Cerise Documentation

Release develop

Lourens Veen

May 29, 2019

Contents:

Cerise is a simple REST service that can run (many) CWL jobs on a remote compute resource, such as a remote server or HPC compute cluster. This documentation explains how to set it up and use it, and also how it works on the inside.

The implementation is fairly complete, and the main things needed are some real-world testing, bug fixing, and polish.

CHAPTER 1

Introduction

Cerise is a generic service for running workflows on compute resources, such as clusters, supercomputers, and simply remote machines. It tries to offer a consistent environment for workflows, so that a workflow sent to resource A will work unchanged on resource B as well.

To achieve this, and to offer a bit of safety and perhaps security, Cerise does not allow running arbitrary command line tools. Instead, it expects the user to submit a workflow document that refers to predefined steps built into the service. Both workflows and steps are defined using the Common Workflow Language (CWL).

Defining these steps, and adding them to the service, is called specialising the service. A specialisation of Cerise is always specific to a project and to a compute resource. The project determines which steps are available and what inputs and outputs they have. The compute resource determines how the steps are implemented. Workflows are written using steps from a particular project, and can then be sent to any specialisation to that project. Where the workflow runs will differ depending on which specialisation is used, but the result should be the same (assuming the calculation is deterministic!).

This site contains the documentation for Cerise.

1.1 Installation

Cerise can be run directly on a host, or in a Docker container. A local installation is created as follows:

clone the repository *git clone git@github.com:MD-Studio/cerise.git*

change into the top-level directory *cd cerise*

install using *pip3 install .*

Steps and supporting files may then be placed in the api/ directory to specialise the service. For a detailed explanation, see *Specialising Cerise*.

To build the Docker image, use

```
docker build -t cerise .
```

and then start a container using

docker run –name=cerise -p 29593:29593 cerise

Note that the docker image gets its config.yml from conf/docker-config.yml in the source tree.

However, this will run a plain, unspecialised Cerise, which is not very useful, as it runs jobs locally inside the container, and it doesn't contain any steps to execute. To use Cerise in Docker, you should make a new, specialised Docker image based on the standard Cerise image, and start that instead. Instructions for how to do so are also under *Specialising Cerise*

1.2 Dependencies

- Python 3.5 or up

On the compute resource:

- Python 2.7 and CWLTool (or another CWL runner), or
- Python3 (using the built-in CWLTiny runner)

1.3 Example usage

In the examples/ directory, you will find some example Python scripts that create jobs and execute them on the job running service.

1.4 Contribution guide

Cerise follows the Google Python style guide, with Sphinxdoc docstrings for module public functions. If you want to contribute to the project please fork it, create a branch including your addition, and create a pull request.

The tests use relative imports and can be run directly after making changes to the code. To run all tests use *pytest* in the main directory. This will also run the integration tests, which take several minutes to complete as a bunch of Docker containers is built, started, and stopped.

Before creating a pull request please ensure the following:

- You have written unit tests to test your additions
- All unit tests pass
- The examples still work and produce the same (or better) results
- The code is compatible with Python 3.5
- An entry about the change or addition is created in CHANGELOG.md
- You've added yourself as contributing author

Contributing authors so far:

- Lourens Veen

CHAPTER 2

Cerise Configuration

2.1 Introduction

Cerise takes configuration information from various sources, with some overriding others. This page describes the configuration files and what can be configured in them.

2.2 Main configuration file

The main configuration file is located at `conf/config.yml`, and contains general configuration information for the Cerise service in YAML format. It looks as follows:

```
database:
  file: run/cerise.db

logging:
  file: /var/log/cerise/cerise_backend.log
  level: INFO

pidfile: run/cerise_backend.pid

client-file-exchange:
  store-location-service: file:///tmp/cerise_files
  store-location-client: file:///tmp/cerise_files

rest-service:
  base-url: http://localhost:29593
  hostname: 127.0.0.1
  port: 29593
```

Cerise uses SQLite to persistently store the jobs that have been submitted to it. SQLite databases consist of a single file, the location of which is given by the `file` key under `database`.

Logging output is configured under the `logging` key. Make sure that the user that Cerise runs under has write access to the given path. If you want to log to /var/log without giving Cerise root rights, making the specified log file on beforehand and then giving ownership to the user Cerise runs under works well. Or you can make a subdirectory and give the user access to that.

The `pidfile` key specifies a path to a file into which Cerise's process identifier (PID) is written. This can be used to shut down a running service, i.e. `kill <pid>` will cleanly shut down Cerise.

Under `client-file-exchange`, the means of communicating files between Cerise and its users is configured. Communication is done using a shared folder accessible to both the users and the Cerise service. If Cerise is running locally, both parties have access to the same file system, and see the shared folder in the same location. Thus, `store-location-service` and `store-location-client` both point to the same on-disk directory.

If the Cerise service does not share a file system with the client, then a directory on the Cerise server must be made available to the client, e.g. via WebDAV. In this case, client and service access the same directory using different URLs, e.g.

```
client-file-exchange:
  store-location-service: file:///home/webdav/files
  store-location-client: http://localhost:29593/files
```

The user is expected to submit references to files that start with the URL in `store-location-client`, Cerise will then fetch the corresponding files from the directory specified in `store-location-service`.

`store-location-client` can be overridden by specifying the environment variable CERISE_STORE_LOCATION_CLIENT. If you want to run multiple Cerise instances in containers, simultaneously, then you need to remap the ports on which they are available to avoid collisions. With this environment variable, the port can be easily injected into the container, removing the need to have a different image for each container. Cerise Client uses this functionality.

Finally, key `rest-service` has the hostname and port on which the REST service should listen, as well as the external URL on which it is available. If you want the service to be available to the outside world, this should be the IP address of the network adaptor to listen on, or `0.0.0.0` to listen on all adaptors. Note that a service running inside a Docker container needs to have `0.0.0.0` for it to be accessible from outside the container.

Since the service needs to pass URLs to the client sometimes, it needs to know at which URL it is available to the client. This is specified by `base-url`, which should contain the first part of the URL to the REST API, before the `/jobs` part. Alternatively, you can set the CERISE_BASE_URL environment variable to this value.

2.3 Compute resource configuration

Information on which compute resource to connect to, and how to transfer files and submit jobs to it, is stored separately from the main service configuration, to make it easier to create specialisations. Furthermore, to enable different users to use the same specialised Cerise installation (e.g. Docker image), credentials can be specified using environment variables. (Cerise Client uses the latter method.) If you are making a specialisation that is to be shared with others, do not put your credentials in this file!

Note: this file is somewhat outdated, but well be updated prior to the 1.0 release.

2.3.1 API configuration file

The API configuration file is located in `api/config.yml`, and has the following format:

```
compute-resource:
  credentials:
    username: None
    password: None
    certfile: None
    passphrase: None

  files:
    credentials:
      username: None
      password: None
      certfile: None
      passphrase: None

    protocol: local
    location: None
    path: /home/$CERISE_USERNAME/.cerise

  jobs:
    credentials:
      username: None
      password: None
      certfile: None
      passphrase: None

    protocol: local
    location: None
    scheduler: none

    queue-name: None      # cluster default
    slots-per-node: None  # cluster default
    cores-per-node: 32
    scheduler-options: None
    cwl-runner: $CERISE_API_FILES/cerise/cwltiny.py

  refresh: 10
```

This file describes the compute resource and how to connect to it. Under the `files` key, file access (staging) is configured, while the `jobs` key has settings on how to submit jobs. `credentials`, and keys `username`, `password`, `certfile` and `passphrase` occurring throughout, refer to credentials, and will be discussed below. Keys may be omitted if they are not needed, e.g. `location` may be omitted if `protocol` is `local`, in which case credentials may also me left out.

For file staging, a protocol, location and path may be specified. Supported protocols are `file`, `sftp`, `ftp`, or `webdav`, where `file` refers to direct access to the local file system.

`location` provides the host name to connect to; to run locally, this may be omitted or empty. `path` configures the remote directory where Cerise will put its files. It may contain the string `$CERISE_USERNAME`, which will be replaced with the user account name that the service is using. This is useful if you want to put Cerise's files into the users home directory, e.g. `/home/$CERISE_USERNAME/.cerise` (which is the default value). Note that user's home directories are not always in `/home` on compute clusters, so be sure to check this.

Job management is configured under the `jobs` key. Here too a protocol may be given, as well as a location, and a few other settings can be made.

For job management, the protocol can be `local` (default) or `ssh`. If the `local` protocol is selected, `location` is ignored, and jobs are run locally. For the `ssh` protocol, `location` is the name of the host, optionally followed by a colon and a port number (e.g. `example.com:2222`).

Jobs can be run directly or via a scheduler. To run jobs directly, either on the local machine or on some remote host via SSH, set the scheduler to `none`. Other valid values for `scheduler` are `slurm`, `torque` and `gridengine` to submit jobs to the respective job management system.

If jobs need to be sent to a particular queue, then you can pass the queue name using the corresponding option; if it is not specified, the default queue is used. If one or more of your steps start MPI jobs, then you may want to set the number of MPI slots per node via `slots-per-node` for better performance. If you need to specify additional scheduler options to e.g. select a GPU node, you can do so using e.g. `scheduler-options: "-C TitanX --gres=gpu:1"`. Ideally, it would be possible to specify this in the CWL file for the step, but support for this in CWL is partial and in-development, and Cerise does not currently support this. Users can specify the number of cores to run on using a CWL ResourceRequirement, but Cerise always allocates whole nodes. It therefore needs to know the number of cores in each node, which you should specify using `cores-per-node`.

Finally, `cwl-runner` specifies the remote path to the CWL runner. It defaults to `$CERISE_API_FILES/cerise/cwltiny.py`, which is Cerise's included simple CWL runner. `$CERISE_API_FILES` will be substituted for the appropriate remote directory by Cerise. See *Specialising Cerise* for more information.

Cerise will regularly poll the compute resource it is connected to, to check if any of the running jobs have finished. The `refresh` setting can be used to set the minimum interval in seconds between checks, so as to avoid putting too much load on the machine.

Credentials may be put into the configuration file as indicated. Valid combinations are:

- No credentials at all (for running locally)
- Only a username
- A username and a password
- A username and a certificate file
- A username, a certificate file, and a passphrase

If the credentials to use for file access and job management are the same, then you should list them under `credentials` and omit them in the other locations. If different credentials are needed for files and jobs, then a `credentials` block can be specified under `files` and `jobs` respectively. Credentials listed here may be overridden by environment variables, as described below.

2.3.2 Environment variables

Cerise checks a set of environment variables for credentials. If found, they override the settings in the configuration file. These variables are:

General credentials

- CERISE_USERNAME
- CERISE_PASSWORD
- CERISE_CERTFILE
- CERISE_PASSPHRASE

Credentials for file access

- CERISE_FILES_USERNAME
- CERISE_FILES_PASSWORD
- CERISE_FILES_CERTFILE
- CERISE_FILES_PASSPHRASE

Credentials for job management

- CERISE_JOBS_USERNAME
- CERISE_JOBS_PASSWORD
- CERISE_JOBS_CERTFILE
- CERISE_JOBS_PASSPHRASE

As in the configuration file, specific credentials go before general ones. Cerise will first try a specific environment variable (e.g. CERISE_JOBS_USERNAME), then the corresponding specific configuration file entry (under `jobs`), then a generic environment variable (e.g. CERISE_USERNAME), and finally the generic configuration file entry (under `credentials`).

