# Ganga Documentation

***Release 8.5.2***

**Ganga Developers**

**Aug 11, 2021**

# For users

Ganga is a tool to make it easy to run data analysis jobs along with managing associated data files.

CHAPTER 1

# User Guide

This tutorial is in several stages that cover all of the basic and more of the advanced functionality that Ganga provides. To follow it properly, you should go through each stage in turn. Please post an issue if you encounter any problems!

## 1.1 What is Ganga

Many scientific computations are too large to take care of by simply running a script from the command line and waiting for it to execute. To get around this many different systems has been used over the years

- Starting tasks in the background on your desktop;
- Using a local batch system or batch system on a central facility;
- Using various grid or cloud systems for submitting your code to.

Getting your work done like this often means that it gets broken into multiple pieces. All these pieces are both tedious and error prone and include things like:

- Ensure that you use same code for testing locally and running on grid system;
- Pack up ancillary files that are required for running on remote system;
- Split your task up into many smaller pieces;
- Submit each of the smaller pieces, keep track of which fail, resubmit them;
- Keep running commands to see if all the jobs have finished;
- Merge all the pieces together.

The idea in Ganga is to take all these problems, provide a Python API for them that allows them to be solved in a clean and programmatic way. In addition, Ganga will provide a service that takes care of monitoring the progress of all tasks that have been submitted. This means that a workflow using Ganga is more like.

- Create your task inside Ganga and test it locally on your desktop;
- Specify to Ganga how you want you task broken up and the results merged;

- Tell Ganga where your task should be executed (Batch, Grid, . . . ) and submit it;
- Let Ganga monitor the progress, resubmit failed pieces and merge the results in the end.

Ganga provides a plugin system that allows groups such as HEP collaborations to expand the API with specific applications that will make it easier to run tasks on remote systems (build shared libraries, find configuration files, interact with data bookkeeping). There is also support for running taks inside docker and singularity containers.

## 1.2 Install and Basic Usage

### 1.2.1 Installation

There are several ways to install and run Ganga:

**NOTE -** Currently Ganga is only available for python3 releases and supported on Linux distributions.

#### Binder

For testing, you can run Ganga within the Binder service. Simply follow the link https://mybinder.org/v2/gh/ganga-devs/ganga/BinderTutorial and in a terminal start Ganga. The parts of Ganga that connects to remote services will be limited due to the restricted connectivity of the Binder containers.

#### CVMFS

If you have access to CVMFS, Ganga can be found at `/cvmfs/ganga.cern.ch/`. This will be kept up-to-date with the latest release that you can run directly with:

`/cvmfs/ganga.cern.ch/runGanga.sh`

This isn't just a link to the start-up script because it needs to be run from the correct directory. However, it will take all the command line options that the normal ganga script takes.

#### PyPI Install

(Recommended)

The best is to install inside a virtual environment

```
# Create a virtualenv
python3 -m venv gangaenv
cd gangaenv/
. bin/activate

# Update installation tools to latest and greatest
python3 -m pip install --upgrade pip setuptools wheel

# Install Ganga
python3 -m pip install ganga
```

To install pip locally if it's not on your system and you don't have admin access please consult: https://pip.pypa.io/en/stable/installing/

Now each time you want to use Ganga in a new shell, you have to activate the virtual environment:

```
cd gangaenv/
. bin/activate
ganga
```

### From Github

```
# Create a virtualenv
python3 -m venv gangaenv
cd gangaenv/
. bin/activate

# Update installation tools to latest and greatest
python3 -m pip install --upgrade pip setuptools wheel

# Clone Ganga
git clone git@github.com:ganga-devs/ganga.git

# Install dependencies
cd ganga
python3 -m pip install -e .
```

## 1.2.2 Starting Ganga

As described above, to run Ganga simply execute `ganga` (for PyPI install), `<installdir>/bin/ganga` (for other installs) or the helper script in CVMFS. This will start Ganga and it's associated threads as well as provide you with a Ganga IPython prompt that gives you access to the Ganga Public Interface (GPI) on top of the usual IPython functionality:

Note that the first time you run Ganga it will ask you to create a default `.gangarc` file which you should probably do. In the future, if you want to recreate this default config file, add the option `-g` to the command line.

In Binder you start ganga in a terminal as illustrated in the image below

## 1.2.3 Getting Help

The documentation for all objects and functions in Ganga can be accessed using the help system:

```
[13:25:29]
Ganga In [1]: help()
************************************

*** Welcome to Ganga ***
Version: 8.3.3
Documentation and support: http://cern.ch/ganga
Type help() or help('index') for online help.

This is free software (GPL), and you are welcome to redistribute it
under certain conditions; type license() for details.

This is an interactive help based on standard pydoc help.

```

(continues on next page)

(continued from previous page)

```
Type 'index'  to see GPI help index.
Type 'python' to see standard python help screen.
Type 'interactive' to get online interactive help from an expert.
Type 'quit'   to return to Ganga.
***********************************
```

Now typing `index` at the prompt will list all the objects, etc. available. You can also directly access the documentation for an object using `help` directly:

```
help(Job)
```

You can also use IPython's tab-complete to help identify members of an object.

### 1.2.4 Hello World with Ganga

We are now in a position to submit our first job. This will take the defaults of the Ganga Job object which is to run `echo 'Hello World'` on the machine you're currently running on:

```
j = Job()
j.submit()
```

If all goes well, you should see the job submit:

```

```

You can view the job in your repository using the `jobs` command which lists all job objects that Ganga knows about:

```
Ganga In [1]: jobs
Ganga Out [1]:

Registry Slice: jobs (1 objects)
--------------
    fqid |     status |       name | subjobs |     application |          backend |        ␣
↪                      backend.actualCE |                             comment
-------------------------------------------------------------------------------------
↪----------------------------------------------------------------------
       0 | completed |            |         |      Executable |            Local |        ␣
↪              epldt017.ph.bham.ac.uk |


[13:34:37]
Ganga In [2]:
```

You can get more info about your job by selecting it from the repository:

```
jobs(0)
```

You can also select specific info about the job object, e.g. the application that was run:

```
jobs(0).application
```

To check the `stdout/stderr` of a job, you can use the peek method

```
j = jobs(0)
j.peek('stdout')
```

### 1.2.5 Copy a job

You can copy and old job, modify its attributes anbd then submit it as a new one

```
j = jobs(0).copy()
j.application.args = ['Hello from a copy']
j.name = 'Copy'
j.submit()
```

### 1.2.6 Job Monitoring

While Ganga is running in interactive mode, a background thread goes through all your active jobs and checks to see what state they are in. Generally, jobs will transition from new -> submitted -> running -> completed/failed. As described above, the *jobs* command will show you the state of your jobs in the Ganga repository.

