ngas Documentation

Release 12.0

ICRAR

Jul 04, 2023

Contents

Contents:

CHAPTER 1

Introduction

The Next Generation Archive System (NGAS) is a very feature rich, archive handling and management system. In its core it is a HTTP based object storage system. It can be deployed on single small servers, or in globally distributed clusters.

Some of its main features are:

- Basic archiving and retrieval of data
- Data checking via various checksum methods
- Server-side data compression and filtering
- Automatic mirroring of data
- Clustering and swarming
- Disk tracking and offline data transfer
- High customisation via user-provided plug-ins

NGAS is written in Python (3.5+, support for 2.7 will be removed soon), making it highly portable. There are a few dependencies on C libraries, which may restrict the ability to install it on some of the more exotic platforms.

CHAPTER 2

Changelog

12.0

- Using `sendfile(2)` when POSTing files through HTTP connections. This should lower the overhead of using python to perform the transfer, bringing benefits to `ngamsPClient` class and command-line tool, and to the subscription thread.
- Improved NGAS's mail sending capabilities. The code is now automatically tested by our unit tests, which allowed us to ensure it sends mails correctly, and works with python3. On top of that mail messages are now created using the standard library modules for correct mail composition, instead of the "hand-written" logic we had previously.
- Fixed a small issue during backlog file archival where the checksum plugin name was incorrectly recorded in the database. This shouldn't affect operational installations, and most database drivers happily worked with the previous code, but the Sybase driver caught this one.
- HTTP requests for archival with `Content-Length` equals to 0, or with missing `Content-Length`, now return an 411 HTTP status code instead of the more generic 400 HTTP status code.
- Cleaned up and aligned the way in which volume information is created and processed by NGAS. The code around volume creation and scanning has been revised, unit tests have been improved to test the functionality more thoroughly, and the existing, not-well maintained script has now been *better integrated* into the NGAS ecosystem. Instructions on how to *set up a volume directory* have also been updated,
- Renamed the old `ngasArchiveClient` tool into `fs-monitor`, and moved it into the `ngamsPClient` package. *This utility* had been kept until now under the unmaintained `ngasUtils` package, and therefore hadn't been ported alongside the rest of the code until now. Not only was this tool renamed, but it was completely overwritten to simplify its maintenance in the long run, and to enable easy unit testing, which we have also added now. Moving it into the `ngamsPClient` ensure it will have continued testing and visibility.
- Added tests to ensure all plugin modules can be correctly imported. This ensures the code is compatible with python 3 up to some degree, and it also increases our code coverage.
- Unit tests don't need to be run from within the `test` directory anymore. This makes using unit test tools like `pytest` or the built-in `unittest` module easier to use.

- Fixed issues with the `BBCPARC` command, which didn't work for remote host transfers, only for `localhost` ones.
- Fixed small issue in `QUERY` command where column names where not correctly aligned with the underlying column data.
- Added support for using HTTPS with both the client and with subscriptions. Support for wrapping the server with TLS has also been added, but this should only be used for testing, rather than in production (we recommend handling HTTPS with more traditional HTTPS servers such as Apache or Nginx).
- Added support for using more diverse authentication plugins for subscription, via the authentication mechanisms provided by requests. Note that with the current setup, the authentication plugins can only be used with HTTPS (this may change in the future).
- Default FITS plug-in doesn't mandate an `ARCFILE` keyword to be present in the incoming FITS file if the `ignore_arcfile=1` option is given in the HTTP parameters. This is useful for archiving FITS files that don't have this keyword, but that want to use the default FITS plug-in.
- Removing the `create_venv.sh` script from the sources, in favour of letting users create one by themselves if they want, or let the fabric tasks create one.
- Added *Exclude* attribute to the *Authorization* element for defining a list of commands that are to be excluded from authorization
- Added new partner sites feature that provides the capability of configuring a remote NGAS cluster as a proxy for retrieving files that are not available in the local NGAS cluster.
- Fixed the mirroring plugin modules: *ngamsCmd_HTTPFETCH.py*, *ngamsCmd_MIRRARCHIVE.py*, *ngamsCmd_MIRREXEC.py* and *ngamsCmd_MIRRTABLE.py*
- Ported NGAS utility scripts: *ngasCheckFileCopies.py*, *ngasCheckFileList.py ngasDiscardFiles.py*, *ngasVerifyCloning.py* and *ngasXSyncTool.py*
- Fixed an issue with the subscription mechanism, where upstream files with checksum values calculated with checksum variants other than `crc32` failed to be pushed downstream.
- Fixed logging of C utilities, and implemented the logic behind the `-v` flag of the `ngamsCClient` program.
- Improved error message sent back by the `REGISTER` command when registration of a file with a MIME type with no configured plug-in is requested.
- The `bad-files` directory now exists on each volume rather that there being a single one outside of any volume directory. This allows for faster movement of files into the `bad-files` directory, a more consistent directory structure, and better traceability of files that end up in `bad-files`.
- The ngamsDaemon.py now checks the *-force* command line option and will forcibly start-up and clean up any existing PID lock files.
- Lots of code clean up for the mirroring plugin code using PEP8 style guidelines. Replaced deprecated rfc822 module with email module in the ngamsDAPIMirroring.py and ngamsAlmaMultipart.py plugins. Added test_dapi_mirroring.py unit tests. General code clean up changes in both mirroring and SDM multipart plug-ins.
- Fixed 'Connection refused' exception on mirror plugin start-up. Now logs a warning when NGAS server is not available. Now uses exception class instead of string for handling stop events.

11.0.2

- Fixed an important bug that was preventing the `STATUS` command from being imported correctly in normal NGAS installations.

11.0.1

- Fixed an important bug that was causing data to be removed from the NGAS root directory.

11.0

- Initial python 3 support. The code not only correctly imports under python 3, but also all unit tests pass correctly. The code is both 2.7/3.5+ compatible, so users don't need to immediately switch to python 3. Given that our test coverage currently sits at about 65%, it is likely that there are code paths that need further work.
- *Command plug-ins* can be implemented as user-provided plug-ins. This was almost the case until now, as they still had the restriction of having to reside on the `ngamsPlugIns` package, which is not the case anymore. Moreover, a single python module can implement the logic of more than one command.
- Unit tests can be run against *arbitrary filesystems*, and they default to run under `/dev/shm` for faster execution.
- Added new CRC variant called `crc32z`. It behaves exactly like `crc32`, except that its values, *as stored in the database*, should be consistent across python 2.7 and 3. The `crc32` variant does not have this property, although we can still (and do) normalize them when checking files' checksums.
- Changed the server to use a thread pool to serve requests instead of creating a brand new thread every time a request comes in.
- Improving how the *RETRIEVE* command works when returning compressed files.
- Adding support to the `CRETRIEVE` command to retrieve all files as a tarball. It internally uses `sendfile(2)` when possible.
- Users can configure NGAS to issue a specific SQL statement at connection-establishment time, similarly to how other connection pools do.
- Fixed a few details regarding expected v/s real datatypes used in some SQL queries. These affected only the Sybase ASE official driver, which is now working correctly.
- Unit tests moved to the top-level `test` directory, and renamed to `test_*.py`. This makes it more straightforward to use unit test runners which usually rely on this layout for test discovery.
- A new sample configuration file replaces the old, large set of configuration files that used to be shipped with NGAS.
- Starting a server in cache mode is now be done via a configuration file preference rather than a command-line argument.
- The subscription code and the cache handling thread update the file status flags atomically. Before they had a race condition which resulted in files not being deleted on the cache server.
- Improving handling of overwriting flags for archiving commands. Now all archiving commands obey the same logic, which has been detached from the individual data-archiving plug-ins.
- Improving and simplifying the `QUERY` command.
- Removed many unnecessary internal usage of `.bsddb` files.
- Added a MacOS build to our Travis CI set up.
- Misc bug fixes and code improvements.

10.0

- The `ARCHIVE`, `QARCHIVE`, `REARCHIVE` and `BBCPARC` commands now use the same underlying code. All the small differences between the commands has been kept, so they should behave exactly as before. This was a required step we needed to take before implementing other improvements/bugfixes.
- The archiving commands listed above are now more efficient in how they calculate the checksum of the incoming data. If the data archiving plug-in promises not to change the data, then the checksum is calculated on the incoming stream instead of calculating it on the file, reducing disk access and response times. This behavior was previously not seen neither on the `ARCHIVE` command, which always dumped all contents to disk and then did a checksum on the on-disk contents, nor in the `QARCHIVE` command, which **unconditionally** calculated the checksum on the incoming stream, irrespective of whether the data archiving plug-in changed the data afterward or not.
- Partial content retrieval for the `RETRIEVE` command has been implemented. This feature was present in the ALMA branch of the NGAS code, and now has been incorporated into ours.
- We merged the latest ALMA mirroring code into our code base. This and the point above should ensure that NGAS is ALMA-compatible.
- Unified and centralized all the CRC checksuming code, and how different variants are chosen.
- We have improved response times for scenarios when many parallel `RETRIEVE` commands are issued. Worst-case scenario times in 100 parallel request scenarios were brought down from tens of seconds to about 2 seconds (i.e., an order of magnitude).
- Moved the *data-check* background thread checksum to a separate pool of processes to avoid hanging up the main process. The checksuming also pauses/resumes depending on whether the server is serving any requests or not to avoid exhausting access to the disk.
- Added the ability to write plug-ins that will react to each file archiving (e.g., to trigger some processing, etc).
- Added support for the latest bbcp release, which includes, among other things, our contributions to add support for the `crc32c` checksum variant, plus other fixes to existing code.
- Fixed a few small problems with different installation scenarios.

9.1

- NGAS is now hosted in our public GitHub repository.
- Travis CI has been set up to ensure that tests runs correctly against SQLite3, MySQL and PostgreSQL.
- User-provided plug-ins do not need to be installed alongside NGAS anymore. This allows users to place their plug-ins in their own personally-owned directories, which in turn allows to install NGAS in isolation, and probably with more strict permissions.
- Project-specific plug-ins under the `ngamsPlugIns` package have been moved to sub-packages (e.g., `ngamsPlugIns.mwa`), and will eventually be phased out as projects take ownership of their own plug-ins.
- *Janitor Thread* changes:
  - Plug-ins: Instead of having a fixed, single module with all the business logic of the Janitor Thread, its individual components have been broken down into separate modules which are loaded and run using a standard interface. This makes the whole Janitor Thread logic simpler. It also allows us to implement users-written plug-ins that can be run as part of the janitor thread.
  - The execution of the Janitor Thread doesn't actually happen in a thread anymore, but in a separate process. This takes some burden out from the main NGAS process. In most places we keep calling it a thread though; this will continue changing continuously as we find these occurrences.

- The NGAS server script, the daemon script and the SystemV init script have been made more flexible, removing the need of having more than one version for each of them.
- Some cleanup has been done on the NGAS client-side HTTP code to remove duplicates and offer a better interface both internally and externally.
- Self-archiving of logfiles is now optional.
- A few occurrences of code incorrectly handling database results have been fixed, making the code behave better across different databases.
- Misc bug fixes and code cleanups.

9.0

- Switched from our `pcc`-based, own home-brewed logging package to the standard python logging module.
- Unified time conversion routines, eliminating heaps of old code
- Removed the entire `pcc` set of modules.
- General bug fixes and improvements.

