDir = job.fileStore.getLocalTempDir()'.
- **docker_dir** – The working directory inside of the docker container which is bind mounted to 'temp_dir'. By default this is 'data'.
- **globs** – A list of expected output files to retrieve as glob patterns that will be returned as hard links to the current working directory.
- **cmd** – A bash command to be written into the bash script and run.
- **job_name** – The job's name, only used to write in a file name identifying the script as written for that job. Will be used to call the script later.

Returns

Nothing, but it writes and deposits a bash script in temp_dir intended to be run inside of a docker container for this job.

toil.wdl.wdl_functions.**process_single_infile**(*wdl_file*, *fileStore*)

Parameters

- **wdl_file** (toil.wdl.wdl_types.WDLFile) –
- **fileStore** (toil.fileStores.abstractFileStore.AbstractFileStore) –

Return type

toil.wdl.wdl_types.WDLFile

toil.wdl.wdl_functions.**process_infile**(*f*, *fileStore*)

Takes any input and imports the WDLFile into the fileStore.

This returns the input importing all WDLFile instances to the fileStore. Toil does not preserve a file's original name upon import and so the WDLFile also keeps track of this.

Parameters

- **f** (*Any*) – A primitive, WDLFile, or a container. A file needs to be a WDLFile instance to be imported.
- **fileStore** (toil.fileStores.abstractFileStore.AbstractFileStore) – The fileStore object that is called to load files into the fileStore.

toil.wdl.wdl_functions.**sub**(*input_str*, *pattern*, *replace*)

Given 3 String parameters *input*, *pattern*, *replace*, this function will replace any occurrence matching *pattern* in *input* by *replace*. *pattern* is expected to be a regular expression. Details of regex evaluation will depend on the execution engine running the WDL.

WDL syntax: String sub(String, String, String)

Parameters

- **input_str** (*str*) –
- **pattern** (*str*) –
- **replace** (*str*) –

Return type
str

toil.wdl.wdl_functions.**defined**(*i*)

toil.wdl.wdl_functions.**process_single_outfile**(*wdl_file*, *fileStore*, *workDir*, *outDir*)

Parameters
wdl_file (toil.wdl.wdl_types.WDLFile) –

Return type
toil.wdl.wdl_types.WDLFile

toil.wdl.wdl_functions.**process_outfile**(*f*, *fileStore*, *workDir*, *outDir*)

toil.wdl.wdl_functions.**abspath_single_file**(*f*, *cwd*)

Parameters

- **f** (toil.wdl.wdl_types.WDLFile) –
- **cwd** (*str*) –

Return type
toil.wdl.wdl_types.WDLFile

toil.wdl.wdl_functions.**abspath_file**(*f*, *cwd*)

Parameters

- **f** (*Any*) –
- **cwd** (*str*) –

toil.wdl.wdl_functions.**read_single_file**(*f*, *tempDir*, *fileStore*, *docker=False*)

Parameters
f (toil.wdl.wdl_types.WDLFile) –

Return type
str

toil.wdl.wdl_functions.**read_file**(*f*, *tempDir*, *fileStore*, *docker=False*)

Parameters

- **f** (*Any*) –
- **tempDir** (*str*) –
- **fileStore** (toil.fileStores.abstractFileStore.AbstractFileStore) –
- **docker** (*bool*) –

toil.wdl.wdl_functions.**process_and_read_file**(*f*, *tempDir*, *fileStore*, *docker=False*)

toil.wdl.wdl_functions.**generate_stdout_file**(*output*, *tempDir*, *fileStore*, *stderr=False*)

Create a stdout (or stderr) file from a string or bytes object.

Parameters

- **output** (*str|bytes*) – A str or bytes object that holds the stdout/stderr text.
- **tempDir** (*str*) – The directory to write the stdout file.
- **fileStore** – A fileStore object.
- **stderr** (*bool*) – If True, a stderr instead of a stdout file is generated.

Returns
The file path to the generated file.

toil.wdl.wdl_functions.**parse_memory**(*memory*)

Parses a string representing memory and returns an integer # of bytes.

Parameters
memory –

Returns

toil.wdl.wdl_functions.**parse_cores**(*cores*)

toil.wdl.wdl_functions.**parse_disk**(*disk*)

toil.wdl.wdl_functions.**is_number**(*s*)

toil.wdl.wdl_functions.**size**(*f=None*, *unit='B'*, *fileStore=None*)

Given a *File* and a *String* (optional), returns the size of the file in Bytes or in the unit specified by the second argument.

Supported units are KiloByte (“K”, “KB”), MegaByte (“M”, “MB”), GigaByte (“G”, “GB”), TeraByte (“T”, “TB”) (powers of 1000) as well as their binary version (https://en.wikipedia.org/wiki/Binary_prefix) “Ki” (“KiB”), “Mi” (“MiB”), “Gi” (“GiB”), “Ti” (“TiB”) (powers of 1024). Default unit is Bytes (“B”).

WDL syntax: Float size(File, [String]) Varieties: Float size(File?, [String])

Float size(Array[File], [String]) Float size(Array[File?], [String])

Parameters

- **f** (*Optional[Union[str, toil.wdl.wdl_types.WDLFile, List[Union[str, toil.wdl.wdl_types.WDLFile]]]]*) –
- **unit** (*Optional[str]*) –

- **fileStore** (*Optional[toil.fileStores.abstractFileStore.AbstractFileStore]*) –

Return type
float

toil.wdl.wdl_functions.**select_first**(*values*)

toil.wdl.wdl_functions.**combine_dicts**(*dict1*, *dict2*)

toil.wdl.wdl_functions.**basename**(*path*, *suffix=None*)
https://software.broadinstitute.org/wdl/documentation/article?id=10554

toil.wdl.wdl_functions.**heredoc_wdl**(*template*, *dictionary={}*, *indent=''*)

toil.wdl.wdl_functions.**floor**(*i*)
Converts a Float value into an Int by rounding down to the next lower integer.

Parameters
i (*Union[int, float]*) –

Return type
int

toil.wdl.wdl_functions.**ceil**(*i*)
Converts a Float value into an Int by rounding up to the next higher integer.

Parameters
i (*Union[int, float]*) –

Return type
int

toil.wdl.wdl_functions.**read_lines**(*path*)
Given a file-like object (*String*, *File*) as a parameter, this will read each line as a string and return an *Array[String]* representation of the lines in the file.

WDL syntax: Array[String] read_lines(String|File)

Parameters
path (*str*) –

Return type
List[str]

toil.wdl.wdl_functions.**read_tsv**(*path*, *delimiter='\t'*)
Take a tsv filepath and return an array; e.g. [[],[],[]].

For example, a file containing:

1 2 3 4 5 6 7 8 9

would return the array: [['1','2','3'], ['4','5','6'], ['7','8','9']]

WDL syntax: Array[Array[String]] read_tsv(String|File)

Parameters

- **path** (*str*) –
- **delimiter** (*str*) –

Return type
List[List[str]]

toil.wdl.wdl_functions.**read_csv**(*path*)

Take a csv filepath and return an array; e.g. [[],[],[]].

For example, a file containing:

1,2,3 4,5,6 7,8,9

would return the array: [['1','2','3'], ['4','5','6'], ['7','8','9']]

Parameters
path (*str*) –

Return type
List[List[str]]

toil.wdl.wdl_functions.**read_json**(*path*)

The *read_json()* function takes one parameter, which is a file-like object
(*String*, *File*) and returns a data type which matches the data structure in the JSON file. See

https://github.com/openwdl/wdl/blob/main/versions/development/SPEC.md#mixed-read_jsonstringfile

WDL syntax: mixed read_json(String|File)

Parameters
path (*str*) –

Return type
Any

toil.wdl.wdl_functions.**read_map**(*path*)

Given a file-like object (*String*, *File*) as a parameter, this will read each line from a file and expect the line to have the format *col1 col2*. In other words, the file-like object must be a two-column TSV file.

WDL syntax: Map[String, String] read_map(String|File)

Parameters
path (*str*) –

Return type
Dict[str, str]

toil.wdl.wdl_functions.**read_int**(*path*)

The *read_int()* function takes a file path which is expected to contain 1 line with 1 integer on it. This function returns that integer.

WDL syntax: Int read_int(String|File)

Parameters
path (*Union[str, toil.wdl.wdl_types.WDLFile]*) –

Return type
int

toil.wdl.wdl_functions.**read_string**(*path*)

The *read_string()* function takes a file path which is expected to contain 1 line with 1 string on it. This function returns that string.

WDL syntax: String read_string(String|File)

Parameters
path (*Union[str, toil.wdl.wdl_types.WDLFile]*) –

Return type
str

toil.wdl.wdl_functions.**read_float**(*path*)

The *read_float()* function takes a file path which is expected to contain 1 line with 1 floating point number on it. This function returns that float.

WDL syntax: Float read_float(String|File)

Parameters
path (*Union[str, toil.wdl.wdl_types.WDLFile]*) –

Return type
float

toil.wdl.wdl_functions.**read_boolean**(*path*)

The *read_boolean()* function takes a file path which is expected to contain 1 line with 1 Boolean value (either "true" or "false" on it). This function returns that Boolean value.

WDL syntax: Boolean read_boolean(String|File)

Parameters
path (*Union[str, toil.wdl.wdl_types.WDLFile]*) –

Return type
bool

toil.wdl.wdl_functions.**write_lines**(*in_lines*, *temp_dir=None*, *file_store=None*)

Given something that's compatible with *Array[String]*, this writes each element to it's own line on a file. with newline `

` characters as line separators.

WDL syntax: File write_lines(Array[String])

Parameters

- **in_lines** (*List[str]*) –
- **temp_dir** (*Optional[str]*) –
- **file_store** (*Optional[toil.fileStores.abstractFileStore.AbstractFileStore]*) –

Return type
str

toil.wdl.wdl_functions.**write_tsv**(*in_tsv*, *delimiter='\t'*, *temp_dir=None*, *file_store=None*)

Given something that's compatible with *Array[Array[String]]*, this writes a TSV file of the data structure.

WDL syntax: File write_tsv(Array[Array[String]])

Parameters

- **in_tsv** (*List[List[str]]*) –
- **delimiter** (*str*) –
- **temp_dir** (*Optional[str]*) –
- **file_store** (*Optional[toil.fileStores.abstractFileStore.AbstractFileStore]*) –

Return type
str

toil.wdl.wdl_functions.**write_json**(*in_json*, *indent=None*, *separators=(', ', ':')*, *temp_dir=None*, *file_store=None*)

Given something with any type, this writes the JSON equivalent to a file. See the table in the definition of https://github.com/openwdl/wdl/blob/main/versions/development/SPEC.md#mixed-read_jsonstringfile

WDL syntax: File write_json(mixed)

Parameters

- **in_json** (*Any*) –
- **indent** (*Union[None, int, str]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **temp_dir** (*Optional[str]*) –
- **file_store** (*Optional[toil.fileStores.abstractFileStore.AbstractFileStore]*) –

Return type
str

toil.wdl.wdl_functions.**write_map**(*in_map*, *temp_dir=None*, *file_store=None*)

Given something that's compatible with *Map[String, String]*, this writes a TSV
file of the data structure.

WDL syntax: File write_map(Map[String, String])

Parameters

- **in_map** (*Dict[str, str]*) –
- **temp_dir** (*Optional[str]*) –
- **file_store** (*Optional[toil.fileStores.abstractFileStore.AbstractFileStore]*) –

Return type
str

toil.wdl.wdl_functions.**wdl_range**(*num*)

Given an integer argument, the range function creates an array of integers of length equal to the given argument.

WDL syntax: Array[Int] range(Int)

Parameters
num (*int*) –

Return type
List[int]

toil.wdl.wdl_functions.**transpose**(*in_array*)

Given a two dimensional array argument, the transpose function transposes the two dimensional array according to the standard matrix transpose rules.

WDL syntax: Array[Array[X]] transpose(Array[Array[X]])

Parameters
in_array (*List[List[Any]]*) –

Return type
List[List[Any]]

toil.wdl.wdl_functions.**length**(*in_array*)

Given an Array, the *length* function returns the number of elements in the Array as an Int.

Parameters
in_array (*List[Any]*) –

Return type
int

toil.wdl.wdl_functions.**wdl_zip**(*left*, *right*)

Return the dot product of the two arrays. If the arrays have different lengths it is an error.

WDL syntax: Array[Pair[X,Y]] zip(Array[X], Array[Y])

Parameters

- **left** (*List[Any]*) –
- **right** (*List[Any]*) –

Return type
List[*toil.wdl.wdl_types.WDLPair*]

toil.wdl.wdl_functions.**cross**(*left*, *right*)

Return the cross product of the two arrays. Array[Y][1] appears before Array[X][1] in the output.

WDL syntax: Array[Pair[X,Y]] cross(Array[X], Array[Y])

Parameters

- **left** (*List[Any]*) –
- **right** (*List[Any]*) –

Return type
List[*toil.wdl.wdl_types.WDLPair*]

toil.wdl.wdl_functions.**as_pairs**(*in_map*)

Given a Map, the *as_pairs* function returns an Array containing each element in the form of a Pair. The key will be the left element of the Pair and the value the right element. The order of the the Pairs in the resulting Array is the same as the order of the key/value pairs in the Map.

WDL syntax: Array[Pair[X,Y]] as_pairs(Map[X,Y])

Parameters
in_map (*dict*) –

Return type
List[*toil.wdl.wdl_types.WDLPair*]

toil.wdl.wdl_functions.**as_map**(*in_array*)

Given an Array consisting of Pairs, the *as_map* function returns a Map in which the left elements of the Pairs are the keys and the right elements the values.

WDL syntax: Map[X,Y] as_map(Array[Pair[X,Y]])

Parameters
in_array (*List[toil.wdl.wdl_types.WDLPair]*) –

Return type
dict

toil.wdl.wdl_functions.**keys**(*in_map*)

Given a Map, the *keys* function returns an Array consisting of the keys in the Map. The order of the keys in the resulting Array is the same as the order of the Pairs in the Map.

WDL syntax: Array[X] keys(Map[X,Y])

Parameters
in_map (*dict*) –

Return type
list

toil.wdl.wdl_functions.**collect_by_key**(*in_array*)

Given an Array consisting of Pairs, the *collect_by_key* function returns a Map in which the left elements of the Pairs are the keys and the right elements the values.

WDL syntax: Map[X,Array[Y]] collect_by_key(Array[Pair[X,Y]])

Parameters
in_array (*List[toil.wdl.wdl_types.WDLPair]*) –

Return type
dict

toil.wdl.wdl_functions.**flatten**(*in_array*)

Given an array of arrays, the *flatten* function concatenates all the member arrays in the order to appearance to give the result. It does not deduplicate the elements.

WDL syntax: Array[X] flatten(Array[Array[X]])

Parameters
in_array (*List[list]*) –

Return type
list

toil.wdl.wdl_synthesis

Module Contents

Classes

SynthesizeWDL	SynthesizeWDL takes the "workflows_dictionary" and "tasks_dictionary" produced by

Attributes

logger

toil.wdl.wdl_synthesis.**logger**

class toil.wdl.wdl_synthesis.**SynthesizeWDL**(*version*, *tasks_dictionary*, *workflows_dictionary*, *output_directory*, *json_dict*, *docker_user*, *jobstore=None*, *destBucket=None*)

SynthesizeWDL takes the "workflows_dictionary" and "tasks_dictionary" produced by wdl_analysis.py and uses them to write a native python script for use with Toil.

A WDL "workflow" section roughly corresponds to the python "main()" function, where functions are wrapped as Toil "jobs", output dependencies specified, and called.

A WDL "task" section corresponds to a unique python function, which will be wrapped as a Toil "job" and defined outside of the "main()" function that calls it.

Generally this handles breaking sections into their corresponding Toil counterparts.

For example: write the imports, then write all functions defining jobs (which have subsections like: write header, define variables, read "File" types into the jobstore, docker call, etc.), then write the main and all of its subsections.

Parameters

- **version** (*str*) –
- **tasks_dictionary** (*dict*) –
- **workflows_dictionary** (*dict*) –
- **output_directory** (*str*) –
- **json_dict** (*dict*) –
- **docker_user** (*str*) –
- **jobstore** (*Optional[str]*) –
- **destBucket** (*Optional[str]*) –

write_modules()

write_main()

Writes out a huge string representing the main section of the python compiled toil script.

Currently looks at and writes 5 sections: 1. JSON Variables (includes importing and preparing files as tuples) 2. TSV Variables (includes importing and preparing files as tuples) 3. CSV Variables (includes importing and preparing files as tuples) 4. Wrapping each WDL "task" function as a toil job 5. List out children and encapsulated jobs by priority, then start job0.

This should create variable declarations necessary for function calls. Map file paths appropriately and store them in the toil fileStore so that they are persistent from job to job. Create job wrappers for toil. And finally write out, and run the jobs in order of priority using the addChild and encapsulate commands provided by toil.

Returns

giant string containing the main def for the toil script.

write_main_header()

write_main_jobwrappers()

Writes out 'jobs' as wrapped toil objects in preparation for calling.

Returns

A string representing this.

write_main_jobwrappers_declaration(*declaration*)

write_main_destbucket()

Writes out a loop for exporting outputs to a cloud bucket.

Returns
A string representing this.

fetch_ignoredifs(*assignments*, *breaking_assignment*)

fetch_ignoredifs_chain(*assignments*, *breaking_assignment*)

write_main_jobwrappers_if(*if_statement*)

write_main_jobwrappers_scatter(*task*, *assignment*)

fetch_scatter_outputs(*task*)

fetch_scatter_inputs(*assigned*)

fetch_scatter_inputs_chain(*inputs*, *assigned*, *ignored_ifs*, *inputs_list*)

write_main_jobwrappers_call(*task*)

fetch_call_outputs(*task*)

write_functions()

Writes out a python function for each WDL "task" object.

Returns
a giant string containing the meat of the job defs.

write_scatterfunctions_within_if(*ifstatement*)

write_scatterfunction(*job*, *scatternname*)

Writes out a python function for each WDL "scatter" object.

write_scatterfunction_header(*scatternname*)

Returns

write_scatterfunction_outputreturn(*scatter_outputs*)

Returns

write_scatterfunction_lists(*scatter_outputs*)

Returns

write_scatterfunction_loop(*job*, *scatter_outputs*)

Returns

write_scatter_callwrapper(*job*, *previous_dependency*)

write_function(*job*)

Writes out a python function for each WDL "task" object.

Each python function is a unit of work written out as a string in preparation to being written out to a file. In WDL, each "job" is called a "task". Each WDL task is written out in multiple steps:

1: Header and inputs (e.g. 'def mapping(self, input1, input2)') 2: Log job name (e.g. 'job.fileStore.logToMaster('initialize_jobs')') 3: Create temp dir (e.g. 'tempDir = fileStore.getLocalTempDir()') 4: import filenames and use readGlobalFile() to get files from the

jobStore

5: Reformat commandline variables (like converting to ' '.join(files)). 6: Commandline call using subprocess.Popen(). 7: Write the section returning the outputs. Also logs stats.

Returns
a giant string containing the meat of the job defs for the toil script.

`write_function_header`(*job*)

Writes the header that starts each function, for example, this function can write and return:

'def write_function_header(self, job, job_declaration_array):'

Parameters

- **job** – A list such that: (job priority #, job ID #, Job Skeleton Name, Job Alias)
- **job_declaration_array** – A list of all inputs that job requires.

Returns
A string representing this.

`json_var`(*var*, *task=None*, *wf=None*)

Parameters

- **var** –
- **task** –
- **wf** –

Returns

`needs_file_import`(*var_type*)

Check if the given type contains a File type. A return value of True means that the value with this type has files to import.

Parameters
var_type (toil.wdl.wdl_types.WDLType) –

Return type
bool

`write_declaration_type`(*var_type*)

Return a string that preserves the construction of the given WDL type so it can be passed into the compiled script.

Parameters
var_type (toil.wdl.wdl_types.WDLType) –

`write_function_bashscriptline`(*job*)

Writes a function to create a bashscript for injection into the docker container.

Parameters

- **job_task_reference** – The job referenced in WDL's Task section.
- **job_alias** – The actual job name to be written.

Returns
A string writing all of this.

write_function_dockercall(*job*)

Writes a string containing the apiDockerCall() that will run the job.

Parameters

- **job_task_reference** – The name of the job calling docker.
- **docker_image** – The corresponding name of the docker image. e.g. "ubuntu:latest"

Returns
A string containing the apiDockerCall() that will run the job.

write_function_cmdline(*job*)

Write a series of commandline variables to be concatenated together eventually and either called with subprocess.Popen() or with apiDockerCall() if a docker image is called for.

Parameters
job – A list such that: (job priority #, job ID #, Job Skeleton Name, Job Alias)

Returns
A string representing this.

write_function_subprocesspopen()

Write a subprocess.Popen() call for this function and write it out as a string.

Parameters
job – A list such that: (job priority #, job ID #, Job Skeleton Name, Job Alias)

Returns
A string representing this.

write_function_outputreturn(*job*, *docker=False*)

Find the output values that this function needs and write them out as a string.

Parameters

- **job** – A list such that: (job priority #, job ID #, Job Skeleton Name, Job Alias)
- **job_task_reference** – The name of the job to look up values for.

Returns
A string representing this.

indent(*string2indent*)

Indent the input string by 4 spaces.

Parameters
string2indent (*str*) –

Return type
str

needsdocker(*job*)

Parameters
job –

Returns

write_python_file(*module_section*, *fn_section*, *main_section*, *output_file*)

Just takes three strings and writes them to output_file.

Parameters

- **module_section** – A string of 'import modules'.
- **fn_section** – A string of python 'def functions()'.
- **main_section** – A string declaring toil options and main's header.
- **job_section** – A string import files into toil and declaring jobs.
- **output_file** – The file to write the compiled toil script to.

toil.wdl.wdl_types

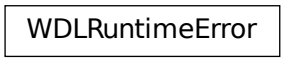
Module Contents

Classes

WDLType	Represents a primitive or compound WDL type:
WDLCompoundType	Represents a WDL compound type.
WDLStringType	Represents a WDL String primitive type.
WDLIntType	Represents a WDL Int primitive type.
WDLFloatType	Represents a WDL Float primitive type.
WDLBooleanType	Represents a WDL Boolean primitive type.
WDLFileType	Represents a WDL File primitive type.
WDLArrayType	Represents a WDL Array compound type.
WDLPairType	Represents a WDL Pair compound type.
WDLMapType	Represents a WDL Map compound type.
WDLFile	Represents a WDL File.
WDLPair	Represents a WDL Pair literal defined at

exception toil.wdl.wdl_types.**WDLRuntimeError**

Bases: RuntimeError



Unspecified run-time error.

class toil.wdl.wdl_types.**WDLType**(*optional=False*)

Represents a primitive or compound WDL type:

https://github.com/openwdl/wdl/blob/main/versions/development/SPEC.md#types

Parameters
optional (*bool*) –

abstract property name: str

Type name as string. Used in display messages / 'mappings.out' if dev mode is enabled.