It does this for each of the four credential components separately, then uses the first complete combination from the top down to connect:

- username + certfile + passphrase
- username + certfile
- username + password
- username
- <no credentials>

CHAPTER 3

Specialising Cerise

(This document assumes some knowledge of the Common Workflow Language. See the CWL User Guide for an introduction.)

Cerise works by letting users submit workflows to a REST API, and then executing those workflows on some HPC compute resource. Users submit CWL Workflow documents, which specify a number of steps to run. Step definitions are not submitted by the user, but are part of the Cerise service. The base Cerise service does not contain any steps, so those have to be added first, before Cerise can be used in a project. This process of adding steps is known as specialisation.

A Cerise specialisation is always specific to a particular project, and to a specific compute resource. The project determines which steps are available and what their inputs and outputs are. Since HPC resources are all different in terms of which software is installed and how things are best set up, the implementation of the steps will be different from machine to machine, and so a separate specialisation will have to be made for each one. (Hopefully Singularity will help, once it sees more use, and in the cloud you have more freedom of course, but we'll address the more difficult situation with traditional HPC resources here.)

To specialise Cerise, you will have to make a configuration file that describes how to connect to a compute resource, design steps, and implement them. The easiest way to use the specialisation is to wrap it up into a Docker container together with Cerise, so you'll have a ready-to-run service.

These steps are described in more detail below, using an example that you can find in docs/examples/specialisation in the source distribution. Note that you'll need Docker installed, if you don't have it yet, see the Docker Community Edition documentation on how to install it.

3.1 The API configuration file

The main directory of a specialisation is called `api/`, because a specialisation specifies what the CWL API looks like and how it's implemented. Inside of this directory is a configuration file named `api/config.yml`. The example's configuration file looks like this:

Listing 1: docs/examples/specialisation/api/config.yml

```
compute-resource:
  files:
    protocol: local
    path: /home/cerise/.cerise

  jobs:
    protocol: local

  refresh: 0.1
```

An API configuration file is a YAML file describing the compute resource to run on. It has a single top-level key *compute-resource*, with below it keys for *files*, *jobs* and *refresh*. There are many other options (see *Compute resource configuration*), what you see here is the simplest possible configuration.

Cerise needs two things from the compute resource it runs on: a place to store files (including workflows, steps, input and output, as well as any files that are installed by this specialisation), and a way of submitting jobs. The files are stored in a directory on the remote system, given by *path* under *files*. In this case, we'll put them into */home/cerise/.cerise*, and we recommend using a *.cerise* directory in the user's home directory. (This is fine even if your user is also using a different specialisation that uses the same *.cerise* directory.)

The other thing Cerise needs is a way to get files into and out of that directory, a *protocol*. In this case, we're using the file system inside of the Docker container, so *local* is appropriate.

Next, we need to specify how to run jobs on the machine. This requires a *protocol* key again, this time specifying the protocol to use to connect to the machine when submitting jobs. We again use *local* here. Since there's no scheduler (like Slurm or Torque, as you'd find on a typical compute cluster) inside of the Docker container, we don't specify one. If no scheduler is specified, Cerise will start jobs directly on the machine, in this case inside of the Docker container.

Finally, we specify a refresh interval in seconds, which tells Cerise how often to check on the progress of running jobs. When you're submitting long-running jobs to a cluster or HPC machine, something like half a minute or a minute is appropriate. That will keep Cerise from hammering the head node with requests, and for a long-running job it's okay if it takes a bit longer for Cerise to realise that the job is done. For our testing purposes it's nice to have a result quickly, and since we're running locally we're not bothering anyone by updating more often, so in the example the refresh interval is set to 0.1 second.

3.2 The Dockerfile

The easiest way to run Cerise is inside of a Docker container. Note that this does not mean that you need Docker support on the compute resource. Cerise runs locally inside the container, and connects to the resource to run workflows there. Docker containers are built using a Dockerfile, which describes the steps needed to install the needed software and set up the system.

Cerise comes with its own Docker image, which is available on Dockerhub, and all that's really needed to specialise it is to copy your API into it. So all our Dockerfile needs to do is to start with the Cerise image, and copy the `api/` dir inside:

```
FROM cerise

COPY api /home/cerise/api
```

To build the test image, make sure you are inside the *specialisation* directory, then type:

```
docker build -t cerise-example .
```

This will build a new Docker image with your API inside it and name it *cerise-example*.

It's possible to test the image by starting it manually and making HTTP requests to it using *curl*, but it's much easier to use Python and Cerise Client. We'll make a virtualenv and install Cerise Client in it first:

```
virtualenv -p python3 ./env
. ./env/bin/activate
pip install cerise_client
```

Next, you can start Python 3 interactively, or make a simple Python script that starts a Cerise service and runs a job:

Listing 2: `docs/examples/specialisation/test_script.py`

```
#!/usr/bin/env python3

import cerise_client.service as cs
import time


srv = cs.require_managed_service('cerise-example-test', 29593, 'cerise-example')

# If your specialisation is not working, the following will print the
# Cerise server log. That should give you an error message describing
# what went wrong.
# print(srv.get_log())

job = srv.create_job('test_job')
job.set_workflow('test_workflow.cwl')
job.run()

while job.is_running():
    time.sleep(0.1)

# If something goes wrong executing the job, this will print the job log.
# print(job.log)

print(job.outputs['output'].text)

srv.destroy_job(job)
cs.destroy_managed_service(srv)
```

In this example, we run a test workflow that looks like this:

Listing 3: `docs/examples/specialisation/test_workflow.cwl`

```
#!/usr/bin/env cwl-runner

cwlVersion: v1.0

class: Workflow

inputs: []
```

(continues on next page)

(continued from previous page)

```
outputs:
  output:
    type: File
    outputSource: hostname/output

steps:
  hostname:
    run: cerise/test/hostname.cwl
    in: []
    out:
      - output
```

This workflow runs a single step, the `cerise/test/hostname.cwl` step, which is built in to the base Cerise image. This step takes no inputs, and produces a single output named *output*, a file containing the output of the *hostname* command. The test script prints its contents, which will be something like *aad7da47e423*, because Docker containers by default have a random host name that looks like that, and we're running inside of the container.

3.3 Adding steps

So far, our specialisation only has a configuration file. For an actual project, you will want to add one or more steps for running the software you need. Here, we'll add two steps, a very simple one that runs a program that is already present on the target machine, and a more complex one that requires installing some software remotely.

The Cerise API is organised by project. A project is simply a collection of CWL steps, plus the additional files needed to make them work, wrapped up in a directory. If multiple people work on specialisations for the same project, then they'll have to coordinate their efforts in order to avoid messing up each other's work, but developers on different projects can do their own thing without getting into each other's way. (Even if they use the same remote working directory. You can run two specialisations for different projects and the same machine simultaneously on the same account.)

While it's possible to have multiple projects in a single specialisation, for example in a Cerise-as-infrastructure case where you have a single Cerise instance with a cluster behind it and multiple users that want to do different things, in most cases you'll have only one project per specialisation. So that is what we'll assume here.

3.3.1 A simple step

Let's start with a simple step that returns the hostname of the machine it is running on, like before, but this time as part of our own project, named `example`. We'll add a new CWL file to the specialisation, in `api/example/steps/example/hostname.cwl`:

Listing 4: `docs/examples/specialisation/api/example/steps/example/hostname.cwl`

```
#!/usr/bin/env cwl-runner

cwlVersion: v1.0

class: CommandLineTool
baseCommand: hostname

inputs: []
```

(continues on next page)

(continued from previous page)

```
stdout: output.txt
outputs:
  output:
    type: File
    outputBinding: { glob: output.txt }
```

Since the *hostname* command is available on any Linux machine, we don't need to do anything else for this step to work. To call it, just modify *test_workflow.cwl* to use `example/hostname.cwl` instead of `cerise/test/hostname.cwl`. Note that the name to use in the workflow is the path relative to your *steps/* directory.

This means that the submitted workflow does not contain the full path to the step. This is good, because the full path changes per machine and per user (who each have a home directory with a different name), and besides it being annoying for the user if they have to look up the full path every time they write a workflow, it would mean that workflows become machine- and user-specific. This is something that Cerise is designed to avoid.

Instead, Cerise expects the user to give a relative path starting with the name of the project. When it copies the user's workflow to the compute resource in preparation for running it, Cerise extracts the project name from the first part in the path, and then prepends the path with the absolute path to the remote *steps/* directory for that project. As a result, the CWL runner that executes the workflow can find the steps. It only works however if your steps are in `<project>/steps/<project>/`, so be sure to follow that pattern!

3.3.2 How Cerise installs the API

In order to find out how to set up more complex steps that require software installation, it's good to know a bit about how Cerise installs your API on the compute resource.

When Cerise is started, it logs in to the configured compute resource, and checks to see if the API has been installed there already. If not, it will create the configured base file path (from your *config.yml*), create an *api/* directory inside it, and copy your project directory into that.

While copying the steps, Cerise will replace any occurrence of *$CERISE_PROJECT_FILES* in the *baseCommand* or *arguments* with the location of your *files/* directory. This allows you to run programs in your files directory, or pass locations of files in your *files/* directory to programs that you run.

After the steps and the files are copied, Cerise will check whether a file named *install.sh* exists in your *files/* directory. If it does, Cerise will run it remotely. This script (or whatever you put there) will run with an environment that has *$CERISE_PROJECT_FILES* set. It's probably a bad idea to modify anything outside of the *files/* directory from this script, so don't do that (if you have a good reason to do so, we'd love to hear from you, please make an issue on GitHub!).

3.3.3 Debugging a specialisation

The new steps you're adding will likely not work immediately. Just like with any kind of programming or configuring, it usually takes a few tries to get it right. If there is something wrong with your install script or your steps, then it may happen that Cerise fails to start. In this case, no jobs can be submitted, and you need the server log to figure out what's going on. You'll find a few commented-out lines in *test_script.py* that print the server log for you. You can also get to the file by hand, provided that the container still exists, using the command

```
docker cp cerise-example-test:/var/log/cerise/cerise_backend.log .
```

This will copy the log file `cerise_backend.log` to your current directory, where you can open it to have a look (it's plain text).

If you want to have a look around inside the running container, do

```
docker exec -ti cerise-examples-test bash
```

If the test script has stopped or crashed, and the container is still running, then you will want to stop the container and remove it, before rebuilding it and trying again. You can do that using

```
docker stop cerise-example-test
docker rm cerise-example-test
```

Finally, if the service starts correctly, but something goes wrong with running the workflow, then you can request the job log to get an error message. There's another commented section in *test_script.py* showing how.