8.0

- Re-structured NGAS python packages. Importing NGAS python packages is now simpler and doesn't alter the python path in any way. The different packages can be installed either as zipped eggs, exploded eggs, or in development mode. This makes NGAS behave like other standard python packages, and therefore easier to install in any platform/environment where setuptools or pip is available.
- `RETRIEVE` command uses `sendfile(2)` to serve files to clients. This is more efficient both in terms of kernel-user interaction (less memory copying), and python performance (less python instructions have to be decoded/interpreted, needing less GIL locking, leading to better performance and less multithread contention).
- Initial support for logical containers. Logical containers are groups of files, similar to how directories group files in a filesystem.
- NGAS server replying with more standard HTTP headers (e.g., `Content-Type` instead of `content-type`). Most HTTP client-side libraries are lenient to these differences though.
- Streamlined `crc32c` support throughout `QARCHIVE` and subscription flows. We use the crc32c module for this, which was previously found as part of NGAS's source code, but that has been separated into its own package for better reusability.
- Stabilization of unit test suite. Now the unit test suite shipped with NGAS runs reliably on most computers. This made it possible to have a continuous integration environment (based on a private Jenkins installation) to monitor the health of the software after each change on the code.
- Improved SQL interaction, making sure we use prepared statements all over the place, and standard PEP-249 python modules for database connectivity.
- Improved server- and client-side connection handling, specially error-handling paths.
- General bug fixes and improvements.

CHAPTER 3

Installation

Contents

3.1 Installing from source

First, get the latest NGAS sources:

```
git clone https://github.com/ICRAR/ngas
```

Installing NGAS from source is pretty straight-forward. There are two ways to perform an installation:

- *Manually*.
- *Using Fabric*. This is the recommended way, if possible, as it automates most of the installation steps while still being highly customisable.

3.1.1 Manual installation

Note: Like any other python package, NGAS can be installed in a virtual environment or as a system-wide package.

To manually install NGAS go to the root directory and simply run:

```
./build.sh
```

Run `./build.sh -h` to see the full set of options. Among the options users can choose whether to skip the build of the C-written clients, to install the python packages in development mode, and to build NGAS without CRC32c support (see *CRC* for details).

The script will (optionally) build and install the C NGAS client first, and then will build and install each of the python modules (i.e., `ngamsCore`, `ngamsPClient`, `ngamsServer` and `ngamsPlugIns`). The python modules will automatically pull and install their dependencies.

Note: If any step of the build fails the script will stop and notify the user. It is the user's responsibility to install any missing build dependencies. To avoid these issues you can also try the *Fabric installation*

In addition, the `build.sh` script recognizes when a virtual environment is loaded, and will install the C client on it, as well as the python modules.

The C client compilation is based on `autotools`, meaning that it can also be manually compiled and installed easily via the usual:

```
$> ./bootstrap
$> ./configure
$> make all
$> make install
```

The Python modules are all setuptool-based packages, meaning that they can also be manually compiled and installed easily via the usual:

```
$> python setup.py install
```

3.1.2 Via Fabric

Note: The installation via Fabric always installs NGAS in a virtualenv

Fabric is a tool that allows to perform commands in one or more hosts, local or remote (via SSH). NGAS comes with a set of fabric modules to ease the installation of NGAS in complex scenarios, and to automate most of the system-level preparation tasks. This enables not only a simple procedure for installing NGAS in any host or hosts at a given time, but also the customization of the hosts as necessary, plus any other extra step required by other scenarios.

Fabric's command-line allows users to specify the username and hosts where tasks will take place, and a set of variables to be defined. For example:

```
fab -H host.company.com -u user some_task --set VAR1=a,VAR2
```

In the example the instructions of the task `some_task` will be carried out in host `host.company.com` with the user `user`, and the `VAR1` variable will be set to the value `a`, while variable `VAR2` will be marked as set.

For a complete list of tasks run `fab -l`. For a detailed description of a task run `fab -d <task>`. For a more complete manual visit Fabric's documentation.

The two main fabric tasks NGAS provides are:

- *User installation*: This task compiles and installs NGAS under a user-owned directory.
- *System installation*: The recommended task if you have *sudo* access to the target machine. This task installs all the necessary dependencies in the system before compiling and installing NGAS.

Basic per-user installation

To compile and install NGAS in a user-owned directory run:

```
fab hl.user_deploy
```

This task will:

- Check that SSH is working on the target host
- Copy the NGAS sources to the target host
- Compile and install NGAS into a virtualenv on the target host
- Create a minimal, working NGAS root directory
- Finally, modify the corresponding `~/.bash_profile` file to automatically load the NGAS virtualenv when the user enters a `bash` shell.

The user on the target host used for running these tasks is the SSH user given to fabric via the command line (`fab -u <user>`).

This task doesn't take care of installing any dependencies needed by NGAS, assuming they all are met. For a more complete automatic procedure that takes care of that see the `hl.operations_deploy` task.

The following fabric variables (set via the `--set` command-line switch) are available to further customize the process:

Variable	Description	Default value
NGAS_SRC_DIR	The directory where the NGAS sources will be extracted on the target host	`~/ngas_src`
NGAS_INSTALL_DIR	The directory where the virtualenv will be created and NGAS installed	`~/ngas_rt`
NGAS_ROOT_DIR	The NGAS root directory created by default by the installation procedure	`~/NGAS`
NGAS_REV	The git revision of the sources used to compile and install NGAS (only for sources from a git repository). Keep in mind that after cloning, you might have a reference to remote branches without having **local** branches with the corresponding names, so using `NGAS_REV=v10` might not work, but `NGAS_REV=origin/v10` will.	`HEAD`
NGAS_OVERWRITE_INSTALLATION	If specified, an existing installation directory will be overwritten	Not specified
NGAS_OVERWRITE_ROOT	If specified, an existing NGAS root directory will be overwritten	Not specified
NGAS_USE_CUSTOM_PIP_CERT	If specified, configure pip to use curl.haxx.se/ca/cacert.pem as the root TLS certificate. In some old platforms this is needed so pip trusts PyPI downloads	Not specified
NGAS_EXTRA_PYTHON_PACKAGES	Comma-separated list of extra python packages to install	Not specified

For example, to install the tip of the `v8` branch as user `foo` in hosts `bar1` and `bar2`, and without compiling the C client, the following command would do:

```
fab hl.user_deploy -u foo -H bar1,bar2 --set NGAS_NO_CLIENT,NGAS_REV=v8
```

Total system setup

Note: `sudo` must be installed and configured in the target host for this task to work properly. Also, the user used with fab (`fab -u <user>`) needs to be properly configured on the target host to use `sudo` commands.

To perform a system-wide setup and NGAS install run:

```
fab hl.operations_deploy
```

This task will:

- Check that SSH is working on the target host
- Check that `sudo` is installed (`sudo` is used to run commands as root).
- Install all necessary system packages (using the OS-specific package manager) for compiling NGAS and its dependencies
- Compile and install a suitable version of python (2.7) if necessary
- Create the `NGAS_USER` if necessary
- Proceed with the normal NGAS compilation and installation as performed by `hl.user_deploy`
- Install an `/etc/init.d` script for automatic startup of the server.

The user on the target host used for running the `sudo` commands is the SSH user given to fabric via the command line (`fab -u <user>`).

On top of the normal fabric variables used by `hl.user_deploy` the following additional variables control this script:

Variable	Description	Default value
NGAS_USER	The user under which the NGAS installation will take place	`ngas`
NGAS_EXTRA_PACKAGES	Comma-separated list of extra system-level packages to install on the target system(s)	Not specified

Currently supported OSs are Ubuntu, Debian, Fedora, CentOS, and MacOSX Darwin, but more might work or could be added in the future.

3.1.3 Other Fabric tasks

On top of the *two main Fabric tasks* to install NGAS, our fabric modules define a number of other optional, high-level tasks that can be useful in other scenarios.

AWS deployment

Note: The `boto` module is required for using this install option.

The fabric modules contain routines to create an NGAS installation on AWS machines. This is performed by running:

```
fab hl.aws_deploy
```

This procedure will: * Create and bring up the required AWS instances * Wait until they are fully operational, and * Perform a `hl.operations_deploy` on the instances.

On top of the normal fabric variables used by `hl.user_deploy` and `hl.operations_deploy` the following additional variables control the AWS-related aspects of the script:

Variable	Description	Default value
AWS_PROFILE	The profile to use when connecting to AWS	`NGAS`
AWS_REGION	The AWS region to connect to	`us-east-1`
AWS_KEY_NAME	The private SSH key to be used to create the instances, and later to connect to them	`icrar_ngas`
AWS_AMI_NAME	The name associated to an AMI (from a predetermined set of AMI IDs) which will be used to create the instance	`Amazon`
AWS_INSTANCES	The number of instances to create	`1`
AWS_INSTANCE_TYPE	The type of instances to create	`t1.micro`
AWS_INSTANCE_NAME	The name of instances to create	`NGAS_<rev>`
AWS_SEC_GROUP	The name of the security group to attach to the instances (will be created if it doesn't exist)	`NGAS`
AWS_ELASTIC_IPS	A comma-separated list of public IPs to associate with the new instances, if specified.	Not specified

For example, to create 3 instances of type `t3.micro` on region `us-east-2` one would run:

```
fab hl.aws_deploy --set AWS_REGION=us-east-2,AWS_INSTANCES=3,AWS_INSTANCE_TYPE=t3.
↪micro
```

To assist with AWS-related procedures the following other tasks are also available:

```
fab aws.list_instances
fab aws.terminate_instance:instance_id=<the-instance-id>
```

Docker Image

Note: These instructions are to *build* a docker image with NGAS. If you simply want to *use* the pre-built images see *Docker container*.

Note: The `docker` python package is required to use of this install option. Also, a local docker daemon must be running and the current user must have access to perform docker operations.

To build a Docker container containing an NGAS installation simply run:

```
fab hl.docker_image
```

This will generate an image called `icrar/ngas:latest` based on CentOS 7. When started, the container by default will run the NGAS server. The NGAS server will look for a configuration file under `/home/ngas/NGAS/cfg/ngamsServer.conf`, which by default needs to be provided via volume mapping.

This task will:

- Create an initial Docker image with SSH on it
- Start a container using that image
- Perform a `hl.operations_deploy` on the container
- Perform some cleanups on the container, including removing the `NGAS_ROOT` directory
- Commit the container and create the final Docker image

On top of the normal fabric variables used by `hl.user_deploy` and `hl.operations_deploy` the following additional variables control the Docker-related aspects of the task:

Variable	Description	Default value
DOCKER_KEEP_NGAS_ROOT	If specified, the NGAS root directory will still be present in the final image	Not specified
DOCKER_KEEP_NGAS_SRC	If specified, the NGAS source directory will still be present in the final image	Not specified
DOCKER_IMAGE_REPOSITORY	The repository for the final image produced by this task	`icrar/ngas`

3.2 Docker container

Alternatively, if you don't need to build NGAS from sources you can use the pre-built docker image we distribute for NGAS. To do so you can do:

```
docker pull icrar/ngas:latest
```

Then follow the instructions in the DockerHub page to get yourself started.

CHAPTER 4

Post-installation

Note: If NGAS was installed in a virtual environment remember to source it before proceeding.

4.1 Setting up an NGAS instance

These steps describe how to set up an NGAS server instance.

When using on of the *fabric-based installation* procedures an NGAS root directory is automatically created and prepared under `~/NGAS` of the user hosting the NGAS installation (or somewhere different if indicated via the `NGAS_ROOT` fabric variable)

4.1.1 Create an NGAS root directory

NGAS's *root* is the top-level directory that will be used to store all its internal files, including the data being stored.

The NGAS root directory can be placed anywhere in the filesystem, and can be totally empty initially. The only requirement is that it is writable by the user running the NGAS server.

To help users create their an NGAS root directory, NGAS comes with a `prepare_ngas_root.sh` script. The script needs at least the name of the target directory that will be used as NGAS root. More options are available, you can use `prepare_ngas_root.sh -h` to learn more.