Return type
str

property default_value: Optional[str]

Default value if optional.

Return type
Optional[str]

create(*value*, *output=False*)

Calls at runtime. Returns an instance of the current type. An error may be raised if the value is not in the correct format.

Parameters

- **value** (*Any*) – a Python object
- **output** (*bool*) –

Return type
Any

__eq__(*other*)

Return self==value.

Parameters
other (*Any*) –

Return type
bool

__str__()

Return str(self).

Return type
str

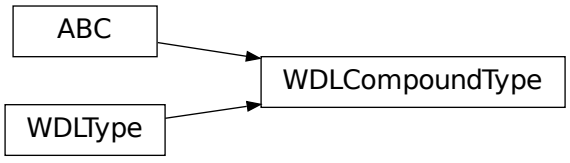
__repr__()

Return repr(self).

Return type
str

class toil.wdl.wdl_types.**WDLCompoundType**(*optional=False*)

Bases: *WDLType*, abc.ABC

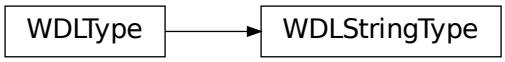


Represents a WDL compound type.

Parameters
optional (*bool*) –

class toil.wdl.wdl_types.**WDLStringType**(*optional=False*)

Bases: *WDLType*



Represents a WDL String primitive type.

Parameters
optional (*bool*) –

property name: str

Type name as string. Used in display messages / 'mappings.out' if dev mode is enabled.

Return type
str

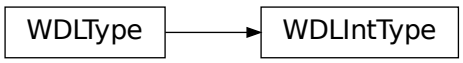
property default_value: str

Default value if optional.

Return type
str

class toil.wdl.wdl_types.**WDLIntType**(*optional=False*)

Bases: *WDLType*



Represents a WDL Int primitive type.

Parameters
optional (*bool*) –

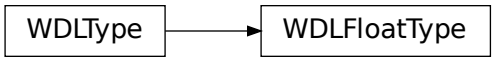
property name: str

Type name as string. Used in display messages / 'mappings.out' if dev mode is enabled.

Return type
str

class toil.wdl.wdl_types.**WDLFloatType**(*optional=False*)

Bases: *WDLType*



Represents a WDL Float primitive type.

Parameters
optional (*bool*) –

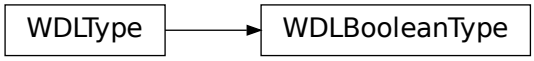
property name: str

Type name as string. Used in display messages / 'mappings.out' if dev mode is enabled.

Return type
str

class toil.wdl.wdl_types.**WDLBooleanType**(*optional=False*)

Bases: *WDLType*



Represents a WDL Boolean primitive type.

Parameters
optional (*bool*) –

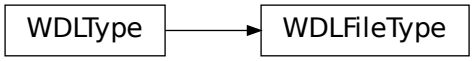
property name: str

Type name as string. Used in display messages / 'mappings.out' if dev mode is enabled.

Return type
str

class toil.wdl.wdl_types.**WDLFileType**(*optional=False*)

Bases: *WDLType*



Represents a WDL File primitive type.

Parameters
optional (*bool*) –

property name: str

Type name as string. Used in display messages / 'mappings.out' if dev mode is enabled.

Return type
str

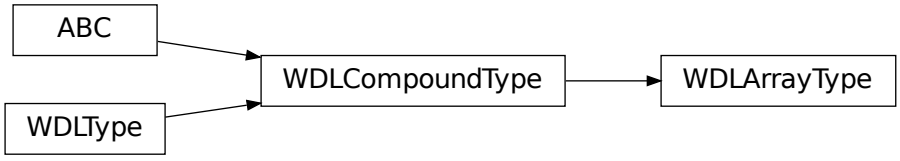
property default_value: str

Default value if optional.

Return type
str

class toil.wdl.wdl_types.**WDLArrayType**(*element*, *optional=False*)

Bases: *WDLCompoundType*



Represents a WDL Array compound type.

Parameters

- **element** (WDLType) –
- **optional** (*bool*) –

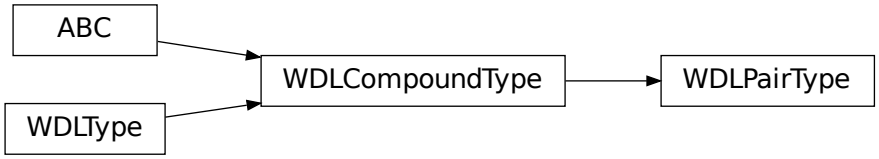
property name: str

Type name as string. Used in display messages / 'mappings.out' if dev mode is enabled.

Return type
str

class toil.wdl.wdl_types.**WDLPairType**(*left*, *right*, *optional=False*)

Bases: *WDLCompoundType*



Represents a WDL Pair compound type.

Parameters

- **left** (WDLType) –
- **right** (WDLType) –
- **optional** (*bool*) –

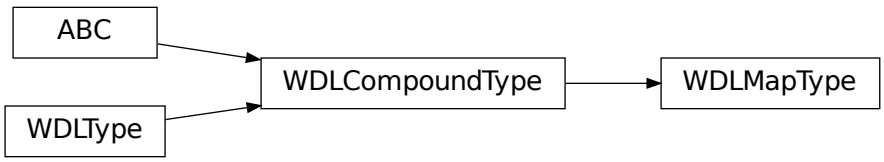
property name: str

Type name as string. Used in display messages / 'mappings.out' if dev mode is enabled.

Return type
str

class toil.wdl.wdl_types.**WDLMapType**(*key*, *value*, *optional=False*)

Bases: *WDLCompoundType*



Represents a WDL Map compound type.

Parameters

- **key** (WDLType) –
- **value** (WDLType) –
- **optional** (*bool*) –

property name: str

Type name as string. Used in display messages / 'mappings.out' if dev mode is enabled.

Return type
str

class toil.wdl.wdl_types.**WDLFile**(*file_path*, *file_name=None*, *imported=False*)

Represents a WDL File.

Parameters

- **file_path** (*str*) –
- **file_name** (*Optional[str]*) –
- **imported** (*bool*) –

class toil.wdl.wdl_types.**WDLPair**(*left*, *right*)

Represents a WDL Pair literal defined at https://github.com/openwdl/wdl/blob/main/versions/development/SPEC.md#pair-literals

Parameters

- **left** (*Any*) –
- **right** (*Any*) –

to_dict()

Return type
Dict[str, Any]

__eq__(*other*)

Return self==value.

Parameters
other (*Any*) –

Return type
Any

__repr__()

Return repr(self).

Return type
str

toil.wdl.wdltoil

Module Contents

Classes

NonDownloadingSize	WDL size() implementation that avoids downloading files.
ToilWDLStdLibBase	Standard library implementation for WDL as run on Toil.
ToilWDLStdLibTaskOutputs	Standard library implementation for WDL as run on Toil, with additional
WDLBaseJob	Base job class for all WDL-related jobs.
WDLTaskJob	Job that runs a WDL task.
WDLWorkflowNodeJob	Job that evaluates a WDL workflow node.
WDLCombineBindingsJob	Job that collects the results from WDL workflow nodes and combines their
WDLNamespaceBindingsJob	Job that puts a set of bindings into a namespace.
WDLSectionJob	Job that can create more graph for a section of the wrokflow.
WDLScatterJob	Job that evaluates a scatter in a WDL workflow. Runs the body for each
WDLArrayBindingsJob	Job that takes all new bindings created in an array of input environments,
WDLConditionalJob	Job that evaluates a conditional in a WDL workflow.
WDLWorkflowJob	Job that evaluates an entire WDL workflow.
WDLOutputsJob	Job which evaluates an outputs section (such as for a workflow).
WDLRootJob	Job that evaluates an entire WDL workflow, and returns the workflow outputs

Functions

potential_absolute_uris(uri, path[, importer])	Get potential absolute URIs to check for an imported file.
toil_read_source(uri, path, importer)	Implementation of a MiniWDL read_source function that can use any
combine_bindings(all_bindings)	Combine variable bindings from multiple predecessor tasks into one set for
log_bindings(log_function, message, all_bindings)	Log bindings to the console, even if some are still promises.
get_supertype(types)	Get the supertype that can hold values of all the given types.
for_each_node(root)	Iterate over all WDL workflow nodes in the given node, including inputs,
recursive_dependencies(root)	Get the combined workflow_node_dependencies of root and everything under
pack_toil_uri(file_id, file_basename)	Encode a Toil file ID and its source path in a URI that starts with the scheme in TOIL_URI_SCHEME.
unpack_toil_uri(toil_uri)	Unpack a URI made by make_toil_uri to retrieve the FileID and the basename
evaluate_named_expression(context, name, ...)	Evaluate an expression when we know the name of it.
evaluate_decl(node, environment, stdlib)	Evaluate the expression of a declaration node, or raise an error.
evaluate_call_inputs(context, expressions, ...)	Evaluate a bunch of expressions with names, and make them into a fresh set of bindings.
evaluate_defaultable_decl(node, environment, stdlib)	If the name of the declaration is already defined in the environment, return its value. Otherwise, return the evaluated expression.
devirtualize_files(environment, stdlib)	Make sure all the File values embedded in the given bindings point to files
virtualize_files(environment, stdlib)	Make sure all the File values embedded in the given bindings point to files
import_files(environment, toil[, path])	Make sure all File values embedded in the given bindings are imported,
drop_missing_files(environment[, ...])	Make sure all the File values embedded in the given bindings point to files
get_file_paths_in_bindings(environment)	Get the paths of all files in the bindings. Doesn't guarantee that
map_over_typed_files_in_bindings(environment, transform)	Run all File values embedded in the given bindings through the given
map_over_files_in_bindings(bindings, transform)	Run all File values' types and values embedded in the given bindings
map_over_typed_files_in_binding(binding, transform)	Run all File values' types and values embedded in the given binding's value through the given
map_over_typed_files_in_value(value, transform)	Run all File values embedded in the given value through the given
main()	A Toil workflow to interpret WDL input files.

Attributes

logger
WDLBindings
TOIL_URI_SCHEME

toil.wdl.wdltoil.**logger**

toil.wdl.wdltoil.**potential_absolute_uris**(*uri*, *path*, *importer=None*)

Get potential absolute URIs to check for an imported file.

Given a URI or bare path, yield in turn all the URIs, with schemes, where we should actually try to find it, given that we want to search under/against the given paths or URIs, the current directory, and the given importing WDL document if any.

Parameters

- **uri** (*str*) –
- **path** (*List[str]*) –
- **importer** (*Optional[WDL.Tree.Document]*) –

Return type
Iterator[str]

async toil.wdl.wdltoil.**toil_read_source**(*uri*, *path*, *importer*)

Implementation of a MiniWDL read_source function that can use any filename or URL supported by Toil.

Needs to be async because MiniWDL will await its result.

Parameters

- **uri** (*str*) –
- **path** (*List[str]*) –
- **importer** (*Optional[WDL.Tree.Document]*) –

Return type
WDL.ReadSourceResult

toil.wdl.wdltoil.**WDLBindings**

toil.wdl.wdltoil.**combine_bindings**(*all_bindings*)

Combine variable bindings from multiple predecessor tasks into one set for the current task.

Parameters
all_bindings (*Sequence[WDLBindings]*) –

Return type
WDLBindings

toil.wdl.wdltoil.**log_bindings**(*log_function*, *message*, *all_bindings*)

Log bindings to the console, even if some are still promises.

Parameters

- **log_function** (*Callable[Ellipsis, None]*) – Function (like logger.info) to call to log data
- **message** (*str*) – Message to log before the bindings
- **all_bindings** (*Sequence[toil.job.Promised[WDLBindings]]*) – A list of bindings or promises for bindings, to log

Return type
None

toil.wdl.wdltoil.**get_supertype**(*types*)

Get the supertype that can hold values of all the given types.

Parameters
types (*Sequence[Optional[WDL.Type.Base]]*) –

Return type
WDL.Type.Base

toil.wdl.wdltoil.**for_each_node**(*root*)

Iterate over all WDL workflow nodes in the given node, including inputs, internal nodes of conditionals and scatters, and gather nodes.

Parameters
root (*WDL.Tree.WorkflowNode*) –

Return type
Iterator[WDL.Tree.WorkflowNode]

toil.wdl.wdltoil.**recursive_dependencies**(*root*)

Get the combined workflow_node_dependencies of root and everything under it, which are not on anything in that subtree.

Useful because section nodes can have internal nodes with dependencies not reflected in those of the section node itself.

Parameters
root (*WDL.Tree.WorkflowNode*) –

Return type
Set[str]

toil.wdl.wdltoil.**TOIL_URI_SCHEME = 'toilfile:'**

toil.wdl.wdltoil.**pack_toil_uri**(*file_id*, *file_basename*)

Encode a Toil file ID and its source path in a URI that starts with the scheme in TOIL_URI_SCHEME.

Parameters
- **file_id** (toil.fileStores.FileID) –
- **file_basename** (*str*) –

Return type
str

toil.wdl.wdltoil.**unpack_toil_uri**(*toil_uri*)

Unpack a URI made by make_toil_uri to retrieve the FileID and the basename (no path prefix) that the file is supposed to have.

Parameters
toil_uri (*str*) –

Return type
Tuple[*toil.fileStores.FileID*, str]

class toil.wdl.wdltoil.**NonDownloadingSize**

Bases: WDL.StdLib._Size

WDL size() implementation that avoids downloading files.

MiniWDL's default size() implementation downloads the whole file to get its size. We want to be able to get file sizes from code running on the leader, where there may not be space to download the whole file. So we override the fancy class that implements it so that we can handle sizes for FileIDs using the FileID's stored size info.

class toil.wdl.wdltoil.**ToilWDLStdLibBase**(*file_store*)

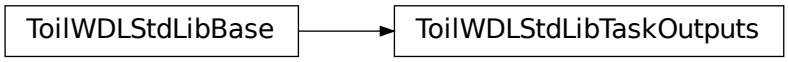
Bases: WDL.StdLib.Base

Standard library implementation for WDL as run on Toil.

Parameters
file_store (toil.fileStores.abstractFileStore.AbstractFileStore) –

class toil.wdl.wdltoil.**ToilWDLStdLibTaskOutputs**(*file_store*, *stdout_path*, *stderr_path*, *current_directory_override=None*)

Bases: *ToilWDLStdLibBase*, WDL.StdLib.TaskOutputs



Standard library implementation for WDL as run on Toil, with additional functions only allowed in task output sections.

Parameters

- **file_store** (toil.fileStores.abstractFileStore.AbstractFileStore) –
- **stdout_path** (*str*) –

- **stderr_path** (*str*) –
- **current_directory_override** (*Optional[str]*) –

toil.wdl.wdltoil.**evaluate_named_expression**(*context*, *name*, *expected_type*, *expression*, *environment*, *stdlib*)

Evaluate an expression when we know the name of it.

Parameters

- **context** (*Union[WDL.Error.SourceNode, WDL.Error.SourcePosition]*) –
- **name** (*str*) –
- **expected_type** (*Optional[WDL.Type.Base]*) –
- **expression** (*Optional[WDL.Expr.Base]*) –
- **environment** (*WDLBindings*) –
- **stdlib** (*WDL.StdLib.Base*) –

Return type
WDL.Value.Base

toil.wdl.wdltoil.**evaluate_decl**(*node*, *environment*, *stdlib*)

Evaluate the expression of a declaration node, or raise an error.

Parameters

- **node** (*WDL.Tree.Decl*) –
- **environment** (*WDLBindings*) –
- **stdlib** (*WDL.StdLib.Base*) –

Return type
WDL.Value.Base

toil.wdl.wdltoil.**evaluate_call_inputs**(*context*, *expressions*, *environment*, *stdlib*)

Evaluate a bunch of expressions with names, and make them into a fresh set of bindings.

Parameters

- **context** (*Union[WDL.Error.SourceNode, WDL.Error.SourcePosition]*) –
- **expressions** (*Dict[str, WDL.Expr.Base]*) –
- **environment** (*WDLBindings*) –
- **stdlib** (*WDL.StdLib.Base*) –

Return type
WDLBindings

toil.wdl.wdltoil.**evaluate_defaultable_decl**(*node*, *environment*, *stdlib*)

If the name of the declaration is already defined in the environment, return its value. Otherwise, return the evaluated expression.

Parameters

- **node** (*WDL.Tree.Decl*) –
- **environment** (*WDLBindings*) –
- **stdlib** (*WDL.StdLib.Base*) –

Return type
WDL.Value.Base

toil.wdl.wdltoil.**devirtualize_files**(*environment*, *stdlib*)

Make sure all the File values embedded in the given bindings point to files that are actually available to command line commands.

Parameters

- **environment** (*WDLBindings*) –
- **stdlib** (*WDL.StdLib.Base*) –

Return type
WDLBindings

toil.wdl.wdltoil.**virtualize_files**(*environment*, *stdlib*)

Make sure all the File values embedded in the given bindings point to files that are usable from other machines.

Parameters

- **environment** (*WDLBindings*) –
- **stdlib** (*WDL.StdLib.Base*) –

Return type
WDLBindings

toil.wdl.wdltoil.**import_files**(*environment*, *toil*, *path=None*)

Make sure all File values embedded in the given bindings are imported, using the given Toil object.

Parameters

- **path** (*Optional[List[str]]*) – If set, try resolving input location relative to the URLs or directories in this list.
- **environment** (*WDLBindings*) –
- **toil** (toil.common.Toil) –

Return type
WDLBindings

toil.wdl.wdltoil.**drop_missing_files**(*environment*, *current_directory_override=None*)

Make sure all the File values embedded in the given bindings point to files that exist, or are null.

Files must not be virtualized.

Parameters

- **environment** (*WDLBindings*) –
- **current_directory_override** (*Optional[str]*) –

Return type
WDLBindings

toil.wdl.wdltoil.**get_file_paths_in_bindings**(*environment*)

Get the paths of all files in the bindings. Doesn't guarantee that duplicates are removed.

TODO: Duplicative with WDL.runtime.task._fspaths, except that is internal and supports Direcotry objects.

Parameters
environment (*WDLBindings*) –

Return type
List[str]

toil.wdl.wdltoil.**map_over_typed_files_in_bindings**(*environment*, *transform*)

Run all File values embedded in the given bindings through the given transformation function.

TODO: Replace with WDL.Value.rewrite_env_paths or WDL.Value.rewrite_files

Parameters

- **environment** (*WDLBindings*) –
- **transform** (*Callable[[WDL.Type.Base, str], Optional[str]]*) –

Return type
WDLBindings

toil.wdl.wdltoil.**map_over_files_in_bindings**(*bindings*, *transform*)

Run all File values' types and values embedded in the given bindings through the given transformation function.

TODO: Replace with WDL.Value.rewrite_env_paths or WDL.Value.rewrite_files

Parameters

- **bindings** (*WDLBindings*) –
- **transform** (*Callable[[str], Optional[str]]*) –

Return type
WDLBindings

toil.wdl.wdltoil.**map_over_typed_files_in_binding**(*binding*, *transform*)

Run all File values' types and values embedded in the given binding's value through the given transformation function.

Parameters

- **binding** (*WDL.Env.Binding[WDL.Value.Base]*) –
- **transform** (*Callable[[WDL.Type.Base, str], Optional[str]]*) –

Return type
WDL.Env.Binding[WDL.Value.Base]

toil.wdl.wdltoil.**map_over_typed_files_in_value**(*value*, *transform*)

Run all File values embedded in the given value through the given transformation function.

If the transform returns None, the file value is changed to Null.

The transform has access to the type information for the value, so it knows if it may return None, depending on if the value is optional or not.

The transform is *allowed* to return None only if the mapping result won't actually be used, to allow for scans. So error checking needs to be part of the transform itself.

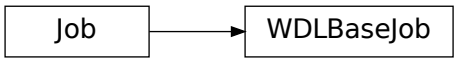
Parameters

- **value** (*WDL.Value.Base*) –
- **transform** (*Callable[[WDL.Type.Base, str], Optional[str]]*) –

Return type
WDL.Value.Base

class toil.wdl.wdltoil.**WDLBaseJob**(***kwargs*)

Bases: *toil.job.Job*



Base job class for all WDL-related jobs.

Parameters
kwargs (*Any*) –

run(*file_store*)

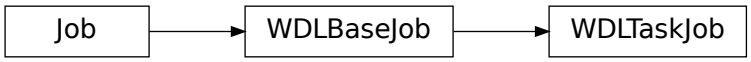
Run a WDL-related job.

Parameters
file_store (toil.fileStores.abstractFileStore.AbstractFileStore) –

Return type
Any

class toil.wdl.wdltoil.**WDLTaskJob**(*task*, *prev_node_results*, *task_id*, *namespace*, ***kwargs*)

Bases: *WDLBaseJob*



Job that runs a WDL task.

Responsible for evaluating the input declarations for unspecified inputs, evaluating the runtime section, rescheduling if resources are not available, running any command, and evaluating the outputs.

All bindings are in terms of task-internal names.

Parameters

- **task** (*WDL.Tree.Task*) –
- **prev_node_results** (*Sequence[toil.job.Promised[WDLBindings]]*) –
- **task_id** (*List[str]*) –
- **namespace** (*str*) –
- **kwargs** (*Any*) –

can_fake_root()

Determie if –fakeroot is likely to work for Singularity.

Return type
bool

run(*file_store*)

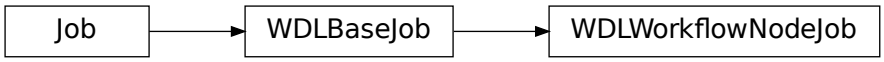
Actually run the task.

Parameters
file_store (toil.fileStores.abstractFileStore.AbstractFileStore) –

Return type
toil.job.Promised[WDLBindings]

class toil.wdl.wdltoil.**WDLWorkflowNodeJob**(*node*, *prev_node_results*, *namespace*, ***kwargs*)

Bases: *WDLBaseJob*



Job that evaluates a WDL workflow node.

Parameters

- **node** (*WDL.Tree.WorkflowNode*) –
- **prev_node_results** (*Sequence[toil.job.Promised[WDLBindings]]*) –
- **namespace** (*str*) –
- **kwargs** (*Any*) –

run(*file_store*)

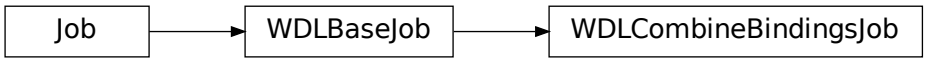
Actually execute the workflow node.