3.3.4 A more complex step

Our second step will run a custom script that will be uploaded by Cerise. The CWL step looks like this:

Listing 5: `docs/examples/specialisation/api/example/steps/example/custom_program.cwl`

```
#!/usr/bin/env cwl-runner

cwlVersion: v1.0

class: CommandLineTool
baseCommand: $CERISE_PROJECT_FILES/add_heading.sh
arguments: [$CERISE_PROJECT_FILES]

inputs:
  in_text:
    type: File
    inputBinding:
      position: 1

stdout: output.txt
outputs:
  out_text:
    type: File
    outputBinding: { glob: output.txt }
```

Note how it uses *$CERISE_PROJECT_FILES* to refer to the *files/* directory for both the executable to be run, and for its first argument. The script itself is in the *files/* directory of course. This is a plain bash script that concatenates two files together:

Listing 6: `docs/examples/specialisation/api/example/files/add_heading.sh`

```
#!/bin/bash

CERISE_PROJECT_FILES="$1"
INPUT_FILE="$2"

cat $CERISE_PROJECT_FILES/heading.txt $INPUT_FILE
```

Of course, you can put anything in here, including say compiled binaries for the machine you're specialising for. A bit of experience: it seems to happen fairly often that you end up writing a shell script which is called from a step, and those shell scripts can have many parameters. If it's more than 9, make sure to use ${10} rather than $10, because the latter will take the first argument and append a 0.

When uploading, Cerise will copy permissions along, so that executable files will remain executable and private files will remain private. Unfortunately, there is a permission issue with Docker: when copying your API into the Docker image, Docker will strip all permissions. In the Dockerfile for the example, we manually make *add_heading.sh* executable again, but for more complex sets of files this gets tedious to make and maintain. In that case, it's probably better to create an archive from the *api/* dir, and use the *ADD* command in the Dockerfile to extract it into the container.

Listing 7: `docs/examples/specialisation/Dockerfile`

```
FROM cerise

COPY api /home/cerise/api
RUN chown -R cerise:cerise /home/cerise/api && \
    chmod +x /home/cerise/api/example/files/add_heading.sh
```

There's one thing missing in the above, the actual heading file, which *add_heading.sh* expects to find at *$CERISE_PROJECT_FILES/heading.txt*. Of course we could just put a *heading.txt* into the *files/* directory, but here we have the install script create the heading file just to show how that works:

Listing 8: `docs/examples/specialisation/api/example/install.sh`

```
#!/bin/bash

echo '=== Here is a heading! ===' >$CERISE_PROJECT_FILES/heading.txt
```

Since this is not a tutorial on shell programming, we keep it simple, printing a one line message to the required file. Note that *$CERISE_PROJECT_FILES* is automatically defined here. In practice, this script can be much more complex, installing various libraries and programs as needed by your steps.

3.4 Alternatives for installing software

If the program you want to run is available by default on the compute resource, then running it is as simple as providing an appropriate CWL CommandLineTool definition for it, as we did with *hostname*. Often however, you'll need to do a bit or a lot of work to get there. There are at least four ways of making the program you need available on the compute resource you are specialising for:

1. Leave it to the user
2. Ask the system's administrator to install the software for you
3. Install it yourself where others can access it
4. Have Cerise stage the necessary files

Option 1 is the easiest of course, but you'll have to provide very precise instructions to your users to ensure that they'll install the software exactly where your step is expecting it. Also, the users may not be very happy about having to jump through a bunch of hoops before being able to run their calculations.

Option 2 is also pretty easy, but it may take a while for the system administrator to get to your request, and they may refuse it for some reason. If your request is granted, the software will typically be installed as a module, so you'll need a `module load` command to make it available. The best solution for this seems to be to stage a small shell script that does just that, and then calls the program, passing on any arguments.

It would be nice if a CWL SoftwareRequirement could be used here to specify which modules to load. Support for SoftwareRequirements in `cwltool` is still in beta however, and `cwltiny` (Cerise's internal runner) does not support it at all yet.

Option 3 can work if you have enough permissions, but has the downside that the existence of the installation will probably depend on the existence of your account. If your account is deleted, your users' services will be stuck without the software they need.

Option 4 is what we did above. It's not any more work than option 3, but makes the installation independent from your involvement; if you put your specialisation in a public version control repository, then anyone can contribute. At the same time, you don't depend on external system administrators' whims, or on your users having a lot of knowledge of HPC.

3.5 Versioning

Your specialisation is effectively a library that gets used by the workflows that your users submit. Like with any library, it is therefore a good idea to put it into a version control system, and to give it a version number that changes every time you change the steps, using semantic versioning. In fact, Cerise requires a version number.

If that sounds complicated and you want a simple way to get started that won't cause problems in the future, just put your steps into `myproject/steps/myproject/` and put a single line `0.0.0.dev` in the `myproject/version` file. Whenever you add or change a step, increment the second number in the `version` file by one (going from 0.9.0 to 0.10.0 and beyond if necessary).

The `.dev` part will make Cerise reinstall your API every time it starts. Note that that means that it will wait for running jobs to finish, reinstall the API, then start running newly submitted jobs. This is very useful for development and debugging, but not when you're running longer jobs, so in that case you will want to remove the `.dev` postfix. If there is no `.dev` at the end of the version number, Cerise will only reinstall if the version of the API on the compute resource is lower than the local one, so be sure to increment the version number if you make changes, otherwise you'll end up mixing different versions, and that will probably end badly.

The only issue with this simple solution is that if you change a step in an incompatible way (for example, when you change the name or the type of an input or an output), your users' workflows will break. If it's just one or two people, you can sit down with them and help them modify their workflows and then upgrade everything at a single point in time, but if you have many users or many workflows, then you have to either avoid this situation (by making new steps rather than changing existing ones), or communicate it clearly.

If you're not the only one using your specialisation, then you should make sure that they know what they can expect. Semantic Versioning is a standardised way of doing this. A semantic version consists of three numbers, separated by periods, and an optional postfix. The first number is incremented when an incompatible change is made, i.e. one that may break things (workflows) that depend on the versioned object (the steps). The second number is incremented when new functionality is added in such a way that existing workflows keep working, and the third number is incremented when there is no new functionality, e.g. for bug fixes. The only postfix supported by Cerise is `.dev`, as described above. Furthermore, there is a general rule that for version numbers starting with 0, anything goes, and there are no guarantees.

To use semantic versioning, put a notice in your documentation saying that you're doing so, and whenever you make changes, update the version number according to the above rules, both in the `version` file and in the documentation. Now, if the users see that you've released a new major version (e.g. 2.0.0), they'll know that their workflow may break.

If you want to be really fancy and expect your project to live for a long time and have many users, then you can version your steps API. You do this by putting your steps in a directory `myproject/steps/myproject/1/step.cwl`. Now if you want to make incompatible changes to your step, you can leave it in place, but make a new version of it at `myproject/steps/myproject/2/step.cwl`. As long as you maintain the old versions, all workflows will keep working. If at some point you want to stop supporting old steps, you can remove them, but be sure to update your major version when you do so, because that can break existing workflows.

3.6 Making a step template

Once you have a specialisation with a few steps and an implementation for your favourite compute resource, you may want to support other machines as well. To do this, you'll need other specialisations, of the same project but for different machines. Also, some way of keeping them in sync is a very good idea, to ensure that any workflow designed for your project will run on any of the specialisations.

The best way to do this is to make a step template. A step template is basically just the steps directory of your specialisation, but containing partially-defined steps. The steps are partially-defined because exactly how a program is executed depends on the machine on which it's running, and since the step template is the same for all specialisations, we don't know that yet. What is important is that the inputs and outputs of each step are defined, that there is a description of what it does exactly, and perhaps you can already specify how to build the command line arguments from them.

The recommended layout of a step template is this:

```
myproject
├── steps
│   └── myproject
│       ├── step1.cwl
│       └── step2.cwl
└── version
```

The `version` file will contain the version of the step template. This will be a two-place version, `major.minor`, where the major number is incremented when there are incompatible changes to the step definitions, and the minor number is incremented when there are compatible additions. There is no third number, because there is no implementation to patch.

New specialisations can now start from this step template, and add a third number to their version. Every time the implementation changes, but the step definitions remain the same, only the third number is incremented. To change the step definitoins, you change the template, increment its first or second number, then update the specialisations to match, resetting them to `x.y.0`, where `x.y` comes from the step template.

If you're now thinking that all this stuff is a bit complicated, well, unfortunately, it can be. When you make a Cerise specialisation, you're making software for others to use, and that's always a bit more complex than making something only for yourself. On the other hand, if you have only a single specialisation and set the version to `0.0.0.dev`, then to `0.1.0` once you're more or less done, then you can still use Cerise just fine by yourself without ever thinking about versions. So how complex it gets depends on how many features you want.

3.7 Remote execution

In the above example, we have set up Cerise to run in a Docker container, and to execute jobs inside of the container. One way of using Cerise is to set it up this way, then run the container on a compute server, and have users connect to the REST API to submit and retrieve jobs. However, chances are you'll want to use a compute cluster or supercomputer that you cannot just install software on. In that case, it's better to run the Cerise container on your local machine, and configure it to talk to the compute resource via the network. Here is an example of such a configuration:

Listing 9: `docs/examples/specialisation/config_remote.yml`

```
compute-resource:
  refresh: 30
  files:
    protocol: sftp
```

(continues on next page)

(continued from previous page)

```
    location: fs0.das5.cs.vu.nl
    path: /home/$CERISE_USERNAME/.cerise

  jobs:
    protocol: ssh
    location: fs0.das5.cs.vu.nl
    scheduler: slurm
```

This configuration is for the DAS-5 supercomputer, a development system in The Netherlands. We connect to it via SSH for starting jobs, and SFTP for file transfer. Both protocols require a location to connect to. The remote path contains another special string, $CERISE_USERNAME, which gets substituted by the user name used to connect to the cluster. This way, each user will have their own directory in their own home directory for Cerise to use. For submitting jobs, the scheduler in use has to be specified, which is Slurm in case of the DAS-5. Cerise also supports Torque/PBS, for which you should specify `torque`. There are some more options in this file, for which we refer to the *Cerise Configuration*.

One thing should be pointed out here though: while it's possible to put credentials (e.g. usernames and passwords) in the configuration file, this is a really bad idea for a public multi-user system such as the DAS-5. If you're running your own cluster or compute server behind your firewall, and run a single instance of the specialisation that connects to the compute resource using a special account, then you can use that functionality, but for a public machine, this is a really bad idea, and almost always against the terms of service. Instead, every user should start their own instance of the specialisation, which runs on their behalf, with their credentials. Those can be injected into the Docker container via environment variables, and Cerise Client will do this automatically. See the Cerise Client documentation for how to do that.

CHAPTER 4

Developer documentation

4.1 Releases

Cerise uses Git on GitHub for version management, using the Git Flow branching model. Making a release involves quite a few steps, so they're listed here to help make the process more reliable. Cerise is not yet on PyPI, so for now it's only a Git branch and a Docker image on DockerHub that are involved.

