Toil Documentation

Release 3.1.7a1

UCSC Computational Genomics Lab

June 06, 2016

Contents

Contents:

Toil

1.1 Features

Toil is a workflow engine entirely written in Python. It features:

- Easy installation, e.g. `pip install toil`.
- A small API

  Easily mastered, the user API is built upon one core class.
- Cross platform support

  Develop and test on your laptop then deploy on any of the following:

  - Commercial clouds + Amazon Web Services (including the spot market) + Microsoft Azure
  - Private clouds + OpenStack
  - High Performance Computing Environments + GridEngine + Apache Mesos + Parasol + Individual multi-core machines
- Complete file and stream management:

  Temporary and persistent file management that abstracts the details of the underlying file system, providing a uniform interface regardless of environment. Supports both atomic file transfer and streaming interfaces, and provides encryption of user data.
- Scalability:

  Toil can easily handle workflows concurrently using hundreds of nodes and thousands of cores.
- Robustness:

  Toil workflows support arbitrary worker and leader failure, with strong check-pointing that always allows resumption.
- Efficiency:

  Caching, fine grained, per task, resource requirement specifications, and support for the AWS spot market mean workflows can be executed with little waste.
- Declarative and dynamic workflow creation:

  Workflows can be declared statically, but new jobs can be added dynamically during execution within any existing job, allowing arbitrarily complex workflow graphs with millions of jobs within them.

- Support for databases and services:

  For example, Apache Spark clusters can be created in seconds and easily integrated within a toil workflow as a service, with precisely defined time start and end times that fits with the flow of other jobs in the workflow.

- Draft Common Workflow Language (CWL) support

  Complete support for the draft 2.0 CWL specification, allowing it to execute CWL workflows.

- Open Source: An Apache license allows unrestricted use.

1.2 Prerequisites

- Python 2.7.x
- pip > 7.x

1.3 Installation

To setup a basic Toil installation use

```
pip install toil
```

Toil uses setuptools' extras mechanism for dependencies of optional features like support for Mesos or AWS. To install Toil with all bells and whistles use

```
pip install toil[aws,mesos,azure,encryption]
```

Here's what each extra provides:

- The `aws` extra provides support for storing workflow state in Amazon AWS. This extra has no native dependencies.
- The `azure` extra stores workflow state in Microsoft Azure Storage. This extra has no native dependencies.
- The `mesos` extra provides support for running Toil on an Apache Mesos cluster. Note that running Toil on SGE (GridEngine), Parasol or a single machine does not require an extra. The `mesos` extra requires the following native dependencies:
  - *Apache Mesos*
  - *Python headers and static libraries*
- The `encryption` extra provides client-side encryption for files stored in the Azure and AWS job stores. This extra requires the following native dependencies:
  - *Python headers and static libraries*
  - *Libffi headers and library*

Apache Mesos

Only needed for the `mesos` extra. Toil has been tested with version 0.25.0. Mesos can be installed on Linux by following the instructions on https://open.mesosphere.com/getting-started/install/. The Homebrew package manager has a formula for Mesos such that running `brew install mesos` is probably the easiest way to install Mesos on OS X. This assumes, of course, that you already have Xcode and Homebrew.
Please note that even though Toil depends on the Python bindings for Mesos, it does not explicitly declare that dependency and they will **not** be installed automatically when you run `pip install toil[mesos]`. You need to install the bindings manually. The Homebrew formula for OS X installs them by default. On Ubuntu you will need to download the appropriate .egg from https://open.mesosphere.com/downloads/mesos/ and install it using `easy_install -a <path_to_egg>`. Note that on Ubuntu Trusty you may need to upgrade `protobuf` via `pip install --upgrade protobuf` **before** running the above `easy_install` command.

Python headers and static libraries

Only needed for the `mesos` and `encryption` extras. The Python headers and static libraries can be installed on Ubuntu/Debian by running `sudo apt-get install build-essential python-dev` and accordingly on other Linux distributions. On Mac OS X, these headers and libraries are installed when you install the Xcode command line tools by running `xcode-select --install`, assuming, again, that you have Xcode installed.