### 1.2.7 Scripting and Batch Mode

You can put your ganga commands into a python script

```
[centos7] ~ % cat >> myfile.py
j = Job()
j.submit()
^D
```

and then execute it from the Ganga prompt like this

```
runfile('myfile.py')
```

In addition, Ganga can be run in batch mode by just providing a script as the last argument:

```
[centos7] ~ % ganga myfile.py
```

Note that by default, the monitoring is turned off while in batch mode.

## 1.3 Running Executables

You can run any executable or script through Ganga using the `Executable` application. This accepts either a full path to an already installed exe (e.g. on CVMFS) or a Ganga `File` object that will be sent with your job. You can also supply arguments and environment settings with the options in the `Executable` object.

As an example:

```
# Already existing Exe
j = Job()
j.application = Executable()
j.application.exe = '/bin/ls'
j.application.args = ['-l', '-h']
```

(continues on next page)

(continued from previous page)

```
j.submit()

# Wait for completion
j.peek("stdout")

# Send a script
open('my_script.sh', 'w').write("""#!/bin/bash
echo 'Current dir: ' `pwd`
echo 'Contents:'
ls -ltr
echo 'Args: ' $@
""")
import os
os.system('chmod +x my_script.sh')

j = Job()
j.application = Executable()
j.application.exe = File('my_script.sh')
j.submit()

# Wait for completion
j.peek("stdout")
```

If your executable requires more than one file to run, you can use the `inputfiles` field of the `Job` object to send this across as well (see *Input And Output Data*).

## 1.4 Tutorial Plugin

Ganga has a tutorial plugin available that serves the purposes of

- Illustrating how specific applications can be used (*Using Different Applications*)
- Using specific splitters to divide a computational task into many pieces (*Using Different Applications*)
- Illustrating how you can write your own plugin package for Ganga that provides new capability.

### 1.4.1 Enable tutorial plugin

In the *Binder* tutorial, the tutorial plugin is already enabled and you do not have to do anything. For any other Ganga session, you can simply start Ganga like

```
ganga -o '[Configuration]RUNTIME_PATH=GangaTutorial'
```

or edit the *RUNTIME_PATH* line of your *~/.gangarc* configuration file.

## 1.5 Using Different Applications

For executing some complicated computation, of ten more than just a simple executable is provided. Maybe certain calibration files are required as well, compilation of code is required first that gives rise to shared libraries and so on. Ganga allows through a plugin system for specific applications to be executed. You can see which ones are available as

```
Ganga In [1]: plugins('applications')
Ganga Out [1]: ['Executable', 'Root', 'Notebook', 'PrimeFactorizer']
```

Large Particle Physics collaborations, such as LHCb and T2K have specific applications written and it is quite easy to write your own.

### 1.5.1 Try it out

The `PrimeFactorizer` application is part of the Tutorial plugin (*Tutorial Plugin*). It illustrates how a specific application can be used instead of the default *Executable* application. You can try and create a job like

```
j = Job(application = PrimeFactorizer(number=1527), inputdata = PrimeTableDataset())
```

and submit it. In the output of the job when finished you should see the prime factors of the number. For the *inputdata* in the job specification above, see *Input And Output Data*.

## 1.6 Using Different Backends

One of the main benefits of Ganga is that you can submit to different clusters/systems (in Ganga, these are termed backends) by only changing one or two lines in your scripts. Though there are often very different ways of submission for each backend, Ganga tries to hide this as much as possible and follow a submission that is more or less identical to what is done for a Local job.

```
plugins("backends")
```

### 1.6.1 Local Backend

This is the default and refers to the machine that Ganga is running on. The job will be spawned as a separate process, independent of Ganga. Typical usage is:

```
j = Job()
j.backend = Local()
j.submit()
```

There are no editable options for the object itself but there are two config options that you can view with `config.Local`. You can quit Ganga and restart and the Local job will still run in the background.

### 1.6.2 Batch Backends

Ganga supplies backend objects for most of the major batch systems around - Condor, Slurm, SGE, LSF and PBS. You should obviously use the one that is relevant to the system you are running on. Typical usage is detailed below though as with all these, you can get more help using `help(<backend>)` and `config.<backend>`. Sometimes a local installation requires that small changes are made to the configuration. Look in the relevant section of the `~/.gangarc` file.

#### LSF

```
j = Job()
j.backend = LSF()
j.backend.queue = '1nh'
```

#### Slurm

Very similar to the LSF backend, this is setup by default to submit to a typical Slurm installation but again, can easily be changed to reflect your specific setup:

```
j = Job()
j.backend = Slurm()
j.submit()
```

#### Condor

Condor is a little different than the other backends but should still submit to most typical installations. There is also a requirements object that can be used to specify memory, architecture, etc.

```
j = Job()
j.backend = Condor()
j.backend.getenv = "True"  # send the environment to the host
j.backend.requirements.memory = 1200
j.submit()
```

Also note that the `getenv` option is defined as a string so in your `.gangarc`, you would need to set it to:

```
[Condor]
getenv = 'True'
```

To avoid Ganga attempting to assign a boolean instead.

### 1.6.3 Dirac Backend

To submit to a Dirac instance, you will need to have the Dirac client installed and a Dirac proxy available.

#### Using GridPP Dirac on CVMFS

There is an installed version of Dirac configured to use the GridPP Dirac instance available on the Ganga CVMFS area. To run with this, do

```
/cvmfs/ganga.cern.ch/runGanga-dirac.sh
```

A few questions about your Virtual Organisation will be asked the first time you run.

#### Installing and Configuring the Dirac Client

If you are not using the GridPP instance of Dirac, or don't have access to CVMFS then you will need to install and configure the Dirac client. See here for instructions.

After successfully doing this, do the following steps to configure Ganga:

- Edit your `.gangarc` file and set the following options:

```
[Configuration]
RUNTIME_PATH = GangaDirac

[DIRAC]
DiracEnvSource = /home/<username>/dirac_ui/bashrc
```

## 1.7 Virtualization

It is possible to run a Ganga job inside a container. This allows you to get a completely well defined environment on the worker node where the job is executed. Each job has a virtualization attribute which defines the image to be used for the container as a required attribute. Images can be either from Docker or Singularity hub, from the images created by Gitlab or in case of Singularity from a file provided as a GangaFile.

Using images can provide an attractive workflow where GitLab continuous integration is used to create Docker images. Those images can then subsequently be used for running jobs where it is assured that they are in the same environment. The image can either be used directly from the repository (using the deploy username/password if private) or can be pulled and converted to a Singularity image.

### 1.7.1 Try it out

```
j1 = Job(name='Weather', \
         virtualization=Docker(image='uegede/weather'), \
         application=Executable(exe='weather', args=['MEL']))
j2 = Job(name='Fedora', \
         virtualization=Docker(image='fedora:latest'), \
         application=Executable(exe='cat', args=['/etc/redhat-release']))
```

### 1.7.2 Singularity class

The Singularity class can be used for either Singularity or Docker images. It requires that singularity is installed on the worker node.