4.1.1 Make release branch

To start the release process, make a release branch

```
git checkout -b release-x.y.z develop
```

Cerise uses Semantic Versioning, so name the new version accordingly.

4.1.2 Update version

Next, the version should be updated. There is a version tag in `setup.py` and two for the documentation in `docs/source/conf.py` (search for `version` and `release`). On the development branch, these should be set to `develop`. On the release branch, they should be set to `x.y.z` (or rather, the actual number of this release of course).

4.1.3 Check documentation

Since we've just changed the documentation build configuration, the buil should be run locally to test:

```
make docs
```

It may give some warnings about missing references; they should disappear if you run the command a second time. Next, point your web browser to `docs/build/index.html` and verify that the documentation built correctly. In

particular, the new version number should be in the browser's title bar as well as in the blue box on the top left of the page.

4.1.4 Run tests

Before we make a commit, the tests should be run, and this is a good idea anyway if we're making a release. So run `make test` and check that everything is in order.

4.1.5 Commit the version update

That's easy:

```
git commit -m 'Set release version'
git push
```

This will trigger the Continuous Integration, so check that that's not giving any errors while we're at it.

4.1.6 Merge into the master branch

If all seems to be well, then we can merge the release branch into the master branch and tag it, thus making a release, at least as far as Git Flow is concerned.

```
git checkout master
git merge --no-ff release-x.y.z
git tag -a x.y.z
git push
```

4.1.7 Add a Docker Hub build

Finally, we need Docker Hub to build a properly tagged Docker image of the new release. To get it do do this, follow these steps:

- Log in to Docker Hub
- Go to Organizations
- Go to `mdstudio/cerise`
- Go to Build Settings
- Add a new Tag-based build with the `x.y.z` tag you just made

Next, pull the image to check that it works:

```
docker pull mdstudio/cerise:x.y.z
```

4.2 Requirements

4.2.1 Introduction

This document describes the requirements placed on Cerise. The aim of this project is to implement all these features. Most of them are currently there, but Cerise is not yet completely done (and new features will probably keep coming

up).

4.2.2 Overview

Cerise provides a REST interface through which CWL workflows can be submitted for execution on a compute resource. The particulars of the compute resource are configurable, with support for local execution as well as SSH-accessed remote machines, and compute clusters using SLURM or TORQUE as their resource manager. There will not be complete support for all possible CWL workflows.

The requirements below are categorised using the MoSCoW system: as either a Must have, Should have, Could have or Won't have requirement.

4.2.3 Functionality

4.2.3.1 User side functionality

- Job management
  - [M] A user can submit a CWL workflow for execution using the Netherlands eScience Center version of the GA4GH workflow execution schema (the REST API).
  - [M] Multiple workflows can be submitted and executing at the same time.
  - [M] The service will execute the submitted workflow on a compute resource.
  - [M] The status of the job, and eventual results of the workflow will be available through the REST API.
  - [M] Running jobs can be cancelled (aborted).
  - [M] Results may be left on the compute resource by copying them to a storage location on the compute resource using explicit commands that are part of the CWL workflow.
  - [M] It must be easy to do the above things from a Python program.
  - [W] Only a subset of CWL files will be supported. The exact subset is currently undefined, but custom input and output types will not be supported.
  - [W] Inputs and outputs are small, on the order of megabytes, not gigabytes. The system does not have to support parallel up/downloads, or ones that take days to complete.
- Workflow definition
  - [M] The workflow will be defined in domain terms, not in terms of command line statements, core hours, or other low-level technical constructs. Different runs requiring different amounts of resources will be dealt with by offering steps for different scenarios, e.g. "run an LIE simulation efficiently using Gromacs for a protein in water" or "run an LIE simulation quickly using Gromacs for a protein in water" or "run a very long simulation of a protein using Gromacs". Thus, submitted workflow definitions are not compute resource specific.
  - [M] Steps for specific scenarios can be configured into the software without changing the software itself.
  - [M] It must be possible to specify multiple related files as an input (secondaryFiles in CWL terms), to have array-valued inputs (including arrays of files), and to have optional inputs and outputs (with default values for inputs).

4.2.3.2 Computing

- Workflow execution
  - [C] Serial execution of workflow steps is acceptable, with parallelism achieved through running many jobs. However, as there are typically limits to how many jobs can be submitted to a scheduler, parallel execution within a workflow would be nice to have.
  - [C] On busy compute resources, queueing times can be long. Support for something like STOPOS (if available, such as on SURFsara Lisa) or pilot jobs could be considered.
- Configuration
  - [M] Configurations may (or should) be compute-resource specific. We will not attempt grid-style resource abstraction, but instead rely on an administrator or developer to set up steps for each compute resource.
  - [M] It must be possible to configure the service to select one compute resource on which submitted CWL workflows are to be run.
  - [M] Given a desired set of workflow steps, it should be easy to configure these into the service. No superfluous hoops to jump through, and good documentation.
  - [M] It should be possible to share configuration that is the same for different compute resources, to avoid duplication.
- Resource type support
  - [M] Remote execution on a compute resource using either SLURM or TORQUE must be supported.
  - [M] It must be possible to select specific resources within a cluster, e.g. GPUs, by submitting to a particular queue, or by specifying a resource constraint. These constraints are to be specified in a step.
  - [S] Remote machines accessible through SSH should be supported
  - [S] Local execution (on the machine the service runs on) should be supported.
  - [S] With an eye towards the future, cloud resources should be supported.
  - [C] With an eye towards the past, grid resources could be supported.
  - [W] It would be nice to be able to distribute work across all available resources, but this is better done in a front-end accessing multiple resources, rather than by having a single service do both access and load balancing.

4.2.3.3 Deployment

- At least the following deployment configurations must be supported:
  - [S] Client, this service, and workflow execution all on the same machine
  - [S] Client on one machine, service and execution on another, where the client can connect to the service, but not vice versa.
  - [M] Client, this service, and workflow execution each on a different machine or resource, where the client can connect to the service, and the service can connect to the compute resource, but no other connections are possible.

4.3 Design overview

This file describes the high level design of Cerise, covering the architecture, functionality,and behaviour of the system. The general principle of operation is to run the standard CWLTool on some remote resource, feeding it the job given

to the service by the user. To allow the job to run correctly, input must be staged to the remote resource, and output must be staged back to some location where the user of the service can access it.

As the implementation is not yet complete, this is a description of how I'd like it to eventually be, not of how it currently is (although most pieces are already in place).

4.3.1 Architecture

Cerise has a simple architecture.

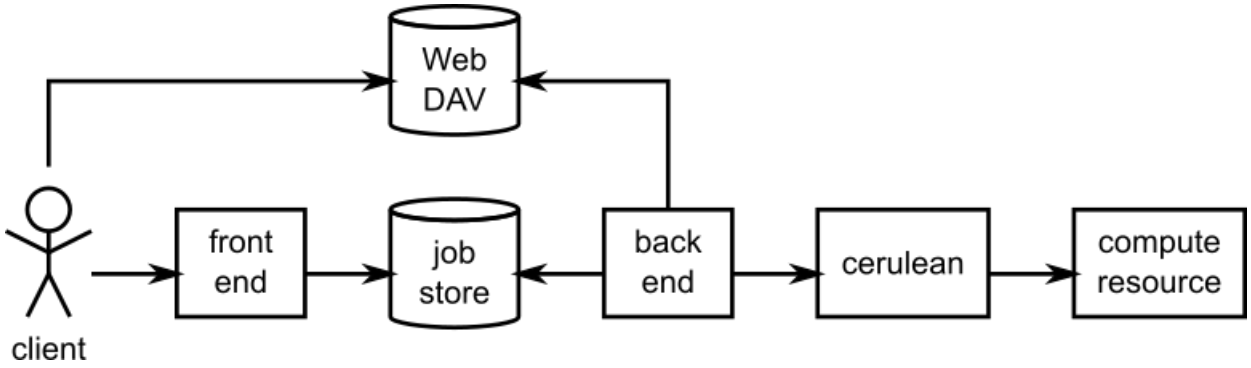

Fig. 1: Architecture of Cerise. Arrows show the direction of calls.

The front end of the system is provided by server-side Python bindings for the REST API, as generated by Swagger (which uses connexion). In the middle is the job store, in which the currently known jobs are registered, and which takes care of synchronisation between the front end and the back end. The back end takes care of staging and running jobs, using the Cerulean library to connect to the compute resource.

4.3.2 Functionality

The front end is mostly generated code. It takes requests from the client, and calls the job store to add or change jobs, or obtain their current status. (See below under *Behaviour*.)

The job store is a simple SQLite database that stores the list of jobs that are currently known to the system. It implements the basic create-read-update-delete cycle for jobs. A job in the Job Store holds all the available information about the job, with the exception of the input and output files, which are stored on disk or in a separate WebDAV service.

The back end comprises four components: Local Files, Remote Files, the Job Runner, and the Execution Manager.

The Local Files component manages the local storage area. This local storage area is used for communicating files with the client. Before submitting a job, the client may upload or copy a file to this area, and then pass a file:// URL to the service referring to it. Alternatively, http:// URLs may be used, which LocalFiles can also access. The local storage area may be a directory on a local file system, or a directory on a WebDAV.

Local Files contains functionality for opening input files for staging, creating directories for job output, and publishing job output.

The Remote Files component manages a remote storage area, presumably on the compute resource, or at least accessible from there. This is used for communication with the compute resource. The remote storage area is simply a directory on a file system that is accessible through any of the Cerulean-supported access methods. Inside this directory, Remote Files keeps one directory per job. It will stage input there, and retrieve output from there. Remote Files also contains functionality for interpreting the job's output, and updating the state of the job based on this.

The Job Runner component contains the functionality for starting jobs on the compute resource, getting the current status (waiting to run, running, done) of remote jobs, and for cancelling them. It does not interpret the result of the job, and cannot tell whether it completed successfully or not, it only knows whether a job is running or not.

The Execution Manager contains the main loop of the back end. It calls the other components to stage in submitted jobs, start jobs that are ready to run, monitor their progress, destage them when done, and cancel or delete them on request.