Parameters
file_store (toil.fileStores.abstractFileStore.AbstractFileStore) –

Return type
toil.job.Promised[WDLBindings]

class toil.wdl.wdltoil.**WDLCombineBindingsJob**(*prev_node_results*, *underlay=None*, *remove=None*, ***kwargs*)

Bases: *WDLBaseJob*



Job that collects the results from WDL workflow nodes and combines their environment changes.

Parameters

- **prev_node_results** (*Sequence[toil.job.Promised[WDLBindings]]*) –
- **underlay** (*Optional[toil.job.Promised[WDLBindings]]*) –

- **remove** (*Optional[toil.job.Promised[WDLBindings]]*) –
- **kwargs** (*Any*) –

run(*file_store*)

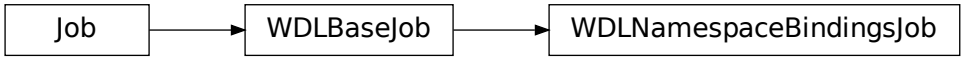
Aggregate incoming results.

Parameters
file_store (toil.fileStores.abstractFileStore.AbstractFileStore) –

Return type
WDLBindings

class toil.wdl.wdltoil.**WDLNamespaceBindingsJob**(*namespace*, *prev_node_results*, ***kwargs*)

Bases: *WDLBaseJob*



Job that puts a set of bindings into a namespace.

Parameters

- **namespace** (*str*) –
- **prev_node_results** (*Sequence[toil.job.Promised[WDLBindings]]*) –
- **kwargs** (*Any*) –

run(*file_store*)

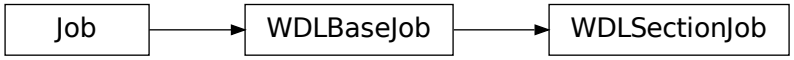
Apply the namespace

Parameters
file_store (toil.fileStores.abstractFileStore.AbstractFileStore) –

Return type
WDLBindings

class toil.wdl.wdltoil.**WDLSectionJob**(*namespace*, ***kwargs*)

Bases: *WDLBaseJob*



Job that can create more graph for a section of the wrokflow.

Parameters

- **namespace** (*str*) –

- **kwargs** (*Any*) –

create_subgraph(*nodes*, *gather_nodes*, *environment*, *local_environment=None*)

Make a Toil job to evaluate a subgraph inside a workflow or workflow section.

Returns

a child Job that will return the aggregated environment after running all the things in the section.

Parameters

- **gather_nodes** (*Sequence[WDL.Tree.Gather]*) – Names exposed by these will always be defined with something, even if the code that defines them does not actually run.
- **environment** (*WDLBindings*) – Bindings in this environment will be used to evaluate the subgraph and will be passed through.
- **local_environment** (*Optional[WDLBindings]*) – Bindings in this environment will be used to evaluate the subgraph but will go out of scope at the end of the section.
- **nodes** (*Sequence[WDL.Tree.WorkflowNode]*) –

Return type

toil.job.Job

make_gather_bindings(*gathers*, *undefined*)

Given a collection of Gathers, create bindings from every identifier gathered, to the given "undefined" placeholder (which would be Null for a single execution of the body, or an empty array for a completely unexecuted scatter).

These bindings can be overlaid with bindings from the actual execution, so that references to names defined in unexecuted code get a proper default undefined value, and not a KeyError at runtime.

The information to do this comes from MiniWDL's "gathers" system: <https://miniwdl.readthedocs.io/en/latest/WDL.html#WDL.Tree.WorkflowSection.gathers>

TODO: This approach will scale O(n^2) when run on n nested conditionals, because generating these bindings for the outer conditional will visit all the bindings from the inner ones.

Parameters

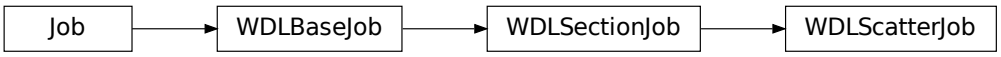
- **gathers** (*Sequence[WDL.Tree.Gather]*) –
- **undefined** (*WDL.Value.Base*) –

Return type

WDLBindings

class toil.wdl.wdltoil.**WDLScatterJob**(*scatter*, *prev_node_results*, *namespace*, ***kwargs*)

Bases: *WDLSectionJob*



Job that evaluates a scatter in a WDL workflow. Runs the body for each value in an array, and makes arrays of the new bindings created in each instance of the body. If an instance of the body doesn't create a binding, it gets a null value in the corresponding array.

Parameters

- **scatter** (*WDL.Tree.Scatter*) –
- **prev_node_results** (*Sequence[toil.job.Promised[WDLBindings]]*) –
- **namespace** (*str*) –
- **kwargs** (*Any*) –

run(*file_store*)

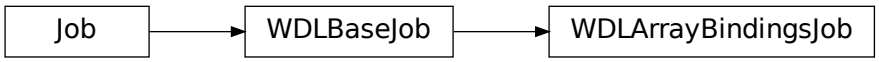
Run the scatter.

Parameters
file_store (toil.fileStores.abstractFileStore.AbstractFileStore) –

Return type
toil.job.Promised[WDLBindings]

class toil.wdl.wdltoil.**WDLArrayBindingsJob**(*input_bindings*, *base_bindings*, ***kwargs*)

Bases: *WDLBaseJob*



Job that takes all new bindings created in an array of input environments, relative to a base environment, and produces bindings where each new binding name is bound to an array of the values in all the input environments.

Useful for producing the results of a scatter.

Parameters

- **input_bindings** (*Sequence[toil.job.Promised[WDLBindings]]*) –
- **base_bindings** (*WDLBindings*) –
- **kwargs** (*Any*) –

run(*file_store*)

Actually produce the array-ified bindings now that promised values are available.

Parameters
file_store (toil.fileStores.abstractFileStore.AbstractFileStore) –

Return type
WDLBindings

class toil.wdl.wdltoil.**WDLConditionalJob**(*conditional*, *prev_node_results*, *namespace*, ***kwargs*)

Bases: *WDLSectionJob*

Job that evaluates a conditional in a WDL workflow.

Parameters

- **conditional** (*WDL.Tree.Conditional*) –
- **prev_node_results** (*Sequence[toil.job.Promised[WDLBindings]]*) –
- **namespace** (*str*) –
- **kwargs** (*Any*) –

run(*file_store*)

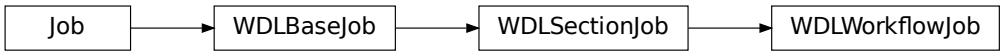
Run the conditional.

Parameters
file_store (toil.fileStores.abstractFileStore.AbstractFileStore) –

Return type
toil.job.Promised[WDLBindings]

class toil.wdl.wdltoil.**WDLWorkflowJob**(*workflow*, *prev_node_results*, *workflow_id*, *namespace*, ***kwargs*)

Bases: *WDLSectionJob*



Job that evaluates an entire WDL workflow.

Parameters

- **workflow** (*WDL.Tree.Workflow*) –
- **prev_node_results** (*Sequence[toil.job.Promised[WDLBindings]]*) –
- **workflow_id** (*List[str]*) –
- **namespace** (*str*) –
- **kwargs** (*Any*) –

run(*file_store*)

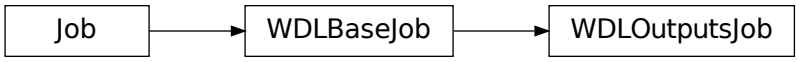
Run the workflow. Return the result of the workflow.

Parameters
file_store (toil.fileStores.abstractFileStore.AbstractFileStore) –

Return type
toil.job.Promised[WDLBindings]

class toil.wdl.wdltoil.**WDLOutputsJob**(*outputs*, *bindings*, ***kwargs*)

Bases: *WDLBaseJob*



Job which evaluates an outputs section (such as for a workflow).

Returns an environment with just the outputs bound, in no namespace.

Parameters

- **outputs** (*List[WDL.Tree.Decl]*) –
- **bindings** (*toil.job.Promised[WDLBindings]*) –
- **kwargs** (*Any*) –

run(*file_store*)

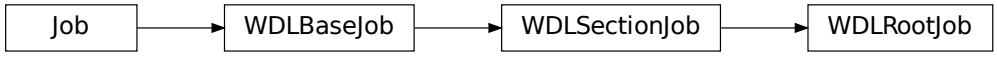
Make bindings for the outputs.

Parameters
file_store (toil.fileStores.abstractFileStore.AbstractFileStore) –

Return type
WDLBindings

class toil.wdl.wdltoil.**WDLRootJob**(*workflow*, *inputs*, ***kwargs*)

Bases: *WDLSectionJob*



Job that evaluates an entire WDL workflow, and returns the workflow outputs namespaced with the workflow name. Inputs may or may not be namespaced with the workflow name; both forms are accepted.

Parameters

- **workflow** (*WDL.Tree.Workflow*) –
- **inputs** (*WDLBindings*) –
- **kwargs** (*Any*) –

run(*file_store*)

Actually build the subgraph.

Parameters
file_store (toil.fileStores.abstractFileStore.AbstractFileStore) –

Return type
toil.job.Promised[WDLBindings]

toil.wdl.wdltoil.**main**()

A Toil workflow to interpret WDL input files.

Return type
None

30.1.2 Submodules

toil.bus

Message types and message bus for leader component coordination.

Historically, the Toil Leader has been organized around functions calling other functions to "handle" different things happening. Over time, it has become very brittle: exactly the right handling functions need to be called in exactly the right order, or it gets confused and does the wrong thing.

The MessageBus is meant to let the leader avoid this by more losely coupling its components together, by having them communicate by sending messages instead of by calling functions.

When events occur (like a job coming back from the batch system with a failed exit status), this will be translated into a message that will be sent to the bus. Then, all the leader components that need to react to this message in some way (by, say, decrementing the retry count) would listen for the relevant messages on the bus and react to them. If a new component needs to be added, it can be plugged into the message bus and receive and react to messages without interfering with existing components' ability to react to the same messages.

Eventually, the different aspects of the Leader could become separate objects.

By default, messages stay entirely within the Toil leader process, and are not persisted anywhere, not even in the JobStore.

The Message Bus also provides an extension point: its messages can be serialized to a file by the leader (see the –writeMessages option), and they can then be decoded using MessageBus.scan_bus_messages() (as is done in the Toil WES server backend). By replaying the messages and tracking their effects on job state, you can get an up-to-date view of the state of the jobs in a workflow. This includes information, such as whether jobs are issued or running, or what jobs have completely finished, which is not persisted in the JobStore.

The MessageBus instance for the leader process is owned by the Toil leader, but the BatchSystem has an opportunity to connect to it, and can send (or listen for) messages. Right now the BatchSystem deos not *have* to send or receive any messages; the Leader is responsible for polling it via the BatchSystem API and generating the events. But a BatchSystem implementation *may* send additional events (like JobAnnotationMessage).

Currently, the MessageBus is implemented using pypubsub, and so messages are always handled in a single Thread, the Toil leader's main loop thread. If other components send events, they will be shipped over to that thread inside the MessageBus. Communication between processes is allowed using MessageBus.connect_output_file() and MessageBus.scan_bus_messages().

Module Contents

Classes

JobIssuedMessage	Produced when a job is issued to run on the batch system.
JobUpdatedMessage	Produced when a job is "updated" and ready to have something happen to it.
JobCompletedMessage	Produced when a job is completed, whether successful or not.
JobFailedMessage	Produced when a job is completely failed, and will not be retried again.
JobMissingMessage	Produced when a job goes missing and should be in the batch system but isn't.
JobAnnotationMessage	Produced when extra information (such as an AWS Batch job ID from the
ExternalBatchIdMessage	Produced when using a batch system, links toil assigned batch ID to
QueueSizeMessage	Produced to describe the size of the queue of jobs issued but not yet
ClusterSizeMessage	Produced by the Toil-integrated autoscaler describe the number of
ClusterDesiredSizeMessage	Produced by the Toil-integrated autoscaler to describe the number of
MessageBus	Holds messages that should cause jobs to change their scheduling states.
MessageBusClient	Base class for clients (inboxes and outboxes) of a message bus. Handles
MessageInbox	A buffered connection to a message bus that lets us receive messages.
MessageOutbox	A connection to a message bus that lets us publish messages.
MessageBusConnection	A two-way connection to a message bus. Buffers incoming messages until you
JobStatus	Records the status of a job.

Functions

message_to_bytes(message)	Convert a plain-old-data named tuple into a byte string.
bytes_to_message(message_type, data)	Convert bytes from message_to_bytes back to a message of the given type.
replay_message_bus(path)	Replay all the messages and work out what they mean for jobs.
gen_message_bus_path()	Return a file path in tmp to store the message bus at.

Attributes

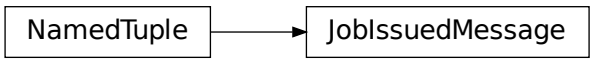
logger

MessageType

toil.bus.**logger**

class toil.bus.**JobIssuedMessage**

Bases: NamedTuple



Produced when a job is issued to run on the batch system.

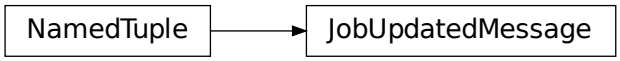
job_type: *str*

job_id: *str*

toil_batch_id: *int*

class toil.bus.**JobUpdatedMessage**

Bases: NamedTuple



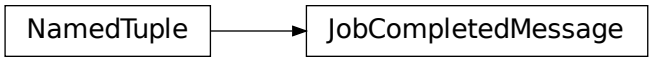
Produced when a job is "updated" and ready to have something happen to it.

job_id: *str*

result_status: *int*

class toil.bus.**JobCompletedMessage**

Bases: NamedTuple



Produced when a job is completed, whether successful or not.

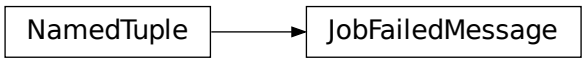
job_type: *str*

job_id: *str*

exit_code: *int*

class toil.bus.**JobFailedMessage**

Bases: NamedTuple



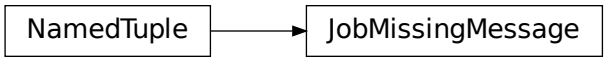
Produced when a job is completely failed, and will not be retried again.

job_type: *str*

job_id: *str*

class toil.bus.**JobMissingMessage**

Bases: NamedTuple

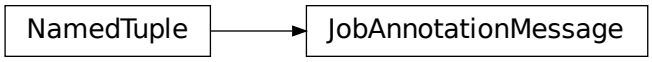


Produced when a job goes missing and should be in the batch system but isn't.

job_id: *str*

class toil.bus.**JobAnnotationMessage**

Bases: NamedTuple



Produced when extra information (such as an AWS Batch job ID from the AWSBatchBatchSystem) is available that goes with a job.

job_id: *str*

annotation_name: *str*

annotation_value: *str*

class toil.bus.**ExternalBatchIdMessage**

Bases: NamedTuple

Produced when using a batch system, links toil assigned batch ID to Batch system ID (Whatever's returned by local implementation, PID, batch ID, etc)

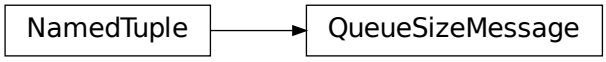
toil_batch_id: *int*

external_batch_id: *str*

batch_system: *str*

class toil.bus.**QueueSizeMessage**

Bases: NamedTuple

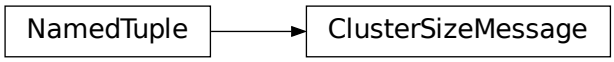


Produced to describe the size of the queue of jobs issued but not yet completed. Theoretically recoverable from other messages.

queue_size: *int*

class toil.bus.**ClusterSizeMessage**

Bases: NamedTuple



Produced by the Toil-integrated autoscaler describe the number of instances of a certain type in a cluster.

instance_type: *str*

current_size: *int*

class toil.bus.**ClusterDesiredSizeMessage**

Bases: NamedTuple

Produced by the Toil-integrated autoscaler to describe the number of instances of a certain type that it thinks will be needed.

instance_type: *str*

desired_size: *int*

toil.bus.**message_to_bytes**(*message*)

Convert a plain-old-data named tuple into a byte string.

Parameters

message (*NamedTuple*) –

Return type

bytes

toil.bus.**MessageType**

toil.bus.**bytes_to_message**(*message_type*, *data*)

Convert bytes from message_to_bytes back to a message of the given type.

Parameters

- **message_type** (*Type[MessageType]*) –
- **data** (*bytes*) –

Return type

MessageType

class toil.bus.**MessageBus**

Holds messages that should cause jobs to change their scheduling states. Messages are put in and buffered, and can be taken out and handled as batches when convenient.

All messages are NamedTuple objects of various subtypes.

Message order is guaranteed to be preserved within a type.

MessageType

publish(*message*)

Put a message onto the bus. Can be called from any thread.

Parameters
message (*Any*) –

Return type
None

check()

If we are in the owning thread, deliver any messages that are in the queue for us. Must be called every once in a while in the main thread, possibly through inbox objects.

Return type
None

subscribe(*message_type*, *handler*)

Register the given callable to be called when messages of the given type are sent. It will be called with messages sent after the subscription is created. Returns a subscription object; when the subscription object is GC'd the subscription will end.

Parameters

- **message_type** (*Type[MessageType]*) –
- **handler** (*Callable[[MessageType], Any]*) –

Return type
pubsub.core.listener.Listener

connect(*wanted_types*)

Get a connection object that serves as an inbox for messages of the given types. Messages of those types will accumulate in the inbox until it is destroyed. You can check for them at any time.

Parameters
wanted_types (*List[type]*) –

Return type
MessageBusConnection

outbox()

Get a connection object that only allows sending messages.

Return type
MessageOutbox

connect_output_file(*file_path*)

Send copies of all messages to the given output file.

Returns connection data which must be kept alive for the connection to persist. That data is opaque: the user is not supposed to look at it or touch it or do anything with it other than store it somewhere or delete it.

Parameters
file_path (*str*) –

Return type
Any

classmethod scan_bus_messages(*stream*, *message_types*)

Get an iterator over all messages in the given log stream of the given types, in order. Discard any trailing partial messages.

Parameters

- **stream** (*IO[bytes]*) –
- **message_types** (*List[Type[NamedTuple]]*) –

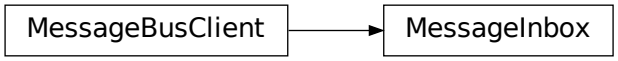
Return type
Iterator[Any]

class toil.bus.**MessageBusClient**

Base class for clients (inboxes and outboxes) of a message bus. Handles keeping a reference to the message bus.

class toil.bus.**MessageInbox**

Bases: *MessageBusClient*



A buffered connection to a message bus that lets us receive messages. Buffers incoming messages until you are ready for them. Does not preserve ordering between messages of different types.

MessageType

count(*message_type*)

Get the number of pending messages of the given type.

Parameters
message_type (*type*) –

Return type
int

empty()

Return True if no messages are pending, and false otherwise.

Return type
bool

for_each(*message_type*)

Loop over all messages currently pending of the given type. Each that is handled without raising an exception will be removed.

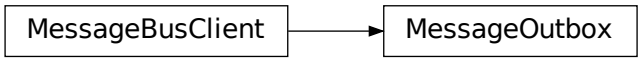
Messages sent while this function is running will not be yielded by the current call.

Parameters
message_type (*Type[MessageType]*) –

Return type
Iterator[MessageType]

class toil.bus.**MessageOutbox**

Bases: *MessageBusClient*



A connection to a message bus that lets us publish messages.

publish(*message*)

Publish the given message to the connected message bus.

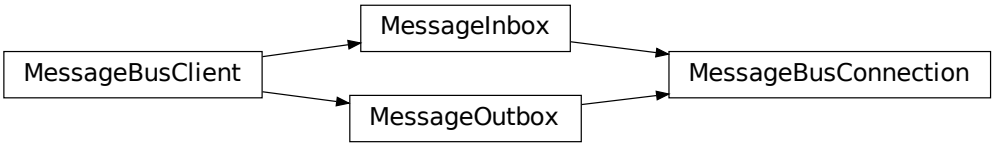
We have this so you don't need to store both the bus and your connection.

Parameters
message (*Any*) –

Return type
None

class toil.bus.**MessageBusConnection**

Bases: *MessageInbox*, *MessageOutbox*



A two-way connection to a message bus. Buffers incoming messages until you are ready for them, and lets you send messages.

class toil.bus.**JobStatus**

Records the status of a job.

job_store_id: str

name: str

exit_code: int

annotations: Dict[str, str]

toil_batch_id: int

external_batch_id: str

batch_system: str

__repr__()

Return repr(self).

Return type
str

toil.bus.**replay_message_bus**(*path*)

Replay all the messages and work out what they mean for jobs.

We track the state and name of jobs here, by ID. We would use a list of two items but MyPy can't understand a list of items of multiple types, so we need to define a new class.

Returns a dictionary from the job_id to a dataclass, JobStatus. A JobStatus contains information about a job which we have gathered from the message bus, including the job store id, name of the job the exit code, any associated annotations, the toil batch id the external batch id, and the batch system on which the job is running.

Parameters
path (*str*) –

Return type
Dict[str, *JobStatus*]

toil.bus.**gen_message_bus_path**()

Return a file path in tmp to store the message bus at. Calling function is responsible for cleaning the generated file.