Libffi headers and library

Libffi is only needed for the `encryption` extra. To install Libffi on Ubuntu, run `sudo apt-get install libffi-dev`. On Mac OS X, run `brew install libffi`. This assumes, of course, that you have Xcode and Homebrew installed.

1.4 Scripting Quick Start

Toil's Job class (*`toil.job.Job`*) contains the Toil API, documented below. To begin, consider this short toil script which illustrates defining a workflow:

```
from toil.job import Job

def helloWorld(message, memory="2G", cores=2, disk="3G"):
    return "Hello, world!, here's a message: %s" % message

j = Job.wrapFn(helloWorld, "woot")

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflow")
    print Job.Runner.startToil(j, options) #Prints Hello, world!, ...
```

The workflow consists of a single job, which calls the helloWorld function. The resource requirements for that job are (optionally) specified by keyword arguments (memory, cores, disk).

The *`toil.job.Job.Runner`* class handles the invocation of Toil workflows. It is fed an options object that configures the running of the workflow. This can be populated by an argument parser object using *`toil.job.Job.Runner.getDefaultArgumentParser()`*, allowing all these options to be specified via the command line to the script. See *User Tutorial* for more details.

1.5 Building & Testing

For developers and people interested in building the project from source the following explains how to setup virtualenv to create an environment to use Toil in.

After cloning the source and `cd`-ing into the project root, create a virtualenv and activate it:

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

Simply running

```
make
```

from the project root will print a description of the available Makefile targets.

If cloning from GitHub, running

```
make develop
```

will install Toil in *editable* mode, also known as development mode. Just like with a regular install, you may specify extras to use in development mode after installing any native dependencies listed in *Installation*.

```
make develop extras=[aws,mesos,azure,encryption]
```

To invoke the tests (unit and integration) use

```
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 azure' src"
```

This will run only the tests that don't depend on the `azure` 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, the `gridengine` and `parasol` features fall into that category. So in order to skip tests involving both the Parasol feature and the Azure extra, the following can be used:

```
make test tests="-m 'not azure and not parasol' src"
```

1.5.1 Running Mesos Tests

See *Apache Mesos*. Be sure to create the virtualenv with `--system-site-packages` to include the Mesos Python bindings. Verify by activating the virtualenv and running .. `pip list | grep mesos`. On OS X, 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-packag
```

CHAPTER 2

User Tutorial

This tutorial will guide you through the features of Toil from a user perspective.

2.1 Job basics

The atomic unit of work in a Toil workflow is a *job* (`toil.job.Job`). User scripts inherit from this base class to define units of work. For example:

```python
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):
        fileStore.logToMaster("Hello, world!, I have a message: %s" %
                              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 logs a message using `toil.job.Job.FileStore.logToMaster()`, which will be registered in the log output of the leader process of the workflow.

2.2 Job.Runner

2.2.1 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. This uses the `toil.job.Job.Runner` class, which is used to start and resume Toil workflows. For example:

```python
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):
        fileStore.logToMaster("Hello, world!, I have a message: %s"
                              % self.message)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    Job.Runner.startToil(HelloWorld("woot"), options)
```

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 an S3 bucket or Azure object store location. By default the job store is deleted if the workflow completes successfully.

On the next line we specify a single option, the log level for the workflow, to ensure the message from the job is reported in the leader's log, which by default will be printed to standard error.

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.

2.2.2 Specifying arguments via the command line

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.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):
        fileStore.logToMaster("Hello, world!, I have a message: %s"
                              % self.message)

if __name__=="__main__":
    parser = Job.Runner.getDefaultArgumentParser()
    options = parser.parse_args()
    options.logLevel = "INFO"
    Job.Runner.startToil(HelloWorld("woot"), options)
```

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.

2.2.3 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. To resume a workflow specify the "restart" option in the options object passed to *`toil.job.Job.Runner.startToil()`*. 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.job.Job.Runner.startToil()`* will raise a `toil.job.leader.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.

2.3 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:

```
from toil.job import Job

def helloWorld(job, message, memory="2G", cores=2, disk="3G"):
    job.fileStore.logToMaster("Hello world, "
    "I have a message: %s" % message) # This uses a logging function
    # of the Job.FileStore class

j = Job.wrapJobFn(helloWorld, "woot")

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    Job.Runner.startToil(j, options)
```

Is equivalent to the complete previous example. Here *helloWorld* is an example of 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`.

The function call:

```
Job.wrapJobFn(helloWorld, "woot")
```

Creates the instance of the `toil.job.JobFunctionWrappingTarget` that wraps the job 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 to as arguments when wrapping a function as a job and will be used to specify resource requirements.

Non-job functions can also be wrapped, for example:

```
from toil.job import Job

def helloWorld2(message):
    return "Hello world, I have a message: %s" % message

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    print Job.Runner.startToil(Job.wrapFn(helloWorld2, "woot"), options)
```

Here the only major difference to note is the line:

```
Job.Runner.startToil(Job.wrapFn(helloWorld, "woot"), options)
```

Which uses the function `toil.job.Job.wrapFn()` to wrap an ordinary function instead of `toil.job.Job.wrapJobFn()` which wraps a job function.

2.4 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.job import Job

def helloWorld(job, message, memory="2G", cores=2, disk="3G"):
    job.fileStore.logToMaster("Hello world, "
    "I have a message: %s" % message) # This uses a logging function
    # of the Job.FileStore class

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)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    Job.Runner.startToil(j1, options)
```

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.job import Job

def helloWorld(job, message, memory="2G", cores=2, disk="3G"):
    job.fileStore.logToMaster("Hello world, "
    "I have a message: %s" % message) # This uses a logging function
    # of the Job.FileStore class

j1 = Job.wrapJobFn(helloWorld, "first")
j2 = j1.addChildJobFn(helloWorld, "second or third")
j3 = j1.addChildJobFn(helloWorld, "second or third")
j4 = j1.addFollowOnJobFn(helloWorld, "last")

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    Job.Runner.startToil(j1, options)
```

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 acylic 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.job import Job

def helloWorld(job, message, memory="2G", cores=2, disk="3G"):
    job.fileStore.logToMaster("Hello world, "
    "I have a message: %s" % message) # This uses a logging function
    # of the Job.FileStore class

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)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    Job.Runner.startToil(j1, options)
```

Note the use of an extra child edge to make j4 a child of both j2 and j3.

2.5 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:

```
from toil.job import Job

def binaryStringFn(job, message="", depth):
    if depth > 0:
        job.addChildJobFn(binaryStringFn, message + "0", depth-1)
        job.addChildJobFn(binaryStringFn, message + "1", depth-1)
    else:
        job.fileStore.logToMaster("Binary string: %s" % message)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    Job.Runner.startToil(Job.wrapJobFn(binaryStringFn, depth=5), options)
```

The 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.

2.6 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. However, it is often desirable to return variables from jobs in a non-recursive or dynamic context. In Toil this is achieved with promises, as illustrated in the following example:

```
from toil.job import Job

def fn(job, i):
    job.fileStore.logToMaster("i is: %s" % i, logLevel=100)
    return i+1

j1 = Job.wrapJobFn(fn, 1)
j2 = j1.addChildJobFn(fn, j1.rv())
j3 = j1.addFollowOnJobFn(fn, j2.rv())

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    Job.Runner.startToil(j1, options)
```

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 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.PromisedJobReturnValue`) is essentially a pointer to 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 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:

```
from toil.job import Job

def binaryStrings(job, message="", depth):
    if depth > 0:
        s = [ job.addChildJobFn(binaryStrings, message + "0",
                                depth-1).rv(),
              job.addChildJobFn(binaryStrings, message + "1",
                                depth-1).rv() ]
        return job.addFollowOnFn(merge, s).rv()
    return [message]

def merge(strings):
    return strings[0] + strings[1]

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    l = Job.Runner.startToil(Job.wrapJobFn(binaryStrings, depth=5), options)
    print l #Prints a list of all binary strings of length 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.

2.7 Job.FileStore: 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.job.Job.FileStore` 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:

```
from toil.job import Job

class LocalFileStoreJob(Job):
    def run(self, fileStore):
        scratchDir = fileStore.getLocalTempDir() #Create a temporary
        # directory safely within the allocated disk space
        # reserved for the job.

        scratchFile = fileStore.getLocalTempFile() #Similarly
        # create a temporary file.