4.3.3 Behaviour

Jobs in Cerise go through a sequence of operations as they are being processed. The behaviour of the system as it processes a job can be described as executing the following state machine:

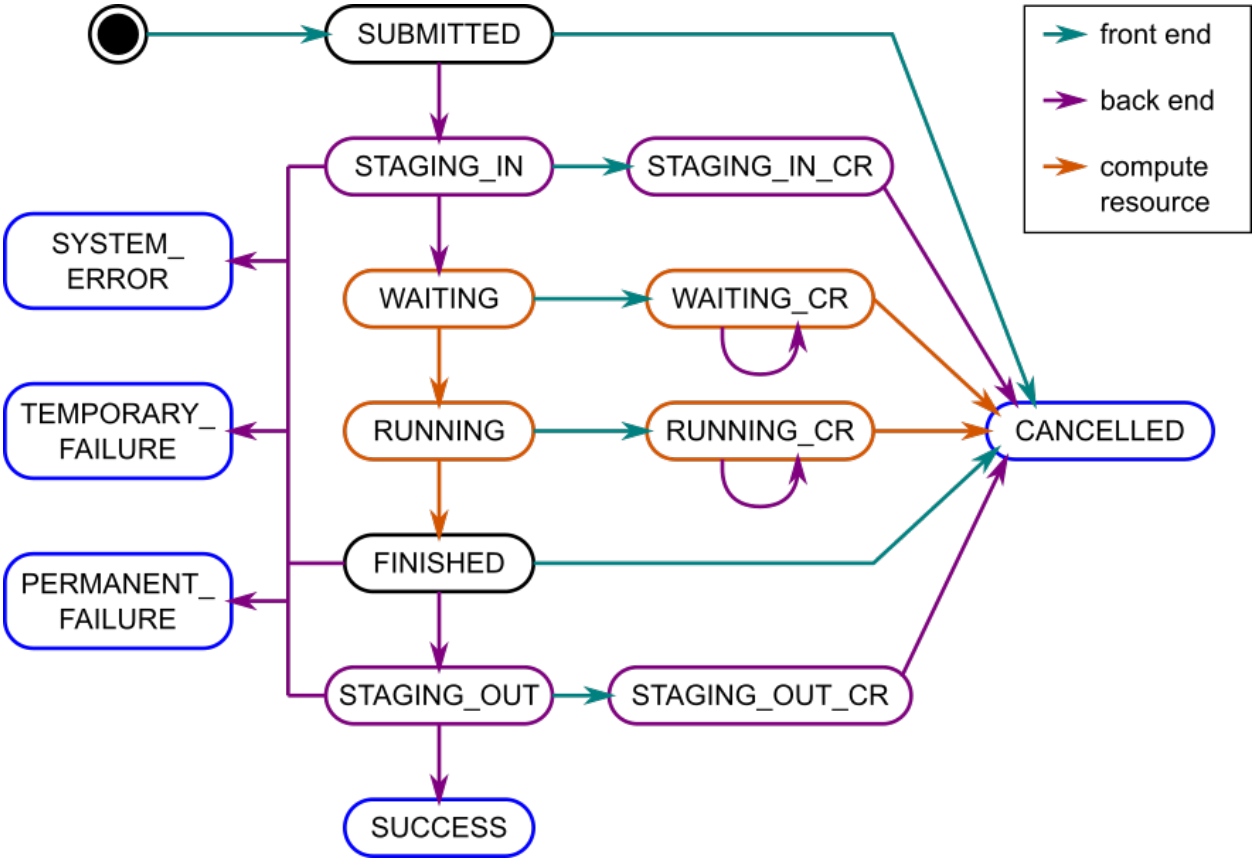

Fig. 2: Internal job states and components that act on them. Black and blue states are rest states, blue states are final states, in purple states the back end is active, and in orange states the compute resource is active (and being observed by the back end).

This state machine is not only used to keep track of where a job is in the process of being executed, but also as a way of synchronising between different threads of execution within the service, through atomic state transitions. More on this below.

In total, there are fifteen internal states that a job may be in:

State	Type	CWL State	Category
SUBMITTED	Rest	Waiting	
STAGING_IN	Active	Waiting	
WAITING	Remote Active	Waiting	
RUNNING	Remote Active	Running	
FINISHED	Rest	Running	
STAGING_OUT	Active	Running	
SUCCESS	Rest	Success	Final
STAGING_IN_CR	Active	Waiting	Cancellation pending
WAITING_CR	Remote Active	Waiting	Cancellation pending
RUNNING_CR	Remote Active	Running	Cancellation pending
STAGING_OUT_CR	Active	Running	Cancellation pending
CANCELLED	Rest	Cancelled	Final
PERMANENT_FAILURE	Rest	PermanentFailure	Final
TEMPORARY_FAILURE	Rest	TemporaryFailure	Final
SYSTEM_ERROR	Rest	SystemError	Final

4.3.3.1 Normal execution

When a job is submitted to the system, an entry is added to the job store representing the job, in state SUBMITTED. The job is then moved to the STAGING_IN state, and staging in commences. In general this may take a while, depending on network speeds and data volumes. Once all files are copied, the job is submitted to the remote resource, and the job is moved into the WAITING state.

At some point, resources will become available at the remote compute resource, and the job is started, putting it into the RUNNING state. When it stops running, it moves on to FINISHED, from where the service will move it into STAGING_OUT and start the staging out process. When that is complete, and assuming all went well, the job ends up in stage SUCCESS.

While the job is being processed, the user may request its status via the REST API. The REST API defines a more limited set of states, to which the internal states are mapped (third column in the table). The mapping is such that the Success state signals that the job finished successfully and results are available, while Waiting and Running signal that the user will have to wait a bit longer.

4.3.3.2 Cancellation

If the user submits a cancel request for a job, processing needs to be stopped. How this is to happen depends on the current state of the job. If the state is a Rest state (second column, black and blue in the diagram), then it is not actively being processed, and it can simply be moved to the CANCELLED state.

If the job is in an Active state (purple in the diagram), it is moved to the corresponding _CR state, processing is stopped, and it is then moved to the CANCELLED state (this to synchronise front end and back end, see below). If it is in a Remote Active state (orange in the diagram), it is moved to the corresponding _CR state, and a cancellation request is sent to the compute resource (purple circular arcs). Once the compute resource has stopped the job, it moves into the CANCELLED state.

Note that all activities done by the remote compute resource are observed by the service's back end, and any state changes are propagated to the service's job store periodically.

4.3.3.3 Errors

If an error occurs during processing, the job will be in an Active or Remote Active state (since in a Rest state nothing happens, and so nothing can go wrong).

During staging in, in state STAGING_IN, permanent errors may occur if an input file is not available (e.g. due to a mistyped URI). Temporary failures are also possible, e.g. if an http URI returns error 503 Resource Temporarily Unavailable. In this case, staging is aborted, and the job moved to the corresponding error state. If an internal error occurs (which it shouldn't, but no program is perfect) the job is put into the SYSTEM_ERROR state.

Unsuccessful workflow runs will result in a CWL error of type PermanentFailure or TemporaryFailure, as signalled by the remote CWL runner. Once a job is in the FINISHED state, this output will be examined, and if an error has occurred it will be moved into PERMANENT_FAILURE, or TEMPORARY_FAILURE as appropriate. If the remote CWL runner does not produce usable output, a SYSTEM_ERROR results.

If an error occurs during staging out, in state STAGING_OUT, then like for staging in, the process is aborted and the job moved into an appropriate error state (PERMANENT_FAILURE, TEMPORARY_FAILURE or SYSTEM_ERROR).

4.3.3.4 Service shutdown

The service may be shut down while it is processing jobs. If this happens, then the shutdown process must ensure that running activities are stopped, and that the jobs are put into a state from where processing may recommence when the service is started again. This is achieved as follows:

- For all jobs in the STAGING_IN state, staging is aborted, and the job is moved into the SUBMITTED state.
- For all jobs in the STAGING_OUT state, staging is aborted, and the job is moved into the FINISHED state.
- For all jobs in the STAGING_IN_CR state, staging is aborted, and the job is moved into the CANCELLED state.
- For all jobs in the STAGING_OUT_CR state, staging is aborted, and the job is moved into the CANCELLED state.

4.3.3.5 Service start-up

On service start-up, the jobs database is checked. If the service was shut down cleanly, all jobs will be in a Rest state, and the service may start up as normal and start processing.

If any jobs are found to be in an Active state, they will be moved to the corresponding Rest state as per the shutdown procedure above. If staging is idempotent (and they should be) this should allow the system to continue processing where it left off. Ideally, staging will check whether a file already exists on the target side, and not upload or download it a second time.

If any jobs are in WAITING_CR or RUNNING_CR and are still running, a cancellation request will be sent for them, as the service may have crashed after transitioning the state, but before sending the cancellation request, or the cancellation request may have failed for some other reason.

4.3.4 Multiprocess implementation

Since this is a web service, multiple clients may access it concurrently. Staging may take a significant amount of time, during which we would like to be able to service requests. Also, even for a single client, a job submission request should not have to wait for completion of staging in to return. Therefore, staging should be done in the background. Furthermore, the remote compute resource should be polled regularly to update the status of running jobs, so that their results can be staged out shortly after they are done.

The service therefore has a front end, which communicates with the user, and a back end, which does most of the work. In the diagram above, state transitions done by the front end are coloured teal, while the ones done by the back end are coloured purple. State transitions performed by the remote resource are coloured orange. These are observed by the back end, and propagated to the job store periodically, since the remote resource cannot access the job store.

4.3.4.1 Front end threads

Front end threads are responsible for state transitions that are made in response to user input. If a client submits a job, the job is created and put into the SUBMITTED state. If a cancellation request is received, and the job is in a Rest state, it will be moved into CANCELLED by the front-end thread. If it is in an Active state, it is moved into the corresponding _CR state (if not already there). If the job is in a Remote Active state, a cancellation request is sent to the remote resource, and the job is moved into the corresponding _CR state (also, if not already there).

Deletion requests are signalled from the front end to the back end via a separate job property, outside of the job state machine. A cancel operation is done first, then deletion is requested.

4.3.4.2 Back end threads

The back end is responsible for staging and job submission. It operates in a loop, finding a job in the SUBMITTED state, moving it into STAGING_IN, and starting the staging process. If during staging the job is moved into STAGING_IN_CR (by a front-end thread), staging is aborted, and the job is moved to CANCELLED. If a shutdown is signalled, staging is aborted and the job is moved back into SUBMITTED.

The back end also regularly polls the remote compute resource, requesting the status of running jobs. Any jobs in the WAITING state that according to the retrieved information are running, are moved into the RUNNING state. Jobs in WAITING_CR go to RUNNING_CR.

If a job is in a Remote Active state, but is found to no longer be running, then if it was in a Cancellation pending state (named _CR) it is moved to CANCELLED. Otherwise, the output is checked to see if the job was successful, and it is moved into an appropriate error state if it was not. If it was successful, is is put into FINISHED.

If the back end finds a job in the FINISHED state, it checks the result. If the job finished successfully, it moves it to the STAGING_OUT state and begins staging out the results. If during staging the job is moved into STAGING_OUT_CR, staging is aborted and the job is moved to CANCELLED. If a shutdown is signalled, staging is aborted and the job is moved back into FINISHED.

4.3.4.3 Synchronisation

To avoid data corruption, there must be a mechanism that keeps multiple threads from working on the same job at the same time. Also, we can't have multiple state transitions occurring at the same time and interfering with each other. Thus, there must be some synchronisation mechanism between the threads.

In the Rest states, no processing is done, and any thread can safely move the job to another state as long as the state transitions are atomic. This can be implemented in the form of a try_transition(from_state, to_state) -> bool function. If two threads try to transition a job simultaneously, one from A to B and the other from A to C, one will succeed, while the other will fail because its from_state does not match the current state. (A transactional system with optimistic concurrency control.)