Return type
str

toil.common

Module Contents

Classes

Config	Class to represent configuration operations for a toil workflow run.
Toil	A context manager that represents a Toil workflow.
ToilMetrics	

Functions

parser_with_common_options([provisioner_options, ...])	
addOptions(parser[, config, jobstore_as_flag])	Add Toil command line options to a parser.
parseBool(val)	
getNodeID()	Return unique ID of the current node (host). The resulting string will be convertable to a uuid.UUID.
parseSetEnv(l)	Parse a list of strings of the form "NAME=VALUE" or just "NAME" into a dictionary.
iC(minValue[, maxValue])	Returns a function that checks if a given int is in the given half-open interval.
fC(minValue[, maxValue])	Returns a function that checks if a given float is in the given half-open interval.
parse_accelerator_list(specs)	Parse a string description of one or more accelerator requirements.
cacheDirName(workflowID)	**return** Name of the cache directory.
getDirSizeRecursively(dirPath)	This method will return the cumulative number of bytes occupied by the files
getFileSystemSize(dirPath)	Return the free space, and total size of the file system hosting *dirPath*.
safeUnpickleFromStream(stream)	

Attributes

defaultTargetTime
SYS_MAX_SIZE
UUID_LENGTH
logger
JOBSTORE_HELP

toil.common.**defaultTargetTime = 1800**

toil.common.**SYS_MAX_SIZE = 9223372036854775807**

toil.common.**UUID_LENGTH = 32**

toil.common.**logger**

class toil.common.**Config**

Class to represent configuration operations for a toil workflow run.

logFile: Optional[str]

logRotating: bool

cleanWorkDir: str

max_jobs: int

max_local_jobs: int

run_local_jobs_on_workers: bool

tes_endpoint: str

tes_user: str

tes_password: str

tes_bearer_token: str

jobStore: str

batchSystem: str

batch_logs_dir: Optional[str]

The backing scheduler will be instructed, if possible, to save logs to this directory, where the leader can read them.

workflowAttemptNumber: int

disableAutoDeployment: bool

workflowID: Optional[str]

This attribute uniquely identifies the job store and therefore the workflow. It is necessary in order to distinguish between two consecutive workflows for which self.jobStore is the same, e.g. when a job store name is reused after a previous run has finished successfully and its job store has been clean up.

prepare_start()

After options are set, prepare for initial start of workflow.

Return type
None

prepare_restart()

Before restart options are set, prepare for a restart of a workflow. Set up any execution-specific parameters and clear out any stale ones.

Return type
None

setOptions(*options*)

Creates a config object from the options object.

Parameters
options (*argparse.Namespace*) –

Return type
None

__eq__(*other*)

Return self==value.

Parameters
other (*object*) –

Return type
bool

__hash__()

Return hash(self).

Return type
int

toil.common.**JOBSTORE_HELP = Multiline-String**

```
"""The location of the job store for the workflow.  A job store holds␣
↪persistent information about the jobs, stats, and files in a workflow. If␣
↪the workflow is run with a distributed batch system, the job store must␣
↪be accessible by all worker nodes. Depending on the desired job store␣
↪implementation, the location should be formatted according to one of the␣
↪following schemes:

file:<path> where <path> points to a directory on the file systen

aws:<region>:<prefix> where <region> is the name of an AWS region like us-
↪west-2 and <prefix> will be prepended to the names of any top-level AWS␣
↪resources in use by job store, e.g. S3 buckets.

 google:<project_id>:<prefix> TODO: explain

For backwards compatibility, you may also specify ./foo (equivalent to␣
↪file:./foo or just file:foo) or /bar (equivalent to file:/bar)."""
```

toil.common.**parser_with_common_options**(*provisioner_options=False*, *jobstore_option=True*)

Parameters

- **provisioner_options** (*bool*) –
- **jobstore_option** (*bool*) –

Return type
argparse.ArgumentParser

toil.common.**addOptions**(*parser*, *config=None*, *jobstore_as_flag=False*)

Add Toil command line options to a parser.

Parameters

- **config** (*Optional[Config]*) – If specified, take defaults from the given Config.
- **jobstore_as_flag** (*bool*) – make the job store option a –jobStore flag instead of a required jobStore positional argument.
- **parser** (*argparse.ArgumentParser*) –

Return type
None

toil.common.**parseBool**(*val*)

Parameters
val (*str*) –

Return type
bool

toil.common.**getNodeID**()

Return unique ID of the current node (host). The resulting string will be convertable to a uuid.UUID.

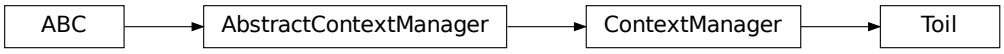
Tries several methods until success. The returned ID should be identical across calls from different processes on the same node at least until the next OS reboot.

The last resort method is uuid.getnode() that in some rare OS configurations may return a random ID each time it is called. However, this method should never be reached on a Linux system, because reading from /proc/sys/kernel/random/boot_id will be tried prior to that. If uuid.getnode() is reached, it will be called twice, and exception raised if the values are not identical.

Return type
str

class toil.common.**Toil**(*options*)

Bases: ContextManager[*Toil*]



A context manager that represents a Toil workflow.

Specifically the batch system, job store, and its configuration.

Parameters
options (*argparse.Namespace*) –

config: *Config*

__enter__()

Derive configuration from the command line options.

Then load the job store and, on restart, consolidate the derived configuration with the one from the previous invocation of the workflow.

Return type
Toil

__exit__(*exc_type*, *exc_val*, *exc_tb*)

Clean up after a workflow invocation.

Depending on the configuration, delete the job store.

Parameters

- **exc_type** (*Optional[Type[BaseException]]*) –
- **exc_val** (*Optional[BaseException]*) –
- **exc_tb** (*Optional[types.TracebackType]*) –

Return type
Literal[False]

`start`(*rootJob*)

Invoke a Toil workflow with the given job as the root for an initial run.

This method must be called in the body of a `with Toil(...) as toil:` statement. This method should not be called more than once for a workflow that has not finished.

Parameters
rootJob (toil.job.Job) – The root job of the workflow

Returns
The root job's return value

Return type
Any

`restart`()

Restarts a workflow that has been interrupted.

Returns
The root job's return value

Return type
Any

`classmethod getJobStore`(*locator*)

Create an instance of the concrete job store implementation that matches the given locator.

Parameters
locator (*str*) – The location of the job store to be represent by the instance

Returns
an instance of a concrete subclass of AbstractJobStore

Return type
toil.jobStores.abstractJobStore.AbstractJobStore

`static parseLocator`(*locator*)

Parameters
locator (*str*) –

Return type
Tuple[str, str]

`static buildLocator`(*name*, *rest*)

Parameters

- **name** (*str*) –
- **rest** (*str*) –

Return type
str

`classmethod resumeJobStore`(*locator*)

Parameters
locator (*str*) –

Return type
toil.jobStores.abstractJobStore.AbstractJobStore

static createBatchSystem(*config*)

Create an instance of the batch system specified in the given config.

Parameters
config (Config) – the current configuration

Returns
an instance of a concrete subclass of AbstractBatchSystem

Return type
toil.batchSystems.abstractBatchSystem.AbstractBatchSystem

importFile(*srcUrl: str*, *sharedFileName: str*, *symlink: bool = True*) → None

importFile(*srcUrl: str*, *sharedFileName: None = None*, *symlink: bool = True*) → *toil.fileStores.FileID*

import_file(*src_uri: str*, *shared_file_name: str*, *symlink: bool = True*) → None

import_file(*src_uri: str*, *shared_file_name: None = None*, *symlink: bool = True*) → *toil.fileStores.FileID*

Import the file at the given URL into the job store.

See *toil.jobStores.abstractJobStore.AbstractJobStore.importFile()* for a full description

exportFile(*jobStoreFileID*, *dstUrl*)

Parameters

- **jobStoreFileID** (toil.fileStores.FileID) –
- **dstUrl** (*str*) –

Return type
None

export_file(*file_id*, *dst_uri*)

Export file to destination pointed at by the destination URL.

See *toil.jobStores.abstractJobStore.AbstractJobStore.exportFile()* for a full description

Parameters

- **file_id** (toil.fileStores.FileID) –
- **dst_uri** (*str*) –

Return type
None

static normalize_uri(*uri*, *check_existence=False*)

Given a URI, if it has no scheme, prepend “file:”.

Parameters

- **check_existence** (*bool*) – If set, raise an error if a URI points to a local file that does not exist.
- **uri** (*str*) –

Return type
str

static getToilWorkDir(*configWorkDir=None*)

Return a path to a writable directory under which per-workflow directories exist.

This directory is always required to exist on a machine, even if the Toil worker has not run yet. If your workers and leader have different temp directories, you may need to set TOIL_WORKDIR.

Parameters
configWorkDir (*Optional[str]*) – Value passed to the program using the –workDir flag

Returns
Path to the Toil work directory, constant across all machines

Return type
str

classmethod get_toil_coordination_dir(*config_work_dir*, *config_coordination_dir*)

Return a path to a writable directory, which will be in memory if convenient. Ought to be used for file locking and coordination.

Parameters

- **config_work_dir** (*Optional[str]*) – Value passed to the program using the –workDir flag
- **config_coordination_dir** (*Optional[str]*) – Value passed to the program using the –coordinationDir flag

Returns
Path to the Toil coordination directory. Ought to be on a POSIX filesystem that allows directories containing open files to be deleted.

Return type
str

classmethod getLocalWorkflowDir(*workflowID*, *configWorkDir=None*)

Return the directory where worker directories and the cache will be located for this workflow on this machine.

Parameters

- **configWorkDir** (*Optional[str]*) – Value passed to the program using the –workDir flag
- **workflowID** (*str*) –

Returns
Path to the local workflow directory on this machine

Return type
str

classmethod get_local_workflow_coordination_dir(*workflow_id*, *config_work_dir*, *config_coordination_dir*)

Return the directory where coordination files should be located for this workflow on this machine. These include internal Toil databases and lock files for the machine.

If an in-memory filesystem is available, it is used. Otherwise, the local workflow directory, which may be on a shared network filesystem, is used.

Parameters

- **workflow_id** (*str*) – Unique ID of the current workflow.

- **config_work_dir** (*Optional[str]*) – Value used for the work directory in the current Toil Config.
- **config_coordination_dir** (*Optional[str]*) – Value used for the coordination directory in the current Toil Config.

Returns
Path to the local workflow coordination directory on this machine.

Return type
str

exception toil.common.**ToilRestartException**(*message*)

Bases: Exception

Common base class for all non-exit exceptions.

Parameters
message (*str*) –

exception toil.common.**ToilContextManagerException**

Bases: Exception

Common base class for all non-exit exceptions.

class toil.common.**ToilMetrics**(*bus*, *provisioner=None*)

Parameters

- **bus** (toil.bus.MessageBus) –
- **provisioner** (*Optional[*toil.provisioners.abstractProvisioner.AbstractProvisioner*]*) –

startDashboard(*clusterName*, *zone*)

Parameters

- **clusterName** (*str*) –
- **zone** (*str*) –

Return type
None

add_prometheus_data_source()

Return type
None

log(*message*)

Parameters
message (*str*) –

Return type
None

logClusterSize(*m*)

Parameters
m (toil.bus.ClusterSizeMessage) –

Return type
None

logClusterDesiredSize(*m*)

Parameters
m (toil.bus.ClusterDesiredSizeMessage) –

Return type
None

logQueueSize(*m*)

Parameters
m (toil.bus.QueueSizeMessage) –

Return type
None

logMissingJob(*m*)

Parameters
m (toil.bus.JobMissingMessage) –

Return type
None

logIssuedJob(*m*)

Parameters
m (toil.bus.JobIssuedMessage) –

Return type
None

logFailedJob(*m*)

Parameters
m (toil.bus.JobFailedMessage) –

Return type
None

logCompletedJob(*m*)

Parameters
m (toil.bus.JobCompletedMessage) –

Return type
None

shutdown()

Return type
None

toil.common.**parseSetEnv**(*l*)

Parse a list of strings of the form "NAME=VALUE" or just "NAME" into a dictionary.

Strings of the latter from will result in dictionary entries whose value is None.

```
>>> parseSetEnv([])
{}
>>> parseSetEnv(['a'])
{'a': None}
>>> parseSetEnv(['a='])
{'a': ''}
>>> parseSetEnv(['a=b'])
{'a': 'b'}
>>> parseSetEnv(['a=a', 'a=b'])
{'a': 'b'}
>>> parseSetEnv(['a=b', 'c=d'])
{'a': 'b', 'c': 'd'}
>>> parseSetEnv(['a=b=c'])
{'a': 'b=c'}
>>> parseSetEnv([''])
Traceback (most recent call last):
...
ValueError: Empty name
>>> parseSetEnv(['=1'])
Traceback (most recent call last):
...
ValueError: Empty name
```

Parameters
l (*List[str]*) –

Return type
Dict[str, Optional[str]]

toil.common.**iC**(*minValue*, *maxValue=SYS_MAX_SIZE*)

Returns a function that checks if a given int is in the given half-open interval.

Parameters

- **minValue** (*int*) –
- **maxValue** (*int*) –

Return type
Callable[[int], bool]

toil.common.**fC**(*minValue*, *maxValue=None*)

Returns a function that checks if a given float is in the given half-open interval.

Parameters

- **minValue** (*float*) –
- **maxValue** (*Optional[float]*) –

Return type
Callable[[float], bool]

toil.common.**parse_accelerator_list**(*specs*)

Parse a string description of one or more accelerator requirements.

Parameters
specs (*Optional[str]*) –

Return type
List[*toil.job.AcceleratorRequirement*]

toil.common.**cacheDirName**(*workflowID*)

Returns
Name of the cache directory.

Parameters
workflowID (*str*) –

Return type
str

toil.common.**getDirSizeRecursively**(*dirPath*)

This method will return the cumulative number of bytes occupied by the files on disk in the directory and its subdirectories.

If the method is unable to access a file or directory (due to insufficient permissions, or due to the file or directory having been removed while this function was attempting to traverse it), the error will be handled internally, and a (possibly 0) lower bound on the size of the directory will be returned.

The environment variable 'BLOCKSIZE'='512' is set instead of the much cleaner –block-size=1 because Apple can't handle it.

Parameters
dirPath (*str*) – A valid path to a directory or file.

Returns
Total size, in bytes, of the file or directory at dirPath.

Return type
int

toil.common.**getFileSystemSize**(*dirPath*)

Return the free space, and total size of the file system hosting *dirPath*.

Parameters
dirPath (*str*) – A valid path to a directory.

Returns
free space and total size of file system

Return type
Tuple[int, int]

toil.common.**safeUnpickleFromStream**(*stream*)

Parameters
stream (*IO[Any]*) –

Return type
Any

toil.deferred

Module Contents

Classes

DeferredFunction	>>> from collections import defaultdict
DeferredFunctionManager	Implements a deferred function system. Each Toil worker will have an

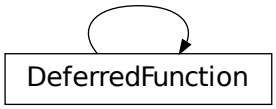
Attributes

logger

toil.deferred.**logger**

class toil.deferred.**DeferredFunction**

Bases: namedtuple('DeferredFunction', 'function args kwargs name module')




```
>>> from collections import defaultdict
>>> df = DeferredFunction.create(defaultdict, None, {'x':1}, y=2)
>>> df
DeferredFunction(defaultdict, ...)
>>> df.invoke() == defaultdict(None, x=1, y=2)
True
```

__repr__

classmethod create(*function*, **args*, ***kwargs*)

Capture the given callable and arguments as an instance of this class.

Parameters

- **function** (*callable*) – The deferred action to take in the form of a function
- **args** (*tuple*) – Non-keyword arguments to the function
- **kwargs** (*dict*) – Keyword arguments to the function

invoke()

Invoke the captured function with the captured arguments.

__str__()

Return str(self).

class toil.deferred.**DeferredFunctionManager**(*stateDirBase*)

Implements a deferred function system. Each Toil worker will have an instance of this class. When a job is executed, it will happen inside a context manager from this class. If the job registers any "deferred" functions, they will be executed when the context manager is exited.

If the Python process terminates before properly exiting the context manager and running the deferred functions, and some other worker process enters or exits the per-job context manager of this class at a later time, or when the DeferredFunctionManager is shut down on the worker, the earlier job's deferred functions will be picked up and run.

Note that deferred function cleanup is on a best-effort basis, and deferred functions may end up getting executed multiple times.

Internally, deferred functions are serialized into files in the given directory, which are locked by the owning process.

If that process dies, other processes can detect that the files are able to be locked, and will take them over.

Parameters
stateDirBase (*str*) –

STATE_DIR_STEM = 'deferred'

PREFIX = 'func'

WIP_SUFFIX = '.tmp'

__del__()

Clean up our state on disk. We assume that the deferred functions we manage have all been executed, and none are currently recorded.

open()

Yields a single-argument function that allows for deferred functions of type `toil.DeferredFunction` to be registered. We use this design so deferred functions can be registered only inside this context manager.

Not thread safe.

classmethod cleanupWorker(*stateDirBase*)

Called by the batch system when it shuts down the node, after all workers are done, if the batch system supports worker cleanup. Checks once more for orphaned deferred functions and runs them.

Parameters
stateDirBase (*str*) –

Return type
None

toil.exceptions

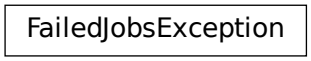
Neutral place for exceptions, to break import cycles.

Module Contents

toil.exceptions.**logger**

exception toil.exceptions.**FailedJobsException**(*job_store*, *failed_jobs*, *exit_code=1*)

Bases: Exception



Common base class for all non-exit exceptions.

Parameters

- **job_store** (toil.jobStores.abstractJobStore.AbstractJobStore) –
- **failed_jobs** (*List[*toil.job.JobDescription*]*) –
- **exit_code** (*int*) –

__str__()

Stringify the exception, including the message.

Return type

str

toil.job

Module Contents

Classes

TemporaryID	Placeholder for a unregistered job ID used by a JobDescription.
AcceleratorRequirement	Requirement for one or more computational accelerators, like a GPU or FPGA.
RequirementsDict	Typed storage for requirements for a job.
Requirer	Base class implementing the storage and presentation of requirements.
JobDescription	Stores all the information that the Toil Leader ever needs to know about a Job.
ServiceJobDescription	A description of a job that hosts a service.
CheckpointJobDescription	A description of a job that is a checkpoint.
Job	Class represents a unit of work in toil.
FunctionWrappingJob	Job used to wrap a function. In its *run* method the wrapped function is called.
JobFunctionWrappingJob	A job function is a function whose first argument is a *Job*
PromisedRequirementFunctionWrappingJob	Handles dynamic resource allocation using *toil.job.Promise* instances.
PromisedRequirementJobFunctionWrappingJob	Handles dynamic resource allocation for job functions.
EncapsulatedJob	A convenience Job class used to make a job subgraph appear to be a single job.
ServiceHostJob	Job that runs a service. Used internally by Toil. Users should subclass Service instead of using this.
Promise	References a return value from a method as a *promise* before the method itself is run.
PromisedRequirement	Class for dynamically allocating job function resource requirements.
UnfulfilledPromiseSentinel	This should be overwritten by a proper promised value.

Functions

parse_accelerator(spec)	Parse an AcceleratorRequirement specified by user code.
accelerator_satisfies(candidate, requirement[, ignore])	Test if candidate partially satisfies the given requirement.
accelerators_fully_satisfy(candidates, requirement[, ...])	Determine if a set of accelerators satisfy a requirement.
unwrap(p)	Function for ensuring you actually have a promised value, and not just a promise.
unwrap_all(p)	Function for ensuring you actually have a collection of promised values,

Attributes

logger
REQUIREMENT_NAMES
ParsedRequirement
ParseableIndivisibleResource
ParseableDivisibleResource
ParseableFlag
ParseableAcceleratorRequirement
ParseableRequirement
T
Promised

toil.job.**logger**

exception toil.job.**JobPromiseConstraintError**(*promisingJob*, *recipientJob=None*)

Bases: RuntimeError

Error for job being asked to promise its return value, but it not available.

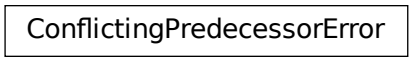
(Due to the return value not yet been hit in the topological order of the job graph.)

Parameters

- **promisingJob** (Job) –
- **recipientJob** (*Optional[Job]*) –

exception toil.job.**ConflictingPredecessorError**(*predecessor*, *successor*)

Bases: Exception



Common base class for all non-exit exceptions.

Parameters

- **predecessor** (Job) –
- **successor** (Job) –

class toil.job.**TemporaryID**

Placeholder for a unregistered job ID used by a JobDescription.

Needs to be held:

- By JobDescription objects to record normal relationships.
- By Jobs to key their connected-component registries and to record predecessor relationships to facilitate EncapsulatedJob adding itself as a child.
- By Services to tie back to their hosting jobs, so the service tree can be built up from Service objects.

__str__()

Return str(self).

Return type
str

__repr__()

Return repr(self).

Return type
str

__hash__()

Return hash(self).

Return type
int

__eq__(*other*)

Return self==value.

Parameters
other (*Any*) –

Return type
bool

__ne__(*other*)

Return self!=value.

Parameters
other (*Any*) –

Return type
bool

class toil.job.**AcceleratorRequirement**

Bases: TypedDict

Requirement for one or more computational accelerators, like a GPU or FPGA.

count: int

How many of the accelerator are needed to run the job.

kind: str

What kind of accelerator is required. Can be "gpu". Other kinds defined in the future might be "fpga", etc.

model: typing_extensions.NotRequired[str]

What model of accelerator is needed. The exact set of values available depends on what the backing scheduler calls its accelerators; strings like "nvidia-tesla-k80" might be expected to work. If a specific model of accelerator is not required, this should be absent.

brand: typing_extensions.NotRequired[str]

What brand or manufacturer of accelerator is required. The exact set of values available depends on what the backing scheduler calls the brands of its accleerators; strings like "nvidia" or "amd" might be expected to work. If a specific brand of accelerator is not required (for example, because the job can use multiple brands of accelerator that support a given API) this should be absent.

api: typing_extensions.NotRequired[str]

What API is to be used to communicate with the accelerator. This can be "cuda". Other APIs supported in the future might be "rocm", "opencl", "metal", etc. If the job does not need a particular API to talk to the accelerator, this should be absent.

toil.job.**parse_accelerator**(*spec*)

Parse an AcceleratorRequirement specified by user code.

Supports formats like:

```
>>> parse_accelerator(8)
{'count': 8, 'kind': 'gpu'}
```

```
>>> parse_accelerator("1")
{'count': 1, 'kind': 'gpu'}
```

```
>>> parse_accelerator("nvidia-tesla-k80")
{'count': 1, 'kind': 'gpu', 'brand': 'nvidia', 'model': 'nvidia-tesla-k80'}
```

```
>>> parse_accelerator("nvidia-tesla-k80:2")
{'count': 2, 'kind': 'gpu', 'brand': 'nvidia', 'model': 'nvidia-tesla-k80'}
```

```
>>> parse_accelerator("gpu")
{'count': 1, 'kind': 'gpu'}
```

```
>>> parse_accelerator("cuda:1")
{'count': 1, 'kind': 'gpu', 'brand': 'nvidia', 'api': 'cuda'}
```

```
>>> parse_accelerator({"kind": "gpu"})
{'count': 1, 'kind': 'gpu'}
```

```
>>> parse_accelerator({"brand": "nvidia", "count": 5})
{'count': 5, 'kind': 'gpu', 'brand': 'nvidia'}
```

Assumes that if not specified, we are talking about GPUs, and about one of them. Knows that "gpu" is a kind, and "cuda" is an API, and "nvidia" is a brand.