The `prepare_ngas_root.sh` script will also create simple, but usable, *volumes* under the NGAS *root* directory. These are sufficient for testing purposes, but you may want to setup proper volumes (see next section).

4.1.2 Setup volumes

Inside the NGAS *root* directory *volumes* should exist where the data will be stored (see *Storage organization* for a full explanation). Volumes need only be directories, so you can either create directories for each volume in simple or

testing scenarios, or symbolic link actual partitions as separate volumes for more demanding, real-world setups – it's your choice.

In any case, volumes need to be tagged as such in order to be recognized by NGAS. This is done by placing a small, hidden file in the root of the volume containing a random UID for the disk using the `ngas-prepare-volume` utility.

For example, if the NGAS *root* directory is under `~/NGAS` and a new volume called `volume1` is created, it can be tagged as such:

```
$ cd ~/NGAS
$ mkdir volume1
$ cd volume1
$ ngas-prepare-volume $PWD
```

Answer Yes and you're done. Try running `ngas-prepare-volume -h` to list all available options.

To let NGAS know about your volumes check the *StorageSets* configuration option.

4.2 Running the server

Note: In case you haven't yet, please review how to *setup an NGAS server instance* before you start to start the server for the first time.

After a successful installation and setup, you should be able to run the server by running:

```
ngamsServer -cfg <configFile>
```

For a full list of all command-line flags run `ngamsServer -h`. In particular, when running manually you will probably want to use the `-autoonline` flag to bring the server to the `ONLINE` state immediately, and `-v 4` to increase the output of the server to the `INFO` logging level.

To start the NGAS server as a daemon run instead:

```
ngamsDaemon start <params>
```

The NGAS daemon accepts also the `stop` and `status` commands. Any parameters given in `<params>` will be passed down verbatim to the `ngamsServer` being started, and thus should include at least the configuration file flag.

4.3 Running the client

To run the NGAS (python) client run the following command:

```
ngamsPClient
```

For a full list of all command-line flags run `ngamsPClient -h`.

Likewise, the NGAS C client (if installed) is run with the following command:

```
ngamsCClient
```

As a more complex test, the following command can be used to execute the client and issue an ARCHIVE command to the server. If successful, this will signal that the whole installation is working fine:

```
ngamsPClient ARCHIVE --file-uri $(which ngamsPClient) --mime-type application/octet-
↪stream -v
```

What comes out should look as follows:

```
--------------------------------------------------------------------------------
↪--------
Host:            icrar-dirp01
Port:            7777
Command:         ARCHIVE

Date:            2015-12-10T16:58:40.759
Error Code:      0
Host ID:         icrar-dirp01
Message:         Successfully handled Archive Push Request for data file with URI␣
↪ngamsPClient
Status:          SUCCESS
State:           ONLINE
Sub-State:       IDLE
NG/AMS Version: v4.1-ALMA/2010-04-14T08:00:00
--------------------------------------------------------------------------------
↪--------
```

4.4 Running the tests

If you want to run the suite of unit tests then you need to install at least one additional package:

```
$> pip install psutil
```

Unit tests are found in the `test_*.py` files under the `test` directory of the ngas source distribution. You can use any unittest runner to execute the tests. In particular, we tend to use pytest, like this:

```
$> pip install pytest
$> pytest
```

Note: Previous versions of NGAS had the restriction that one had to be **inside** the test/ directory to run the tests. This restriction does not exist anymore, although one can *still* run the tests from within the directory. Tools `pytest` or the `unittest` built-in module should also be able to automatically discover and execute unit tests.

4.4.1 Using alternative filesystem

The NGAS unit tests, and the servers that are spawned from within, look through the following possible directories to host any temporary files that need to be written to disk:

- The value of the `NGAS_TESTS_TMP_DIR_BASE` environment variable.
- `/dev/shm`, if present (it usually is in Linux systems) and has at least 1 [GB] of space available.
- The return value of `tempfile.gettempdir()`, which returns a well standardized location temporary directory.

Whatever value is used, a final `/ngas` path element is added at the end. For example, in a normal Linux system the tests would write their temporary files to `/dev/shm/ngas/`.

The rationale of prioritizing `/dev/shm` over the system-dependent standard temporary directory location is to speedup the test execution. `/dev/shm`, when present, is often backed up by an in-memory filesystem, and therefore I/O operations run much faster there. If you system has such location in the filesystem but doesn't correspond to an in-memory filesystem, or if you want to use your system's temporary directory (usually `/tmp` under Linux) you can still set the `NGAS_TESTS_TMP_DIR_BASE` environment variable to point to it.

4.4.2 Using alternative databases

By default the unit tests will run against a temporary on-disk sqlite database. If users want to run the tests against a different database they can do so by setting the `NGAS_TESTDB` environment variable to contain a *Db* XML element with the correct information.

For example, in Travis we run our tests against the local MySQL database like this:

```
export NGAS_TESTDB='<Db Id="blah" Snapshot="0" Interface="MySQLdb" host="127.0.0.1"␣
↪db="ngas" user="ngas" passwd="ngas"/>'
pip install psutil pytest-cov
pytest --cov
```

4.4.3 Keeping intermediate results

Tests generate a number of temporary files and directories on disk that are automatically removed after each test finishes, regardless of whether they fail or succeed. If the output of **the last test executed** needs to be kept users can set the `NGAS_TESTS_NO_CLEANUP` to a value different than `0`. This will keep all files under *the temporary directory* untouched so users can go and get more details about the test execution.

CHAPTER 5

NGAS tools

This section lists the command-line tools installed by the different NGAS python packages. A very brief description of each tool is given. For more details on each of them you can follow the relevant links, or get the corresponding command-line help message for each of them.

5.1 Server tools

These are programs and scripts that are only relevant for the server-side of NGAS.

5.1.1 `ngamsServer`

The main workhorse of NGAS, the `ngamsServer` tool starts up an NGAS server.

For details on how to start the server, run `ngamsServer -h`. Alternatively you can read *Running the server*. For more documentation on the server itself, its organization and features, please check the *server documentation*.

5.1.2 `ngamsDaemon`

The `ngamsDaemon` tool starts an NGAS server is daemon mode.

For details on how to start a daemon run `ngamsDaemon -h`. Alternatively you can read *Running the server*.

5.1.3 `ngas-prepare-volume`

Note: This tools was previous known as `ngasPrepareVolume` but had not been properly kept up to date.

The `ngas-prepare-volume` tool prepares a directory to be used as an *NGAS volume*. This preparation consists simply on recording some meta-data about the volume into a specific place and format.

5.2 Client tools

These are programs relevant for the client-side of NGAS.

5.2.1 ngamsPClient

The `ngamsPClient` tools is a generic NGAS client written in python, and accessible via the command-line for easy use and integration.

Use `ngamsPClient -h` for more help.

5.2.2 ngamsCClient

The `ngamsCClient` tools is a generic NGAS client written in C, and accessible via the command-line for easy use and integration. The C client (and corresponding library) are compiled optionally, so they might not be available for use (refer to the *NGAS installation* for details).

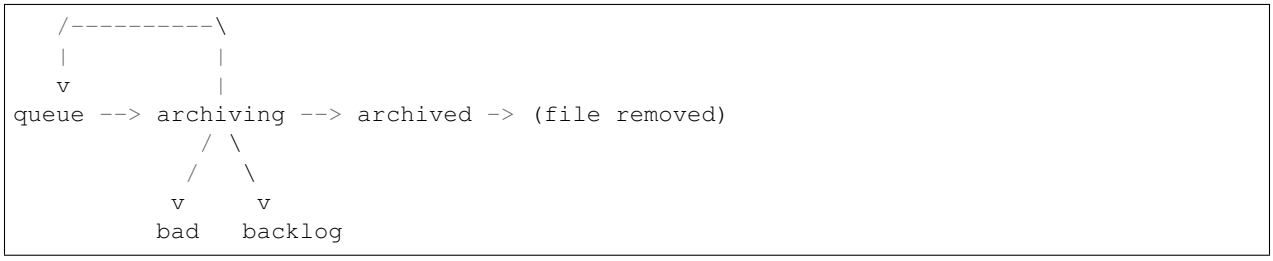
Use `ngamsCClient -h` for help on how to use the C client.

5.2.3 ngas-fs-monitor-client

Note: This tool was previously known as `ngasArchiveClient`, but had not been properly kept up to date.

The `ngas-fs-monitor-client` tool continuously scans files in a specific directory, archives them into an NGAS server as they appear, and performs a check on the server to ensure the file has been received successfully. After a successful check, the file is locally removed.

For a given file, its lifecycle looks like this:



For more information, run `ngas-fs-monitor-client -h`.

CHAPTER 6

Server

The NGAS server is the heart of NGAS.

6.1 Configuration

The NGAS server is configured via an XML configuration file, which is indicated to the server at startup time via the `-cfg` command-line flag (see *Running the server*).

To see more details about the XML documentation go to the *Configuration* section.

6.2 Running Modes

The NGAS server can be run in three different modes:

- As a *cache* server
- As a *data-mover* (or read-only) server
- As a normal server

Selecting which mode will be used is done by editing the server configuration file.

Additionally, server can be configured to be allowed to perform a specific set of actions. For details see *Permissions*.

6.2.1 Cache mode

When started in *cache* mode, an NGAS server starts its *cache control* thread, enabling it to periodically remove files from its underlying storage after they have been successfully transmitted to the configured subscribers. This behavior effectively turns the NGAS server into a temporary cache for data in transit to some other location.

To start an NGAS server with its cache control thread enabled you need the configure the *caching* element of the server configuration file.

Note: A server in *cache* mode was historically started by running an *ngamsCacheServer* executable. Since v11.0 this alternative doesn't exist anymore, and centralizing the server's starting mode in its configuration file.

6.2.2 Data mover mode

When started in *data-mover* mode, an NGAS server configures itself to operate in a *read-only* mode.

This mode is meant to be used as a complementing background-processing system to a main NGAS server instance. While the main NGAS server instance manages the data in and out of the volumes, a data-mover server can still read data out and perform other tasks, like taking care of subscriptions or perform background processing.

To start an NGAS server in *data-mover* mode you need to use the `-datamover` command-line switch when running the `ngamsServer` script.

6.3 Proxy behavior

When NGAS servers are deployed in a cluster configuration (i.e., many servers connected to a single central database), clients can issue commands to any server in the cluster.

In this scenario it can happen that clients contact a server with a command that can only be fulfilled by a different server (e.g., fetching data). In those cases the server contacted by the client can either respond with an HTTP redirect answer to the client, or it can act as a proxy, issuing the command to the second server on behalf of the client, and forwarding the response as it comes. This behaviour can be set in a per-server basis via their corresponding configuration file. See *Server* for details.

6.4 Storage organization

6.4.1 Volumes

When NGAS was first designed, data was mostly transported manually by swapping physical disks in and out of server. Thus, data was organized by *disks*, which were mounted onto the filesystem hierarchy into a particular directory.

Nowadays transport happens mostly through the network, even for long-distance transmissions, with disks staying fixed, or rarely replaced. Data still gets archived into a directory though, usually corresponding to the root of a filesystem mount point. Thus, we usually prefer to refer to these top-level data directories as **volumes**, a more generic term.

Volumes are directories in the filesystem, usually under the NGAS root directories (makes it easier to see them all together). Because they are used to organize how data gets archived, users will still want to map them to disk partitions or mounted filesystems via symbolic links. Otherwise (specially in testing scenarios) a simple directory can also be used.