Jobs are moved into Active states (STAGING_IN or STAGING_OUT) by the back end, which subsequently owns it until it moves it into another state. The only exception is that during this process, the job may be moved into STAGING_IN_CR or STAGING_OUT_CR by a front-end thread. Effectively, the state machine functions here as a compare-and-exchange based mutual exclusion mechanism.

4.3.5 Known issues/failure modes

If the service crashes or is killed while a job is being staged, and this happens just after submission of the job to the compute resource, but before the transition from STAGING_IN to WAITING, the job will be started again on start-up of the service. This may be undesirable; maybe the service could check as part of error recovery whether the job is already running, or has run anyway.

All synchronisation goes via a single job store component, which means that it may become a bottleneck. However, jobs only spend a fraction of their time in state transitions, jobs are independent of one another, and the total amount of data stored is small (kilobytes per job, at most), so this is unlikely to affect scalability.

4.4 Source code

4.4.1 cerise package

4.4.1.1 Subpackages

cerise.back_end package

Subpackages

cerise.back_end.test package

Submodules

cerise.back_end.test.conftest module

cerise.back_end.test.mock_job module

class cerise.back_end.test.mock_job.**MockJob**(*job_id*, *name*, *workflow*, *job_input*)

Bases: object

This class provides an in-memory implementation of a job.

It's used for testing, so that we don't need to bother with a database there.

Creates a new Job object.

The state of a newly created job is JobState.SUBMITTED.

Parameters

- **id** (*str*) – The id of the job, a string containing a GUID
- **name** (*str*) – The name of the job, as given by the user
- **workflow** (*str*) – The URI of the workflow file
- **job_input** (*str*) – An input definition for the job

add_log(*level*, *message*)

Add a message to the job's log.

Parameters

- **level** (*logging.LogLevel*) – Level of importance

- **message** (*str*) – The log message.

critical(*message*)
Add a message to the job's log at level CRITICAL.

Parameters **message** (*str*) – The log message.

debug(*message*)
Add a message to the job's log at level DEBUG.

Parameters **message** (*str*) – The log message.

error(*message*)
Add a message to the job's log at level ERROR.

Parameters **message** (*str*) – The log message.

id = None
Job id, a string containing a UUID.

Type str

info(*message*)
Add a message to the job's log at level INFO.

Parameters **message** (*str*) – The log message.

local_input = None
Input JSON string, as specified by the submitter.

Type str

local_output = None
The serialised JSON output object describing the destaged outputs.

Type str

log = None
Log output as of last update.

Type str

name = None
Name, as specified by the submitter.

Type str

please_delete = None
Whether deletion of the job has been requested.

Type bool

remote_error = None
cwl-runner stderr output as of last update.

Type str

remote_input_path = None
The absolute remote path of the input description file.

Type str

remote_job_id = None
The id the remote scheduler gave to this job.

Type str

remote_output = None
cwl-runner output as of last update.

Type str

remote_stderr_path = None
The absolute remote path of the standard error dump.

Type str

remote_stdout_path = None
The absolute remote path of the standard output dump.

Type str

remote_system_err_path = None
The absolute remote path of the standard error dump.

Type str

remote_system_out_path = None
The absolute remote path of the system output dump.

Type str

remote_workdir_path = None
The absolute remote path of the working directory.

Type str

remote_workflow_path = None
The absolute remote path of the CWL workflow file.

Type str

required_num_cores = None
The number of cores to reserve for this workflow. If 0, use cluster default.

resolve_retry_count = None
Number of times we've tried to resolve inputs.

Type int

state = None
Current state of the job.

Type *JobState*

time_limit = None
The time to reserve, in seconds. If 0, use cluster default.

try_transition(*from_state*, *to_state*)
Attempts to transition the job's state to a new one.

If the current state equals from_state, it is set to to_state, and True is returned, otherwise False is returned and the current state remains what it was.

Parameters

- **from_state** (JobState) – The expected current state
- **to_state** (JobState) – The desired next state

Returns True iff the transition was successful.

warning(*message*)
Add a message to the job's log at level WARNING.

Parameters **message** (*str*) – The log message.

workflow = None
Workflow file URI, as specified by the submitter.

Type str

workflow_content = None
The content of the workflow description file, or None if it has not been resolved yet.

Type Union[bytes, NoneType]

cerise.back_end.test.test_cwl module

cerise.back_end.test.test_job_planner module

cerise.back_end.test.test_job_planner.**test_job_planner_init**(*mock_config*, *mock_store_resolved*, *local_api_dir*)

cerise.back_end.test.test_job_planner.**test_plan_job**(*mock_config*, *mock_store_resolved*, *local_api_dir*)

cerise.back_end.test.test_job_runner module

cerise.back_end.test.test_local_files module

cerise.back_end.test.test_remote_api module

cerise.back_end.test.test_remote_job_files module

Module contents

Submodules

cerise.back_end.cwl module

cerise.back_end.cwl.**get_cwltool_result**(*cwltool_log: str*) → cerise.job_store.job_state.JobState
Parses cwltool log output and returns a JobState object describing the outcome of the cwl execution.

Parameters **cwltool_log** – The standard error output of cwltool

Returns Any of JobState.PERMANENT_FAILURE, JobState.TEMPORARY_FAILURE or JobState.SUCCESS, or JobState.SYSTEM_ERROR if the output could not be interpreted.

cerise.back_end.cwl.**get_files_from_binding**(*cwl_binding: Dict[str, Any]*) → List[cerise.back_end.file.File]
Parses a CWL input or output binding an returns a list containing name: path pairs. Any non-File objects are omitted.

Parameters **cwl_binding** – A dict structure parsed from a JSON CWL binding

Returns

A list of File objects describing the input files described in the binding.

cerise.back_end.cwl.**get_required_num_cores**(*cwl_content: bytes*) → int
Takes a CWL file contents and extracts number of cores required.

Parameters **cwl_content** – The contents of a CWL file.

Returns The number of cores required, or 0 if not specified.

cerise.back_end.cwl.**get_secondary_files**(*secondary_files: List[Dict[str, Any]]*) → List[cerise.back_end.file.File]
Parses a list of secondary files, recursively.

Parameters **secondary_files** – A list of values from a CWL secondaryFiles attribute.

Returns A list of secondary input files.

cerise.back_end.cwl.**get_time_limit**(*cwl_content: bytes*) → int
Takes a CWL file contents and extracts cwl1.1-dev1 time limit.

Supports only two of three possible ways of writing this. Returns 0 if no value was specified, in which case the default should be used.

Parameters **cwl_content** – The contents of a CWL file.

Returns Time to reserve in seconds.

cerise.back_end.cwl.**get_workflow_step_names**(*workflow_content: bytes*) → List[str]
Takes a CWL workflow and extracts names of steps.

This assumes that the steps are not inlined, but referenced by name, as we require for workflows submitted to Cerise. Also, this is not the name of the step in the workflow document, but the name of the step in the API to run. It's the content of the `run` attribute, not that of the `id` attribute.

Parameters **workflow_content** – The contents of the workflow file.

Returns A list of step names.

cerise.back_end.cwl.**is_workflow**(*workflow_content: bytes*) → bool
Takes CWL file contents and checks whether it is a CWL Workflow (and not an ExpressionTool or CommandLineTool).

Parameters **workflow_content** – a dict structure parsed from a CWL file.

Returns

True iff the top-level Process in this CWL file is an instance of Workflow.

cerise.back_end.execution_manager module

class cerise.back_end.execution_manager.**ExecutionManager**(*config: cerise.config.Config*, *local_api_dir: cerulean.path.Path*)
Bases: `object`

Handles the execution of jobs on the remote resource. The execution manager monitors the job store for files that are ready to be staged in, started, cancelled, staged out, or deleted, and performs the required activity. It also monitors the remote resource, ensuring that any remote state changes are propagated to the job store correctly.

Set up the execution manager.

Parameters

- **config** – The configuration.
- **local_api_dir** – The path to the local API directory.

execute_jobs() → None
Run the main backend execution loop.

This repeatedly processes jobs, but does not check the remote compute resource more often than specified in the remote_refresh configuration parameter.

shutdown() → None
Requests the execution manager to execute a clean shutdown.

cerise.back_end.file module

class cerise.back_end.file.**File**(*name: Optional[str], index: Optional[int], location: str, secondary_files: List[File]*)

Bases: object

Create a File object.

This describes a file, and is the result of resolving input files from the user-submitted input description, or output generated by the CWL runner. It is used by the staging machinery to stage these files, and update the input description with remote paths.

Parameters

- **name** – The name of the input for which this file is.
- **index** – The index of this file into an array of Files.
- **location** – A URL with the (local) location of the file.
- **secondary_files** – A list of secondary files.

index = None
The index of this file, if it is in an array of files.

location = None
Local URL of the file.

name = None
The input name for which this file is.

secondary_files = None
CWL secondary files.

source = None
The source of the file.

cerise.back_end.job_planner module

exception cerise.back_end.job_planner.**InvalidJobError**
Bases: RuntimeError

class cerise.back_end.job_planner.**JobPlanner**(*job_store: cerise.job_store.sqlite_job_store.SQLiteJobStore, local_api_dir: cerulean.path.Path*)

Bases: object

Handles workflow execution requirements.

This class keeps track of which hardware is needed for each available step, then analyses a workflow and decides which resources it needs based on this.

Create a JobPlanner.

Parameters

- **job_store** – The job store to act on.
- **local_api_dir** – Path of local api directory.

plan_job(*job_id: str*) → None
Figures out which resources a job needs.

Resources are identified by strings. Currently, there is `num_cores`, the number of cores to run on, and `time_limit`, the amount of time to reserve in seconds.

Parameters **job_id** – Id of the job to plan.

cerise.back_end.job_runner module

class cerise.back_end.job_runner.**JobRunner**(*job_store: cerise.job_store.sqlite_job_store.SQLiteJobStore*, *config: cerise.config.Config*, *remote_cwlrunner: str*)

Bases: `object`

Create a JobRunner object.

Parameters

- **job_store** – The job store to get jobs from.
- **config** – The configuration.
- **remote_cwlrunner** – The location of the CWL runner to use.

cancel_job(*job_id: str*) → bool
Cancel a running job.

Job must be cancellable, i.e. in JobState.RUNNING or JobState.WAITING. If it isn't cancellable, this function does nothing.

Cancellation may not happen immediately. If the cancellation request has been executed immediately and the job is now gone, this function returns False. If the job will be cancelled soon, it returns True.

Parameters **job_id** – The id of the job to cancel.

Returns Whether the job is still running.

start_job(*job_id: str*) → None
Get a job from the job store and start it on the compute resource.

Parameters **job_id** – The id of the job to start.

update_job(*job_id: str*) → None
Get status from compute resource and update store.

Parameters **job_id** – ID of the job to get the status of.

cerise.back_end.local_files module

class cerise.back_end.local_files.**LocalFiles**(*job_store: cerise.job_store.sqlite_job_store.SQLiteJobStore*, *config: cerise.config.Config*)

Bases: object

Create a LocalFiles object. Sets up local directory structure as well.