For Singularity images you provide the image name and tag from Singularity hub like

```
j=Job()
j.application=Executable(exe=File('my/full/path/to/executable'))
j.virtualization = Singularity("shub://image:tag")
```

Notice how the executable is given as a `File` object. This ensures that it is copied to the working directory and thus will be accessible inside the container.

The container can also be provided as a Docker image from a repository. The default repository is Docker hub.

```
j.virtualization = Singularity("docker://gitlab-registry.cern.ch/lhcb-core/lbdocker/
↪centos7-build:v3")
j.virtualization = Docker("docker://fedora:latest")
```

Another option is to provide a `GangaFile` Object which points to a singularity file. In that case the singularity image file will be copied to the worker node. The first example is with an image located on some shared disk. This will be effective for running on a local backend or a batch system with a shared disk system.

```
imagefile = SharedFile('myimage.sif', locations=['/my/full/path/myimage.sif'])
j.virtualization = Singularity(image= imagefile)
```

while a second example is with an image located in the Dirac Storage Element. This will be effective when using the Dirac backend.

```
imagefile = DiracFile('myimage.sif', lfn=['/some/lfn/path'])
j.virtualization = Singularity(image= imagefile)
```

If the image is a private image, the username and password of the deploy token can be given like the example below. Look inside Gitlab setting for how to set this up. The token will only need access to the images and nothing else.

```
j.virtualization.tokenuser = 'gitlab+deploy-token-123'
j.virtualization.tokenpassword = 'gftrh84dgel-245^ghHH'
```

Directories can be mounted from the host to the container using key-value pairs to the mounts option. If the directory is not vailable on the host, a warning will be written to stderr of the job and no mount will be attempted.

```
j.virtualization.mounts = {'/cvmfs':'/cvmfs'}
```

By default the container is started in singularity with the `--nohome` option. Extra options can be provided through the `options` attribute. See the Singularity documentation for what is possible.

### 1.7.3 Docker class

You can define a docker container by providing an image name and tag. Using that ganga will fetch the image from the docker hub.

```
j=Job()
j.virtualization = Docker("image:tag")
```

Ganga will try to run the container using Docker if Docker is availabe in the worker node and if the user has the permission to run docker containers. If not ganga will download UDocker which provides the ability to run docker containers in userspace. The runmode in Udocker can be changed as seen in the documentation. Using Singualarity as the run mode is not recommended; use the `Singularity` class above instead.

### 1.7.4 Issues to keep in mind

Awareness should be given to the load that using containers will impose on the system where they are running

- If the file system is shared (like for the `Batch` and `Local` backends, the images pulled down from a remote repository will be cached locally.
- If the file system is not shared (like for the `LCG` and `Dirac` backends), then images from remote repositories will be pulled for each job. This might put an excessive load on the network and/or the repository.
- If the image for `Singularity` is given as a file, it will be copied to the worker node. If provided as a `DiracFile` object, it can be replicated to the sites where the job will be asked to run to limit the impact of pulling the image.

## 1.8 Splitters

One of the main benefits of Ganga is it's ability to split a job description across many subjobs, changing the input data or arguments appropriately for each. Ganga then keeps these subjobs organised with the parent master job but keeps track of all their status, etc. individually. There are two main splitters that are provided in Ganga Core which are detailed below. You can see which splitter are available with

```
Ganga In [1]: plugins('splitters')
Ganga Out [1]:
['ArgSplitter',
 'GenericSplitter',
 'GangaDatasetSplitter',
 'PrimeFactorizerSplitter']
```

### 1.8.1 Try it out:

Using the prime factorisation example from the Tutorial plugin (*Tutorial Plugin*). We can split up the factorisation of a very large number up into 5 different tasks.

```
j = Job(application = PrimeFactorizer(number=268709474635016474894472456), \
        inputdata = PrimeTableDataset(table_id_lower=1, table_id_upper=30), \
        splitter = PrimeFactorizerSplitter(numsubjobs=10))
```

After the job and been submitted and finished, the output of each of the subjobs will be available. Remember that ganga is just a standard Python prompt, so we can use standard Python syntax

```
for js in j.subjobs: js.peek('stdout','cat')
```

See the section *PostProcessors* for how we can merge the output into a single file.

### 1.8.2 ArgSplitter

For a job that is using an *Executable* application, it is very common that you want to run it multiple times with a different set of arguments (like a random number seed). The *ArgSplitter* can do exactly that. For each of the subjobs created, it will replace the arguments fot he job with one from the array of array of arguments provided to the splitter. So

```
j = Job()
j.splitter=ArgSplitter(args=[['Hello 1'], ['Hello 2']])
```

will create two subjobs where the `Hello World` of the default executable argument will be replaced by `Hello 1` and `Hello 2` respectively.

### 1.8.3 GenericSplitter

The `GenericSplitter` is a useful tool to split a job based on arguments or parameters in an application or backend. You can specify whatever attribute you want to split over within the job as a string using the `attribute` option. The example below illustrate how you can use it to do the same as the `ArgSplitter`.

```
j = Job()
j.splitter = GenericSplitter()
j.splitter.attribute = 'application.args'
```

(continues on next page)

(continued from previous page)

```
j.splitter.values = [['hello', 1], ['world', 2], ['again', 3]]
j.submit()
```

This produces 3 subjobs with the arguments:

```
echo hello 1     # subjob 1
echo world 2     # subjob 2
echo again 3     # subjob 3
```

Each subjob is essentially another `Job` object with all the parameters set appropriately for the subjob. You can check each one by using:

```
j.subjobs
j.subjobs(0).peek("stdout")
```

There may be times where you want to split over multiple sets of attributes though, for example the `args` and the `env` options in the `Executable` application. This can be done with the `multi_attrs` option that takes a dictionary with each key being the attribute values to change and the lists being the values to change. Give the following a try:

```
j = Job()
j.splitter = GenericSplitter()
j.splitter.multi_attrs = {'application.args': ['hello1', 'hello2'],
                          'application.env': [{'MYENV':'test1'}, {'MYENV':'test2'}]}
j.submit()
```

This will produce subjobs with the exe and environment:

```
echo hello1 ; MYENV = test1   # subjob 1
echo hello2 ; MYENV = test2   # subjob 2
```

### 1.8.4 GangaDatasetSplitter

The `GangaDatasetSplitter` is provided as an easy way of splitting over a number input data files given in the `inputdata` field of a job. The splitter will create a subjob with the maximum number of file specified (default is 5). A typical example is:

```
j = Job()
j.application.exe = 'more'
j.application.args = ['__GangaInputData.txt__']
j.inputdata = GangaDataset( files=[ LocalFile('*.txt') ] )
j.splitter = GangaDatasetSplitter()
j.splitter.files_per_subjob = 2
j.submit()
```

If you check the output you will see the list of files that each subjob was given using `j.subjobs()` as above.