Raises

- **ValueError** – if it gets somethign it can't parse
- **TypeError** – if it gets something it can't parse because it's the wrong type.

Parameters

spec (*Union[int, str, Dict[str, Union[str, int]]]*) –

Return type

AcceleratorRequirement

toil.job.**accelerator_satisfies**(*candidate*, *requirement*, *ignore=[]*)

Test if candidate partially satisfies the given requirement.

Returns

True if the given candidate at least partially satisfies the given requirement (i.e. check all fields other than count).

Parameters

- **candidate** (AcceleratorRequirement) –
- **requirement** (AcceleratorRequirement) –
- **ignore** (*List[str]*) –

Return type

bool

toil.job.**accelerators_fully_satisfy**(*candidates*, *requirement*, *ignore=[]*)

Determine if a set of accelerators satisfy a requirement.

Ignores fields specified in ignore.

Returns

True if the requirement AcceleratorRequirement is fully satisfied by the ones in the list, taken together (i.e. check all fields including count).

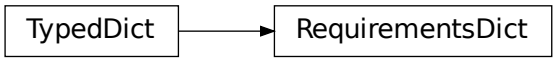
Parameters

- **candidates** (*Optional[List[AcceleratorRequirement]]*) –
- **requirement** (AcceleratorRequirement) –
- **ignore** (*List[str]*) –

Return type
bool

class toil.job.**RequirementsDict**

Bases: TypedDict



Typed storage for requirements for a job.

Where requirement values are of different types depending on the requirement.

cores: typing_extensions.NotRrequired[Union[int, float]]

memory: typing_extensions.NotRrequired[int]

disk: typing_extensions.NotRrequired[int]

accelerators: typing_extensions.NotRrequired[List[*AcceleratorRequirement*]]

preemptible: typing_extensions.NotRrequired[bool]

toil.job.**REQUIREMENT_NAMES** = ['disk', 'memory', 'cores', 'accelerators', 'preemptible']

toil.job.**ParsedRequirement**

toil.job.**ParseableIndivisibleResource**

toil.job.**ParseableDivisibleResource**

toil.job.**ParseableFlag**

toil.job.**ParseableAcceleratorRequirement**

toil.job.**ParseableRequirement**

class toil.job.**Requirer**(*requirements*)

Base class implementing the storage and presentation of requirements.

Has cores, memory, disk, and preemptability as properties.

Parameters
requirements (*Mapping[str, ParseableRequirement]*) –

property requirements: *RequirementsDict*

Get dict containing all non-None, non-defaulted requirements.

Return type
RequirementsDict

property disk: int

Get the maximum number of bytes of disk required.

Return type
int

property memory: int

Get the maximum number of bytes of memory required.

Return type
int

property cores: Union[int, float]

Get the number of CPU cores required.

Return type
Union[int, float]

property preemptible: bool

Whether a preemptible node is permitted, or a nonpreemptible one is required.

Return type
bool

property accelerators: List[*AcceleratorRequirement*]

Any accelerators, such as GPUs, that are needed.

Return type
List[*AcceleratorRequirement*]

assignConfig(*config*)

Assign the given config object to be used to provide default values.

Must be called exactly once on a loaded JobDescription before any requirements are queried.

Parameters
config (toil.common.Config) – Config object to query

Return type
None

__getstate__()

Return the dict to use as the instance's __dict__ when pickling.

Return type
Dict[str, Any]

__copy__()

Return a semantically-shallow copy of the object, for `copy.copy()`.

Return type
Requirer

__deepcopy__(*memo*)

Return a semantically-deep copy of the object, for `copy.deepcopy()`.

Parameters
memo (*Any*) –

Return type
Requirer

preemptable(*val*)

Parameters
val (*ParseableFlag*) –

Return type
None

scale(*requirement*, *factor*)

Return a copy of this object with the given requirement scaled up or down.

Only works on requirements where that makes sense.

Parameters

- **requirement** (*str*) –
- **factor** (*float*) –

Return type
Requirer

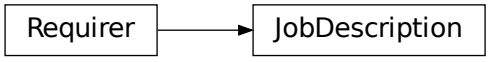
requirements_string()

Get a nice human-readable string of our requirements.

Return type
str

class toil.job.**JobDescription**(*requirements*, *jobName*, *unitName=''*, *displayName=''*, *command=None*, *local=None*)

Bases: *Requirer*



Stores all the information that the Toil Leader ever needs to know about a Job.

(requirements information, dependency information, commands to issue, etc.)

Can be obtained from an actual (i.e. executable) Job object, and can be used to obtain the Job object from the JobStore.

Never contains other Jobs or JobDescriptions: all reference is by ID.

Subclassed into variants for checkpoint jobs and service jobs that have their specific parameters.

Parameters

- **requirements** (*Mapping[str, Union[int, str, bool]]*) –
- **jobName** (*str*) –
- **unitName** (*Optional[str]*) –
- **displayName** (*Optional[str]*) –
- **command** (*Optional[str]*) –

- **local** (*Optional[bool]*) –

property services

Get a collection of the IDs of service host jobs for this job, in arbitrary order.

Will be empty if the job has no unfinished services.

property remainingTryCount

Get the number of tries remaining.

The try count set on the JobDescription, or the default based on the retry count from the config if none is set.

serviceHostIDsInBatches()

Find all batches of service host job IDs that can be started at the same time.

(in the order they need to start in)

Return type
Iterator[List[str]]

successorsAndServiceHosts()

Get an iterator over all child, follow-on, and service job IDs.

Return type
Iterator[str]

allSuccessors()

Get an iterator over all child, follow-on, and chained, inherited successor job IDs.

Follow-ons will come before children.

Return type
Iterator[str]

successors_by_phase()

Get an iterator over all child/follow-on/chained inherited successor job IDs, along with their phase numbere on the stack.

Phases ececute higher numbers to lower numbers.

Return type
Iterator[Tuple[int, str]]

nextSuccessors()

Return the collection of job IDs for the successors of this job that are ready to run.

If those jobs have multiple predecessor relationships, they may still be blocked on other jobs.

Returns None when at the final phase (all successors done), and an empty collection if there are more phases but they can't be entered yet (e.g. because we are waiting for the job itself to run).

Return type
Set[str]

filterSuccessors(*predicate*)

Keep only successor jobs for which the given predicate function approves.

The predicate function is called with the job's ID.

Treats all other successors as complete and forgets them.

Parameters
predicate (*Callable[[str], bool]*) –

Return type
None

`filterServiceHosts`(*predicate*)

Keep only services for which the given predicate approves.

The predicate function is called with the service host job's ID.

Treats all other services as complete and forgets them.

Parameters
predicate (*Callable[[str], bool]*) –

Return type
None

`clear_nonexistent_dependents`(*job_store*)

Remove all references to child, follow-on, and associated service jobs that do not exist.

That is to say, all those that have been completed and removed.

Parameters
job_store (toil.jobStores.abstractJobStore.AbstractJobStore) –

Return type
None

`clear_dependents`()

Remove all references to successor and service jobs.

Return type
None

`is_subtree_done`()

Check if the subtree is done.

Returns
True if the job appears to be done, and all related child, follow-on, and service jobs appear to be finished and removed.

Return type
bool

`replace`(*other*)

Take on the ID of another JobDescription, retaining our own state and type.

When updated in the JobStore, we will save over the other JobDescription.

Useful for chaining jobs: the chained-to job can replace the parent job.

Merges cleanup state and successors other than this job from the job being replaced into this one.

Parameters
other (JobDescription) – Job description to replace.

Return type
None

`addChild`(*childID*)

Make the job with the given ID a child of the described job.

Parameters
childID (*str*) –

Return type
None

addFollowOn(*followOnID*)

Make the job with the given ID a follow-on of the described job.

Parameters
followOnID (*str*) –

Return type
None

addServiceHostJob(*serviceID*, *parentServiceID=None*)

Make the ServiceHostJob with the given ID a service of the described job.

If a parent ServiceHostJob ID is given, that parent service will be started first, and must have already been added.

hasChild(*childID*)

Return True if the job with the given ID is a child of the described job.

Parameters
childID (*str*) –

Return type
bool

hasFollowOn(*followOnID*)

Test if the job with the given ID is a follow-on of the described job.

Parameters
followOnID (*str*) –

Return type
bool

hasServiceHostJob(*serviceID*)

Test if the ServiceHostJob is a service of the described job.

Return type
bool

renameReferences(*renames*)

Apply the given dict of ID renames to all references to jobs.

Does not modify our own ID or those of finished predecessors. IDs not present in the renames dict are left as-is.

Parameters
renames (*Dict[TemporaryID, str]*) – Rename operations to apply.

Return type
None

addPredecessor()

Notify the JobDescription that a predecessor has been added to its Job.

Return type
None

onRegistration(*jobStore*)

Perform setup work that requires the JobStore.

Called by the Job saving logic when this JobDescription meets the JobStore and has its ID assigned.

Overridden to perform setup work (like hooking up flag files for service jobs) that requires the JobStore.

Parameters
jobStore (toil.jobStores.abstractJobStore.AbstractJobStore) – The job store we are being placed into

Return type
None

setupJobAfterFailure(*exit_status=None*, *exit_reason=None*)

Configure job after a failure.

Reduce the remainingTryCount if greater than zero and set the memory to be at least as big as the default memory (in case of exhaustion of memory, which is common).

Requires a configuration to have been assigned (see *toil.job.Requirer.assignConfig()*).

Parameters

- **exit_status** (*Optional[int]*) – The exit code from the job.
- **exit_reason** (*Optional[toil.batchSystems.abstractBatchSystem.BatchJobExitReason]*) – The reason the job stopped, if available from the batch system.

Return type
None

getLogFileHandle(*jobStore*)

Create a context manager that yields a file handle to the log file.

Assumes logJobStoreFileID is set.

clearRemainingTryCount()

Clear remainingTryCount and set it back to its default value.

Returns
True if a modification to the JobDescription was made, and False otherwise.

Return type
bool

__str__()

Produce a useful logging string identifying this job.

Return type
str

__repr__()

Return repr(self).

pre_update_hook()

Run before pickling and saving a created or updated version of this job.

Called by the job store.

Return type
None

get_job_kind()

Return an identifying string for the job.

The result may contain spaces.

Returns: Either the unit name, job name, or display name, which identifies
the kind of job it is to toil. Otherwise "Unknown Job" in case no identifier is available

Return type
str

class toil.job.**ServiceJobDescription**(**args*, ***kwargs*)

Bases: *JobDescription*

A description of a job that hosts a service.

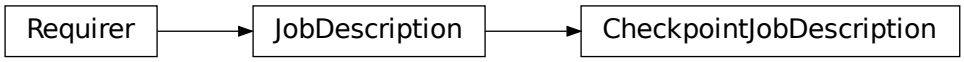
onRegistration(*jobStore*)

Setup flag files.

When a ServiceJobDescription first meets the JobStore, it needs to set up its flag files.

class toil.job.**CheckpointJobDescription**(**args*, ***kwargs*)

Bases: *JobDescription*



A description of a job that is a checkpoint.

restartCheckpoint(*jobStore*)

Restart a checkpoint after the total failure of jobs in its subtree.

Writes the changes to the jobStore immediately. All the checkpoint's successors will be deleted, but its try count will *not* be decreased.

Returns a list with the IDs of any successors deleted.

Parameters
jobStore (toil.jobStores.abstractJobStore.AbstractJobStore) –

Return type
List[str]

class toil.job.Job(*memory=None*, *cores=None*, *disk=None*, *accelerators=None*, *preemptible=None*, *preemptable=None*, *unitName=''*, *checkpoint=False*, *displayName=''*, *descriptionClass=None*, *local=None*)

Class represents a unit of work in toil.

Parameters

- **memory** (*Optional[ParseableIndivisibleResource]*) –
- **cores** (*Optional[ParseableDivisibleResource]*) –
- **disk** (*Optional[ParseableIndivisibleResource]*) –
- **accelerators** (*Optional[ParseableAcceleratorRequirement]*) –
- **preemptible** (*Optional[ParseableFlag]*) –
- **preemptable** (*Optional[ParseableFlag]*) –
- **unitName** (*Optional[str]*) –
- **checkpoint** (*Optional[bool]*) –
- **displayName** (*Optional[str]*) –
- **descriptionClass** (*Optional[type]*) –
- **local** (*Optional[bool]*) –

class Runner

Used to setup and run Toil workflow.

static getDefaultArgumentParser()

Get argument parser with added toil workflow options.

Returns
The argument parser used by a toil workflow with added Toil options.

Return type
argparse.ArgumentParser

static getDefaultOptions(*jobStore*)

Get default options for a toil workflow.

Parameters
jobStore (*str*) – A string describing the jobStore for the workflow.

Returns
The options used by a toil workflow.

Return type
argparse.Namespace

static addToilOptions(*parser*)

Adds the default toil options to an optparse or argparse parser object.

Parameters
parser (*Union[optparse.OptionParser, argparse.ArgumentParser]*) – Options object to add toil options to.

Return type
None

static startToil(*job*, *options*)

Run the toil workflow using the given options.

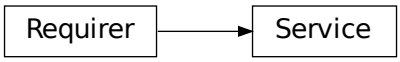
Deprecated by toil.common.Toil.start.

(see Job.Runner.getDefaultOptions and Job.Runner.addToilOptions) starting with this job. :param job: root job of the workflow :raises: toil.exceptions.FailedJobsException if at the end of function there remain failed jobs. :return: The return value of the root job's run function.

Parameters
job (Job) –
Return type
Any

class Service(*memory=None*, *cores=None*, *disk=None*, *accelerators=None*, *preemptible=None*, *unitName=None*)

Bases: *Requirer*



Abstract class used to define the interface to a service.

Should be subclassed by the user to define services.

Is not executed as a job; runs within a ServiceHostJob.

abstract start(*job*)

Start the service.

Parameters
job (Job) – The underlying host job that the service is being run in. Can be used to register deferred functions, or to access the fileStore for creating temporary files.
Returns
An object describing how to access the service. The object must be pickleable and will be used by jobs to access the service (see *toil.job.Job.addService()*).
Return type
Any

abstract stop(*job*)

Stops the service. Function can block until complete.

Parameters
job (Job) – The underlying host job that the service is being run in. Can be used to register deferred functions, or to access the fileStore for creating temporary files.
Return type
None

check()

Checks the service is still running.

Raises
exceptions.RuntimeError – If the service failed, this will cause the service job to be labeled failed.
Returns
True if the service is still running, else False. If False then the service job will be terminated, and considered a success. Important point: if the service job exits due to a failure, it should raise a RuntimeError, not return False!
Return type
bool

property jobStoreID: Union[str, *TemporaryID*]

Get the ID of this Job.

Return type
Union[str, *TemporaryID*]

property description: *JobDescription*

Expose the JobDescription that describes this job.

Return type
JobDescription

property disk: int

The maximum number of bytes of disk the job will require to run.

Return type
int

property memory

The maximum number of bytes of memory the job will require to run.

property cores: Union[int, float]

The number of CPU cores required.

Return type
Union[int, float]

property accelerators: List[*AcceleratorRequirement*]

Any accelerators, such as GPUs, that are needed.

Return type
List[*AcceleratorRequirement*]

property preemptible: bool

Whether the job can be run on a preemptible node.

Return type
bool

property checkpoint: bool

Determine if the job is a checkpoint job or not.

Return type
bool

property tempDir: str

Shortcut to calling `job.fileStore.getLocalTempDir()`.

Temp dir is created on first call and will be returned for first and future calls :return: Path to tempDir. See *job.fileStore.getLocalTempDir*

Return type
str

__str__()

Produce a useful logging string to identify this Job and distinguish it from its JobDescription.

preemptable()

assignConfig(*config*)

Assign the given config object.

It will be used by various actions implemented inside the Job class.

Parameters
config (toil.common.Config) – Config object to query

Return type
None

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters
fileStore (toil.fileStores.abstractFileStore.AbstractFileStore) – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns
The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

Return type
Any

addChild(*childJob*)

Add a childJob to be run as child of this job.

Child jobs will be run directly after this job's *toil.job.Job.run()* method has completed.

Returns
childJob: for call chaining

Parameters
childJob (Job) –

Return type
Job

hasChild(*childJob*)

Check if childJob is already a child of this job.

Returns
True if childJob is a child of the job, else False.

Parameters
childJob (Job) –

Return type
bool

addFollowOn(*followOnJob*)

Add a follow-on job.

Follow-on jobs will be run after the child jobs and their successors have been run.

Returns
followOnJob for call chaining

Parameters
followOnJob (Job) –

Return type
Job

hasPredecessor(*job*)

Check if a given job is already a predecessor of this job.

Parameters
job (Job) –

Return type
bool

hasFollowOn(*followOnJob*)

Check if given job is already a follow-on of this job.

Returns
True if the followOnJob is a follow-on of this job, else False.

Parameters
followOnJob (Job) –

Return type
bool

addService(*service*, *parentService=None*)

Add a service.

The *toil.job.Job.Service.start()* method of the service will be called after the run method has completed but before any successors are run. The service's *toil.job.Job.Service.stop()* method will be called once the successors of the job have been run.

Services allow things like databases and servers to be started and accessed by jobs in a workflow.

Raises
toil.job.JobException – If service has already been made the child of a job or another service.

Parameters

- **service** (Service) – Service to add.
- **parentService** (*Optional[Service]*) – Service that will be started before 'service' is started. Allows trees of services to be established. parentService must be a service of this job.

Returns
a promise that will be replaced with the return value from *toil.job.Job.Service.start()* of service in any successor of the job.

Return type
Promise

hasService(*service*)

Return True if the given Service is a service of this job, and False otherwise.

Parameters
service (Service) –

Return type
bool

addChildFn(*fn*, **args*, ***kwargs*)

Add a function as a child job.

Parameters
fn (*Callable*) – Function to be run as a child job with `*args` and `**kwargs` as arguments to this function. See toil.job.FunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Returns
The new child job that wraps fn.

Return type
FunctionWrappingJob

addFollowOnFn(*fn*, **args*, ***kwargs*)

Add a function as a follow-on job.

Parameters
fn (*Callable*) – Function to be run as a follow-on job with `*args` and `**kwargs` as arguments to this function. See toil.job.FunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Returns
The new follow-on job that wraps fn.

Return type
FunctionWrappingJob

addChildJobFn(*fn*, **args*, ***kwargs*)

Add a job function as a child job.

See *toil.job.JobFunctionWrappingJob* for a definition of a job function.

Parameters
fn (*Callable*) – Job function to be run as a child job with `*args` and `**kwargs` as arguments to this function. See toil.job.JobFunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Returns
The new child job that wraps fn.

Return type
FunctionWrappingJob

addFollowOnJobFn(*fn*, **args*, ***kwargs*)

Add a follow-on job function.

See *toil.job.JobFunctionWrappingJob* for a definition of a job function.

Parameters
fn (*Callable*) – Job function to be run as a follow-on job with `*args` and `**kwargs` as arguments to this function. See toil.job.JobFunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Returns
The new follow-on job that wraps fn.

Return type
FunctionWrappingJob

log(*text*, *level=logging.INFO*)

Log using `fileStore.logToMaster()`.

Parameters
text (*str*) –

Return type
None

static wrapFn(*fn*, **args*, ***kwargs*)

Makes a Job out of a function.

Convenience function for constructor of *toil.job.FunctionWrappingJob*.

Parameters
fn – Function to be run with `*args` and `**kwargs` as arguments. See toil.job.JobFunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Returns
The new function that wraps fn.

Return type
FunctionWrappingJob

static wrapJobFn(*fn*, **args*, ***kwargs*)

Makes a Job out of a job function.

Convenience function for constructor of *toil.job.JobFunctionWrappingJob*.

Parameters
fn – Job function to be run with `*args` and `**kwargs` as arguments. See toil.job.JobFunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Returns
The new job function that wraps fn.

Return type
JobFunctionWrappingJob

encapsulate(*name=None*)

Encapsulates the job, see *toil.job.EncapsulatedJob*. Convenience function for constructor of *toil.job.EncapsulatedJob*.

Parameters
name (*Optional[str]*) – Human-readable name for the encapsulated job.

Returns
an encapsulated version of this job.

Return type
EncapsulatedJob

rv(**path*)

Create a *promise* (*toil.job.Promise*).

The "promise" representing a return value of the job's run method, or, in case of a function-wrapping job, the wrapped function's return value.

Parameters
path ((*Any*)) – Optional path for selecting a component of the promised return value. If

absent or empty, the entire return value will be used. Otherwise, the first element of the path is used to select an individual item of the return value. For that to work, the return value must be a list, dictionary or of any other type implementing the *__getitem__()* magic method. If the selected item is yet another composite value, the second element of the path can be used to select an item from it, and so on. For example, if the return value is *[6,{'a':42}]*, *.rv(0)* would select *6* , *rv(1)* would select *{'a':3}* while *rv(1,'a')* would select *3*. To select a slice from a return value that is slicable, e.g. tuple or list, the path element should be a *slice* object. For example, assuming that the return value is *[6, 7, 8, 9]* then *.rv(slice(1, 3))* would select *[7, 8]*. Note that slicing really only makes sense at the end of path.

Returns
A promise representing the return value of this jobs *toil.job.Job.run()* method.

Return type
Promise

registerPromise(*path*)

prepareForPromiseRegistration(*jobStore*)

Set up to allow this job's promises to register themselves.

Prepare this job (the promisor) so that its promises can register themselves with it, when the jobs they are promised to (promisees) are serialized.

The promissee holds the reference to the promise (usually as part of the job arguments) and when it is being pickled, so will the promises it refers to. Pickling a promise triggers it to be registered with the promissor.

Parameters
jobStore (toil.jobStores.abstractJobStore.AbstractJobStore) –

Return type
None

checkJobGraphForDeadlocks()

Ensures that a graph of Jobs (that hasn't yet been saved to the JobStore) doesn't contain any pathological relationships between jobs that would result in deadlocks if we tried to run the jobs.

See *toil.job.Job.checkJobGraphConnected()*, toil.job.Job.checkJobGraphAcyclic() and *toil.job.Job.checkNewCheckpointsAreLeafVertices()* for more info.

Raises
toil.job.JobGraphDeadlockException – if the job graph is cyclic, contains multiple roots or contains checkpoint jobs that are not leaf vertices when defined (see toil.job.Job.checkNewCheckpointsAreLeaves()).

getRootJobs()

Return the set of root job objects that contain this job.

A root job is a job with no predecessors (i.e. which are not children, follow-ons, or services).

Only deals with jobs created here, rather than loaded from the job store.