6.4.2 Storage sets

Volumes are grouped in **storage sets**. A storage set will usually consist of only one volume, the *main volume*, but it can optionally contain also a *replication volume*. Replication volumes are usually not required nowadays (because data is replicated via the network, and/or because mount points are backed up by different RAID setups), and therefore you will very unlikely need one.

6.4.3 Streams

Storage sets form the base for organizing data storage. An NGAS server is configured to store certain types of data into certain storage sets. Such mappings from a data type (i.e., a MIME type) and one or more storage sets is called a **stream**.

Streams made it easy to collect all data of a certain type in one or more disks, which then could be swapped out for data movement. Because of this, in practice only the *ARCHIVE* command follows this configuration to determine the target disk to host the incoming data. On the other hand, the *QARCHIVE* command doesn't obey these rules, as it was designed with network transport as means of replication. With network-based replication the physical volume hosting the data locally does not have a great impact anymore, and therefore the system tries to fill them evenly.

6.5 CRC

When a file is being archived into NGAS the server will calculate its CRC as part of the archiving process. The CRC is saved into the database as an integer value, and is used later to check the integrity of the file.

Three CRC variants are currently supported by the NGAS server, which in the future might expand:

- `crc32`: This is the original implementation. It uses python's `binascii.crc32` method to calculate the CRC, and therefore it is fully implemented in software. This variant **does not mask the value with 0xffffffff**, and therefore in python 2.7 the integer value that gets stored in the database can be negative (while in python 3 the value is strictly an unsigned integer).
- `crc32c`: A hardware-based implementation available as part of Intel's SSE 4.2 instruction set. This variant will only be available if the crc32c package is installed.
- `crc32z`: Like `crc32`, but generates the same integer values even across different python versions. Users should prefer this variant over `crc32`, which is still maintained for backwards-compatibility reasons.

Note: The `crc32c` package is automatically installed by the *NGAS installation script*, unless the `NGAS_NO_CRC32C` environment variable is defined.

Note: Care has been put in ensuring that even the values produced by `crc32` are correctly compared, so even if users choose that method checksum comparisons should still yield the correct answer.

Depending on your environment choosing to use one method over the other might bring significant improvements on archiving times for large files. To configure which method should be used across an entire NGAS installation change the `ArchiveHandling.CRCVariant` setting on the *NGAS configuration*.

Also, users can install NGAS without `crc32c` support if their CPUs do not support the SSE 4.2 instruction set. (see inst for details).

6.6 Processing

When a *RETRIEVE* request is issued by a client, the data sent by the server can optionally be processed on-the-fly before serving it back to the client.

This behavior can be provided by users with potentially different plug-ins used to process different mime-types. See *Processing* for details.

6.7 Archiving events

The NGAS server features an *archiving event* mechanism. Each time a new file is archived, a new archiving event is generated, and a list of *event handlers* is invoked with the given event. The NGAS server has its won internal event handlers, but users can also provide their own via plug-ins. This mechanism is a flexible way of enabling archiving notifications and reacting on these events.

Users wanting to implement their own event handlers should *write a python class* to handle it, and *configure the server* to use that class.

6.8 States

An NGAS server can be in one of two states at any given time: **ONLINE** and **OFFLINE**. The state is meant to represent the availability of the NGAS service. In addition, an **IDLE** or **BUSY** sub-state represents the activity that is going on on the server.

States are used by the different *Commands* to decide whether a particular action can take place or not. If the current state/sub-state allows the operation it will continue without interruptions; otherwise the user will receive an error stating that the server is in the wrong state/sub-state.

The NGAS server starts by default on the **OFFLINE** state. If the server is started with the `-autoOnline` command-line flag (see how to *run the server*) it will move itself automatically to the **ONLINE** state after initializing. At runtime the state can be toggled via different *Commands*.

6.9 Requests database

The NGAS server keeps a rotating set of all incoming client requests for future status querying. When a client request comes in, it is first registered into a *requests database*. After the request is served as usual, the corresponding item in the request database is updated to reflect the final state of the request. If a request is asynchronous in nature (e.g., it spawns a background task that will finish later in time), the entry in the requests database may also be updated as it logic is executed, even if the initial response has already been sent to the user. This, together with the *STATUS* command, are the basis for asynchronous command execution and monitoring (used only the *CLONE* command).

The requests database has three different implementations. The implementation used by the server is configured by the `RequestDbBackend` attribute in the *Server* configuration element. The first, a BSDDB-based one, is the most expensive to use, as it needs to lock during I/O access, but it provides persistence across executions. A second, memory-based implementation is also available. This is faster as it doesn't involve disk I/O, but doesn't provide persistence. Finally, a null implementation is provided. This implementation is provided for cases when a request database is known not to be needed (e.g., no asynchronous commands are ever issued).

6.10 Logical Containers

NGAS supports the concepts of *logical containers*. They are called *logical* to distinguish them from *physical* containers. Physical containers are currently only envisioned and not implemented, so for the rest of the document we use *container* and *logical containers* interchangeably.

Logical containers are a way of grouping files together, which in turn allows to perform container-wise operations like retrieval or archiving. Files can be added to or removed from a container independently, but can belong to only one container (or none) at a time. Finally, containers can be hierarchically organized, with one parent container (or none) allowed per container.

Container thus allow to organize files stored in NGAS in a filesystem-like structure, where directories are NGAS containers and files are NGAS files.

Containers are handled via the different *container commands*.

6.11 Authorization

NGAS supports authentication via the standard HTTP `Authorization` header. Currently only `Basic` authentication is supported, but more authentication methods could be added in the future. On top of authentication, a binary authorization scheme is implemented which allows users or not to run a command.

In other words, NGAS can be set up to allow different users to run different commands. Details on how to set up this configuration can be found in *Authorization*.

6.12 Logging

The NGAS server outputs its logs to two different places: the standard output, and a logfile. Users will mostly be interested in the logfile, as it provides a persistent location to inspect logs. To avoid cluttering, the NGAS server rotates these logfiles after a fixed amount of time, and after each time the server starts.

Each time the logfile is rotated, its name is first changed to make space for the next logfile. If the `Log.ArchiveRotatedLogFiles` option is set in the configuration file, then the logfile is archived into the NGAS server itself for easier retrieval. Finally, users can also write more code to handle a rotated logfile.

Details on how to configure logging in NGAS can be found in *Log*. To learn how to write logfile handler plug-ins see *Logfile handling*.

6.13 Email Notifications

The NGAS server can be configured to send emails to different email addresses when particular events happen.

For further explanation on the full range of options and how to configure them, see *Notification*.

6.14 Suspension

An NGAS server can be configured to put itself into suspension mode after it detects it has been idle (i.e., not serving external requests) for a certain amount of time.

Later on, if a request arrives that ultimately needs to be served by a currently idle server, a *Wake Up server* will be contacted and tasked with waking up the idle server.

Note that host suspension makes sense in scenarios where only a couple of NGAS servers are public-facing and acting like proxies to a bigger cluster of NGAS server. In such setup, idling internal servers in the cluster can be safely managed. If all servers in the cluster were exposed publicly, sending a server into idling would not work as expected, as direct requests going into that host will not automatically cause it to wake up.

Host suspension is configured in the server configuration file as shown in *HostSuspension*.

CHAPTER 7

Background tasks

Apart from serving requests to users, the NGAS server executes different tasks in the background with a given period.

This section describes the purpose of these tasks, how they work, and which of their aspects are configurable.

7.1 Janitor thread

The Janitor thread is a background task taking care of generic, routine tasks. Among other things, it checks that sufficient disk space is available for each of the configured volumes, cleans up old data from temporary directories, sends pending notifications, archives rotated logfiles, and more. Moreover, additional tasks to be carried out by the janitor thread can be specified by users via the configuration file and implemented as user-provided plug-ins.

7.2 Data check thread

The data check thread is a background task that periodically checks the integrity of the files sitting on disk by comparing their checksum as stored in the database against a freshly computed checksum. All aspects of the data check thread are configured via the NGAS server configuration file (see *DataCheckThread*).

Data checking can be expensive depending on your setup, as it re-reads the full contents of the ingested data. Not all systems might be able to carry out such a task; thus data checking can be enabled or disabled.

The data check thread continuously runs a full data check cycle with a configurable period. At the beginning of each data check cycle a lists all files on disk is constructed, then their checksums are calculated (using the same CRC variant that was used to archive the file) and compared against the database-stored values. Any checksum failures or files found to be unregistered are then notified. Finally, the data check thread waits until the period finishes to start a new cycle.

The data check thread actually uses a pool of processes to carry out the checksum calculations, increasing its performance when more than one core is available in the system. This parallel execution of checksum checking also takes into account the volumes to which the files belong to, reading files in parallel from different volumes when possible.

Finally, all data checking workload is fully paused whenever the server is serving a user request. This prevents user requests to be slowed down due to resource exhaustion produced by the data checking processes (in particular, CPU and disk reading).

7.3 Cache control

The Cache control task, if enabled, periodically removes local files from the server. This is useful in setups where an NGAS server acts as a buffer to received data locally before replicating it to different, remote locations. Being able to remove local files automatically keeps the overall disk in check, allowing users to decide what their space needs are depending on the buffering capabilities needed by the system.

A number of criteria control how and when local files are removed from NGAS, which can be configured through the *Caching* element in the server's configuration file:

- A per-file time limit has been reached. If configured, files are removed from the server after a given amount of time has passed since the file was originally archived.
- A maximum amount of storage capacity has been hit. When configured, files are removed when their total volume exceeds the specified maximum value. Older files are deleted first.
- A maximum number of files has been hit. When this option is set, files are removed when their total number exceeds the configured limit. Older files are deleted first.
- A user-provided plug-in makes the decision. Users can write *ad-hoc* code to decide whether particular files should be deleted (or not).

More than one rule can be active at a given time, in which case they are processed in the order given above. On top of that, if the Subscription service is enabled files will only be eligible for deletion after they are successfully transmitted to all their subscribers (this cannot be overridden).

CHAPTER 8

Plug-ins

A central feature of NGAS is its extensibility. By writing different types of plug-ins users can modify the default behaviour of NGAS, add functionality that is specific to their projects, or adjust some details for specific platforms.

8.1 Installation

Regardless of the type of plug-in you want to install, NGAS must be able to find it. Because it is not always possible to install user-written code alongside NGAS itself (e.g., NGAS installation could be read-only, or writeable only by `root`) NGAS loads user-provided code from any arbitrary, user-defined location. This is indicated in the server's configuration file with the `PluginsPath` variable. For details on how this works see the *Server* configuration element.

8.2 Developing plug-ins

Depending on the type of plug-in you want to develop, different interfaces must be obeyed. Also, even though all plug-ins can be *installed* in the same area, configuring the server to actually use them is a different task, and which is different from one plug-in type to another.

The following sub-sections detail each plug-in type, what is the interface they should obey, and how to configured the server to use it.

8.2.1 Commands

NGAS allows users to write their own modules to implement new NGAS commands.

Interface

The only requirement a command plug-in needs to satisfy is to implement a method at the module level:

```
def handleCmd(server, request, http_ref)
```

The arguments are the following:

- `server` is a reference to the *ngamsServer* instance this command is running in. From the `server` object a pointer to the database object and to the configuration object can be obtained (via `server.db` and `server.cfg` respectively).
- `request` is an instance of `ngamsReqProps`, and encapsulates information that exists in the scope of a request.
- `http_ref` is a reference to the HTTP connection. It can be used to read content from the incoming data stream, and send responses.

Any uncaught exception thrown by the `handleCmd` method will be interpreted as an error by the NGAS server and will generate an error status code being sent to the client, together with an XML NGAS status document indicating the failure.