### 1.8.5 Resplitting a job

Sometimes a job that has been split will have some of the subjobs failing. This might for example be due to that they run out of CPU time and are required to be split into smaller units. To support this, it is possible to apply a new splitter to a subjob which is in the `completed` or `failed` state. In the example below can be seen how the first subjob is subsequently split into a further two subjobs.

```
j = Job(splitter=ArgSplitter(args=[ [0],[0] ]))
j.submit()

# wait for jobs to complete
j.subjobs

Registry Slice: jobs(8).subjobs (2 objects)
--------------
    fqid |    status       |                       comment |
-------------------------------------------------------
     8.0 | completed       |                               |
     8.1 | completed       |                               |

j.subjobs(0).resplit(ArgSplitter(args=[ [1], [1] ]))

# wait for jobs to complete
j.subjobs

--------------
    fqid |    status       |                       comment |
-------------------------------------------------------
     8.0 |completed_frozen |             - has been resplit |
     8.1 | completed       |                               |
     8.2 | completed       |               - resplit of 8.0 |
     8.3 | completed       |               - resplit of 8.0 |
```

Any splitter can be used for the resplitting. The subjob that was the origin of the resplit is clearly marked as seen above.

## 1.9 PostProcessors

Ganga can be instructed to do many things after a job completes. Each object can be added to the `postprocessors` field of the `Job` object and they will be carried out in order. The available Post-Processing options are detailed below:

### 1.9.1 Try it out

When using the prime factorisation example from the Tutorial plugin (*Tutorial Plugin*) it was not satisfactory that the individual prime factors were distributed over different files. A simple `TextMerger` can collate the numbers into a single file.

```
j = Job(application = PrimeFactorizer(number=268709474635016474894472456), \
        inputdata = PrimeTableDataset(table_id_lower=1, table_id_upper=30), \
        splitter = PrimeFactorizerSplitter(numsubjobs=10), \
        postprocessors = TextMerger(files=['factors.dat']))
```

When the job has finished, there will now be a single file that we can look at

```
j.peek('factors.dat')
```

See below for how a `CustomMerger` could be used to provide a more unified output.

### 1.9.2 Mergers

A merger is an object which will merge files from each subjobs and place it the master job output folder. The method to merge depends on the type of merger object (or file type). For example, if each subjob produces a root file 'thesis_data.root' and you would like this to be merged you can attach a RootMerger object to your job:

```
j.postprocessors.append(RootMerger(files = ['thesis_data.root'],ignorefailed = True,
↪overwrite = True))
```

When the job is finished this merger object will then merge the root files and place them in `j.outputdir`. The `ignorefailed` flag toggles whether the merge can proceed if a subjob has failed. The overwrite flag toggles whether to overwrite the output if it already exists. If a merger fails to merge, then the merger will fail the job and subsequent postprocessors will not run. Also, be aware that the merger will only run if the files are available locally, Ganga won't automatically download them for you (unless you use Tasks) to avoid running out of local space. You can always run the mergers separately though:

```
j.postprocessors[0].merge()
```

There are several mergers available:

#### TextMerger

```
TextMerger(compress = True)
```

Used for merging `.txt`, `.log`, etc. In addition to the normal attributes, you can also choose to compress the output with

#### RootMerger

```
TextMerger(compress = True)
```

Used for root files. In addition to the normal attributes, you can also pass additional arguments to hadd.

#### CustomMerger

A custom merger where you can define your own merge function. For this merger to work you must supply a path to a python module which carries out the merge with

```
CustomMerger().module = '~/mymerger.py'
```

In `mymerger.py` you must define a function called mergefiles(file_list,output_file), e.g:

```
import os
def mergefiles(file_list,output_file):
      f_out = file(output_file,'w')
      for f in file_list:
            f_in = file(f)
            f_out.write(f_in.read())
            f_in.close()
      f_out.flush()
      f_out.close()
```

This function would mimic the TextMerger, but with more control to the user. Note that the `overwrite` and `ignorefailed` flags will still work here as a normal merger object.

### SmartMerger

The final merger object which can be used is the `SmartMerger()`, which will choose a merger object based on the output file extension. It supports different file types. For example the following SmartMerger would use a RootMerger for 'thesis_data.root' and a TextMerger for 'stdout'.

```
SmartMerger(files = ['thesis_data.root','stdout'],overwrite = True)
```

Note that:

```
j.postprocessors.append(SmartMerger(files = ['thesis_data.root','stdout'],overwrite =␣
↪True))
```

is equivalent to doing:

```
j.postprocessors.append(TextMerger(files = ['stdout'],overwrite = True))
j.postprocessors.append(RootMerger(files = ['thesis_data.root'],overwrite = False))
```

However in the second instance you gain more control as you have access to the `Root/TextMerger` specific attributes, but at the expense of more code. Choose which objects work best for you.

## 1.9.3 Checkers

A checker is an object which will fail otherwise completed jobs based on certain conditions. However, if a checker is misconfigured the default is to do nothing (pass the job), this is different to the merger. Currently there are three Checkers:

### FileChecker

Checks the list of output files and fails job if a particular string is found (or not found). For example, you could do:

```
fc = FileChecker(files = ['stdout'], searchStrings = ['Segmentation'])
```

You can also enforce that your file must exist, by setting `filesMustExists` to `True`:

```
fc.filesMustExist = True
```

If a job does not produce a stdout file, the checker will fail the job. This FileChecker will look in your stdout file and grep the file for the string 'Segmentation'. If it finds it, the job will fail. If you want to fail the job a string doesn't exist, then you can do:

```
fc.searchStrings = ['SUCCESS']
fc.failIfFound = False
```

In this case the FileChecker will fail the job if the string 'SUCCESS' is not found.

### RootFileChecker

This checks that all your ROOT files are closed properly and have nonzero size. Also checks the merging procedure worked properly. Adding a RootFileChecker to your job will add some protection against hadd failures, and ensure that your ROOT files are mergable. If you do:

This checker will check that each ROOT file has non-zero file size and is not a zombie. If you also have a merger, it will check the output from hadd, ensure that the sum of the subjob entries is the same as the master job entries, and check that each ROOT file has the same file structure. `RootFileChecker` inherits from `FileChecker` so you can also ensure that the ROOT files must exist.