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    #Create an instance of FooJob which will
    # have at least 10 gigabytes of storage space.
    j = LocalFileStoreJob(disk="10G")
    #Run the workflow
    Job.Runner.startToil(j, options)
```

Job functions can also access the file store for the job. The equivalent of the LocalFileStoreJob class is equivalently:

```
def localFileStoreJobFn(job):
    scratchDir = job.fileStore.getLocalTempDir()
    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:

```
from toil.job import Job
import os

def globalFileStoreJobFn(job):
    job.fileStore.logToMaster("The following example exercises all the"
                              " methods provided by the"
                              " Job.FileStore class")

    scratchFile = job.fileStore.getLocalTempFile() # Create a local
    # temporary file.

    with open(scratchFile, 'w') as fH: # Write something in the
        # scratch file.
        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.
    fileID = job.fileStore.writeGlobalFile(scratchFile) #This write
    # is asynchronous by default

    # 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("Out brief candle")

    # 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.fileStore.getLocalTempDir(), "foo.txt")
    job.fileStore.readGlobalFile(fileID, 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__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    Job.Runner.startToil(Job.wrapJobFn(globalFileStoreJobFn), options)
```

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.

2.8 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:

```
from toil.job import Job

class DemoService(Job.Service):

    def start(self):
        # Start up a database/service here
        return "loginCredentials" # Return a value that enables another
        # process to connect to the database

    def stop(self):
        # Cleanup the database here
        pass

j = Job()
s = DemoService()
loginCredentialsPromise = j.addService(s)

def dbFn(loginCredentials):
    # Use the login credentials returned from the service's start method
    # to connect to the service
    pass

j.addChildFn(dbFn, loginCredentialsPromise)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    Job.Runner.startToil(j, options)
```

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.

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.

2.9 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.:

```
from toil.job import Job

# 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)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    Job.Runner.startToil(Ap, options)
```

An *encapsulated job* of E(A) of A saves making Ap, instead we can write:

```
from toil.job import Job

# A
A = Job()
A.addChild(Job())
A.addFollowOn(Job())

#Encapsulate A
A = A.encapsulate()
```

```
# 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)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    Job.Runner.startToil(A, options)
```

Note the call to *`toil.job.Job.encapsulate()`* creates the `toil.job.Job.EncapsulatedJob`.

2.10 Toil Utilities

TODO: Cover clean, kill, restart, stats and status. Note these should have APIs to access them as well as the utilities.

CHAPTER 3

Toil API

3.1 Job Methods

Jobs are the units of work in Toil which are composed into workflows.

class `toil.job.`**`Job`**(*memory=None*, *cores=None*, *disk=None*, *cache=None*)

Class represents a unit of work in toil.

This method must be called by any overiding constructor.

Parameters

- **`memory`** (*`int or string convertable by bd2k.util.humanize.human2bytes to an int`*) – the maximum number of bytes of memory the job will require to run.
- **`cores`** (*`int or string convertable by bd2k.util.humanize.human2bytes to an int`*) – the number of CPU cores required.
- **`disk`** (*`int or string convertable by bd2k.util.humanize.human2bytes to an int`*) – the amount of local disk space required by the job, expressed in bytes.
- **`cache`** (*`int or string convertable by bd2k.util.humanize.human2bytes to an int`*) – the amount of disk (so that cache <= disk), expressed in bytes, for storing files from previous jobs so that they can be accessed from a local copy.

`addChild`(*childJob*)

Adds 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.

Parameters **`childJob`** (`toil.job.Job`) –

Returns childJob

Return type *toil.job.Job*

`addChildFn`(*fn*, **args*, ***kwargs*)

Adds a function as a child job.

Parameters **`fn`** – 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 *toil.job.FunctionWrappingJob*

addChildJobFn (*fn*, **args*, ***kwargs*)
Adds a job function as a child job. See `toil.job.JobFunctionWrappingJob` for a definition of a job function.

Parameters **fn** – 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 *toil.job.JobFunctionWrappingJob*

addFollowOn (*followOnJob*)
Adds a follow-on job, follow-on jobs will be run after the child jobs and their successors have been run.

Parameters **followOnJob** (`toil.job.Job`) –

Returns followOnJob

Return type *toil.job.Job*

addFollowOnFn (*fn*, **args*, ***kwargs*)
Adds a function as a follow-on job.