If the module returns a string, this will be used to create an XML NGAS status document that will contain that message, and that will be returned to the client with a `200` HTTP status code.

If the module returns `None`, a generic success XML NGAS status document will be sent to the client.

To send other kind of replies, please refer to the *ngamsHttpRequestHandler* class documentation.

Registration

Modules implementing commands must be registered with NGAS in order to be picked up. This is done by listing them in the *Commands element* of the server XML configuration.

8.2.2 Archiving event handlers

Apart from the built-in *Archiving events*, users can provide their own code to handle them.

Interface

Archiving event handler are implemented by writing a python class with a `handle_event` method.

```
class MyEventHandler(object):

    def __init__(**kwargs):
        pass

    def handle_event(evt):
        pass
```

The class constructor should accept keyword arguments (corresponding to the parameters set in the configuration, see below). The class' `handle_event` method accepts a unique parameter (the archiving event), and gets invoked for each archiving event. The event has two members, `file_id` and `file_version`, with the ID of the file just archived and its version.

Note: Archiving event handlers run synchronously as part of the archiving process. Therefore it is vital they are fast, otherwise they can block the server.

Registration

Archive event handler classes must be registered with NGAS in order to be picked up. See *ArchiveHandling* for details on how to do this.

8.2.3 Logfile handling

As explained in *Logging*, users can write their own logfile handling plug-ins. These plug-ins will be invoked each time a logfile is rotated, which happens after a fixed amount of time, and every time the server is started.

Interface

Logfile handling is implemented by writing a python module with a `run` method.

```
def run(srv, fname):
    pass
```

The `srv` argument is an *ngamsServer* object, and the `fname` is the path to the rotated logfile. Logfile handler plug-ins run asynchronously as part of the *janitor thread*, and therefore it is acceptable that they take some time to run.

Registration

Logfile handling plug-ins must be registered with NGAS in order to be picked up. See *Log* for details on how to do this.

8.2.4 Subscription filtering

NGAS allows users to write their own modules to provide subscription filtering logic.

Interface

The only requirement a subscription filtering plug-in has is to implement one method at the module level with the same name of the module itself:

```
def my_plugin_name(server, plugin_pars, file_name, file_id, file_version)
```

`my_plugin_name` needs to match the name of the module; in other words, the file containing it must be called `my_plugin_name.py`.

The arguments are the following:

- `server` is a reference to the *ngamsServer* instance this command is running in. From the `server` object a pointer to the database object and to the configuration object can be obtained (via `server.db` and `server.cfg` respectively).
- `plugin_pars` is the comma-separated, `key=value` pairs of parameters as stored by the subscription in the database.
- `file_name`, `file_id` and `file_version` are the name, ID and version of the file delivery being assessed.

The method should return `True` if the file should be delivered, and `False` otherwise.

Please note that filtering is done on a per-file basis. Calculating the list of files that will be fed into the plug-in is outside of the scope of this plug-in, and depends on the subscription settings, like its start date.

Registration

Modules implementing commands must be registered with NGAS in order to be picked up. This is done by listing them in the *Commands element* of the server XML configuration.

CHAPTER 9

Commands

An NGAS server works by replying to HTTP requests sent to it. All URLs used to contact NGAS have the following structure:

```
http://<server>:<port>/<command>?<parameters>
```

The `server` and `port` parts indicate where the NGAS server is listening for requests. The `command` part is a simple name, usually in uppercase, like `ARCHIVE` or `RETRIEVE`, and indicates the action to be performed by the server. An optional list of parameters, separated by an ampersand (&) sign, and optionally each having a value, provides further details about the action to be performed.

NGAS currently supports the `POST`, `GET` and `PUT` HTTP methods only. In general the method used to invoke a command does not make a difference, but some commands have different behavior depending on the HTTP Method being used. Refer to the documentation of each command for more details.

All commands return an HTTP status code that reflects the outcome of the operation (200 for success, 3xx for redirections, 4xx for errors). Additionally some commands return a *status* XML document with a more detailed description of the operation result.

One can configure an NGAS server to accept HTTP requests only from authenticated users, and moreover to allow certain commands to certain users only. Please refer to *Authorization* for more details.

The following is a list of the most relevant commands supported by NGAS. More commands can be added in the form of *plug-ins* (see *Commands* for details).

There are three ways in which commands are found by NGAS:

- There is a fixed set of built-in commands. Users cannot override these with their own. Commands like `RETRIEVE` and `STATUS` belong to this category.
- Modules with a name following the pattern `ngamsPlugin.ngamsCmd_<CMD>`. For historical reasons there are a number of commands that are shipped with NGAS, but that are implemented as plug-ins, and are named following this pattern. In future releases these will be shipped as regular built-in commands, and therefore this special pattern will no longer be considered.
- User-written plug-ins that are registered in the *Commands* section of the server configuration.

9.1 Core

9.1.1 STATUS

The **STATUS** command is the most basic of all. It can be used (and is used) to confirm that an NGAS server instance is correctly running. It simply returns a status XML document containing information about the server runtime, like its state, disks, etc.

In particular, it can also be used to query the status of a previous client request when given a *request_id* URL query parameter. See *Requests database* for more details.

9.1.2 OFFLINE

Sends the NGAS server to the **OFFLINE** state. See *States*.

9.1.3 ONLINE

Sends the NGAS server to the **ONLINE** state. See *States*.

9.1.4 EXIT

Stops the NGAS server. The server must be in the **OFFLINE** state for the **EXIT** command to be successful.

9.2 Storage

9.2.1 ARCHIVE

Archive data files within an NGAS node, calculating and storing the CRC of the archived file in the NGAS database.

After a successful archiving of a file, all archiving event handlers are invoked. Read *Archiving events* for more information about these events and how to handle them.

The `ARCHIVE` command supports two modes of operation: *pull* and *push*. A pull request is issued by using the `GET` method, and tells an NGAS node to fetch and archive a file based on a valid URI. A push request, on the other hand, is issued by using the `POST` method, and requires the client to send the file contents as a byte stream to the NGAS server.

When incoming files matching an existing file ID in the NGAS DB, their contents are stored using a new version number if file versioning is on. If file versioning is off, they will overwrite an existing file's contents instead. When overwriting, users can specify a specific file version to overwrite, which must exist **locally** in the server receiving the request.

Parameters

- `filename`: a valid URI i.e. `file://`, `http://`, `ftp://` for Pull or filename ie. `test.fits` for Push.
- `mime_type`: describes the content-type of the file. If not given, NGAS tries to guess it based on the filename's extension, and the *internal mime-type information* stored in the NGAS configuration.
- `versioning`: used to switch the automatic versioning on (`1`, the default behavior) or off (`0`).

- `no_versioning`: The inverse of `versioning`. This is kept for backwards compatibility. If both are specified, `versioning` takes precedence.
- `file_version`: specifies which file version to overwrite. Only taken into account when `versioning=0`/`no_versioning=1`.
- `crc_variant`: used to explicitly choose which CRC variant will be used to checksum the file, overriding the system-wide configuration. See *CRC* for details

Archive Pull Example

In this case the NGAS server will attempt to retrieve and archive the file `remote.fits` from the remote http server:

```
curl http://<host>:<port>/ARCHIVE?filename=http://<remotehost>:<remoteport>/remote.
↪fits
```

Archive Push Example

In this example it is expected that the client uploads the file content as a byte stream to the NGAS server:

```
curl -X POST -i -H "Content-Type: application/octet-stream" --data-binary "@/tmp/file.
↪fits" http://<host>:<port>/ARCHIVE?filename=file.fits
```

9.2.2 QARCHIVE

Like *ARCHIVE*, but with the following differences:

- The target volume is selected at random from the available volumes in the server. This bypasses the server's *stream configuration*, but should yield a more even loading of the available volumes.
- No file replication is carried out, even if a storage set declares a replication disk.

The `QARCHIVE` command was initially implemented separately from the `ARCHIVE` command, and therefore they used to differ in more ways. In particular `QARCHIVE` originally was the only one implementing on-stream checksuming, while `ARCHIVE` didn't. Nowadays they share the same underlying logic though, and only the differences documented above remain.

9.2.3 RETRIEVE

Retrieve archived data files from an NGAS server or cluster.

Parameters

- `file_id`: ID of the file to retrieve.
- `file_version`: version of the file to retrieve.
- `processing_pars`: invoke a processing plug-in by name that will operate on the file requested. Note that NGAS will send back the result of the processing which may or may not be a file stream.

If multiple files of the same ID exist and `file_version` is not specified then the file with the highest version number will be retrieved by default.

If the file was compressed internally by NGAS at archiving time, at `RETRIEVE` time the contents are still returned compressed. Different clients will handle this differently: if the client supports `Content-Encoding: XYZ` responses (with `XYZ` being the compression used internally in NGAS) then a filename without the corresponding compression extension will be presented, as the client automatically decompresses the incoming content before presenting it to the user. In the case the client doesn't support this, a filename with the corresponding extension will be presented to ensure clients associate the filename and its contents with the corresponding utilities.

Note that only one file can be retrieved per RETRIEVE request.

Example

Get the latest version of a file if it exists:

```
curl http://<host>:<port>/RETRIEVE?file_id=file.fits
```

Get specific version of a file if it exists:

```
curl http://<host>:<port>/RETRIEVE?file_id=file.fits&file_version=2
```

9.2.4 QUERY

The QUERY command is used to list different types of elements currently known by the server. Users can use the QUERY command to find out which files have been archived into the server, which subscriptions have been created, which disks are present in the system, and more.

Parameters

- `query`: The query to run, the only required argument. Valid values are:
  - `files_list`: a list of all files in the system.
  - `subscribers_list`: a list of all subscriptions in the system.
  - `subscribers_like`: a list of subscriptions in the database belonging to hosts matching the given `like` value (see below).
  - `disks_list`: a list of all disks in the system.
  - `hosts_list`: a list of all hosts that are part of the same cluster.
  - `files_like`: a list of files whose ID matches the given `like` value (see below).
  - `files_location`: a list of all files in the system, but with information about their physical location on disk.
  - `lastver_location`: like `files_location`, but only listing the last version of each file.
  - `files_between`: a list of files archived into the system between `start` and `end` date (see below).
  - `files_stats`: the total size of all archived files, in [MB].
  - `files_list_recent`: a list of the last 300 archived files.
- `format`: the format in which the result of the query will be returned to the client. Valid values are `list` (a textual, table-like representation), `pickle` (a python pickled version of the data), `json` (a json representation of the data), and `python-list` (a `str` representation of the direct result of the query).
- `like`: indicate the value to use in the `*_like` queries. If no string is given, `%` will be used, therefore matching all values for the corresponding attribute.
- `start` and `end`: indicate the beginning and the end of the time interval used for the `files_between` query. Both parameters must be specified in order for the interval to be properly defined. If any of the two is (or both are) missing, no interval is used.

Example

Get list of all subscriptions the system in json format:

```
curl http://<host>:<port>/QUERY?query=subscribers_list&format=json
```

Get list of all files in the system:

```
curl http://<host>:<port>/QUERY?query=files_list&format=list
```

9.2.5 CLONE

The CLONE Command is used to create copies of a single file or sets of files. In order for the CLONE Command to be accepted by an NGAS node, the system must be configured to accept Archive Requests. NGAS will calculate if there is enough space to execute the request, if not then an error is returned. If the files to be cloned are located on other NGAS host, these will be requested automatically during the cloning (if possible). If the NGAS hosts are suspended, they will be woken up automatically.

Parameters

- `disk_id`: disk ID where the files to be cloned exist.
- `file_id`: ID of the files to be cloned.
- `file_version`: file version of the files to be cloned.
- `notif_email`: list of comma separated email addresses to where the Clone Status Report can be sent.