#### CustomChecker

This is probably the most useful checker and allows the user to use private python code to decide if a job should fail or not. The `CustomChecker` will execute your script and fail the job based on the output. For example, you can make a checker in your home directory called `mychecker.py`. In this file you must define a function called `check(j)`, which takes in your job as input and returns `True` (pass) or `False` (fail)

```
import os

def check(j):
    outputfile = os.path.join(j.outputdir,'thesis_data.root')
    return os.path.exists(outputfile)
```

Then in ganga do:

```
cc = CustomChecker(module = '~/mychecker.py')
```

This checker will then fail jobs which don't produce a 'thesis_data.root' root file.

### 1.9.4 Notifier

The notifier is an object which will email you about your jobs upon completion. The default behaviour is to email when master jobs have finished and when subjobs have failed. Emails are not sent upon failure if the auto-resubmit feature is used. Important note: Emails will only be sent when ganga is running, and so the Notifier is only useful if you have ganga running in the background (e.g. screen session, `GangaService`). To make a notifier, just do something like:

```
n = Notifier(address = 'myaddress.cern.ch')
```

If you want emails about every subjob, do

```
n = Notifier(address = 'myaddress.cern.ch')
```

### 1.9.5 Management of post processors with your job

You can add multiple post processors to a Job and Ganga will order them to some degree. Mergers appear first, then checkers, then finally the notifier. It will preserve the order within each class though (e.g. The ordering of the #checkers is defined by the user). To add some postprocessors to your job, you can do something like

```
tm = TextMerger(files=['stdout'], compress=True)
rm = RootMerger(files=['thesis_data.root'], args='-f6')
fc = FileChecker(files=['stdout'], searchStrings=['Segmentation'])
cc = CustomChecker(module='~/mychecker.py')
n = Notifier(address='myadress.cern.ch')

j.postprocessors = [tm, rm, fc, cc, n]
```

or:

```
j.postprocessors.append(fc)
j.postprocessors.append(tm)
j.postprocessors.append(rm)
j.postprocessors.append(cc)
j.postprocessors.append(n)
```

You can also remove postprocessors:

```
In [21]:j.postprocessors
Out[21]: [SmartMerger (
 files = [] ,
 ignorefailed = False ,
 overwrite = False
 ), FileChecker (
 files = [] ,
 checkSubjobs = False ,
 searchStrings = [] ,
 failIfFound = True
 ), Notifier (
 verbose = False ,
 address = ''
 )]

In [22]:j.postprocessors.remove(FileChecker())

In [23]:j.postprocessors
Out[23]: [SmartMerger (
 files = [] ,
 ignorefailed = False ,
 overwrite = False
 ), Notifier (
 verbose = False ,
 address = ''
 )]
```

## 1.10 Input And Output Data

Ganga tries to simplify sending input files and getting output files back as much as possible. You can specify not only what files you want but where they should be retrieved/put. There are three fields that are relevant for your job:

1. **Input Files** Files that are sent with the job and are available in the same directory on the worker node that runs it
2. **Input Data** A dataset or list of files that the job will run over but which are NOT transferred to the worker. Typically the running job will stream this data.
3. **Output Files** The name, type and location of the job output

### 1.10.1 Basic Input/Output File usage

To start with, we'll show a job that sends an input text file with a job and then sends an output text file back:

```
# create a script to send
open('my_script2.sh', 'w').write("""#!/bin/bash
```

(continues on next page)

(continued from previous page)

```
ls -ltr
more "my_input.txt"
echo "TESTING" > my_output.txt
""")
import os
os.system('chmod +x my_script2.sh')

# create a script to send
open('my_input.txt', 'w').write('Input Testing works!')

j = Job()
j.application.exe = File('my_script2.sh')
j.inputfiles = [ LocalFile('my_input.txt') ]
j.outputfiles = [ LocalFile('my_output.txt') ]
j.submit()
```

After the job completes, you can then view the output directory and see the output file:

```
j.peek()   # list output dir contents
j.peek('my_output.txt')
```

If the job doesn't produce the output Ganga was expecting, it will mark the job as failed:

```
# This job will fail
j = Job()
j.application.exe = File('my_script2.sh')
j.inputfiles = [ LocalFile('my_input.txt') ]
j.outputfiles = [ LocalFile('my_output_FAIL.txt') ]
j.submit()
```

You can also use wildcards in the files as well:

```
# This job will pick up both 'my_input.txt' and 'my_output.txt'
j = Job()
j.application.exe = File('my_script2.sh')
j.inputfiles = [LocalFile('my_input.txt')]
j.outputfiles = [LocalFile('*.txt')]
j.submit()
```

After completion, the output files found are copied as above but they are also recorded in the job appropriately:

```
j.outputfiles
```

This will also work for all backends as well - Ganga handles the changes in protocol behind the scenes, e.g.:

```
j = Job()
j.application.exe = File('my_script2.sh')
j.inputfiles = [ LocalFile('my_input.txt') ]
j.outputfiles = [ LocalFile('my_output.txt') ]
j.backend = Dirac()
j.submit()
```

### 1.10.2 Input Data Usage

Generally, input data for a job is quite experiment specific. However, Ganga provides by default some basic input data functionality that can be used to process a set of remotely stored files without copying them to the worker. This is done with the `GangaDataset` object that takes a list of `GangaFiles` (as you would supply to the `inputfiles` field) and instead of copying them, a flat text file is created on the worker (`__GangaInputData.txt__`) that lists the paths of the given input data. This is useful to access files from Mass or Shared Storage using the mechanisms within the running program, e.g. opening them with directly with Root.

As an example:

```
# Create a test script
open('my_script3.sh', 'w').write("""#!/bin/bash
echo $PATH
ls -ltr
more __GangaInputData.txt__
echo "MY TEST FILE" > output_file.txt
""")
import os
os.system('chmod +x my_script3.sh')

# Submit a job
j = Job()
j.application.exe = File('my_script3.sh')
j.inputdata = GangaDataset(files=[LocalFile('*.sh')])
j.backend = Local()
j.submit()
```

### 1.10.3 File Types Available

Ganga provides several File types for accessing data from various sources. To find out what's available, do:

```
plugins('gangafiles')
```

#### LocalFile

This is a basic file type that refers to a file on the submission host that Ganga runs on. As an input file, it will pick up the file and send it with your job, as an output file it will be returned with your job and put in the `j.outputdir` directory.