Return type
Set[*Job*]

checkJobGraphConnected()

Raises
toil.job.JobGraphDeadlockException – if *toil.job.Job.getRootJobs()* does not contain exactly one root job.

As execution always starts from one root job, having multiple root jobs will cause a deadlock to occur.

Only deals with jobs created here, rather than loaded from the job store.

checkJobGraphAcylic()

Raises
toil.job.JobGraphDeadlockException – if the connected component of jobs containing this job contains any cycles of child/followOn dependencies in the *augmented job graph* (see below). Such cycles are not allowed in valid job graphs.

A follow-on edge (A, B) between two jobs A and B is equivalent to adding a child edge to B from (1) A, (2) from each child of A, and (3) from the successors of each child of A. We call each such edge an edge an "implied" edge. The augmented job graph is a job graph including all the implied edges.

For a job graph G = (V, E) the algorithm is `O(|V|^2)`. It is `O(|V| + |E|)` for a graph with no follow-ons. The former follow-on case could be improved!

Only deals with jobs created here, rather than loaded from the job store.

checkNewCheckpointsAreLeafVertices()

A checkpoint job is a job that is restarted if either it fails, or if any of its successors completely fails, exhausting their retries.

A job is a leaf it is has no successors.

A checkpoint job must be a leaf when initially added to the job graph. When its run method is invoked it can then create direct successors. This restriction is made to simplify implementation.

Only works on connected components of jobs not yet added to the JobStore.

Raises
toil.job.JobGraphDeadlockException – if there exists a job being added to the graph for which checkpoint=True and which is not a leaf.

Return type
None

defer(*function*, **args*, ***kwargs*)

Register a deferred function, i.e. a callable that will be invoked after the current attempt at running this job concludes. A job attempt is said to conclude when the job function (or the *toil.job.Job.run()* method for class-based jobs) returns, raises an exception or after the process running it terminates abnormally. A deferred function will be called on the node that attempted to run the job, even if a subsequent attempt is made on another node. A deferred function should be idempotent because it may be called multiple times on the same node or even in the same process. More than one deferred function may be registered per job attempt by calling this method repeatedly with different arguments. If the same function is registered twice with the same or different arguments, it will be called twice per job attempt.

Examples for deferred functions are ones that handle cleanup of resources external to Toil, like Docker containers, files outside the work directory, etc.

Parameters

- **function** (*callable*) – The function to be called after this job concludes.
- **args** (*list*) – The arguments to the function
- **kwargs** (*dict*) – The keyword arguments to the function

Return type
None

getUserScript()

Return type
toil.resource.ModuleDescriptor

getTopologicalOrderingOfJobs()

Returns
a list of jobs such that for all pairs of indices i, j for which i < j, the job at index i can be run before the job at index j.

Return type
List[*Job*]

Only considers jobs in this job's subgraph that are newly added, not loaded from the job store.

Ignores service jobs.

saveBody(*jobStore*)

Save the execution data for just this job to the JobStore, and fill in the JobDescription with the information needed to retrieve it.

The Job's JobDescription must have already had a real jobStoreID assigned to it.

Does not save the JobDescription.

Parameters
jobStore (toil.jobStores.abstractJobStore.AbstractJobStore) – The job store to save the job body into.

Return type
None

saveAsRootJob(*jobStore*)

Save this job to the given jobStore as the root job of the workflow.

Returns
the JobDescription describing this job.

Parameters
jobStore (toil.jobStores.abstractJobStore.AbstractJobStore) –

Return type
JobDescription

classmethod loadJob(*jobStore*, *jobDescription*)

Retrieves a *toil.job.Job* instance from a JobStore

Parameters

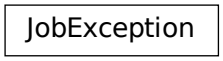
- **jobStore** (toil.jobStores.abstractJobStore.AbstractJobStore) – The job store.
- **jobDescription** (JobDescription) – the JobDescription of the job to retrieve.

Returns
The job referenced by the JobDescription.

Return type
Job

exception toil.job.**JobException**(*message*)

Bases: Exception

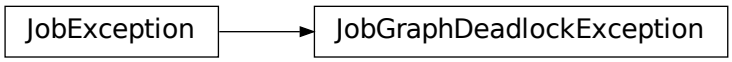


General job exception.

Parameters
message (*str*) –

exception toil.job.**JobGraphDeadlockException**(*string*)

Bases: *JobException*



An exception raised in the event that a workflow contains an unresolvable dependency, such as a cycle. See *toil.job.Job.checkJobGraphForDeadlocks()*.

class toil.job.**FunctionWrappingJob**(*userFunction*, **args*, ***kwargs*)

Bases: *Job*

Job used to wrap a function. In its *run* method the wrapped function is called.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

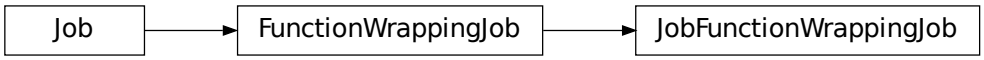
Parameters
fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns
The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

getUserScript()

class toil.job.**JobFunctionWrappingJob**(*userFunction*, **args*, ***kwargs*)

Bases: *FunctionWrappingJob*



A job function is a function whose first argument is a *Job* instance that is the wrapping job for the function. This can be used to add successor jobs for the function and perform all the functions the *Job* class provides.

To enable the job function to get access to the *toil.fileStores.abstractFileStore.AbstractFileStore* instance (see *toil.job.Job.run()*), it is made a variable of the wrapping job called fileStore.

To specify a job's resource requirements the following default keyword arguments can be specified:

- memory
- disk
- cores
- accelerators
- preemptible

For example to wrap a function into a job we would call:

```
Job.wrapJobFn(myJob, memory='100k', disk='1M', cores=0.1)
```

property fileStore

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

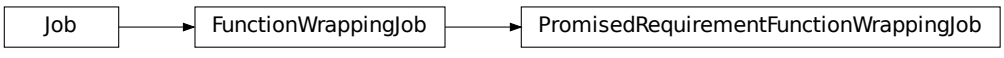
fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

class toil.job.**PromisedRequirementFunctionWrappingJob**(*userFunction*, **args*, ***kwargs*)

Bases: *FunctionWrappingJob*



Handles dynamic resource allocation using *toil.job.Promise* instances. Spawns child function using parent function parameters and fulfilled promised resource requirements.

classmethod create(*userFunction*, **args*, ***kwargs*)

Creates an encapsulated Toil job function with unfulfilled promised resource requirements. After the promises are fulfilled, a child job function is created using updated resource values. The subgraph is encapsulated to ensure that this child job function is run before other children in the workflow. Otherwise, a different child may try to use an unresolved promise return value from the parent.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

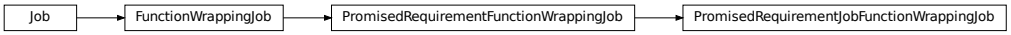
Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

evaluatePromisedRequirements()

class toil.job.**PromisedRequirementJobFunctionWrappingJob**(*userFunction*, **args*, ***kwargs*)

Bases: *PromisedRequirementFunctionWrappingJob*



Handles dynamic resource allocation for job functions. See *toil.job.JobFunctionWrappingJob*

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

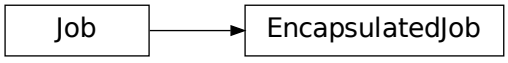
fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

class toil.job.**EncapsulatedJob**(*job*, *unitName=None*)

Bases: *Job*



A convenience Job class used to make a job subgraph appear to be a single job.

Let A be the root job of a job subgraph and B be another job we'd like to run after A and all its successors have completed, for this use encapsulate:

```
#  Job A and subgraph, Job B
A, B = A(), B()
Aprime = A.encapsulate()
Aprime.addChild(B)
#  B will run after A and all its successors have completed, A and its subgraph of
# successors in effect appear to be just one job.
```

If the job being encapsulated has predecessors (e.g. is not the root job), then the encapsulated job will inherit these predecessors. If predecessors are added to the job being encapsulated after the encapsulated job is created then the encapsulating job will NOT inherit these predecessors automatically. Care should be exercised to ensure the encapsulated job has the proper set of predecessors.

The return value of an encapsulated job (as accessed by the *toil.job.Job.rv()* function) is the return value of the root job, e.g. A().encapsulate().rv() and A().rv() will resolve to the same value after A or A.encapsulate() has been run.

addChild(*childJob*)

Add a childJob to be run as child of this job.

Child jobs will be run directly after this job's *toil.job.Job.run()* method has completed.

Returns
childJob: for call chaining

addService(*service*, *parentService=None*)

Add a service.

The *toil.job.Job.Service.start()* method of the service will be called after the run method has completed but before any successors are run. The service's *toil.job.Job.Service.stop()* method will be called once the successors of the job have been run.

Services allow things like databases and servers to be started and accessed by jobs in a workflow.

Raises
toil.job.JobException – If service has already been made the child of a job or another service.

Parameters

- **service** – Service to add.
- **parentService** – Service that will be started before 'service' is started. Allows trees of services to be established. parentService must be a service of this job.

Returns
a promise that will be replaced with the return value from *toil.job.Job.Service.start()* of service in any successor of the job.

addFollowOn(*followOnJob*)

Add a follow-on job.

Follow-on jobs will be run after the child jobs and their successors have been run.

Returns
followOnJob for call chaining

rv(**path*)

Create a *promise* (*toil.job.Promise*).

The "promise" representing a return value of the job's run method, or, in case of a function-wrapping job, the wrapped function's return value.

Parameters

path (*(Any)*) – Optional path for selecting a component of the promised return value. If absent or empty, the entire return value will be used. Otherwise, the first element of the path is used to select an individual item of the return value. For that to work, the return value must be a list, dictionary or of any other type implementing the *__getitem__()* magic method. If the selected item is yet another composite value, the second element of the path can be used to select an item from it, and so on. For example, if the return value is *[6,{'a':42}]*, *.rv(0)* would select *6* , *rv(1)* would select *{'a':3}* while *rv(1,'a')* would select *3*. To select a slice from a return value that is slicable, e.g. tuple or list, the path element should be a *slice* object. For example, assuming that the return value is *[6, 7, 8, 9]* then *.rv(slice(1, 3))* would select *[7, 8]*. Note that slicing really only makes sense at the end of path.

Returns

A promise representing the return value of this jobs *toil.job.Job.run()* method.

Return type

Promise

prepareForPromiseRegistration(*jobStore*)

Set up to allow this job's promises to register themselves.

Prepare this job (the promisor) so that its promises can register themselves with it, when the jobs they are promised to (promisees) are serialized.

The promissee holds the reference to the promise (usually as part of the job arguments) and when it is being pickled, so will the promises it refers to. Pickling a promise triggers it to be registered with the promissor.

__reduce__()

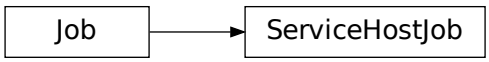
Called during pickling to define the pickled representation of the job.

We don't want to pickle our internal references to the job we encapsulate, so we elide them here. When actually run, we're just a no-op job that can maybe chain.

getUserScript()

class toil.job.**ServiceHostJob**(*service*)

Bases: *Job*



Job that runs a service. Used internally by Toil. Users should subclass Service instead of using this.

property fileStore

Return the file store, which the Service may need.

addChild(*child*)

Add a childJob to be run as child of this job.

Child jobs will be run directly after this job's *toil.job.Job.run()* method has completed.

Returns
childJob: for call chaining

addFollowOn(*followOn*)

Add a follow-on job.

Follow-on jobs will be run after the child jobs and their successors have been run.

Returns
followOnJob for call chaining

addService(*service*, *parentService=None*)

Add a service.

The *toil.job.Job.Service.start()* method of the service will be called after the run method has completed but before any successors are run. The service's *toil.job.Job.Service.stop()* method will be called once the successors of the job have been run.

Services allow things like databases and servers to be started and accessed by jobs in a workflow.

Raises
toil.job.JobException – If service has already been made the child of a job or another service.

Parameters

- **service** – Service to add.
- **parentService** – Service that will be started before 'service' is started. Allows trees of services to be established. parentService must be a service of this job.

Returns
a promise that will be replaced with the return value from *toil.job.Job.Service.start()* of service in any successor of the job.

saveBody(*jobStore*)

Serialize the service itself before saving the host job's body.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters
fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns
The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

getUserScript()

class toil.job.**Promise**(*job*, *path*)

References a return value from a method as a *promise* before the method itself is run.

References a return value from a *toil.job.Job.run()* or *toil.job.Job.Service.start()* method as a *promise* before the method itself is run.

Let T be a job. Instances of *Promise* (termed a *promise*) are returned by T.rv(), which is used to reference the return value of T's run function. When the promise is passed to the constructor (or as an argument to a wrapped function) of a different, successor job the promise will be replaced by the actual referenced return value. This mechanism allows a return values from one job's run method to be input argument to job before the former job's run function has been executed.

Parameters

- **job** (*Job*) –
- **path** (*Any*) –

filesToDelete

A set of IDs of files containing promised values when we know we won't need them anymore

__reduce__()

Return the Promise class and construction arguments.

Called during pickling when a promise (an instance of this class) is about to be be pickled. Returns the Promise class and construction arguments that will be evaluated during unpickling, namely the job store coordinates of a file that will hold the promised return value. By the time the promise is about to be unpickled, that file should be populated.

toil.job.**T**

toil.job.**Promised**

toil.job.**unwrap**(*p*)

Function for ensuring you actually have a promised value, and not just a promise. Mostly useful for satisfying type-checking.

The "unwrap" terminology is borrowed from Rust.

Parameters
p (*Promised[T]*) –

Return type
T

toil.job.**unwrap_all**(*p*)

Function for ensuring you actually have a collection of promised values, and not any remaining promises. Mostly useful for satisfying type-checking.

The "unwrap" terminology is borrowed from Rust.

Parameters
p (*Sequence[Promised[T]]*) –

Return type
Sequence[T]

class toil.job.**PromisedRequirement**(*valueOrCallable*, **args*)

Class for dynamically allocating job function resource requirements.

(involving *toil.job.Promise* instances.)

Use when resource requirements depend on the return value of a parent function. PromisedRequirements can be modified by passing a function that takes the *Promise* as input.

For example, let f, g, and h be functions. Then a Toil workflow can be defined as follows:: A = Job.wrapFn(f) B = A.addChildFn(g, cores=PromisedRequirement(A.rv()) C = B.addChildFn(h, cores=PromisedRequirement(lambda x: 2*x, B.rv()))

getValue()

Return PromisedRequirement value.

static convertPromises(*kwargs*)

Return True if reserved resource keyword is a Promise or PromisedRequirement instance.

Converts Promise instance to PromisedRequirement.

Parameters
kwargs (*Dict[str, Any]*) – function keyword arguments

Return type
bool

class toil.job.**UnfulfilledPromiseSentinel**(*fulfillingJobName*, *file_id*, *unpickled*)

This should be overwritten by a proper promised value.

Throws an exception when unpickled.

Parameters

- **fulfillingJobName** (*str*) –
- **file_id** (*str*) –
- **unpickled** (*Any*) –

static __setstate__(*stateDict*)

Only called when unpickling.

This won't be unpickled unless the promise wasn't resolved, so we throw an exception.

Parameters
stateDict (*Dict[str, Any]*) –

Return type
None

toil.leader

The leader script (of the leader/worker pair) for running jobs.

Module Contents

Classes

Leader	Represents the Toil leader.

Attributes

logger

toil.leader.**logger**

class toil.leader.**Leader**(*config*, *batchSystem*, *provisioner*, *jobStore*, *rootJob*, *jobCache=None*)

Represents the Toil leader.

Responsible for determining what jobs are ready to be scheduled, by consulting the job store, and issuing them in the batch system.

Parameters

- **config** (toil.common.Config) –
- **batchSystem** (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem) –
- **provisioner** (*Optional*[toil.provisioners.abstractProvisioner.AbstractProvisioner]) –
- **jobStore** (toil.jobStores.abstractJobStore.AbstractJobStore) –
- **rootJob** (toil.job.JobDescription) –
- **jobCache** (*Optional*[*Dict*[*Union*[*str*, toil.job.TemporaryID], toil.job.JobDescription]]) –

run()

Run the leader process to issue and manage jobs.

Raises
toil.exceptions.FailedJobsException if failed jobs remain after running.

Returns
The return value of the root job's run function.

Return type
Any

create_status_sentinel_file(*fail*)

Create a file in the jobstore indicating failure or success.

Parameters
fail (*bool*) –

Return type
None

innerLoop()

Process jobs.

This is the leader's main loop.

checkForDeadlocks()

Check if the system is deadlocked running service jobs.

feed_deadlock_watchdog()

Note that progress has been made and any pending deadlock checks should be reset.

Return type
None

issueJob(*jobNode*)

Add a job to the queue of jobs currently trying to run.

Parameters
jobNode (toil.job.JobDescription) –

Return type
None

issueJobs(*jobs*)

Add a list of jobs, each represented as a jobNode object.

issueServiceJob(*service_id*)

Issue a service job.

Put it on a queue if the maximum number of service jobs to be scheduled has been reached.

Parameters
service_id (*str*) –

Return type
None

issueQueingServiceJobs()

Issues any queuing service jobs up to the limit of the maximum allowed.

getNumberOfJobsIssued(*preemptible=None*)

Get number of jobs that have been added by issueJob(s) and not removed by removeJob.

Parameters
preemptible (*Optional[bool]*) – If none, return all types of jobs. If true, return just the number of preemptible jobs. If false, return just the number of non-preemptible jobs.

Return type
int

removeJob(*jobBatchSystemID*)

Remove a job from the system by batch system ID.

Returns
Job description as it was issued.

Parameters
jobBatchSystemID (*int*) –

Return type
toil.job.JobDescription

getJobs(*preemptible=None*)

Get all issued jobs.

Parameters
preemptible (*Optional[bool]*) – If specified, select only preemptible or only non-preemptible jobs.

Return type
List[*toil.job.JobDescription*]

killJobs(*jobsToKill*)

Kills the given set of jobs and then sends them for processing.

Returns the jobs that, upon processing, were reissued.

reissueOverLongJobs()

Check each issued job.

If a job is running for longer than desirable issue a kill instruction. Wait for the job to die then we pass the job to process_finished_job.

Return type
None

reissueMissingJobs(*killAfterNTimesMissing=3*)

Check all the current job ids are in the list of currently issued batch system jobs.

If a job is missing, we mark it as so, if it is missing for a number of runs of this function (say 10).. then we try deleting the job (though its probably lost), we wait then we pass the job to process_finished_job.

processRemovedJob(*issuedJob*, *result_status*)

process_finished_job(*batch_system_id*, *result_status*, *wall_time=None*, *exit_reason=None*)

Process finished jobs.

Called when an attempt to run a job finishes, either successfully or otherwise.

Takes the job out of the issued state, and then works out what to do about the fact that it succeeded or failed.

Returns
True if the job is going to run again, and False if the job is fully done or completely failed.

Return type
bool

process_finished_job_description(*finished_job*, *result_status*, *wall_time=None*, *exit_reason=None*, *batch_system_id=None*)

Process a finished JobDescription based upon its succees or failure.

If wall-clock time is available, informs the cluster scaler about the job finishing.

If the job failed and a batch system ID is available, checks for and reports batch system logs.

Checks if it succeeded and was removed, or if it failed and needs to be set up after failure, and dispatches to the appropriate function.

Returns
True if the job is going to run again, and False if the job is fully done or completely failed.

Parameters

- **finished_job** (toil.job.JobDescription) –
- **result_status** (*int*) –
- **wall_time** (*Optional[float]*) –
- **exit_reason** (*Optional[toil.batchSystems.abstractBatchSystem.BatchJobExitReason]*) –
- **batch_system_id** (*Optional[int]*) –

Return type
bool

getSuccessors(*job_id*, *alreadySeenSuccessors*)

Get successors of the given job by walking the job graph recursively.

Parameters

- **alreadySeenSuccessors** (*Set[str]*) – any successor seen here is ignored and not traversed.
- **job_id** (*str*) –

Returns
The set of found successors. This set is added to alreadySeenSuccessors.

Return type
Set[str]

processTotallyFailedJob(*job_id*)

Process a totally failed job.

Parameters
job_id (*str*) –

Return type
None

toil.realtimeLogger

Implements a real-time UDP-based logging system that user scripts can use for debugging.

Module Contents

Classes

LoggingDatagramHandler	Receive logging messages from the jobs and display them on the leader.
JSONDatagramHandler	Send logging records over UDP serialized as JSON.
RealtimeLoggerMetaclass	Metaclass for RealtimeLogger that lets add logging methods.
RealtimeLogger	Provide a logger that logs over UDP to the leader.

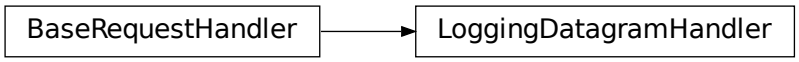
Attributes

logger	

toil.realtimeLogger.**logger**

class toil.realtimeLogger.**LoggingDatagramHandler**(*request*, *client_address*, *server*)

Bases: socketserver.BaseRequestHandler



Receive logging messages from the jobs and display them on the leader.

Uses bare JSON message encoding.

handle()

Handle a single message. SocketServer takes care of splitting out the messages.

Messages are JSON-encoded logging module records.

Return type
None

class toil.realtimeLogger.**JSONDatagramHandler**(*host*, *port*)

Bases: logging.handlers.DatagramHandler

Send logging records over UDP serialized as JSON.

They have to fit in a single UDP datagram, so don't try to log more than 64kb at once.

makePickle(*record*)

Actually, encode the record as bare JSON instead.

Parameters
record (*logging.LogRecord*) –

Return type
bytes

class toil.realtimeLogger.**RealtimeLoggerMetaclass**

Bases: type

Metaclass for RealtimeLogger that lets add logging methods.

Like RealtimeLogger.warning(), RealtimeLogger.info(), etc.

__getattr__(*name*)

Fallback to attributes on the logger.

Parameters
name (*str*) –

Return type
Any

class toil.realtimeLogger.**RealtimeLogger**(*batchSystem*, *level=defaultLevel*)

Provide a logger that logs over UDP to the leader.

To use in a Toil job, do:

```
>>> from toil.realtimeLogger import RealtimeLogger
>>> RealtimeLogger.info("This logging message goes straight to the leader")
```

That's all a user of Toil would need to do. On the leader, Job.Runner.startToil() automatically starts the UDP server by using an instance of this class as a context manager.

Parameters

- **batchSystem** (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem) –
- **level** (*str*) –

envPrefix = 'TOIL_RT_LOGGING_'

defaultLevel = 'INFO'

lock

loggingServer

serverThread

initialized = 0

logger

classmethod getLogger()

Get the logger that logs real-time to the leader.