Parameters **fn** – 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 *toil.job.FunctionWrappingJob*

addFollowOnJobFn (*fn*, **args*, ***kwargs*)
Add a follow-on job function. See `toil.job.JobFunctionWrappingJob` for a definition of a job function.

Parameters **fn** – 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 *toil.job.JobFunctionWrappingJob*

addService (*service*)
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.

Parameters **service** (`toil.job.Job.Service`) – Service to add.

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 *toil.job.PromisedJobReturnValue*

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!

`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.

`checkJobGraphForDeadlocks``()`

Raises ***`toil.job.JobGraphDeadlockException`*** – if the job graph is cyclic or contains multiple roots.

See *`toil.job.Job.checkJobGraphConnected()`* and `toil.job.Job.checkJobGraphAcyclic()` for more info.

`effectiveRequirements``(`*config*`)`

Determine and validate the effective requirements for this job, substituting a missing explict requirement with a default from the configuration.

Return type Expando

Returns a dictionary/object hybrid with one entry/attribute for each requirement

`encapsulate``()`

Encapsulates the job, see *`toil.job.EncapsulatedJob`*. Convenience function for constructor of *`toil.job.EncapsulatedJob`*.

Returns an encapsulated version of this job.

Return type toil.job.EncapsulatedJob.

`getRootJobs``()`

Returns The roots of the connected component of jobs that contains this job. A root is a job with no predecessors.

:rtype : set of toil.job.Job instances

`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

`getUserScript``()`

`hasChild``(`*childJob*`)`

Check if childJob is already a child of this job.

Parameters **`childJob`** (`toil.job.Job`) –

Returns True if childJob is a child of the job, else False.

Return type Boolean

`run``(`*fileStore*`)`

Override this function to perform work and dynamically create successor jobs.

Parameters **`fileStore`** (`toil.job.Job.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()`*.

`rv` (*argIndex=None*)

Gets a *promise* (*`toil.job.PromisedJobReturnValue`*) representing a return value of the job's run function.

Parameters **`argIndex`** (*`int or None`*) – If None the complete return value will be returned, if argIndex is an integer it is used to refer to the return value as indexable (tuple/list/dictionary, or in general an object that implements __getitem__), hence rv(i) would refer to the ith (indexed from 0) member of the return value.

Returns A promise representing the return value of the *`toil.job.Job.run()`* function.

Return type toil.job.PromisedJobReturnValue, referred to as a "promise"

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 *toil.job.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 *toil.job.JobFunctionWrappingJob*

3.2 Job.FileStore

The FileStore is an abstraction of a Toil run's shared storage.

class `Job.`**`FileStore`** (*jobStore*, *jobWrapper*, *localTempDir*, *inputBlockFn*)

Class used to manage temporary files, read and write files from the job store and log messages, passed as argument to the *`toil.job.Job.run()`* method.

This constructor should not be called by the user, FileStore instances are only provided as arguments to the run function.

Parameters

- **`jobStore`** (*`toil.jobStores.abstractJobStore.JobStore`*) – The job store for the workflow.
- **`jobWrapper`** (*`toil.jobWrapper.JobWrapper`*) – The jobWrapper for the job.

- **`localTempDir`** (*`string`*) – A temporary directory in which local temporary files will be placed.
- **`inputBlockFn`** (*`method`*) – A function which blocks and which is called before the fileStore completes atomically updating the jobs files in the job store.

`deleteGlobalFile` (*fileStoreID*)

Deletes a global file with the given job store ID.

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 job store ID of the file to be deleted.

`getLocalTempDir`()

Get a new local temporary directory in which to write files that 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 string

`getLocalTempFile`()

Get a new local temporary file that will persist for the duration of the job.

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 string

`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

- **`string`** – The string to log.
- **`level`** (*`int`*) – The logging level.

`readGlobalFile` (*fileStoreID*, *userPath=None*, *cache=True*)

Get a copy of a file in the job store.

Parameters

- **`userPath`** (*`string`*) – a path to the name of file to which the global file will be copied or hard-linked (see below).
- **`cache`** (*`boolean`*) – If True will use caching (see below). Caching will attempt to keep copies of files between sequences of jobs run on the same worker.