Parameters

- **job_store** – The job store to use
- **config** – The configuration.

create_output_dir(*job_id: str*) → None
Create an output directory for a job.

Parameters **job_id** – The id of the job to make a work directory for.

delete_output_dir(*job_id: str*) → None
Delete the output directory for a job. This will remove the directory and everything in it.

Parameters **job_id** – The id of the job whose output directory to delete.

publish_job_output(*job_id: str, output_files: List[cerise.back_end.file.File]*) → None
Write output files to the local output dir for this job.

Uses the .output_files property of the job to get data, and updates its .output property with URLs pointing to the newly published files, then sets .output_files to None.

Parameters

- **job_id** – The id of the job whose output to publish.
- **output_files** – List of output files to publish.

resolve_input(*job_id: str*) → List[cerise.back_end.file.File]
Resolves input (workflow and input files) for a job.

This function will read the job from the database, add a .workflow_content attribute with the contents of the workflow, and return a list of File objects describing the input files.

This function will accept local file:// URLs as well as remote http:// URLs.

Parameters **job_id** – The id of the job whose input to resolve.

Returns A list of File objects to stage.

resolve_secondary_files(*secondary_files: List[cerise.back_end.file.File]*) → None
Makes a File object for each secondary file.

Works recursively, so nested secondaryFiles work.

Parameters **secondary_files** – List of secondary files.

Returns Resulting Files, with contents.

cerise.back_end.remote_api module

class cerise.back_end.remote_api.**RemoteApi**(*config: cerise.config.Config*, *local_api_dir: cerulean.path.Path*)

Bases: object

Manages the remote API installation.

This class manages the remote directories in which the CWL API is installed, which is <basedir>/api/

Within this, there is a directory per project, with entries

<project>/version <project>/steps/... <project>/files/... <project>/install.sh

Create a RemoteApiFiles object. Sets up remote directory structure as well, but refuses to create the top-level directory.

Parameters

- **config** – The configuration.
- **local_api_dir** – The path to the local API dir to install from.

get_projects() → List[str]
Return names and versions of the installed projects.

Returns

A list of strings, one for each project, with name and version.

install() → None
Install the API onto the compute resource.

Copies subdirectories steps/ and files/ of the given local api dir to the compute resource, copies files/ to the compute resource, and runs the install script.

translate_runner_location(*runner_location: str*) → str
Perform macro substitution on CWL runner location.

This replaces $CERISE_API with the API base dir.

Parameters **runner_location** (*str*) – Location of the runner as configured by the user.

Returns (str) A remote path with variables substituted.

translate_workflow(*workflow_content: bytes*) → bytes
Parse workflow content, check that it calls steps, and insert the location of the steps on the remote resource so that the remote runner can find them.

Also converts YAML to JSON, for cwltiny compatibility.

Parameters **workflow_content** – The raw workflow data

Returns The modified workflow data, serialised as JSON

update_available() → bool
Returns whether the remote API is older than the local one.

Returns True iff an update is available/required.

cerise.back_end.remote_job_files module

class cerise.back_end.remote_job_files.**RemoteJobFiles**(*job_store: cerise.job_store.sqlite_job_store.SQLiteJobStore*, *config: cerise.config.Config*)

Bases: object

Manages a remote directory structure. Expects to be given a remote dir to work within. Inside this directory, it makes a jobs/ directory, and inside that there is a directory for every job.

Within each job directory are the following files:

- jobs/<job_id>/name.txt contains the user-given name of the job

- jobs/<job_id>/workflow.cwl contains the workflow to run
- jobs/<job_id>/work/ contains input and output files, and is the working directory for the job.
- jobs/<job_id>/stdout.txt is the standard output of the CWL runner
- jobs/<job_id>/stderr.txt is the standard error of the CWL runner

Create a RemoteJobFiles object. Sets up remote directory structure as well, but refuses to create the top-level directory.

Parameters

- **job_store** – The job store to use.
- **config** – The configuration.

delete_job (*job_id: str*) → None
Remove the work directory for a job. This will remove the directory and everything in it, if it exists.

Parameters **job_id** – The id of the job whose work directory to delete.

destage_job_output (*job_id: str*) → List[cerise.back_end.file.File]
Download results of the given job from the compute resource.

Parameters **job_id** – The id of the job to download results of.

Returns A list of (name, path, content) tuples.

stage_job (*job_id: str, input_files: List[cerise.back_end.file.File], workflow_content: bytes*) → None
Stage a job. Copies any necessary files to the remote resource.

Parameters

- **job_id** – The id of the job to stage
- **input_files** – A list of input files to stage.
- **workflow_content** – Translated contents of the workflow to be run.

update_job (*job_id: str*) → None
Get status from remote resource and update store.

Parameters **job_id** – ID of the job to get the status of.

Module contents

cerise.front_end package

Subpackages

cerise.front_end.controllers package

Submodules

cerise.front_end.controllers.default_controller module

Module contents

cerise.front_end.models package

Submodules

cerise.front_end.models.base_model_ module

cerise.front_end.models.job module

cerise.front_end.models.job_description module

cerise.front_end.models.workflow_binding module

Module contents

Submodules

cerise.front_end.encoder module

cerise.front_end.util module

Module contents

cerise.job_store package

Submodules

cerise.job_store.job_state module

class cerise.job_store.job_state.**JobState**

Bases: enum.Enum

Enum JobState

CANCELLED = 'Cancelled'

FINISHED = 'Finished'

PERMANENT_FAILURE = 'PermanentFailure'

RUNNING = 'Running'

RUNNING_CR = 'RunningCR'

STAGING_IN = 'StagingIn'

STAGING_IN_CR = 'StagingCR'

STAGING_OUT = 'Destaging'

STAGING_OUT_CR = 'DestagingCR'

SUBMITTED = 'Submitted'

SUCCESS = 'Success'

SYSTEM_ERROR = 'SystemError'

TEMPORARY_FAILURE = 'TemporaryFailure'

WAITING = 'Waiting'

WAITING_CR = 'WaitingCR'

cancellation_active = <function JobState.cancellation_active>

is_final = <function JobState.is_final>

is_remote = <function JobState.is_remote>

to_cwl_state_string = <function JobState.to_cwl_state_string>

cerise.job_store.sqlite_job module

class cerise.job_store.sqlite_job.**SQLiteJob**(*store: Any*, *job_id: str*)
Bases: object

This class provides the internal representation of a job. These are stored inside the service. Note that there is also a JobDescription, which is defined in the Swagger definition and part of the REST API, and a Cerulean JobDescription class, which describes a job to start on a remote compute resource.

Creates a new SQLiteJob object.

This contains only a job id and a reference to the store; the data about the job are in the database.

Parameters

- **store** (SQLiteJobStore) – The store this job is stored by
- **id** – The id of the job, a string containing a GUID

add_log(*level: int, message: Union[str, List[str]]*) → None
Add a message to the job's log.

Parameters

- **level** – Level of importance
- **message** – The log message.

critical(*message: Union[str, List[str]]*) → None
Add a message to the job's log at level CRITICAL.

Parameters **message** – The log message.

debug(*message: Union[str, List[str]]*) → None
Add a message to the job's log at level DEBUG.

Parameters **message** – The log message.

error(*message: Union[str, List[str]]*) → None
Add a message to the job's log at level ERROR.

Parameters **message** – The log message.

id = None
Job id, a string containing a UUID.

Type str

info(*message: Union[str, List[str]]*) → None
Add a message to the job's log at level INFO.

Parameters **message** – The log message.

local_input
Input JSON string, as specified by the submitter.

local_output
The serialised JSON output object describing the destaged outputs.

log
Log output as of last update.

name
Name, as specified by the submitter.

please_delete
Whether the job should be deleted.

remote_error
cwl-runner stderr output as of last update.

remote_input_path
The absolute remote path of the input description file.

remote_job_id
The id the remote scheduler gave to this job.

remote_output
cwl-runner output as of last update.

remote_stderr_path
The absolute remote path of the standard error dump.

remote_stdout_path
The absolute remote path of the standard output dump.

remote_system_err_path
The absolute remote path of the system error dump.

remote_system_out_path
The absolute remote path of the system out dump.

remote_workdir_path
The absolute remote path of the working directory.

remote_workflow_path
The absolute remote path of the CWL workflow file.

required_num_cores
The number of cores to reserve for this workflow.

resolve_retry_count
How many times we've tried to resolve.

state
Current state of the job.

time_limit
The time to reserve, in seconds. If 0, use cluster default.

try_transition(*from_state: cerise.job_store.job_state.JobState*, *to_state: cerise.job_store.job_state.JobState*) → bool
Attempts to transition the job's state to a new one.

If the current state equals from_state, it is set to to_state, and True is returned, otherwise False is returned and the current state remains what it was.

Parameters

- **from_state** – The expected current state
- **to_state** – The desired next state

Returns True iff the transition was successful.

warning(*message: Union[str, List[str]]*) → None
Add a message to the job's log at level WARNING.

Parameters **message** – The log message.

workflow
Workflow file URI, as specified by the submitter.

workflow_content
The content of the workflow description file, or None if it has not been resolved yet.

cerise.job_store.sqlite_job_store module

exception cerise.job_store.sqlite_job_store.**JobNotFound**
Bases: RuntimeError

class cerise.job_store.sqlite_job_store.**SQLiteJobStore**(*dbfile: str*)
Bases: object

A JobStore that stores jobs in a SQLite database. You must acquire the store to do anything with it or the jobs stored in it. It's a context manager, so use a with statement:

with self._store: job = self._store.get_job(id) # go ahead and modify job

don't touch self._store or keep any references to jobs

Having multiple nested with statements is okay, so you can call other functions that use the store and acquire it themselves without incident.

Parameters **dbfile** (*str*) – The path to the file storing the database.

create_job(*name: str*, *workflow: str*, *job_input: str*) → str
Create a job.

Parameters

- **name** – The user-assigned name of the job
- **workflow** – A string containing a URL pointing to the workflow
- **job_input** – A string containing a json description of a json string.

Returns A string containing the job id.

delete_job(*job_id: str*) → None
Delete the job with the given id.

Parameters **job_id** – A string containing the id of the job to be deleted.

get_job(*job_id: str*) → cerise.job_store.sqlite_job.SQLiteJob
Return the job with the given id.

Parameters **job_id** – A string containing a job id, as obtained from create_job() or list_jobs().

Returns The job object corresponding to the given id.

list_jobs() → List[cerise.job_store.sqlite_job.SQLiteJob]
Return a list of all currently known jobs.

Returns A list of SQLiteJob objects.