Note that if the returned logger is used on the leader, you will see the message twice, since it still goes to the normal log handlers, too.

Return type
logging.Logger

__enter__()

Return type
None

__exit__(*exc_type*, *exc_val*, *exc_tb*)

Parameters

- **exc_type** (*Optional[Type[BaseException]]*) –
- **exc_val** (*Optional[BaseException]*) –
- **exc_tb** (*Optional[types.TracebackType]*) –

Return type
None

toil.resource

Module Contents

Classes

Resource	Represents a file or directory that will be deployed to each node before any jobs in the user script are invoked.
FileResource	A resource read from a file on the leader.
DirectoryResource	A resource read from a directory on the leader.
VirtualEnvResource	A resource read from a virtualenv on the leader.
ModuleDescriptor	A path to a Python module decomposed into a namedtuple of three elements

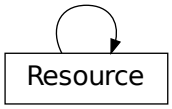
Attributes

logger

toil.resource.**logger**

class toil.resource.**Resource**

Bases: namedtuple('Resource', ('name', 'pathHash', 'url', 'contentHash'))



Represents a file or directory that will be deployed to each node before any jobs in the user script are invoked.

Each instance is a namedtuple with the following elements:

The pathHash element contains the MD5 (in hexdigest form) of the path to the resource on the leader node. The path, and therefore its hash is unique within a job store.

The url element is a "file:" or "http:" URL at which the resource can be obtained.

The contentHash element is an MD5 checksum of the resource, allowing for validation and caching of resources.

If the resource is a regular file, the type attribute will be 'file'.

If the resource is a directory, the type attribute will be 'dir' and the URL will point at a ZIP archive of that directory.

abstract property localPath: str

Get the path to resource on the worker.

The file or directory at the returned path may or may not yet exist. Invoking download() will ensure that it does.

Return type
str

property localDirPath: str

The path to the directory containing the resource on the worker.

Return type
str

resourceEnvNamePrefix = 'JTRES_'

rootDirPathEnvName

classmethod create(*jobStore*, *leaderPath*)

Saves the content of the file or directory at the given path to the given job store and returns a resource object representing that content for the purpose of obtaining it again at a generic, public URL. This method should be invoked on the leader node.

Parameters

- **jobStore** (toil.jobStores.abstractJobStore.AbstractJobStore) –
- **leaderPath** (*str*) –

Return type
Resource

refresh(*jobStore*)

Parameters
jobStore (toil.jobStores.abstractJobStore.AbstractJobStore) –

Return type
Resource

classmethod prepareSystem()

Prepares this system for the downloading and lookup of resources. This method should only be invoked on a worker node. It is idempotent but not thread-safe.

Return type
None

classmethod cleanSystem()

Remove all downloaded, localized resources.

Return type
None

register()

Register this resource for later retrieval via lookup(), possibly in a child process.

Return type
None

classmethod lookup(*leaderPath*)

Return a resource object representing a resource created from a file or directory at the given path on the leader.

This method should be invoked on the worker. The given path does not need to refer to an existing file or directory on the worker, it only identifies the resource within an instance of toil. This method returns None if no resource for the given path exists.

Parameters
leaderPath (*str*) –

Return type
Optional[*Resource*]

download(*callback=None*)

Download this resource from its URL to a file on the local system.

This method should only be invoked on a worker node after the node was setup for accessing resources via prepareSystem().

Parameters
callback (*Optional[Callable[[str], None]]*) –

Return type
None

pickle()

Return type
str

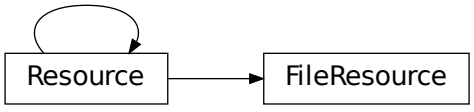
classmethod unpickle(*s*)

Parameters
s (*str*) –

Return type
Resource

class toil.resource.**FileResource**

Bases: *Resource*



A resource read from a file on the leader.

property localPath: str

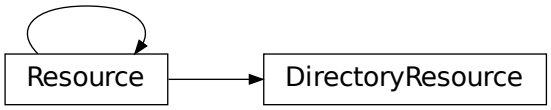
Get the path to resource on the worker.

The file or directory at the returned path may or may not yet exist. Invoking download() will ensure that it does.

Return type
str

class toil.resource.**DirectoryResource**

Bases: *Resource*



A resource read from a directory on the leader.

The URL will point to a ZIP archive of the directory. All files in that directory (and any subdirectories) will be included. The directory may be a package but it does not need to be.

property localPath: str

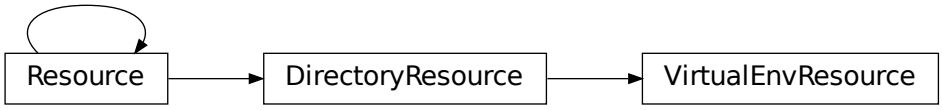
Get the path to resource on the worker.

The file or directory at the returned path may or may not yet exist. Invoking download() will ensure that it does.

Return type
str

class toil.resource.**VirtualEnvResource**

Bases: *DirectoryResource*

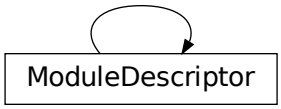


A resource read from a virtualenv on the leader.

All modules and packages found in the virtualenv's site-packages directory will be included.

class toil.resource.**ModuleDescriptor**

Bases: namedtuple('ModuleDescriptor', ('dirPath', 'name', 'fromVirtualEnv'))



A path to a Python module decomposed into a namedtuple of three elements

- dirPath, the path to the directory that should be added to sys.path before importing the module,
- moduleName, the fully qualified name of the module with leading package names separated by dot and

```
>>> import toil.resource
>>> ModuleDescriptor.forModule('toil.resource')
ModuleDescriptor(dirPath='/.../src', name='toil.resource', fromVirtualEnv=False)
```

```
>>> import subprocess, tempfile, os
>>> dirPath = tempfile.mkdtemp()
>>> path = os.path.join( dirPath, 'foo.py' )
>>> with open(path,'w') as f:
...     _ = f.write('from toil.resource import ModuleDescriptor\n'
...                 'print(ModuleDescriptor.forModule(__name__))')
>>> subprocess.check_output([ sys.executable, path ])
b"ModuleDescriptor(dirPath='...', name='foo', fromVirtualEnv=False)\n"
```

```
>>> from shutil import rmtree
>>> rmtree( dirPath )
```

Now test a collision. 'collections' is part of the standard library in Python 2 and 3. >>> dirPath = tempfile.mkdtemp() >>> path = os.path.join(dirPath, 'collections.py') >>> with open(path,'w') as f: ... _ = f.write('from toil.resource import ModuleDescriptorn' ... 'ModuleDescriptor.forModule(__name__)')

This should fail and return exit status 1 due to the collision with the built-in module: >>> subprocess.call([sys.executable, path]) 1

Clean up >>> rmtree(dirPath)

property belongsToToil: bool

True if this module is part of the Toil distribution

Return type
bool

dirPath: str

name: str

classmethod forModule(*name*)

Return an instance of this class representing the module of the given name.

If the given module name is "__main__", it will be translated to the actual file name of the top-level script without the .py or .pyc extension. This method assumes that the module with the specified name has already been loaded.

Parameters
name (*str*) –

Return type
ModuleDescriptor

saveAsResourceTo(*jobStore*)

Store the file containing this module–or even the Python package directory hierarchy containing that file–as a resource to the given job store and return the corresponding resource object. Should only be called on a leader node.

Parameters
jobStore (*toil.jobStores.abstractJobStore.AbstractJobStore*) –

Return type
Resource

localize()

Check if this module was saved as a resource.

If it was, return a new module descriptor that points to a local copy of that resource. Should only be called on a worker node. On the leader, this method returns this resource, i.e. self.

Return type
ModuleDescriptor

globalize()

Reverse the effect of localize().

Return type
ModuleDescriptor

toCommand()

Return type
Sequence[str]

classmethod fromCommand(*command*)

Parameters
command (*Sequence[str]*) –

Return type
ModuleDescriptor

makeLoadable()

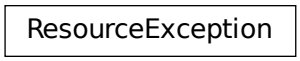
Return type
ModuleDescriptor

load()

Return type
Optional[types.ModuleType]

exception toil.resource.**ResourceException**

Bases: Exception



Common base class for all non-exit exceptions.

toil.serviceManager

Module Contents

Classes

ServiceManager	Manages the scheduling of services.

Attributes

logger

toil.serviceManager.**logger**

class toil.serviceManager.**ServiceManager**(*job_store*, *toil_state*)

Manages the scheduling of services.

Parameters

- **job_store** (toil.jobStores.abstractJobStore.AbstractJobStore) –
- **toil_state** (toil.toilState.ToilState) –

services_are_starting(*job_id*)

Check if services are being started.

Returns
True if the services for the given job are currently being started, and False otherwise.

Parameters
job_id (*str*) –

Return type
bool

get_job_count()

Get the total number of jobs we are working on.

(services and their parent non-service jobs)

Return type
int

start()

Start the service scheduling thread.

Return type
None

put_client(*client_id*)

Schedule the services of a job asynchronously.

When the job's services are running the ID for the job will be returned by toil.leader.ServiceManager.get_ready_client.

Parameters
client_id (*str*) – ID of job with services to schedule.

Return type
None

get_ready_client(*maxWait*)

Fetch a ready client, waiting as needed.

Parameters
maxWait (*float*) – Time in seconds to wait to get a JobDescription before returning

Returns
the ID of a client whose services are running, or None if no such job is available.

Return type
Optional[str]

get_unservable_client(*maxWait*)

Fetch a client whos services failed to start.

Parameters
maxWait (*float*) – Time in seconds to wait to get a JobDescription before returning

Returns
the ID of a client whose services failed to start, or None if no such job is available.

Return type
Optional[str]

get_startable_service(*maxWait*)

Fetch a service job that is ready to start.

Parameters
maxWait (*float*) – Time in seconds to wait to get a job before returning.

Returns
the ID of a service job that the leader can start, or None if no such job exists.

Return type
Optional[str]

kill_services(*service_ids*, *error=False*)

Stop all the given service jobs.

Parameters

- **services** – Service jobStoreIDs to kill
- **error** (*bool*) – Whether to signal that the service failed with an error when stopping it.
- **service_ids** (*Iterable[str]*) –

Return type
None

`is_active`(*service_id*)

Return true if the service job has not been told to terminate.

Parameters
service_id (*str*) – Service to check on

Return type
bool

`is_running`(*service_id*)

Return true if the service job has started and is active.

Parameters

- **service** – Service to check on
- **service_id** (*str*) –

Return type
bool

`check`()

Check on the service manager thread.

Raises
RuntimeError – If the underlying thread has quit.

Return type
None

`shutdown`()

Terminate worker threads cleanly; starting and killing all service threads.

Will block until all services are started and blocked.

Return type
None

toil.statsAndLogging

Module Contents

Classes

StatsAndLogging	A thread to aggregate statistics and logging.

Functions

set_log_level(level[, set_logger])	Sets the root logger level to a given string level (like "INFO").
add_logging_options(parser)	Add logging options to set the global log level.
configure_root_logger()	Set up the root logger with handlers and formatting.
log_to_file(log_file, log_rotation)	
set_logging_from_options(options)	
suppress_exotic_logging(local_logger)	Attempts to suppress the loggers of all non-Toil packages by setting them to CRITICAL.

Attributes

logger
root_logger
toil_logger
DEFAULT_LOGLEVEL

toil.statsAndLogging.**logger**

toil.statsAndLogging.**root_logger**

toil.statsAndLogging.**toil_logger**

toil.statsAndLogging.**DEFAULT_LOGLEVEL**

class toil.statsAndLogging.**StatsAndLogging**(*jobStore*, *config*)

A thread to aggregate statistics and logging.

Parameters

- **jobStore** (toil.jobStores.abstractJobStore.AbstractJobStore) –
- **config** (toil.common.Config) –

start()

Start the stats and logging thread.

Return type
None

classmethod formatLogStream(*stream*, *job_name=None*)

Given a stream of text or bytes, and the job name, job itself, or some other optional stringifyable identity info for the job, return a big text string with the formatted job log, suitable for printing for the user.

We don't want to prefix every line of the job's log with our own logging info, or we get prefixes wider than any reasonable terminal and longer than the messages.

Parameters

- **stream** (*Union[IO[str], IO[bytes]]*) – The stream of text or bytes to print for the user.
- **job_name** (*Optional[str]*) –

Return type
str

classmethod logWithFormatting(*jobStoreID*, *jobLogs*, *method=logger.debug*, *message=None*)

Parameters

- **jobStoreID** (*str*) –
- **jobLogs** (*Union[IO[str], IO[bytes]]*) –
- **method** (*Callable[[str], None]*) –
- **message** (*Optional[str]*) –

Return type
None

classmethod writeLogFiles(*jobNames*, *jobLogList*, *config*, *failed=False*)

Parameters

- **jobNames** (*List[str]*) –
- **jobLogList** (*List[str]*) –
- **config** (toil.common.Config) –
- **failed** (*bool*) –

Return type
None

classmethod statsAndLoggingAggregator(*jobStore*, *stop*, *config*)

The following function is used for collating stats/reporting log messages from the workers. Works inside of a thread, collates as long as the stop flag is not True.

Parameters

- **jobStore** (toil.jobStores.abstractJobStore.AbstractJobStore) –
- **stop** (*threading.Event*) –
- **config** (toil.common.Config) –

Return type
None

check()

Check on the stats and logging aggregator. :raise RuntimeError: If the underlying thread has quit.

Return type
None

shutdown()

Finish up the stats/logging aggregation thread.

Return type
None

toil.statsAndLogging.**set_log_level**(*level*, *set_logger=None*)

Sets the root logger level to a given string level (like "INFO").

Parameters

- **level** (*str*) –
- **set_logger** (*Optional[logging.Logger]*) –

Return type
None

toil.statsAndLogging.**add_logging_options**(*parser*)

Add logging options to set the global log level.

Parameters
parser (*argparse.ArgumentParser*) –

Return type
None

toil.statsAndLogging.**configure_root_logger**()

Set up the root logger with handlers and formatting.

Should be called before any entry point tries to log anything, to ensure consistent formatting.

Return type
None

toil.statsAndLogging.**log_to_file**(*log_file*, *log_rotation*)

Parameters

- **log_file** (*Optional[str]*) –
- **log_rotation** (*bool*) –

Return type
None

toil.statsAndLogging.**set_logging_from_options**(*options*)

Parameters
options (*Union[toil.common.Config, argparse.Namespace]*) –

Return type
None

toil.statsAndLogging.**suppress_exotic_logging**(*local_logger*)

Attempts to suppress the loggers of all non-Toil packages by setting them to CRITICAL.

For example: 'requests_oauthlib', 'google', 'boto', 'websocket', 'oauthlib', etc.

This will only suppress loggers that have already been instantiated and can be seen in the environment, except for the list declared in "always_suppress".

This is important because some packages, particularly boto3, are not always instantiated yet in the environment when this is run, and so we create the logger and set the level preemptively.

Parameters
local_logger (*str*) –

Return type
None

toil.toilState

Module Contents

Classes

ToilState	Holds the leader's scheduling information.

Attributes

logger

toil.toilState.**logger**

class toil.toilState.**ToilState**(*jobStore*)

Holds the leader's scheduling information.

But onlt that which does not need to be persisted back to the JobStore (such as information on completed and outstanding predecessors)

Holds the true single copies of all JobDescription objects that the Leader and ServiceManager will use. The leader and service manager shouldn't do their own load() and update() calls on the JobStore; they should go through this class.

Everything in the leader should reference JobDescriptions by ID.

Only holds JobDescription objects, not Job objects, and those JobDescription objects only exist in single copies.

Parameters
jobStore (toil.jobStores.abstractJobStore.AbstractJobStore) –

load_workflow(*rootJob*, *jobCache=None*)

Load the workflow rooted at the given job.

If jobs are loaded that have updated and need to be dealt with by the leader, JobUpdatedMessage messages will be sent to the message bus.

The jobCache is a map from jobStoreID to JobDescription or None. Is used to speed up the building of the state when loading initially from the JobStore, and is not preserved.

Parameters

- **rootJob** (toil.job.JobDescription) – The description for the root job of the workflow being run.
- **jobCache** (*Optional[Dict[str, toil.job.JobDescription]]*) – A dict to cache downloaded job descriptions in, keyed by ID.

Return type
None

job_exists(*job_id*)

Test if the givin job exists now.

Returns True if the given job exists right now, and false if it hasn't been created or it has been deleted elsewhere.

Doesn't guarantee that the job will or will not be gettable, if racing another process, or if it is still cached.

Parameters
job_id (*str*) –

Return type
bool

get_job(*job_id*)

Get the one true copy of the JobDescription with the given ID.

Parameters
job_id (*str*) –

Return type
toil.job.JobDescription

commit_job(*job_id*)

Save back any modifications made to a JobDescription.

(one retrieved from get_job())

Parameters
job_id (*str*) –

Return type
None

delete_job(*job_id*)

Destroy a JobDescription.

May raise an exception if the job could not be cleaned up (i.e. files belonging to it failed to delete).

Parameters
job_id (*str*) –

Return type
None

reset_job(*job_id*)

Discard any local modifications to a JobDescription.

Will make modifications from other hosts visible.

Parameters
job_id (*str*) –

Return type
None

successors_pending(*predecessor_id*, *count*)

Remember that the given job has the given number more pending successors.

(that have not yet succeeded or failed.)

Parameters

- **predecessor_id** (*str*) –
- **count** (*int*) –

Return type
None

successor_returned(*predecessor_id*)

Remember that the given job has one fewer pending successors.

(because one has succeeded or failed.)

Parameters
predecessor_id (*str*) –

Return type
None

count_pending_successors(*predecessor_id*)

Count number of pending successors of the given job.

Pending successors are those which have not yet succeeded or failed.

Parameters
predecessor_id (*str*) –

Return type
int

toil.version

Module Contents

toil.version.**baseVersion** = '5.11.0'

toil.version.**cgcloudVersion** = '1.6.0a1.dev393'

toil.version.**version** = '5.11.0-9a04dabb36d6ab13ed1ac7c711dbdc8c71724dc9'

toil.version.**distVersion** = '5.11.0'

toil.version.**exactPython** = 'python3.7'

toil.version.**python** = 'python3.7'

toil.version.**dockerTag** = '5.11.0-9a04dabb36d6ab13ed1ac7c711dbdc8c71724dc9-py3.7'

toil.version.**currentCommit** = '9a04dabb36d6ab13ed1ac7c711dbdc8c71724dc9'

toil.version.**dockerRegistry** = 'quay.io/ucsc_cgl'

toil.version.**dockerName** = 'toil'

toil.version.**dirty** = False

toil.version.**cwltool_version** = '3.1.20230425144158'

toil.worker

Module Contents

Classes

StatsDict	Subclass of MagicExpando for type-checking purposes.

Functions

nextChainable(predecessor, jobStore, config)	Returns the next chainable job's JobDescription after the given predecessor
workerScript(jobStore, config, jobName, jobStoreID[, ...])	Worker process script, runs a job.
parse_args(args)	Parse command-line arguments to the worker.
in_contexts(contexts)	Unpickle and enter all the pickled, base64-encoded context managers in the
main([argv])	

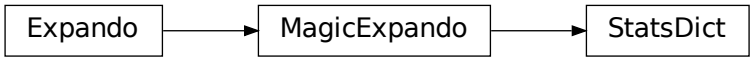
Attributes

logger

toil.worker.**logger**

class toil.worker.**StatsDict**(**args*, ***kwargs*)

Bases: *toil.lib.expando.MagicExpando*



Subclass of MagicExpando for type-checking purposes.

jobs: List[str]

toil.worker.**nextChainable**(*predecessor*, *jobStore*, *config*)

Returns the next chainable job's JobDescription after the given predecessor JobDescription, if one exists, or None if the chain must terminate.

Parameters

- **predecessor** (*toil.job.JobDescription*) – The job to chain from

- **jobStore** (toil.jobStores.abstractJobStore.AbstractJobStore) – The JobStore to fetch JobDescriptions from.
- **config** (toil.common.Config) – The configuration for the current run.

Return type
Optional[*toil.job.JobDescription*]

toil.worker.**workerScript**(*jobStore*, *config*, *jobName*, *jobStoreID*, *redirectOutputToLogFile=True*)

Worker process script, runs a job.

Parameters

- **jobStore** (toil.jobStores.abstractJobStore.AbstractJobStore) – The JobStore to fetch JobDescriptions from.
- **config** (toil.common.Config) – The configuration for the current run.
- **jobName** (*str*) – The "job name" (a user friendly name) of the job to be run
- **jobStoreID** (*str*) – The job store ID of the job to be run
- **redirectOutputToLogFile** (*bool*) –

Return int
1 if a job failed, or 0 if all jobs succeeded

Return type
int

toil.worker.**parse_args**(*args*)

Parse command-line arguments to the worker.

Parameters
args (*List[str]*) –

Return type
argparse.Namespace

toil.worker.**in_contexts**(*contexts*)

Unpickle and enter all the pickled, base64-encoded context managers in the given list. Then do the body, then leave them all.

Parameters
contexts (*List[str]*) –

Return type
Iterator[None]

toil.worker.**main**(*argv=None*)

Parameters
argv (*Optional[List[str]]*) –

Return type
None

30.1.3 Package Contents

Functions

retry([intervals, infinite_retries, errors, ...])	Retry a function if it fails with any Exception defined in "errors".
which(cmd[, mode, path])	Return the path with conforms to the given mode on the Path.
toilPackageDirPath()	Return the absolute path of the directory that corresponds to the top-level toil package.
inVirtualEnv()	Test if we are inside a virtualenv or Conda virtual environment.
resolveEntryPoint(entryPoint)	Find the path to the given entry point that *should* work on a worker.
physicalMemory()	Calculate the total amount of physical memory, in bytes.
physicalDisk(directory)	
applianceSelf([forceDockerAppliance])	Return the fully qualified name of the Docker image to start Toil appliance containers from.
customDockerInitCmd()	Return the custom command set by the `TOIL_CUSTOM_DOCKER_INIT_COMMAND` environment variable.
customInitCmd()	Return the custom command set by the `TOIL_CUSTOM_INIT_COMMAND` environment variable.
lookupEnvVar(name, envName, defaultValue)	Look up environment variables that control Toil and log the result.
checkDockerImageExists(appliance)	Attempt to check a url registryName for the existence of a docker image with a given tag.
parseDockerAppliance(appliance)	Derive parsed registry, image reference, and tag from a docker image string.
checkDockerSchema(appliance)	
requestCheckRegularDocker(origAppliance, registryName, ...)	Check if an image exists using the requests library.
requestCheckDockerIo(origAppliance, imageName, tag)	Check docker.io to see if an image exists using the requests library.
logProcessContext(config)	

Attributes

memoize	Memoize a function result based on its parameters using this decorator.
currentCommit	
log	
KNOWN_EXTANT_IMAGES	
cache_path	

toil.**memoize**

Memoize a function result based on its parameters using this decorator.