If cache=True and userPath is either: (1) a file path contained within a directory or, recursively, a subdirectory of a temporary directory returned by Job.FileStore.getLocalTempDir(), or (2) a file path returned by Job.FileStore.getLocalTempFile() then the file will be cached and returned file will be read only (have permissions 444).

If userPath is specified and the file is already cached, the userPath file will be a hard link to the actual location, else it will be an actual copy of the file.

If the cache=False or userPath is not either of the above the file will not be cached and will have default permissions. Note, if the file is already cached this will result in two copies of the file on the system.

Returns an absolute path to a local, temporary copy of the file keyed by fileStoreID.

:rtype : string

`readGlobalFileStream`(*fileStoreID*)

Similar to readGlobalFile, but allows a stream to be read from the job store.

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.

`writeGlobalFile`(*localFileName*, *cleanup=False*)

Takes a file (as a path) and uploads it to the job store.

If the local file is a file returned by `toil.job.Job.FileStore.getLocalTempFile()` or is in a directory, or, recursively, a subdirectory, returned by `toil.job.Job.FileStore.getLocalTempDir()` then the write is asynchronous, so further modifications during execution to the file pointed by localFileName will result in undetermined behavior. Otherwise, the method will block until the file is written to the file store.

Parameters

- **`localFileName`** (`string`) – The path to the local file to upload.
- **`cleanup`** (`Boolean`) – 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.

`writeGlobalFileStream`(*cleanup=False*)

Similar to writeGlobalFile, but allows the writing of a stream to the job store.

Parameters **`cleanup`** (`Boolean`) – is as in `toil.job.Job.FileStore.writeGlobalFile()`.

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.

3.3 Job.Runner

The Runner contains the methods needed to configure and start a Toil run.

class `Job.`**`Runner`**

Used to setup and run Toil workflow.

static `addToilOptions`(*parser*)

Adds the default toil options to an `optparse` or `argparse` parser object.

Parameters **`parser`** (`optparse.OptionParser or argparse.ArgumentParser`) – Options object to add toil options to.

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`** (`string`) – A string describing the jobStore for the workflow.

Returns The options used by a toil workflow.

Return type argparse.ArgumentParser values object

static **`startToil`**`(`*job*, *options*`)`

Runs the toil workflow using the given options (see Job.Runner.getDefaultOptions and Job.Runner.addToilOptions) starting with this job. :param toil.job.Job job: root job of the workflow :raises: toil.leader.FailedJobsException if at the end of function their remain failed jobs. :returns: return value of job's run function

3.4 Job.Service

The Service class allows databases and servers to be spawned within a Toil workflow.

class `Job.`**`Service`**`(`*memory=None*, *cores=None*, *disk=None*`)`

Abstract class used to define the interface to a service.

Memory, core and disk requirements are specified identically to as in `toil.job.Job.__init__()`.

`start``()`

Start the service.

Returns An object describing how to access the service. Must be pickleable. Will be used by jobs to access the service (see *`toil.job.Job.addService()`*).

`stop``()`

Stops the service.

Function can block until complete.

3.5 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.

Parameters **`userFunction`** – The function to wrap. The userFunction will be called with the `*args` and `**kwargs` as arguments.

The keywords "memory", "cores", "disk", "cache" are reserved keyword arguments that if specified will be used to determine the resources 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).

`getUserScript``()`

`run``(`*fileStore*`)`

3.6 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.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.Job` class provides.

To enable the job function to get access to the *`toil.job.Job.FileStore`* instance (see `toil.job.Job.Run()`), it is made a variable of the wrapping job called fileStore.

Parameters **`userFunction`** – The function to wrap. The userFunction will be called with the `*args` and `**kwargs` as arguments.

The keywords “memory”, “cores”, “disk”, “cache” are reserved keyword arguments that if specified will be used to determine the resources 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*)

3.7 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*)

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:

```
A, B = A(), B() #Job A and subgraph, Job B
A' = A.encapsulate()
A'.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.
```

The return value of an encapsulatd 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.

Parameters **`job`** (`toil.job.Job`) – the job to encapsulate.

`addChild`(*childJob*)

`addFollowOn`(*followOnJob*)

`addService`(*service*)

`rv`(*argIndex=None*)

3.8 Promise

The class used to reference return values of jobs/services not yet run/started.

class `toil.job.`**`PromisedJobReturnValue`**(*promiseCallBackFunction*)

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 PromisedJobReturnValue (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.