The actions of the various combinations of these parameters are explained below:

disk_id	file_id	file_version	Action
	•		Clone one file with the given ID. Latest version of the file is taken.
•	•		Clone one file stored on the given disk. Latest version on that disk is taken.
	•	•	Clone all files found with the given File Version. Storage location (Disk ID) is not taken into account.
•	•	•	Clone one file on the given disk with the given File Version.
•			Clone all files from the disk with the given ID.
•		•	Clone all files with the given File Version from the disk with the ID given.
		•	Illegal. Not accepted to clone arbitrarily files given by only the File Version.

9.2.6 CHECKFILE

The CHECKFILE command is used to check the consistency of a specific file.

Parameters

- `disk_id`: disk ID where the file to be checked exists.

- file_id: ID of the file to check.
- file_version: version of the file to check.

9.2.7 CACHEDEL

The CACHEDEL command is used to remove a file from an NGAS cluster. Only the ngamsCacheServer version supports this command.

WARNING: Once the command completes successfully the file is permanently deleted from the NGAS database and the underlying file system.

Parameters

- disk_id: disk ID where the file to be deleted exists.
- file_id: ID of the file to be deleted.
- file_version: version of the file to be deleted.

9.2.8 REMDISK

The REMDISK command is used to remove storage media from an NGAS node. The command removes both the information about the storage media and the files stored on said media. NGAS will not remove the files from the system unless there are at least three (3) independent copies of the files. Three independent copies refers to three copies of the file stored on three independent storage media. In order for the REMDISK command to be accepted the system must be configured to allow remove requests i.e. NgamsCfg.Server:AllowRemoveReq is set in the configuration file. If the command is executed without the execute parameter, the information about the disk is not deleted, but a report is generated indicating what will be deleted if the execution is requested i.e. execute = 1.

WARNING: Once the command completes successfully the files associated with the storage media are permanently deleted from the NGAS database and the underlying file system.

Parameters

- disk_id: ID of disk/media to remove from NGAS node.
- execute: (0 or 1) 0: is a dummy run which will only report what will happen if the command is executed. 1: executes the command which will deleted the storage media and the associated files.
- notif_email: list of comma separated email addresses to where the REMDISK Status Report can be sent.

9.2.9 REMFILE

The REMFILE command removes a single file from an NGAS node. NGAS will not remove the files from the system unless there are at least three (3) independent copies of the files. In order for the REMFILE command to be accepted the system must be configured to allow remove requests i.e. NgamsCfg.Server:AllowRemoveReq is set in the configuration file.

Parameters

- disk_id: disk ID where the file to be deleted exists.
- file_id: ID of the file to be deleted.
- file_version: version of the file to be deleted.
- execute: (0 or 1) 0: is a dummy run which will only report what will happen if the command is executed. 1: executes the command which will delete the file.

- `notif_email`: list of comma separated email addresses to where the REMFILE Status Report can be sent.

The actions of the various combinations of these parameters are explained below:

disk_id	file_id	file_version	Action
	•		All files matching the given File ID pattern on the contacted NGAS host are selected.
•	•		All files with the given File ID on the disk with the given ID will be selected.
	•	•	All files with the given File ID pattern and the given File Version are selected.
•	•	•	The referenced file with the given File ID and File Version on the given ID is selected (if this exists).
•			Illegal.
•		•	No files are selected.
		•	No files are selected.

9.2.10 REGISTER

The REGISTER command is used to register files already stored on an NGAS disk. It is possible to register single files or entire sets of files by specifying a root path. Only files that are known to NGAS (with a mime-type defined in the configuration) will be taking into account. It is also possible to explicitly specify a comma separated list of mime-types that will be registered. Files with other mime-types than specified in this list will be ignored.

Parameters

- `mime_type`: comma separated list of mime-types. A single mime-type can also be specified.
- `path`: The root path under which NGAS will look for candidate files to register. It is also possible to specify a complete path to a single file.
- `notif_email`: email address to send file registration report.

NGAS can be configured to run specific code when registering a file. See *Register* for details.

9.2.11 REARCHIVE

The purpose of the REARCHIVE command is to register a file in the NGAS DB that has already been generated when the file was archived with the QARCHIVE command. This means that the process of extracting the meta-information and other processing can be skipped whilst re-archiving the file making the processing more efficient.

The meta-information about the file is contained in the special HTTP header named `NGAS-File-Info`. It is stored as a `base64` encoded NGAS XML block for the file (NGAS File Info). This encoding can be accomplished by means of the Python module `base64` using `base64.b64encode()`.

The command does not require any parameters but the data to be re-archived should be contained in the body of the HTTP request similar to QARCHIVE Push or Pull.

9.3 Containers

The following commands deal with *logical containers*. For an explanation on containers see *Logical Containers*.

For container-related commands in general, when dealing with existing containers the following rule applies: if the container can be uniquely identified by name the `container_name` parameter is enough to describe it; otherwise the `container_id` parameter must be given.

9.3.1 CCREATE

Creates one or more containers but without adding files into them.

To create a single container the `container_name` parameter must be present, optionally the `parent_container_id` can be given, and the request must be performed using the `GET` method. To create multiple containers an XML document must be sent in the request body containing elements with a `name` attribute, and optionally a `parentContainerId` attribute at the root level. Nested elements are allowed to create a hierarchy of containers.

9.3.2 CARCHIVE

Archives files and creates the necessary containers for them.

This command reads a MIME Multipart message from the request body. The `Content-Disposition` header of the multipart message contains the name of the container. The messages inside the multipart message each contains in turn a `Container-Disposition` header indicating the name of the file they represent, and their payload is the file's content. A multipart message may also contain multipart messages inside, creating a hierarchy of containers.

9.3.3 CAPPEND

Appends an existing file into an existing container.

If using the `GET` method the `file_id` parameter must point to a file that will be added to the container. Multiple files can also be added at once when using the `POST` method and sending an XML document in the request body consisting of a list of `File` elements, each with a `FileId` attribute in them pointing to an existing file.

9.3.4 CDESTROY

Destroys a single container, without removing its files.

If the optional `recursive` parameter is set to `1` the children containers will also be removed recursively.

9.3.5 CREMOVE

Removes an existing file from an existing container.

File specifications follow the same rules followed by *CAPPEND*.

9.3.6 CRETRIEVE

Retrieves all the contents of a container.

See *CARCHIVE* for a description of the format used by the response body to transmit the contents of the container. Alternatively, if a `format` parameter with the value `application/x-tar` is given, the contents of the container will be retrieved as an uncompressed tarfile instead.

9.3.7 CLIST

Returns a status XML document containing the container hierarchy rooted at the specified container.

9.4 Others

9.4.1 SUBSCRIBE

The Data Subscription Service of NGAS makes it possible to synchronize a full or partial set of data files to remote Data Subscribers which can be other NGAS nodes. A client subscribing for data is referred to as a Data Subscriber. An NGAS Server, which delivers data to such a Subscriber, is referred to as a Data Provider.

A Data Subscriber can specify to receive data files from a certain point in the past to the present day. If the time is not defined then only newly archived files will be delivered to the subscriber. It is also possible for the Data Subscriber to specify a Filter Plug-In which is applied to a data file before it is delivered.

The client subscribes itself by supplying a Subscriber URL to the NGAS Data Provider. NGAS delivers data to the client by performing a HTTP POST on the Subscriber URL. The client must be ready to handle data file HTTP POST requests from the Data Provider. Any HTTP based server can be used, from a simple customized implementation to an existing and widely used server like Apache.

An NGAS Server can be configured to subscribe to another NGAS Server. In this case the Subscriber URL should be the URL used when performing an Archive Push Request:

```
http://<host>:<port>/QARCHIVE
```

Note that `NgamsCfg.SubscriptionDef:Enable` must be set to `1` to enable subscription. It is possible to instruct an NGAS Server to un-subscribe itself automatically when it goes Offline `NgamsCfg.SubscriptionDef:AutoUnsubscribe` set to `1`.

When a client has first subscribed itself to a certain type of file, NGAS guarantees that all files of that type (with the matching time constraint) will be delivered to the client. If it is impossible to deliver a file e.g. the client has terminated or the network is down, NGAS maintains a subscription back-log which will try to periodically deliver the files to the client. `NgamsCfg.SubscriptionDef:SuspensionTime` determines how often the NGAS server will process the back-log.

It is possible to specify an expiration time indicating for how long files should be kept in the back-log `NgamsCfg.SubscriptionDef.BackLogExpTime`. Files residing longer than the expiration time will be deleted and thus never delivered. The name of the subscription back-log is defined by the parameter `<NgamsCfg.Server:BackLogBufferDirectory>/subscr-back-log`.

A simple scheme has been implemented to avoid the scenario where the same data file is delivered to the same subscriber multiple times. This scheme is based on recording the ingestion date for the last file delivered. i.e., only files with a more recent ingestion date will be taken into account. This remembered 'last ingestion date' for each subscriber will be reset if a start date for the subscription 'older' than this date is specified by a client.

SUBSCRIBE is the command used by Data Subscribers to subscribe to an NGAS server.

Parameters

- `subscr_id`: Subscription ID that should be unique.
- `url`: The URL to which the archived file(s) will be delivered.
- `concurrent_threads`: Number of simultaneous file data delivery threads.
- `start_date`: Date from which the data to deliver is taken into account. If not specified the time when the SUBSCRIBE command was received is taken as start date.
- `priority`: Priority for delivering data to this Data Subscriber. The lower the number, the higher the priority. Clients with a higher priority, get more CPU time in connection with the data delivery.
- `filter_plug_in`: Name of a Filter Plug-In to invoke on the file(s).
- `plug_in_pars`: A set of parameters to transfer to the Filter Plug-In when it is invoked.

Example:

```
curl http://localhost:8000/SUBSCRIBE?subscr_id=TEST&url=http://localhost:8889/
↪QARCHIVE&priority=1&start_date=2016-03-01T00:00:00.000&concurrent_threads=4
```

This example has two NGAS instances running on the same node. One instance is bound to port `8000` and the other to port `8889`. The subscriber is telling the NGAS instance on `8000` to deliver all the files it ingested from the date `2016-03-01` to the present day using `4` concurrent delivery threads at the highest priority to NGAS instance `8889`.

9.4.2 UNSUBSCRIBE

Used by Data Subscribers to unsubscribe to a previously established subscription. If NGAS holds a back-log of data files for that subscription, that back-log will be cleared and data delivery will stop.

Parameters

- `subscr_id`: Subscription ID to unsubscribe.

CHAPTER 10

Configuration

This section details the contents of the XML configuration file used by NGAS.

Each sub-section describes an XML Element, while the items listed on each subsection refer to an attribute unless specified otherwise.

All elements share an *Id* attribute to uniquely identify them.

For a sample configuration file see the sample configuration file shipped with NGAS.

10.1 Server

Contains the overall server configuration.

- *RootDirectory*: The root directory which most of the other configuration items are relative to.
- *ArchiveName*: The logical name under which disks found by the server are grouped into. Using this, disks found in different servers may belong to the same logically distributed archive.
- *BlockSize*: The block size used for disk and network access, and checksum calculation. In the future different configuration options may be offered for these different operations.
- *IpAddress*: The IP address to bind the server to. If not specified the server will bind itself to `127.0.0.1`. To bind the server to all interfaces `0.0.0.0` can be set.
- *PortNo*: The port to bind the server to. It defaults to 7777 if unspecified.
- *VolumeDirectory*: The base directory where volumes are searched for. It relative, it is considered relative to the NGAS root directory. Defaults to `..`
- *MaxSimReqs*: The maximum number of requests the server can be serving at a given time. If a new request comes in and the server has reached the limit already, it will respond with an `503` HTTP code.
- *PluginsPath*: A colon-separated list of directories where external python code, like NGAS plug-ins or database drivers, can be loaded from.