For example, this can be used in place of lazy initialization. If the decorating function is invoked by multiple threads, the decorated function may be called more than once with the same arguments.

toil.**retry**(*intervals=None*, *infinite_retries=False*, *errors=None*, *log_message=None*, *prepare=None*)

Retry a function if it fails with any Exception defined in "errors".

Does so every x seconds, where x is defined by a list of numbers (ints or floats) in "intervals". Also accepts ErrorCondition events for more detailed retry attempts.

Parameters

- **intervals** (*Optional[List]*) – A list of times in seconds we keep retrying until returning failure. Defaults to retrying with the following exponential back-off before failing: 1s, 1s, 2s, 4s, 8s, 16s
- **infinite_retries** (*bool*) – If this is True, reset the intervals when they run out. Defaults to: False.
- **errors** (*Optional[Sequence[Union[ErrorCondition, Type[Exception]]]]*) – A list of exceptions OR ErrorCondition objects to catch and retry on. ErrorCondition objects describe more detailed error event conditions than a plain error. An ErrorCondition specifies: - Exception (required) - Error codes that must match to be retried (optional; defaults to not checking) - A string that must be in the error message to be retried (optional; defaults to not checking) - A bool that can be set to False to always error on this condition.

  If not specified, this will default to a generic Exception.
- **log_message** (*Optional[Tuple[Callable, str]]*) – Optional tuple of ("log/print function()", "message string") that will precede each attempt.
- **prepare** (*Optional[List[Callable]]*) – Optional list of functions to call, with the function's arguments, between retries, to reset state.

Returns

The result of the wrapped function or raise.

Return type

Callable[[Any], Any]

toil.**currentCommit = '9a04dabb36d6ab13ed1ac7c711dbdc8c71724dc9'**

toil.**log**

toil.**which**(*cmd*, *mode=os.F_OK | os.X_OK*, *path=None*)

Return the path with conforms to the given mode on the Path.

[Copy-pasted in from python3.6's shutil.which().]

mode defaults to os.F_OK | os.X_OK. *path* defaults to the result of os.environ.get("PATH"), or can be overridden with a custom search path.

Returns
The path found, or None.

Return type
Optional[str]

toil.**toilPackageDirPath**()

Return the absolute path of the directory that corresponds to the top-level toil package.

The return value is guaranteed to end in '/toil'.

Return type
str

toil.**inVirtualEnv**()

Test if we are inside a virtualenv or Conda virtual environment.

Return type
bool

toil.**resolveEntryPoint**(*entryPoint*)

Find the path to the given entry point that *should* work on a worker.

Returns
The path found, which may be an absolute or a relative path.

Parameters
entryPoint (*str*) –

Return type
str

toil.**physicalMemory**()

Calculate the total amount of physical memory, in bytes.

```
>>> n = physicalMemory()
>>> n > 0
True
>>> n == physicalMemory()
True
```

Return type
int

toil.**physicalDisk**(*directory*)

Parameters
directory (*str*) –

Return type
int

toil.**applianceSelf**(*forceDockerAppliance=False*)

Return the fully qualified name of the Docker image to start Toil appliance containers from.

The result is determined by the current version of Toil and three environment variables: TOIL_DOCKER_REGISTRY, TOIL_DOCKER_NAME and TOIL_APPLIANCE_SELF.

TOIL_DOCKER_REGISTRY specifies an account on a publicly hosted docker registry like Quay or Docker Hub. The default is UCSC's CGL account on Quay.io where the Toil team publishes the official appliance images. TOIL_DOCKER_NAME specifies the base name of the image. The default of *toil* will be adequate in most cases. TOIL_APPLIANCE_SELF fully qualifies the appliance image, complete with registry, image name and version tag, overriding both TOIL_DOCKER_NAME and *TOIL_DOCKER_REGISTRY`* as well as the version tag of the image. Setting TOIL_APPLIANCE_SELF will not be necessary in most cases.

Parameters
forceDockerAppliance (*bool*) –

Return type
str

toil.**customDockerInitCmd**()

Return the custom command set by the TOIL_CUSTOM_DOCKER_INIT_COMMAND environment variable.

The custom docker command is run prior to running the workers and/or the primary node's services.

This can be useful for doing any custom initialization on instances (e.g. authenticating to private docker registries). Any single quotes are escaped and the command cannot contain a set of blacklisted chars (newline or tab).

Returns
The custom commmand, or an empty string is returned if the environment variable is not set.

Return type
str

toil.**customInitCmd**()

Return the custom command set by the TOIL_CUSTOM_INIT_COMMAND environment variable.

The custom init command is run prior to running Toil appliance itself in workers and/or the primary node (i.e. this is run one stage before TOIL_CUSTOM_DOCKER_INIT_COMMAND).

This can be useful for doing any custom initialization on instances (e.g. authenticating to private docker registries). Any single quotes are escaped and the command cannot contain a set of blacklisted chars (newline or tab).

returns: the custom command or n empty string is returned if the environment variable is not set.

Return type
str

toil.**lookupEnvVar**(*name*, *envName*, *defaultValue*)

Look up environment variables that control Toil and log the result.

Parameters

- **name** (*str*) – the human readable name of the variable
- **envName** (*str*) – the name of the environment variable to lookup
- **defaultValue** (*str*) – the fall-back value

Returns
the value of the environment variable or the default value the variable is not set

Return type
str

toil.**checkDockerImageExists**(*appliance*)

Attempt to check a url registryName for the existence of a docker image with a given tag.

Parameters
appliance (*str*) – The url of a docker image's registry (with a tag) of the form: 'quay.io/<repo_path>:<tag>' or '<repo_path>:<tag>'. Examples: 'quay.io/ucsc_cgl/toil:latest', 'ubuntu:latest', or 'broadinstitute/genomes-in-the-cloud:2.0.0'.

Returns
Raises an exception if the docker image cannot be found or is invalid. Otherwise, it will return the appliance string.

Return type
str

toil.**parseDockerAppliance**(*appliance*)

Derive parsed registry, image reference, and tag from a docker image string.

Example: "quay.io/ucsc_cgl/toil:latest" Should return: "quay.io", "ucsc_cgl/toil", "latest"

If a registry is not defined, the default is: "docker.io" If a tag is not defined, the default is: "latest"

Parameters
appliance (*str*) – The full url of the docker image originally specified by the user (or the default). e.g. "quay.io/ucsc_cgl/toil:latest"

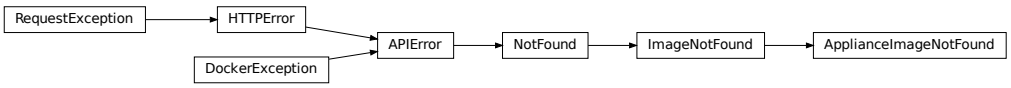
Returns
registryName, imageName, tag

Return type
Tuple[str, str, str]

toil.**checkDockerSchema**(*appliance*)

exception toil.**ApplianceImageNotFound**(*origAppliance*, *url*, *statusCode*)

Bases: docker.errors.ImageNotFound



Error raised when using TOIL_APPLIANCE_SELF results in an HTTP error.

Parameters

- **origAppliance** (*str*) – The full url of the docker image originally specified by the user (or the default). e.g. "quay.io/ucsc_cgl/toil:latest"
- **url** (*str*) – The URL at which the image's manifest is supposed to appear
- **statusCode** (*int*) – the failing HTTP status code returned by the URL

toil.**KNOWN_EXTANT_IMAGES**

toil.**requestCheckRegularDocker**(*origAppliance*, *registryName*, *imageName*, *tag*)

Check if an image exists using the requests library.

URL is based on the docker v2 schema.

This has the following format: `https://{websitehostname}.io/v2/{repo}/manifests/{tag}`

Does not work with the official (docker.io) site, because they require an OAuth token, so a separate check is done for docker.io images.

Parameters

- **origAppliance** (*str*) – The full url of the docker image originally specified by the user (or the default).

  e.g. `quay.io/ucsc_cgl/toil:latest`

- **registryName** (*str*) – The url of a docker image's registry. e.g. `quay.io`
- **imageName** (*str*) – The image, including path and excluding the tag. e.g. `ucsc_cgl/toil`
- **tag** (*str*) – The tag used at that docker image's registry. e.g. `latest`

Raises
ApplianceImageNotFound if no match is found.

Returns
Return True if match found.

Return type
bool

toil.**requestCheckDockerIo**(*origAppliance*, *imageName*, *tag*)

Check docker.io to see if an image exists using the requests library.

URL is based on the docker v2 schema. Requires that an access token be fetched first.

Parameters

- **origAppliance** (*str*) – The full url of the docker image originally specified by the user (or the default). e.g. "ubuntu:latest"
- **imageName** (*str*) – The image, including path and excluding the tag. e.g. "ubuntu"
- **tag** (*str*) – The tag used at that docker image's registry. e.g. "latest"

Raises
ApplianceImageNotFound if no match is found.

Returns
Return True if match found.

Return type
bool

toil.**logProcessContext**(*config*)

Parameters
config (common.Config) –

Return type
None

toil.**cache_path** = '~/.cache/aws/cached_temporary_credentials'

30.2 tutorial_docker

30.2.1 Module Contents

tutorial_docker.**align**

tutorial_docker.**jobstore: str**

30.3 tutorial_managing2

30.3.1 Module Contents

Functions

globalFileStoreJobFn(job)	

Attributes

jobstore	

tutorial_managing2.**globalFileStoreJobFn**(*job*)

tutorial_managing2.**jobstore: str**

30.4 tutorial_helloworld

30.4.1 Module Contents

Functions

helloWorld(message[, memory, cores, disk])	

Attributes

parser	

tutorial_helloworld.**helloWorld**(*message*, *memory='1G'*, *cores=1*, *disk='1G'*)

tutorial_helloworld.**parser**

30.5 tutorial_discoverfiles

30.5.1 Module Contents

Classes

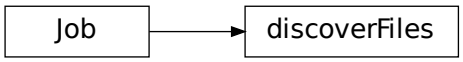
discoverFiles	Views files at a specified path using ls.

Functions

main()

class tutorial_discoverfiles.**discoverFiles**(*path*, **args*, ***kwargs*)

Bases: *toil.job.Job*



Views files at a specified path using ls.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

tutorial_discoverfiles.**main**()

30.6 tutorial_multiplejobs2

30.6.1 Module Contents

Functions

helloWorld(job, message[, memory, cores, disk])

Attributes

parser

tutorial_multiplejobs2.**helloWorld**(*job*, *message*, *memory='2G'*, *cores=2*, *disk='3G'*)

tutorial_multiplejobs2.**parser**

30.7 tutorial_dynamic

30.7.1 Module Contents

Functions

binaryStringFn(job, depth[, message])

Attributes

jobstore

tutorial_dynamic.**binaryStringFn**(*job*, *depth*, *message=''*)

tutorial_dynamic.**jobstore**: str

30.8 tutorial_invokeworkflow2

30.8.1 Module Contents

Classes

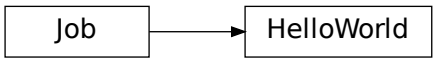
HelloWorld	Class represents a unit of work in toil.

Attributes

jobstore

class tutorial_invokeworkflow2.**HelloWorld**(*message*)

Bases: *toil.job.Job*



Class represents a unit of work in toil.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

tutorial_invokeworkflow2.**jobstore**: str

30.9 tutorial_jobfunctions

30.9.1 Module Contents

Functions

helloWorld(job, message)

Attributes

jobstore

tutorial_jobfunctions.**helloWorld**(*job*, *message*)

tutorial_jobfunctions.**jobstore**: str

30.10 tutorial_managing

30.10.1 Module Contents

Classes

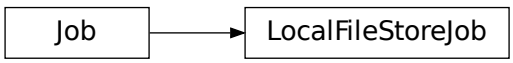
LocalFileStoreJob	Class represents a unit of work in toil.

Attributes

jobstore

class tutorial_managing.**LocalFileStoreJob**(*memory=None*, *cores=None*, *disk=None*, *accelerators=None*, *preemptible=None*, *preemptable=None*, *unitName=''*, *checkpoint=False*, *displayName=''*, *descriptionClass=None*, *local=None*)

Bases: *toil.job.Job*



Class represents a unit of work in toil.

Parameters

- **memory** (*Optional[ParseableIndivisibleResource]*) –
- **cores** (*Optional[ParseableDivisibleResource]*) –
- **disk** (*Optional[ParseableIndivisibleResource]*) –
- **accelerators** (*Optional[ParseableAcceleratorRequirement]*) –
- **preemptible** (*Optional[ParseableFlag]*) –
- **preemptable** (*Optional[ParseableFlag]*) –
- **unitName** (*Optional[str]*) –
- **checkpoint** (*Optional[bool]*) –
- **displayName** (*Optional[str]*) –
- **descriptionClass** (*Optional[type]*) –
- **local** (*Optional[bool]*) –

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

tutorial_managing.**jobstore: str**

30.11 example_alwaysfail

30.11.1 Module Contents

Functions

main()	This workflow always fails.
explode(job)	

example_alwaysfail.**main**()

This workflow always fails.

Invoke like:

python examples/example_alwaysfail.py ./jobstore

Then you can inspect the job store with tools like *toil status*:

toil status –printLogs ./jobstore

example_alwaysfail.**explode**(*job*)

30.12 example_cachingbenchmark

This workflow collects statistics about caching.

Invoke like:

python examples/example_cachingbenchmark.py ./jobstore –realTimeLogging –logInfo

python examples/example_cachingbenchmark.py ./jobstore –realTimeLogging –logInfo –disableCaching

python examples/example_cachingbenchmark.py aws:us-west-2:cachingjobstore –realTimeLogging –logInfo

python examples/example_cachingbenchmark.py aws:us-west-2:cachingjobstore –realTimeLogging –logInfo –disableCaching

30.12.1 Module Contents

Functions

main()	
root(job, options)	
poll(job, options, file_id, number[, cores, disk, memory])	
report(job, views)	

example_cachingbenchmark.**main**()

example_cachingbenchmark.**root**(*job*, *options*)

example_cachingbenchmark.**poll**(*job*, *options*, *file_id*, *number*, *cores=0.1*, *disk='200M'*, *memory='512M'*)

example_cachingbenchmark.**report**(*job*, *views*)

30.13 tutorial_quickstart

30.13.1 Module Contents

Functions

helloWorld(message[, memory, cores, disk])	

Attributes

jobstore	

tutorial_quickstart.**helloWorld**(*message*, *memory='2G'*, *cores=2*, *disk='3G'*)

tutorial_quickstart.**jobstore**: str

30.14 tutorial_encapsulation2

30.14.1 Module Contents

tutorial_encapsulation2.**A**

30.15 tutorial_multiplejobs3

30.15.1 Module Contents

Functions

helloWorld(job, message[, memory, cores, disk])

Attributes

parser

tutorial_multiplejobs3.**helloWorld**(*job*, *message*, *memory='2G'*, *cores=2*, *disk='3G'*)

tutorial_multiplejobs3.**parser**

30.16 tutorial_cwlexample

30.16.1 Module Contents

Functions

initialize_jobs(job)
runQC(job, cwl_file, cwl_filename, yml_file, ...)

Attributes

jobstore	

tutorial_cwlexample.**initialize_jobs**(*job*)

tutorial_cwlexample.**runQC**(*job*, *cwl_file*, *cwl_filename*, *yml_file*, *yml_filename*, *outputs_dir*, *output_num*)

tutorial_cwlexample.**jobstore**: str

30.17 tutorial_encapsulation

30.17.1 Module Contents

tutorial_encapsulation.**A**

30.18 tutorial_invokeworkflow

30.18.1 Module Contents

Classes

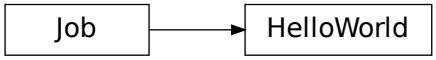
HelloWorld	Class represents a unit of work in toil.

Attributes

jobstore	

class tutorial_invokeworkflow.**HelloWorld**(*message*)

Bases: *toil.job.Job*



Class represents a unit of work in toil.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

tutorial_invokeworkflow.**jobstore**: str

30.19 tutorial_requirements

30.19.1 Module Contents

Functions

parentJob(job)	
stageFn(job, url[, cores])	
analysisJob(job, fileStoreID[, cores])	

Attributes

jobstore	

tutorial_requirements.**parentJob**(*job*)

tutorial_requirements.**stageFn**(*job*, *url*, *cores=1*)

tutorial_requirements.**analysisJob**(*job*, *fileStoreID*, *cores=2*)

tutorial_requirements.**jobstore**: str

30.20 tutorial_staging

30.20.1 Module Contents

Classes

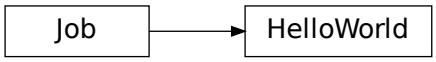
HelloWorld	Class represents a unit of work in toil.

Attributes

jobstore

class tutorial_staging.**HelloWorld**(*id*)

Bases: *toil.job.Job*



Class represents a unit of work in toil.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

tutorial_staging.**jobstore**: str

30.21 tutorial_promises

30.21.1 Module Contents

Functions

fn(job, i)

Attributes

jobstore

tutorial_promises.**fn**(*job*, *i*)

tutorial_promises.**jobstore**: str

30.22 tutorial_services

30.22.1 Module Contents

Classes

DemoService	Abstract class used to define the interface to a service.

Functions

dbFn(loginCredentials)

Attributes

j

s

loginCredentialsPromise

jobstore

class tutorial_services.**DemoService**(*memory=None*, *cores=None*, *disk=None*, *accelerators=None*, *preemptible=None*, *unitName=None*)

Bases: *toil.job.Job.Service*

Abstract class used to define the interface to a service.

Should be subclassed by the user to define services.

Is not executed as a job; runs within a ServiceHostJob.

start(*fileStore*)

Start the service.

Parameters

job – The underlying host job that the service is being run in. Can be used to register deferred functions, or to access the fileStore for creating temporary files.

Returns
An object describing how to access the service. The object must be pickleable and will be used by jobs to access the service (see *toil.job.Job.addService()*).

check()

Checks the service is still running.

Raises
exceptions.RuntimeError – If the service failed, this will cause the service job to be labeled failed.

Returns
True if the service is still running, else False. If False then the service job will be terminated, and considered a success. Important point: if the service job exits due to a failure, it should raise a RuntimeError, not return False!

stop(*fileStore*)

Stops the service. Function can block until complete.

Parameters
job – The underlying host job that the service is being run in. Can be used to register deferred functions, or to access the fileStore for creating temporary files.

tutorial_services.**j**

tutorial_services.**s**

tutorial_services.**loginCredentialsPromise**

tutorial_services.**dbFn**(*loginCredentials*)

tutorial_services.**jobstore: str**

30.23 tutorial_promises2

30.23.1 Module Contents

Functions

binaryStrings(job, depth[, message])	
merge(strings)	

Attributes

jobstore

tutorial_promises2.**binaryStrings**(*job*, *depth*, *message=''*)

tutorial_promises2.**merge**(*strings*)

tutorial_promises2.**jobstore:** str

30.24 tutorial_multiplejobs

30.24.1 Module Contents

Functions

helloWorld(job, message[, memory, cores, disk])	

Attributes

parser	

tutorial_multiplejobs.**helloWorld**(*job*, *message*, *memory='2G'*, *cores=2*, *disk='3G'*)

tutorial_multiplejobs.**parser**

30.25 tutorial_arguments

30.25.1 Module Contents

Classes

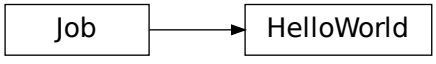
HelloWorld	Class represents a unit of work in toil.

Attributes

parser	

class tutorial_arguments.**HelloWorld**(*message*)

Bases: *toil.job.Job*



Class represents a unit of work in toil.

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters

fileStore – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns

The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

tutorial_arguments.**parser**

30.26 mkFile

30.26.1 Module Contents

Functions

main()	

mkFile.**main**()

30.27 debugWorkflow

30.27.1 Module Contents

Functions

initialize_jobs(job)	Stub function used to start a toil workflow since toil workflows can only
writeA(job, mkFile)	Runs a program, and writes a string 'A' into A.txt using mkFile.py.
writeB(job, mkFile, B_file)	Runs a program, extracts a string 'B' from an existing file, B_file.txt, and
writeC(job)	Creates/writes a file, C.txt, containing the string 'C'.
writeABC(job, A_dict, B_dict, C_dict, filepath)	Takes 3 files (specified as dictionaries) and writes their contents to ABC.txt.
finalize_jobs(job, num)	Does nothing but should be recorded in stats, status, and printDot().
broken_job(job, num)	A job that will always fail. To be used for a tutorial.

Attributes

logger	This workflow's purpose is to create files and jobs for viewing using stats,
jobStorePath	

debugWorkflow.**logger**

This workflow's purpose is to create files and jobs for viewing using stats, status, and printDot() in toilDebugTest.py. It's intended for future use in a debugging tutorial containing a broken job. It is also a minor integration test.

debugWorkflow.**initialize_jobs**(*job*)

Stub function used to start a toil workflow since toil workflows can only start with one job (but afterwards can run many in parallel).

debugWorkflow.**writeA**(*job*, *mkFile*)

Runs a program, and writes a string 'A' into A.txt using mkFile.py.

debugWorkflow.**writeB**(*job*, *mkFile*, *B_file*)

Runs a program, extracts a string 'B' from an existing file, B_file.txt, and writes it into B.txt using mkFile.py.

debugWorkflow.**writeC**(*job*)

Creates/writes a file, C.txt, containing the string 'C'.

debugWorkflow.**writeABC**(*job*, *A_dict*, *B_dict*, *C_dict*, *filepath*)

Takes 3 files (specified as dictionaries) and writes their contents to ABC.txt.

debugWorkflow.**finalize_jobs**(*job*, *num*)

Does nothing but should be recorded in stats, status, and printDot().

debugWorkflow.**broken_job**(*job*, *num*)

A job that will always fail. To be used for a tutorial.

debugWorkflow.**jobStorePath**

30.28 fake_mpi_run

30.28.1 Module Contents

Classes

Runner	

Functions

make_parser()

Attributes

args

fake_mpi_run.**make_parser**()

class fake_mpi_run.**Runner**

run_once(*args*)

Parameters
args (*List[str]*) –

run_many(*n*, *args*)

Parameters

- **n** (*int*) –
- **args** (*List[str]*) –

fake_mpi_run.**args**

- genindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z