3.9 Exceptions

Toil specific exceptions.

class `toil.job.`**`JobException`**`(`*message*`)`

General job exception.

class `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()`*.

CHAPTER 4

The batch system interface

4.1 Implementing the batch system interface tutorial

The batch system interface is used by Toil to abstract over different ways of running batches of jobs, for example GridEngine, Mesos, Parasol and a single node.

This tutorial will guide you through the batch system class (*`toil.batchSystems.abstractBatchSystem.AbstractBatch`* functions and how to implement them to support a new batch system.

TODO

4.2 Toil Abstract Batch System API

The *`toil.batchSystems.abstractBatchSystem.AbstractBatchSystem`* API is implemented to run jobs using a given job management system, e.g. Mesos.

class `toil.batchSystems.abstractBatchSystem.`**`AbstractBatchSystem`**(*config*, *maxCores*, *maxMemory*, *maxDisk*)

An abstract (as far as python currently allows) base class to represent the interface the batch system must provide to the toil.

This method must be called. The config object is setup by the toilSetup script and has configuration parameters for the jobtree. You can add stuff to that script to get parameters for your batch system.

`checkResourceRequest`(*memory*, *cores*, *disk*)

Check resource request is not greater than that available.

`environment` **= None**

:type dict[str,str]

`getIssuedBatchJobIDs`()

A list of jobs (as jobIDs) currently issued (may be running, or maybe just waiting). Despite the result being a list, the ordering should not be depended upon.

classmethod **`getRescueBatchJobFrequency`**()

Gets the period of time to wait (floating point, in seconds) between checking for missing/overlong jobs.

`getRunningBatchJobIDs`()

Gets a map of jobs (as jobIDs) currently running (not just waiting) and a how long they have been running for (in seconds).

getUpdatedBatchJob(*maxWait*)

Gets a job that has updated its status, according to the job manager. Max wait gives the number of seconds to pause waiting for a result. If a result is available returns (jobID, exitValue) else it returns None. Does not return anything for jobs that were killed.

issueBatchJob(*command*, *memory*, *cores*, *disk*)

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 cpu cores needed for the job and error-file is the path of the file to place any std-err/std-out in.

killBatchJobs(*jobIDs*)

Kills the given job IDs.

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.

NB: Only the Mesos and single-machine batch systems support passing environment variables. On other batch systems, this method has no effect. See https://github.com/BD2KGenomics/toil/issues/547.

shutdown()

Called at the completion of a toil invocation. Should cleanly terminate all worker threads.

static supportsHotDeployment()

Whether this batch system supports hot deployment of the user script and toil itself. If it does, the __init__ method will have to accept two optional parameters in addition to the declared ones: userScript and toilDistribution. Both will be instances of toil.common.HotDeployedResource that represent the user script and a source tarball (sdist) of toil respectively.

CHAPTER 5

The job store interface

5.1 Implementing the job store interface tutorial

The job store interface is an abstraction layer that that hides the specific details of file storage, for example standard file systems, S3, etc. This tutorial will guide you through the job store (`toil.jobStores.abstractJobStore.AbstractJobStore`) functions and how to implement them to support a new file store.

TODO

5.2 Toil Abstract Job Store API

The `toil.jobStores.abstractJobStore.AbstractJobStore` API is implemented to support a give file store, e.g. S3.

class `toil.jobStores.abstractJobStore.`**`AbstractJobStore`**(*config=None*)

Represents the physical storage for the jobs and associated files in a toil.

Parameters **`config`** – If config is not None then the given configuration object will be written to the shared file "config.pickle" which can later be retrieved using the readSharedFileStream. See writeConfigToStore. If this file already exists it will be overwritten. If config is None, the shared file "config.pickle" is assumed to exist and is retrieved. See loadConfigFromStore.

`clean`(*rootJobWrapper*, *jobCache=None*)

Function to cleanup the state of a jobStore after a restart. Fixes jobs that might have been partially updated. Resets the try counts. Removes jobs that are not successors of the rootJobWrapper.