- *ProxyMode*: Whether this server should act as a proxy when serving requests that are addressed to a different server within the same cluster (`1`) or not (`0`). See *Proxy behavior* for details.
- *RequestDbBackend*: The implementation of the request database that should be used. Allowed values are `memory`, `bsddb` and `null`. See *Requests database* for details. Defaults to `null`.

10.2 Permissions

This element defines the set of actions this server is allowed to perform.

- *AllowArchiveReq*: Whether archiving is allowed on this server.
- *AllowProcessingReq*: Whether processing is allowed on this server.
- *AllowRemoveReq*: Whether removal of files is allowed on this server.
- *AllowRetrieveReq*: Whether retrieval of files is allowed on this server.

10.3 Db

This element contains the database connection parameters.

- *Interface*: The python module implementing the PEP-249 Database API Specification v2.0.
- *MaxPoolConnections*: The maximum number of connections to be contained in the connection pool.
- *Snapshot*: Whether the *snapshoting* feature of NGAS will be turned on or off. It is recommended to leave it off.
- *UseFileIgnore*: Whether the code should use `file_ignore` or simply `ignore` as the column name to store the `ignore` flag of files in the `ngas_files` table. The latter was used by some particular combinations of old versions of the NGAS code and database engines, while the former is the default nowadays.
- *SessionSql*: Zero or more XML sub-elements, each with an `sql` attribute denoting an SQL statement that will be executed whenever a physical connection is established by the connection pool to the database server. Usually these will not be required, but can be useful, for instance, if one needs to execute a command to switch to a different database.

The rest of the attributes on the *Db* element are used as keyword arguments to create connection from the database module selected with the *Interface* attribute, and therefore don't have fixed names on them as they depend on the module in use.

For example, to connect to a PostgreSQL database using the `psycopg2` module one could use:

```
<Db Id="db-config"
    Snapshot="0"
    UseFileIgnore="false"
    Interface="psycopg2"
    host="db_host.example.com"
    dbname="ngas_db"
    user="ngas_user"
    password="ngas_password"
/>
```

In the example, the `Snapshot`, `UserFileIgnore` and `Interface` attributes work as described above, while `host`, `dbname`, `user` and `password` are keyword arguments accepted by the `psycopg2.connect` method.

10.4 Commands

This element lists user-defined command plug-ins. For details on commands in general see the *commands overview* section. For details on command plug-ins see the *commands plug-in* section.

The `Comands` element contains zero or more XML sub-elements named `Command`, each of which must define the following attributes:

- *Name*: The command name, case-sensitive.
- *Module*: The python module implementing this command.

10.5 MimeTypes

Lists a mapping of filename extensions and mime types. It contains one or more `MimeTypeMap` elements, each one listing the following attributes:

- *Extension*: A filename extension.
- *MimeType*: The mime-type associated to that filename extension.

This information is used, for example, by the *ARCHIVE* command when no mime-type information has been sent by the user.

10.6 StorageSets

Lists the storage sets (i.e., groups of disks) available to NGAS. Inside the `StorageSets` element one or many `StorageSet` elements can be found, each one listing the following attributes:

- *StorageSetId*: The name this storage set can be referenced by.
- *MainDiskSlotId*: The name of the directory where the data will be stored. If a relative path is given, it is considered to be relative to the NGAS volumes directory.
- *RepDiskSlotId*: The name of the directory where the data will be replicated. If a relative path is given, it is considered to be relative to the NGAS volumes directory.

For an explanation on volumes, main/replication disks, directories and storage sets please read *Storage organization*.

10.7 Streams

Lists the mappings from data types to storage sets. This element contains one or more `Stream` elements, each of which lists the following attributes:

- *MimeType*: The data type of this stream.
- *PlugIn*: The plug-in used to process incoming data of this type.
- *PlugInPars*: An optional, comma-separated, key=value string with parameters that can be communicated to the plug-in.

References to storage sets are included by adding `StorageSetRef` sub-elements, each of which should have a `StorageSetId` attribute pointing to the corresponding storage set.

For an explanation on streams please read *Storage organization*.

10.8 ArchiveHandling

Contains archiving-related configuration. For an explanation on most of these terms see *Storage organization* for reference.

- *PathPrefix*: The top-level directory on each volume under which NGAS will store incoming data.
- *Replication*: Whether data will be replicated during archiving from the Main disk to a Replication disk
- *BackLogBuffering*: whether data stored during a failed `ARCHIVE` command *might* be temporarily kept in storage to try to finish its archiving later on in the background.
- *BackLogBufferDirectory*: The top-level directory on each volume where backlogged files will be temporarily stored.
- *CRCVariant*: The CRC algorithm (and implementation) to use to calculate the checksum of incoming files. See *CRC* for details. If not specified the server will use the `crc32` variant. If specified, `0` means `crc32`, `1` means `crc32c` and `2` means `crc32z`.
- *EventHandlerPlugIn*: Zero or more sub-elements defining additional modules that will handle *archiving events*. Each element should have a `Name` attribute with the fully-qualified class name implementing *the plug-in*, and an optional `PlugInPars` attribute with a comma-separated `key=value` definitions, which are passed down to the class constructor as keyword arguments.
- *FreeSpaceDiskChangeMb*: How much available free space in a disk will trigger an error notification to change that disk (see *Notification* for details).
- *MinFreeSpaceWarningMb*: Minimum amount of free space a disk should have. If a disk has less free space than that a warning email is sent (see *Notification*).

10.9 Processing

The `Processing` element defines the behavior of the optional *on-the-fly processing capabilities* attached to the *RETRIEVE* command. The following attributes are supported:

- *ProcessingDirectory*: The directory (potentially relative to the NGAS root directory) where a `processing` directory will be created on, under which temporary files used during on-the-fly processing will be put under.

Under the `Processing` element, one or more `PlugIn` sub-elements can be placed, one per processing plug-in to be declared. Each `PlugIn` element accepts the following attributes:

- *Name*: The name of the python module (with a similarly-named function) where the plug-in is implemented.
- *PlugInPars*: A comma-separated list of `key=value` parameter definitions to be passed to the plug-in.

Finally, inside each `PlugIn` element one or more `MimeType` elements can be added to specify which MIME types will be processed by the plug-in. Each `MimeType` element needs to have a `Name` attribute with specifying the MIME type.

10.10 Register

The `Register` element configures the plug-ins to be used by the *REGISTER* command.

Plug-ins are configured per mime-type. Like *Processing*, one or more `PlugIn` sub-elements can be placed under the `Register` element, following the same guidelines.

10.11 Notification

The `Notification` element defines the behavior of the server *email notifications*. The following attributes are available:

- *Active*: Whether notifications are enabled or not. Note that even if disabled, there are some notifications (that are considered too important to be missed) that will still be sent.
- *SmtpHost*: The SMTP host to use as the email agent.
- *Sender*: The email address that will appear in the `Sender:` field of emails sent by this mechanism.
- *MaxRetentionTime*: Maximum amount of time an undelivered email will be internally kept for before the system decides not to deliver it.
- *MaxRetentionSize*: Maximum amount of undelivered emails the system will keep internally before it starts dropping old emails.

Emails resulting from different events can be configured to be sent to one or more email addresses. This is done by defining `EmailRecipient` elements, each with an `Address` attribute whose value is the target email address. These `EmailRecipient` elements are then added as children of the following sub-elements of `Notification`:

- *AlertNotification*: (*Deprecated*) Never sent.
- *ErrorNotification*: Sent in a number of different error situations.
- *DiskSpaceNotification*: Sent when, during operations, one or more disk are found to have less free space than the configured amount (see *ArchiveHandling*).
- *DiskChangeNotification*: Sent when a disk is full, potentially requiring a change.
- *NoDiskSpaceNotification*: Sent when, during operations, no sufficient space can be found in one or more disks.
- *DataCheckNotification*: Sent by the *Data check thread* informing about the results of the data checking process. Normally sent only if there are errors to be reported, but can be configured to be always sent (see *DataCheckThread*)

Below is an example illustrating a valid configuration:

```
<Notification Id="Notification"
              Active="0" MaxRetentionSize="1" MaxRetentionTime="00T00:30:00"
              Sender="ngas@host.com" SmtpHost="localhost">
  <AlertNotification>
    <EmailRecipient Address="address@example.com"/>
  </AlertNotification>
  <ErrorNotification>
    <EmailRecipient Address="address@example.com"/>
  </ErrorNotification>
  <DiskSpaceNotification>
    <EmailRecipient Address="address@example.com"/>
  </DiskSpaceNotification>
  <DiskChangeNotification>
    <EmailRecipient Address="address@example.com"/>
  </DiskChangeNotification>
  <NoDiskSpaceNotification>
    <EmailRecipient Address="address@example.com"/>
  </NoDiskSpaceNotification>
  <DataCheckNotification>
    <EmailRecipient Address="address@example.com"/>
  </DataCheckNotification>
</Notification>
```

10.12 JanitorThread

The `JanitorThread` element defines the behavior of the *Janitor Thread* (now actually implemented as a separate process). The following attributes are available:

- *SuspensionTime*: The sleep time after a janitor cycle.
- *MinSpaceSysDirMb*: The minimum space to be found on each volume during each cycle. If not enough space is found the system is sent to OFFLINE state.
- *PlugIn*: An XML sub-element with a *Name* attribute, naming a python module where a Janitor plug-in resides. Multiple *Plugin* elements can be defined.

10.13 DataCheckThread

The `DataCheckThread` element defines the behavior of the *Data check thread*. The following attributes are available:

- *Active*: Whether the data-check thread should be allowed to run or not.
- *MaxProcs*: Maximum number of worker processes used to carry out the data checking work load.
- *MinCycle*: The time to leave between data-check cycles.
- *ForceNotif*: Forces the sending of a notification report after each data-check cycle, even if not problems were found.
- *Scan*: Whether files should be scanned only (1) or actually checksumed (0).

The following attributes are present in old configuration files but are not used anymore: *FileSeq*, *DiskSeq*, *LogSummary*, *Prio*, *ChecksumPlugIn* (see *CRCVariant* instead) and *ChecksumPlugInPars*.

10.14 Caching

The `Caching` element defines the behavior of the *cache control thread*. When enabled, it is said that the NGAS server is running in *cache mode*. The following attributes are available:

- *Enable*: Whether the cache control thread should run or not.
- *Period*: The period at which the cache control thread runs.
- *MaxTime*: The maximum time files can stay in the cache.
- *MaxCacheSize*: The maximum total allowed volume of files in the cache.
- *MaxFiles*: The maximum allowed number of files in the cache.
- *CacheControlPlugIn*: A user-provided cache deletion plug-in that decides whether individual files should be marked for deletion.
- *CacheControlPlugInPars*: Parameters for the plug-in above.
- *CheckCanBeDeleted*: Check if a file marked for deletion has been sent to all subscribers yet before actual deletion occurs.

10.15 Log

The server outputs its logs to stdout, to a file, and to syslog, all of which are optional. The `Log` element of the configuration file contains the details to configure the server logging output.