#### DiracFile

This will store/retrieve files from Dirac data storage. This will require a bit of configuration in `~/.gangarc` to set the correct LFN paths and also where you want the data to go:

```
config.DIRAC.DiracLFNBase
config.DIRAC.DiracOutputDataSE
```

To use a DiracFile, do something similar to:

```
j = Job()
j.application.exe = File('my_script2.sh')
j.inputfiles = [ LocalFile('my_input.txt') ]
j.outputfiles = [ DiracFile('my_output.txt') ]
j.backend = Dirac()
j.submit()
```

Ganga won't retrieve the output to the submission node so if you need it locally, you will have to do.

```
j.outputfiles.get()
```

Often it might be better to simply stream the data from its remote destination. You can get th `URL` for this as

```
j.outputfiles[0].accessURL()
```

### GoogleFile

This will store files to the user's Google Drive. This requires the user to authenticate and give restricted access to Google Drive. To use a GangaFile, do something similar to:

```
j = GangaFile("mydata.txt")
j.localDir = "~/temp"
j.put()
print(j)
GoogleFile (
    namePattern = mydata.txt,
    localDir = /home/dumbmachine/temp,
    failureReason = ,
    compressed = False,
    downloadURL = https://drive.google.com/file/d/1dS_XqANroclWAqgIvLU7q5rbzen17mSf
)
```

The urls are generated by using the *id* of the file.

This will upload the local file "~/temp/mydata.txt" to the user's Google Drive inside a folder names *Ganga*. The File object also supports for glob patterns, which can be supplied as *j.namePattern* = '**.ROOT*'.

Upon first usage, the user will be asked to authenticate and allow access to create new files and edit these files only. While the default client ID of *Ganga* can be used, it is recommended to create you own client ID. Tjhis will prevent getting rate limited by other users. See GoogleOauth for how to do this.

Only files created by Ganga can be deleted (or restored after deletion).

```
j = GangaFile("mydata.txt")
j.localDir = "~/temp"
j.put()

# if the file is required to be deleted
j.remove() # will send the file to trash, use permanent=True for deletion
# to restore the file from trash
j.restore()
```

To download files previously uploaded by ganga, use the *get* method:

```
# consider "mydata.txt" file was previously uploaded by ganga
j = GangaFile("mydata.txt")
j.localDir = "~/temp" # folder where the file should be downloaded
j.get()
```

## 1.11 Job Manipulation

There are several ways to control and manipulate your jobs within Ganga.

### 1.11.1 Copying Jobs

You can copy jobs using the `copy` method or using the cop-constructor in the Job creation. The job status is always set to `new`:

```
j = Job(name = 'original')
j2 = j.copy()
j2.name = 'copy'
j.submit()
j3 = Job(j, name = 'copy2')
jobs
```

```
Ganga Out [3]:
Registry Slice: jobs (4 objects)
--------------
    fqid |     status |        name | subjobs |     application |          backend |           ␣
↪                    backend.actualCE |                         comment
-------------------------------------------------------------------------------------
↪-----------------------------------------------------------------------
       0 | completed |             |         |      Executable |            Local |           ␣
↪             epldt017.ph.bham.ac.uk |
       1 | completed |    original |         |      Executable |            Local |           ␣
↪             epldt017.ph.bham.ac.uk |
       2 |       new |        copy |         |      Executable |            Local |           ␣
↪                                    |
       3 |       new |       copy2 |         |      Executable |            Local |           ␣
↪                                    |
```

### 1.11.2 Accessing Jobs in the Repository

As shown before, you can view all the jobs that Ganga is aware of using the `jobs` command. To access a specific job from the repo with the parentheses, use it's `id` number or:

```
jobs(2)
```

You can also use the square bracket (`[]`) notation to specify single jobs, lists of jobs or a job by a (unique) name:

```
jobs[2]
jobs[2:]
jobs['copy2']
```

### 1.11.3 Resubmitting Jobs

Jobs can fail for any number of reasons and often it's a transient problem that resubmitting the job will solve. To do this in Ganga, simply call the `resubmit` method:

```
jobs(0).resubmit()
```

Note that, as above, this can also be used on `completed` jobs, though it's backend and application dependent.

### 1.11.4 Forcing to Failed

Sometimes you may encounter a problem where the job has been marked `completed` by the backend but you notice in the logs that there was a problem which renders the output useless. To mark this job as `failed`, you can do:

```
jobs(1).force_status('failed')
```

Note that there are PostProcessors in Ganga that can help with a lot of these kind of problems.

### 1.11.5 Removing Jobs

As you submit more jobs, your Ganga repository will grow and could become quite large. If you have finished with jobs it is good practise to remove them from the repository:

```
jobs(2).remove()
```

This will remove all associated files and directories from disk.

### 1.11.6 Performing Bulk Job Operations

There are several job operations you can perform in bulk from a set of jobs. To obtain a list of jobs, you can either use the array syntax described above or the `select` method:

```
# can select on ids, name, status, backend, application
jobs.select(status='new')
jobs.select(backend='Local')
jobs.select(ids=[1,3])

# can restrict on min/max id
jobs.select(1,3, application='Executable')
```

Given this selection, you can then perform a number of operations on all of the jobs at once:

```
jobs.select(status='new').submit()
```

Available operations are: `submit`, `copy`, `kill`, `resubmit`, `remove`. These also take the `keep_going` argument which, if set to `True` will mean that it will keep looping through the jobs even if an error occurs performing the operation on one of them. These operations can also be performed on subjobs as well - see SplittersAndPostprocessors for more info.

### 1.11.7 Export and Import of Ganga Objects

Ganga is able to export a Job object (or a `selection` of Job objects) or any other Ganga object using the `export` method which will create a human readable text file that you can edit manually and then load in using `load`:

```
export(jobs(0), 'my_job.txt')
jlist = load('my_job.txt')
jlist[0].submit()
```

As in the above example, any jobs loaded will be put into the `new` state.

## 1.12 Configuration

There are several ways that you can configure and control how Ganga behaves. There are 3 different ways to do this:

1. Edit the options in your `~/.gangarc` file
2. Supply command line options: `ganga -o[Logging]Ganga.Lib=DEBUG`
3. At runtime using the *config* variable:

```
# print full config
config

# print config section
config.Logging

# edit a config option
config.Logging['GangaCore.Lib'] = 'DEBUG'
```

The config system also provides a set of `default_` options for each Ganga object which override what values the object starts with on creation. e.g.

```
config.defaults_Executable.exe = 'ls'
```

In addition to this, you can also supply a `~/.ganga.py` file that will be executed just as if you'd typed the commands when Ganga starts up e.g. this will show all running jobs when you start Ganga if put into the `~/.ganga.py` file:

```
slice = jobs.select(status='running')
print(slice)
```

## 1.13 Miscellaneous Functionality

Ganga provides quite a lot of additional functionality to help with job management. Below are the main ones:

### 1.13.1 Job Templates

If there is a version of a job that you use a lot, it can be beneficial to store this as a job template and then you can easily retrieve and then only alter a few parameters of. To create a template you do exactly what you would do for a normal job except you create a `JobTemplate` object instead of a `Job` object:

```
j = JobTemplate()
j.name = 'LsExeLocal'
j.application.exe = 'ls'
j.backend = Local()
```