Module contents

cerise.test package

Submodules

cerise.test.fixture_jobs module

class cerise.test.fixture_jobs.**BrokenJob**
Bases: object

A simple job with no inputs or outputs, and an invalid command. And an invalid scheme in the input description.

input_content = {}

local_input()

local_input_files = []

output_content = {}

output_files = []

remote_input()

remote_input_files = []

remote_output()

required_num_cores = 0

time_limit = 0

workflow = b'#!/usr/bin/env cwl-runner\n\ncwlVersion: v1.0\nclass: CommandLineTool\nl

class cerise.test.fixture_jobs.**FileArrayJob**
Bases: object

A simple job with an array of input files.

input_content = {'hello_world.2nd': b'Hello, file arrays!', 'hello_world.txt': b'Hel

local_input()

local_input_files = [<cerise.back_end.file.File object>, <cerise.back_end.file.File ob

local_output = '{{ "counts": {{ "class": "File", "location": "output.txt" }} }}\n'

output_content = {'output.txt': b' 4 11 58 hello_world.txt'}

output_files = [<cerise.back_end.file.File object>]

remote_input()

remote_input_files = [('files', '01_hello_world.txt', b'Hello, World!\n\nHere is a tes

remote_output()

required_num_cores = 0

time_limit = 60

workflow = b'#!/usr/bin/env cwl-runner\n\ncwlVersion: v1.0\nclass: Workflow\ninputs:

class cerise.test.fixture_jobs.**HostnameJob**
Bases: object

A simple job with no inputs and one output.

input_content = {}

local_input()

local_input_files = []

local_output = '{ "host": { "class": "File", "location": "output.txt" }}\n'

output_content = {'output.txt': b'hostname\n'}

output_files = [<cerise.back_end.file.File object>]

remote_input()

remote_input_files = []

remote_output()

required_num_cores = 2

time_limit = 101

workflow = b'#!/usr/bin/env cwl-runner\n\ncwlVersion: v1.0\nclass: Workflow\ninputs:

class cerise.test.fixture_jobs.**InstallScriptTestJob**
Bases: object

input_content = {}

local_input()

local_input_files = []

local_output = '{ "host": { "class": "File", "location": "output.txt" } }\n'

output_content = [('output.txt', b'Testing API installation\n')]

output_files = [<cerise.back_end.file.File object>]

remote_output()

required_num_cores = 0

time_limit = 0

workflow = b'#!/usr/bin/env cwl-runner\n\ncwlVersion: v1.0\nclass: Workflow\nsteps:

class cerise.test.fixture_jobs.**LongRunningJob**
Bases: object

local_input()

local_input_files = []

local_output = '{}\n'

output_content = {}

output_files = []

required_num_cores = 0

time_limit = 0

workflow = b'#!/usr/bin/env cwl-runner\n\ncwlVersion: v1.0\nclass: Workflow\nsteps:

class cerise.test.fixture_jobs.**MissingInputJob**
Bases: object

A broken job that references an input file that doesn't exist.

input_content = {}

local_input()

local_input_files = [<cerise.back_end.file.File object>]

remote_input()

remote_input_files = []

required_num_cores = 0

time_limit = 60

workflow = b'#!/usr/bin/env cwl-runner\n\ncwlVersion: v1.0\nclass: Workflow\ninputs:

class cerise.test.fixture_jobs.**NoSuchStepJob**
Bases: object

input_content = {}

local_input()

local_input_files = []

local_output = '{}\n'

output_content = {}

output_files = []

remote_input()

remote_input_files = []

remote_output()

required_num_cores = 0

time_limit = 0

workflow = b'#!/usr/bin/env cwl-runner\n\ncwlVersion: v1.0\nclass: Workflow\nsteps:

class cerise.test.fixture_jobs.**NoWorkflowJob**
Bases: object

A job without a workflow.

input_content = {}

local_input()

local_input_files = []

output_content = {}

output_files = []

remote_input()

remote_input_files = []

remote_output()

required_num_cores = 0

time_limit = 0

workflow = None

class cerise.test.fixture_jobs.**PartiallyFailingJob**
Bases: object

input_content = {}

local_input()

local_input_files = []

local_output = '{ "output": { "class": "File", "location": "output.txt" }, "missing

output_content = [('output.txt', b'Running on host: hostname\n')]

output_files = [<cerise.back_end.file.File object>]

remote_input()

remote_input_files = []

remote_output()

required_num_cores = 0

time_limit = 0

workflow = b'#!/usr/bin/env cwl-runner\n\ncwlVersion: v1.0\nclass: Workflow\ninputs:

class cerise.test.fixture_jobs.**PassJob**
Bases: object

A simple job with no inputs or outputs.

input_content = {}

local_input()
Argument is local input dir for this job.

That's normally local_exchange / input / job_name.

local_input_files = []

local_output = '{}'

output_content = {}

output_files = []

remote_input()

remote_input_files = []

remote_output()
Argument is remote work dir for this job.

required_num_cores = 0

time_limit = 0

workflow = b'#!/usr/bin/env cwl-runner\n\ncwlVersion: v1.0\nclass: CommandLineTool\nl

class cerise.test.fixture_jobs.**SecondaryFilesJob**
Bases: object

A simple job with an input file with a secondary file.

input_content = {'hello_world.2nd': b'Hello, secondaryFiles!', 'hello_world.txt': b'

local_input()

local_input_files = [<cerise.back_end.file.File object>]

local_output = '{ "counts": { "class": "File", "location": "output.txt" } }\n'

output_content = {'output.txt': b' 4 11 58 hello_world.txt'}

output_files = [<cerise.back_end.file.File object>]

remote_input()

remote_input_files = [('file', '01_hello_world.txt', b'Hello, World!\n\nHere is a test

remote_output()

required_num_cores = 0

time_limit = 0

workflow = b'#!/usr/bin/env cwl-runner\n\ncwlVersion: v1.0\nclass: Workflow\ninputs:

class cerise.test.fixture_jobs.**SlowJob**

Bases: object

input_content = {}

local_input()

local_input_files = []

local_output = '{}'

output_content = {}

output_files = []

remote_input()

remote_input_files = []

remote_output()

required_num_cores = 0

time_limit = 0

workflow = b'#!/usr/bin/env cwl-runner\n\ncwlVersion: v1.0\nclass: Workflow\nsteps:\

class cerise.test.fixture_jobs.**WcJob**

Bases: object

A simple job with an input file and an output file.

input_content = {'hello_world.txt': b'Hello, World!\n\nHere is a test file for the st

local_input()

local_input_files = [<cerise.back_end.file.File object>]

local_output = '{ "output": { "class": "File", "location": "output.txt" } }\n'

output_content = {'output.txt': b' 4 11 58 hello_world.txt'}

output_files = [<cerise.back_end.file.File object>]

remote_input()

remote_input_files = [('file', '01_hello_world.txt', b'Hello, World!\n\nHere is a test

remote_output()

required_num_cores = 3

time_limit = 60

workflow = b'#!/usr/bin/env cwl-runner\n\ncwlVersion: v1.0\nclass: Workflow\ninputs:

cerise.test.test_config module

cerise.test.test_service module

Module contents

4.4.1.2 Submodules

4.4.1.3 cerise.config module

class cerise.config.**Config**(*config: Dict[str, Any], api_config: Dict[str, Any]*)
Bases: object

Create a configuration object.

Parameters

- **config** (*dict*) – A main configuration dict.
- **api_config** (*dict*) – An API configuration dict.

close_file_systems() → None
Close any open connections and free resources.

This function is to be called on shutdown, to ensure that the remote file system managed by Config is shut down properly.

get_base_url() → str
Returns the service's base url.

This is the URL of the REST API, before the /jobs part, e.g. if listing jobs is done by a GET to http://localhost/jobs, then this should be set to http://localhost. Obtained from the configuration file or the CERISE_BASE_URL environment variable.

get_basedir() → cerulean.path.Path
Returns the configured remote base directory to use.

Returns The remote path to the base directory.

Return type (str)

get_cores_per_node() → int
Returns the number of cores per node.

This depends on the available compute hardware, and should be configured in the specialisation. The incoming workflow specifies a number of cores, but we reserve nodes, so we need to convert.

The default is 32, which is probably more than what you have, as a result of which we'll allocate fewer nodes than the user specified if no value is given. That'll slow things down, but at least we won't be burning core hours needlessly.

Returns The number of cores per node on this machine.

Return type (int)

get_database_location() → str
Returns the local path to the database file.

Returns The path.

Return type (str)

Raises KeyError – No database path was set.

get_file_system() → cerulean.file_system.FileSystem
Returns a remote file system as configured by the user.

Returns (cerulean.FileSystem) A new filesystem

get_log_file() → str
Returns the configured path for the log file. Use has_logging() to see if logging has been configured first.

Returns The path.

Return type (str)

get_log_level() → int
Returns the configured log level. Use has_logging() to see if logging has been configured first.

Returns The log level, following Python's built-in logging library.

Return type (int)

get_pid_file() → Optional[str]
Returns the location of the PID file, if any.

Returns The configured path, or None

Return type (Union[str,None])

get_queue_name() → Optional[str]
Returns the name of the queue to submit jobs to, or None if no queue name was configured.

Returns The queue name.

Return type (Union[str,None])

get_remote_cwl_runner() → str
Returns the configured remote path to the CWL runner to use.

No macro substitution is done; this gives the configured path as-is.

Returns The path.

Return type (str)

get_remote_refresh() → float
Returns the interval in between checks of the remote job status, in seconds.

Returns How often to check remote job status.

Return type (float)

get_scheduler(*run_on_head_node: bool = False*) → cerulean.scheduler.Scheduler
Returns a scheduler as configured by the user.

Parameters **run_on_head_node** (*bool*) – If True, will create a scheduler using the ssh adaptor instead of the configured one if the configured adaptor is a cluster scheduler (i.e. slurm, torque or gridengine).

Returns A new scheduler

Return type (cerulean.Scheduler)

get_scheduler_options() → Optional[str]
Returns the additional scheduler options to use.

Returns The options as a single string.

Return type (str)

get_service_host() → str
Return the host interface Cerise should listen on.

Returns The IP address of the interface to listen on.

Return type str

get_service_port() → int
Return the port on which Cerise should listen.

Returns The port number to listen on.

Return type int

get_slots_per_node() → int
Returns the configured number of MPI slots per node.

Returns The number of slots to use.

Return type (int)

get_store_location_client() → str
Returns the file exchange location access point for the client.

Returns A URL.

Return type (str)

Raises KeyError – The location was not set.

get_store_location_service() → cerulean.path.Path
Returns the file exchange location access point for the service.

Returns The local base directory for file exchange with the client.

Raises KeyError – The location was not set.

get_username(*kind: str*) → Optional[str]
Return the username used to connect to the specified kind of resource.

Parameters **kind** (*str*) – Either 'files' or 'jobs'

Returns The configured username

Return type (str)

has_logging() → bool
Returns if logging is configured.

Returns True iff a logging section is available in the configuration.

cerise.config.**make_config**() → cerise.config.Config
Make a configuration object.

Uses the configuration files and environment variables to determine the configuration.

Returns The Cerise configuration.

Return type *Config*

4.4.1.4 cerise.run_back_end module

4.4.1.5 cerise.run_front_end module

4.4.1.6 cerise.util module

4.4.1.7 Module contents

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

F

G

H

I

J

L

M

N

O

P

R

S

T

U

W