If jobCache is passed, it must be a dict from job ID to JobWrapper object. Jobs will be loaded from the cache (which can be downloaded from the jobStore in a batch) instead of piecemeal when recursed into.

`config`

`create`(*command*, *memory*, *cores*, *disk*, *predecessorNumber=0*)

Creates a job, adding it to the store.

Command, memory, cores and predecessorNumber are all arguments to the job's constructor.

:rtype : toil.jobWrapper.JobWrapper

`delete`(*jobStoreID*)

Removes from store atomically, can not then subsequently call load(), write(), update(), etc. with the job.

This operation is idempotent, i.e. deleting a job twice or deleting a non-existent job will succeed silently.

`deleteFile` (*jobStoreFileID*)
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.

`deleteJobStore` ()
Removes the jobStore from the disk/store. Careful!

`exists` (*jobStoreID*)
Returns true if the job is in the store, else false.

:rtype : bool

`fileExists` (*jobStoreFileID*)
:rtype : True if the jobStoreFileID exists in the jobStore, else False

`getEmptyFileStoreID` (*jobStoreID=None*)
:rtype : string, the ID of a new, empty file.

jobStoreID is the id of a job, or None. If specified, when delete(job) is called all files written with the given job.jobStoreID will be removed from the jobStore.

Call to fileExists(getEmptyFileStoreID(jobStoreID)) will return True.

`getEnv` ()
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]

`getPublicUrl` (*fileName*)
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.

`getSharedPublicUrl` (*sharedFileName*)
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.

`jobs` ()
Returns iterator on all jobs in the store. The iterator will contain all jobs, but may also contain orphaned jobs that have already finished succesfully and should not be rerun. To guarantee you only get jobs that can be run, instead construct a ToilState object

:rtype : iterator

`load` (*jobStoreID*)
Loads a job for the given jobStoreID and returns it.

Return type toil.jobWrapper.JobWrapper

Raises NoSuchJobException if there is no job with the given jobStoreID

`publicUrlExpiration` **= datetime.timedelta(365)**

`readFile` (*jobStoreFileID*, *localFilePath*)
Copies 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.

`readFileStream` (**args*, ***kwds*)
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.

`readSharedFileStream`(**args*, ***kwds*)
Returns a context manager yielding a readable file handle to the global file referenced by the given name.

`readStatsAndLogging`(*callback*, *readAll=False*)
Reads stats/logging strings accumulated by the writeStatsAndLogging() method. For each stats/logging string this method calls the given callback function with an open, readable file handle from which the stats string can be read. Returns the number of stats/logging strings processed. Each stats/logging string is only processed once unless the readAll parameter is set, in which case the given callback will be invoked for all existing stats/logging strings, including the ones from a previous invocation of this method.

`sharedFileNameRegex` **= <_sre.SRE_Pattern object>**

`update`(*job*)
Persists the job in this store atomically.

`updateFile`(*jobStoreFileID*, *localFilePath*)
Replaces the existing version of a file in the jobStore. Throws an exception if the file does not exist.

Raises **`ConcurrentFileModificationException`** – if the file was modified concurrently during an invocation of this method

`updateFileStream`(*jobStoreFileID*)
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.

Raises **`ConcurrentFileModificationException`** – if the file was modified concurrently during an invocation of this method

`writeConfigToStore`()
Re-writes the config attribute to the jobStore, so that its values can be retrieved if the jobStore is reloaded.

`writeFile`(*localFilePath*, *jobStoreID=None*)
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.

jobStoreID is the id of a job, or None. If specified, when delete(job) is called all files written with the given job.jobStoreID will be removed from the jobStore.

`writeFileStream`(**args*, ***kwds*)
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.

`writeSharedFileStream`(**args*, ***kwds*)
Returns a context manager yielding a writable file handle to the global file referenced by the given name.

Parameters

- **`sharedFileName`** – A file name matching AbstractJobStore.fileNameRegex, unique within the physical storage represented by this job store
- **`isProtected`** – True if the file must be encrypted, None if it may be encrypted or False if it must be stored in the clear.

Raises **`ConcurrentFileModificationException`** – if the file was modified concurrently during an invocation of this method

`writeStatsAndLogging`(*statsAndLoggingString*)
Adds the given statistics/logging string to the store of statistics info.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index