To view the `templates` available, just do:

```
templates
```

You can then create a job from this template by doing:

```
j = Job(templates[0], name = 'JobFromTemplate')
j.submit()
```

### 1.13.2 Job Trees

As you submit more jobs of different types, it can become quite difficult to keep track of them. Ganga supports a *directory tree* like structure for jobs so you can easily keep track of which jobs are associated with different calibrations, analyses, etc. Jobs are stored by *id* and can be thought of as soft links to the main Ganga Job Repository.

```
# show the current job tree (empty to start with)
jobtree

# make some dirs and subdirs
jobtree.mkdir('test_old')
jobtree.mkdir('test')
jobtree.mkdir('prod')
jobtree.mkdir('/test/jan')
jobtree.mkdir('/prod/full')

# have a look at the tree
jobtree.printtree()

# remove a dir
jobtree.rm('test_old')

# create some jobs and add them
jobtree.cd('/test/jan')
jobtree.add( Job() )
jobtree.cd('/prod/full')
jobtree.add(Job())
jobtree.add(Job())

# look at the tree again
jobtree.printtree()

# submit the some jobs
jobtree.getjobs().submit()
```

### 1.13.3 GangaBox

## 1.14 Queues

Many tasks in Ganga can take a lot of time from job submission to output download. Several things are already handled in the background by the Monitoring System, but you may have user tasks that you want to also push into the background that can run in parallel. This is where `queues` can be used.

To start with, you can view the state of the background threads by just typing `queues`:

```
Ganga In [38]: queues
Ganga Out [38]:
                        Ganga user threads:                          |            ␣
↪Ganga monitoring threads:
                        ------------------                           |         ---
↪---------------------
Name                       Command                    Timeout        | Name       ␣
↪            Command                      Timeout
----                       -------                    -------        | ----       ␣
↪            -------                      -------
```

(continues on next page)

(continued from previous page)

```
User_Worker_0              idle                        N/A               | Ganga_Worker_0 ␣
↪           idle                   N/A
User_Worker_1              idle                        N/A               | Ganga_Worker_1 ␣
↪           idle                   N/A
User_Worker_2              idle                        N/A               | Ganga_Worker_2 ␣
↪           idle                   N/A

Ganga user queue:
----------------
[]
Ganga monitoring queue:
----------------------
[]

[12:57:37]
Ganga In [39]:
```

To add a function call to the queue such as a submit call, do the following:

```
for i in range(1, 10):
    j = Job()
    queues.add(j.submit)
```

You can also supply your own functions as well as provide arguments to these functions:

```
def f(x):
    print(x)

queues.add(f, args=(123,))
```

Queues can also be used to submit subjobs in parallel:

```
j = Job()
j.splitter = GenericSplitter()
j.splitter.attribute = 'application.args'
j.splitter.values = [i for i in range(0, 10)]
j.parallel_submit = True
j.submit()
```

## 1.15 Tasks

### 1.15.1 Introduction to Tasks

Even with Ganga, you can find that you may find managing a large set of jobs and steps in an analysis to be a bit cumbersome. The GangaTasks package was developed to help with these larger scale analyses and remove as much of the 'busy work' as possible. It can automatically submit jobs to keep a set number running, it can create new jobs when others complete and chain their data together, it can automatically transfer data around as required and a number of other things as well. As with all of Ganga it is based on the plugin system and so you can easily extend some elements of it to better suit your requirements.

GangaTasks essentially adds 3 new objects that control all aspects of the overall task:

- **Task** This is overall 'container' for the steps in your analysis. It is fairly light weight but is used to aggregate the overall status of the task and control overall settings, numbers of jobs, etc.

- **Transform** This is where most things occur. It is in some ways analogous to a Job Template in that it mostly contains the objects that will be assigned to the created jobs. This is where new Units are created and data is transferred between steps. You will generally have a Transform per 'step' or 'type' of job that you want to run.
- **Unit** This is the 'control class' for any created jobs and contains all the job-specific information (e.g. input data, application settings, etc.) that each actual Job will be setup with. After all the units in a Transform are created, each unit then creates a new Job and attempts to submit it. It will monitor the status of the job and will do any necessary actions (e.g. download output data) upon completion. If the job fails and it seems sensible to do so, it will also resubmit or recreate the job.

A typical example of how this structure works would be in a two stage analysis where you generate some MC in the first step and then run some analysis code on the output of this data. You would create an overall Task to manage both steps. Each step would have an associated Transform with the first being setup as MC generation and the second doing the analysis. You would set the input data of the second transform to be the output data of the first. Then, while running your Task, Units will be created to cover the number of events you wanted to create and jobs will be submitted for each of these. As these complete new units and jobs will be created by the analysis Transform to cover that step.

### 1.15.2 Basic Core Usage

It's quite likely you will want to develop your own plugins to maximise your use of GangaTasks, however there is a set of generalised classes that can get you started. Typical use of these is shown below:

```
# First create the overall Task
t = CoreTask()

# Now create the Transform ( -> Job template)
trf = CoreTransform()
trf.application = Executable()
trf.backend = Local()

# Set the unit splitter (unique to CoreTransform - you may have better ways of␣
↪creating units in your own
# plugins). This will create a unit based on the splitting of any given splitter
# If you put in your own splitter here, use the trf.fields_to_copy string list to␣
↪tell Tasks which fields of
# a Job to preserve from the split. Here, Tasks already knows about GenericSplitter␣
↪and knows that we want to
# change the 'application' object for each Unit/Master Job
trf.unit_splitter = GenericSplitter()
trf.unit_splitter.attribute = "application.args"
trf.unit_splitter.values = ['arg 1', 'arg 2', 'arg 3']

# Append the transform
t.appendTransform(trf)

# set the maximum number of active jobs to have running (allows for throttling)
t.float = 100

# run the Task
t.run()
```

After running the above commands you won't see much happen initially as Tasks runs on a separate monitoring loop that triggers every 30s (configurable in `~/.gangarc`). Eventually though you will see the units created and then jobs for each of these units will be submitted. To see the progress of your tasks use:

```
tasks
tasks(0).overview()
```

Tasks can also take advantage of using queues for submission as well. Simply add:

```
# note - done at the transform level rather than task level as different backends may␣
↪not need it
trf.max_active_threads = 10  # optional - specifies the max number of submissions to␣
↪queue up
trf.submit_with_threads = True
```

### 1.15.3 Job Chaining

The Tasks package also allows you to chain jobs together, i.e. have the output of one job be the input of another. This is done by setting the input data of the dependant Transform to be `TaskChainInput` type and giving the ID of the Transform is depends on. You can have multiple transforms feed into one Transform by specifying more `TaskChainInput` datasets.

A typical example is shown below:

```
# Create a test script
open('my_script3.sh', 'w').write("""#!/bin/bash
echo $PATH
ls -ltr
more __GangaInputData.txt__
echo "MY TEST FILE" > output_file.txt
sleep 120
""")

# Create the parent task
t = CoreTask()

# Create the first transform
trf1 = CoreTransform()
trf1.application = Executable()
trf1.application.exe = File('my_script3.sh')
trf1.outputfiles = [LocalFile("*.txt")]
d = GangaDataset()
d.files = [LocalFile("*.txt")]
d.treat_as_inputfiles = True
trf1.addInputData(d)
trf1.files_per_unit = 1
trf1.submit_with_threads = True

trf1.splitter = GangaDatasetSplitter()
trf1.splitter.files_per_subjob = 2

trf1.backend = Local()
t.appendTransform(trf1)

# Create the second transform
trf2 = CoreTransform()
trf2.application = Executable()
trf1.application.exe = File('my_script3.sh')
trf2.submit_with_threads = True
```

(continues on next page)

(continued from previous page)

```
d = TaskChainInput()
d.input_trf_id = trf1.getID()
trf2.addInputData(d)

trf2.splitter = GangaDatasetSplitter()
trf2.splitter.files_per_subjob = 2

trf2.backend = Local()
t.appendTransform(trf2)

# Set the Task running
t.float = 1
t.run()
```

Sections still to be added:

- Pure Python Usage
- Ganga as a Service
- Developing a New Application

# CHAPTER 2

# GPI

The Ganga Public Interface provides a wrapper around the internals of the Python implementation to provide safety and to increase ease-of-use.

classes

functions

plugins

Sections still to be added:

- Pure Python Usage
- Ganga as a Service
- Developing a New Application

CHAPTER 3

# Guide for System Administrators

This section of the manual is intended for system administrators or interested individuals to describe how to install and manage Ganga.

## 3.1 Installation

Historically Ganga was installed via a custom `ganga-install` script which would fetch the latest version and its dependencies. We have since migrated away from that and there are two primary ways to get access to Ganga, one of which is mostly of interest only to particle physicists.

### 3.1.1 pip

At its simplest it is possbile to install ganga using the standard Python `pip` tool with a simple

```
pip install ganga
```

### 3.1.2 CVMFS

CVMFS is a read-only file system intended for distributing software originally developed for the CERN virtual machine infrastructure.

`/cvmfs/ganga.cern.ch/`

## 3.2 Site config

It's often the case that you want to specify default configuration settings for your users, perhaps on a group-by-group basis. You can do this by placing `.gangarc`-style INI files in a common directory on your system and pointing Ganga at it. The order of precedence for a particular setting goes `default` → `site config` → `user config`

→ `runtime setting` with those later in the chain overriding those earlier. The location that Ganga looks for the site config is controlled with an environment variable, `GANGA_SITE_CONFIG_AREA`, which you could set in your users' default shell setup.

```
GANGA_SITE_CONFIG_AREA=/some/physics/subgroup
```

Files in this directory should be named after the Ganga version that you want to affect. They should start with the version number with the `.` replaced with `-` and can have any extension. So if you have three config files:

```
$ ls $GANGA_SITE_CONFIG_AREA
6-0-44.ini  6-1-6.ini  6-1-10.ini
```

and the user is running Ganga 6.1.6 then `6-0-44` and `6-1-6` will be loaded and `6-1-10` will be ignored.

CHAPTER 4

# Guide for developers

This document is intended to detail some of the inner workings of Ganga to both document what we have done as well as make it easier for new developers to get on-board quicker.

## 4.1 GangaObject

At the core of a lot of Ganga is `GangaObject`. This is a class which provides most of the core functionality of Ganga including persistency, typed attribute checking and simplified construction.

**Note:** There is currently some work being done to replace the existing implementation if `GangaObject` with a simpler version. The user-facing interface should not change at all but more modern Python features will be used to simplify the code. This will also affect how schemas are defined but not how they are presented or persisted.

## 4.2 Schema

The schema of a `GangaObject` defines the set of attributes belonging to that class along with their allowed types, access control, persistency etc. Each `GangaObject` must define a schema which consists of a schema version number and a dictionary of `Items`. Schema items must define their name and a default value and can optionally define a lot more such as a list of possible types and documentation string.

## 4.3 Proxy objects

In order to provide a nice interface to users, Ganga provides a Ganga Public Interface which fulfils two main purposes. Firstly it is a reduced set of objects so that the user is not bombarded with implementation details such as `Node`. Secondly, all `GangaObjects` available through the GPI are wrapped in a runtime-generated class called a *proxy*.

These proxy classes exist for a number of reasons but primarily they are there for access control. While a `GangaObject` can has as many functions and attributes as it likes, only those attributes in the schema and those methods which are explicitly exported will be available to users of the proxy class.

When working on internal Ganga code, you shuold never have to deal with any proxy objects at all. Proxies should be added to objects as they are passed to the GPI and should be removed as they are passed back.

### 4.3.1 Attributes on proxy objects

Proxy classes and the object that they are proxying have a set number of attributes which should be present.

If an object inherits from `GangaObject` the class can have the property `_proxyClass` set which will point to the relevant `GPIProxyObject` subclass. This is created on demand in the `addProxy` and `GPIProxyObjectFactory` methods. The proxy class (which is a subclass of `GPIProxyObject` and created using `GPIProxyClassFactory()`) will have the attribute *_impl* set to be the relevant `GangaObject` subclass.

When an instance of a proxy class is created, the *_impl* attribute of the instance will point to the instance of the `GangaObject` that is being proxied.

## 4.4 Repository

A repository is the physical storage of data on disk (usually persisted `GangaObjects`) as well as library interface to it.

## 4.5 Registry

A registry is an in-memory data-store which is backed by a repository.

## 4.6 Job monitoring

## 4.7 IGangaFile

All file types as of Ganga 6.1 inherit from `IGangaFile`. This main exception to this is the `File` object which as of 05/05/2016 is used as it still has more features than the `IGangaFile` inheirted classes do.

| Script Generator | When is it used? |
|---|---|
| getWNScriptDownload-Command | This generates a script which will make the file accessible from the WN when the job starts running |
| getWNInjectedScript | This generates a script which will send the file to the remote directory from the WN with no client intervention |

| Special attr | Use/Doc | Return type |
|---|---|---|
| lfn | Unique to the DiracFile. This is the LFN of the file in the DFC | str |
| getReplicas | Unique to DiracFile returns a list of SE where the file is replicated | list of str |
| ‘_list_get__match__()’ | IGangaFile, performs a type match on file objects. can we remove this? | bool |

# CHAPTER 5

# API Documentation

The documentation on these pages is automatically generated from the source code. It provides an overview of all the Python objects in the code base and as such is intended for Ganga core and extension developers. If you are a user of Ganga then you probably want the *GPI documentation*.

All Ganga modules