- *LocalLogFile*: The file where the logs are dumped to. If given as a relative path it is relative to the NGAS root directory.
- *LocalLogLevel* An integer from 1 to 5 indicating the log levels that the server should output to `LocalLogFile`.
- *LogRotateInt*: The interval after which the `LocalLogFile` is rotated. Specified as `THH:mm:SS`. Defaults to 10 minutes.
- *LogRotateCache*: The amount of rotated files to retain. If more rotated files are found, they are removed by the system.
- *SysLog*: An integer indicating whether syslog logging is enabled (`1`) or disabled (`0`).
- *SysLogPrefix*: The string used as prefix for all syslog messages.
- *SysLogAddress*: The address where the syslog messages should be sent to. If not specified a platform-dependent default value is used.
- *ArchiveRotatedLogfiles*: An integer indicating whether rotated logfiles should be locally archived by NGAS (`1`) or not (`0`). Defaults to `0`.
- *LogfileHandlerPlugIn*: Zero or more sub-elements defining additional modules that will handle rotated logfiles. Each element should have a `Name` attribute with the fully-qualified module name implementing the plug-in inside a `run` method, and a `PlugInPars` element with a comma-separated, `key=value` pairs.

10.16 Authorization

The `Authorization` element defines the authentication and authorization rules that the NGAS server will follow when receiving commands from clients. For details see *Authorization*.

The `Authorization` element has an `Enable` attribute which determines whether authentication and authorization is enabled (`1`) or not (`0`). The `Authorization` element also has an `Exclude` attribute for defining a list of commands that are to be excluded from authoriztion. Zero or more `User` XML sub-elements also describe a different user recognized by NGAS. Each `User` element should have the following attributes:

- *Name*: The username.
- *Password*: The base64-encoded password.
- *Commands*: A comma-separated list of commands this user is allowed to execute. The special value `*` is interpreted as all commands.

10.17 SubscriptionAuth

The `SubscriptionAuth` element defines the authentication/authorisation configuration to use when acting as a client when using the subscription service. Currently it has only one element `PlugInName`, which follows the usual rules for plugins as noted above, with `PlugInName` being the name of the module to import. This module should have a callable which matches with the signature:

ngas_subscriber_auth (*filename*, *url*)
Provides authentication information needed to send `filename` to `url`.

This function should return an object that can be handled by the `auth` keyword argument of requests.requests, which is generally either a string, or an instance of `requests.auth.AuthBase`. `None` can be returned in the case where the authentication is not needed.

Parameters

- **`filename`** (*`str`*) – The filename to be sent
- **`url`** (*`str`*) – The url to send the filename to

Returns An object used by requests to authenticate the connection

Return type requests.auth.AuthBase, None, str

10.18 HostSuspension

The `HostSuspension` element defines the behavior of the *server suspension*. The following attributes are defined:

- *IdleSuspension*: Whether suspension is enabled (`1`) or not (`0`).
- *IdleSuspensionTime*: The amount of idle time after which a server will suspend itself.
- *SuspensionPlugIn* and *SuspensionPlugInPars*: The plug-in used to perform suspension, and its parameters.
- *WakeUpServerHost*: The server in charge of waking up server that are idling.
- *WakeUpPlugIn* and *WakeUpPlugInPars*: The plug-in used to perform the wake-up, and its parameters.
- *WakeUpCallTimeOut*: Maximum amount of time that a wake up call should take. If a server cannot be woken up after this timeout it is considered to be still idling.

10.19 SystemPlugIns

The `SystemPlugIns` element defines a collection of system-level plug-ins. These plug-ins are used for different purposes, either by a command or by the core system. The `*PlugIn` attributes name a python module that offers a function with the same name, while the `*PlugInPars` attributes are a comma-separated key=value parameter pairs:

- *LabelPrinterPlugIn* and *LabelPrinterPlugInPars*: The plug-in that brings hardware-specific capabilities to the `LABEL` command.
- *OfflinePlugIn* and *OfflinePlugInPars*: The plug-in used to bring the server to `OFFLINE` state (see *States*).
- *OnlinePlugIn* and *OnelinePlugInPars*: The plug-in used to bring the server to `ONLINE` state (see *States*).
- *DiskSyncPlugIn* and *DiskSyncPlugInPars*: The plug-in used to perform a full disk sync.

10.20 PartnerSites

The `PartnerSites` element defines a list of alternative (remote) NGAS servers belonging to separate NGAS archive installations. When a request to retrieve a file cannot be found on the local NGAS archive the request is redirected to the NGAS servers included in the partner sites list. The `ProxyMode` attribute can be used to enable/disable partner sites. Several `PartnerSite` child elements can be added containing the `Address` attribute for the remote NGAS server address.

CHAPTER 11

API

This is a very reduced version of the API documentation of NGAS. It should be helpful for users wanting to develop plug-ins for the system.

11.1 Database access

Contents

11.1.1 Core database access

These are the core classes implementing the core database access logic, including dealing with connections, cursors and transactions. For the higher-level database object offered by the `ngamsServer.ngamsServer.ngamsServer` class please see *Higher-level abstractions*.

class ngamsLib.ngamsDbCore.**ngamsDbCore**(*interface*, *parameters={}*, *createSnapshot=1*, *maxpoolcons=6*, *use_file_ignore=True*, *session_sql=None*)

Core class for the NG/AMS DB interface.

close()

Close the DB pool.

Returns: Void.

dbCursor(*sqlQuery*, *args=()*)

Create a cursor on the given query and return the cursor object.

query2(*sqlQuery*, *args=()*)
Takes an SQL query and a tuple of arguments to bind to the query

transaction()
Creates a new transaction object and return it

class ngamsLib.ngamsDbCore.**cursor2**(*pool*, *query*, *args*)
A cursor that yields values and acts as a context manager

close()
Closes the underlying cursor and connection

fetch(*howmany*)
Fetches at most howmany results from the database at a given time, yielding them instead of returning them as a sequence. This makes client code simpler to write.

class ngamsLib.ngamsDbCore.**transaction**(*db_core*, *pool*)
A context manager that allows multiple SQL queries to be executed within a single transaction

execute(*sql*, *args=()*)
Executes *sql* using *args*

11.1.2 Higher-level abstractions

ngamsLib.ngamsDb.**from_config**(*cfg*, *maxpool=None*)
Create a database object from a configuration object. If *maxpool* is not *None*, it overrides the value loaded from the configuration for the number of connections held by the pool.

class ngamsLib.ngamsDb.**ngamsDb**(*interface*, *parameters={}*, *createSnapshot=1*, *maxpoolcons=6*, *use_file_ignore=True*, *session_sql=None*)
Front-end class for the DB access module.

This class inherits from all the DB sub-classes (which in turn inherit from ngamsDbCore), thus exposing to the rest of the software a single class that implements all the database logic.

addDbChangeEvt(*evtObj*)
Add an Event Object (threading.Event) that will be triggered to indicate other threads that DB changes where introdiced.

evtObj: Event object (threading.Event).

Returns: Reference to object itself.

addDiskHistEntry(*hostId*, *diskId*, *synopsis*, *descrMimeType=None*, *descr=None*, *origin=None*, *date=None*)
Add an entry in the NGAS Disks History Table (ngas_disks_hist) indicating a major action or event occurring in the context of a disk.

dbConObj: Instance of NG/AMS DB class (ngamsDbBase).

diskId: Disk ID for the disk concerned (string).

synopsis: A short description of the action (string/max. 255 char.s).

descrMimeType: The mime-type of the contents of the description field. Must be specified when a description is given (string).

descr: An arbitrary long description of the action or event in the life-time of the disk (string).

origin: Origin of the history log entry. Can either be the name of an application or the name of an operator. If not specified (= None) it will be set to 'NG/AMS - <host name>' (string).

date: Date for adding the log entry. If not specified (set to None), the function takes the current date and writes this in the new entry (string/ISO 8601).

Returns: Void.

addFileToContainer(*containerId*, *fileId*, *force*)
Adds the file pointed by fileId to the container pointed by containerId. If the file doesn't exist an error will be raised. If the file is currently associated with a container and the force flag is not True an error will be raised also.

This method returns the uncompressed size of the file just added to the container. This can then be used to update the total size of the container

Parameters

- **containerId** (*str*) – the id of the container where the file will be added
- **fileId** (*str*) – the id of the file to add to the container
- **force** (*bool*) – force the operation

Returns the uncompressed file size of the file denoted by *fileId*

addSrvList(*srvList*)
Add a server list in the NGAS Server List Table and allocate a unique server list ID for it.

srvList: List of servers to add (string).

Returns: New server list ID (integer).

addSubscrBackLogEntry(*hostId*, *portNo*, *subscrId*, *subscrUrl*, *fileId*, *fileName*, *fileVersion*, *ingestionDate*, *format*)
Adds a Back-Log Entry in the DB. If there is already an entry for that file/Subscriber, a new entry is not created.

hostId: Host ID for NGAS host where Data Provider concerned is running (string).

portNo: Port number used by Data Provider concerned (integer).

subscrUrl: Susbcriber URl to where the files are delivered (string).

subscrId: Subscriber ID (string).

fileId: File ID (string).

fileName: Filename, i.e., name of file as stored in the Subscription Back-Log Area (string).

fileVersion: File Version (integer).

ingestionDate: File Ingestion Date (string/ISO 8601).

format: Mime-type of file (string).

Returns: Void.

addToContainerSize(*containerId*, *amount*)
Updates the size of the indicated container by the given amount

asTimestamp(*t*)
Returns *None* if timestamp is *None*, otherwise calls convertTimeStamp

buildFileSummary1Query(*columns, hostId=None, diskIds=[], fileIds=[], ignore=None, fileStatus=['00000000'], lowLimIngestDate=None, order=1*)
Builds the SQL query for a File Summary1 query. The fields to be selected are left open (specified as %s).

For a description of the input parameters, check the man-page of ngamsDbBase.getFileSummary1().

Returns: SQL query for a File Summary 1 Query (string).

close()
Close the DB pool.

Returns: Void.

closeContainer(*containerId*)
Marks the container as "closed"; that is, it sets an ingestion date on it equals to the current time

containerExists(*containerId*)
Returns whether the container with ID containerId exists (True) or not (False).

Parameters **containerId** – string

convertTimeStamp(*t*)
Convert a timestamp given in one of the following formats:

createContainer(*containerName*, *containerSize=0*, *ingestionDate=None*, *parentContainerId=None*, *parentKnownToExist=False*)
Creates a single container with name containerName. The ingestionDate parameter given to this method is a floating point number representing the number of seconds since the UNIX epoch, as returned by time.time()

If parentContainerId is given the new container will point to it as its parent. The parent container ID is checked for existence, unless parentKnownToExist indicates that the check is not necessary

Parameters

- **containerName** – string
- **containerSize** – integer
- **ingestionDate** – float
- **parentContainerId** – string
- **parentKnownToExist** – bool

Returns uuid.uuid4

createDbFileChangeStatusDoc(*hostId*, *operation*, *fileInfoObjList*, *diskInfoObjList=[]*)
The function creates a pickle document in the '<Disk Mt Pt>/.db/cache' directory from the information in the 'fileInfoObj' object.

operation: Has to be either ngams.NGAMS_DB_CH_FILE_INSERT or ngams.NGAMS_DB_CH_FILE_DELETE (string).

fileInfoObj: List of instances of NG/AMS File Info Object containing the information about the file (list/ngamsFileInfo).

diskInfoObjList: It is possible to give the information about the disk(s) in question via a list of ngamsDiskInfo objects (list/ngamsDiskInfo).

Returns: Void.

createDbRemFileChangeStatusDoc(*diskInfoObj*, *fileInfoObj*)
The function creates a File Removal Status Document with the information about a file, which has been removed from the DB and which should be removed from the DB Snapshot for the disk concerned.

diskInfoObj: Disk Info Object with info for disk concerned (ngamsDiskInfo).

fileInfoObj: Instance of NG/AMS File Info Object containing the information about the file (ngamsFileInfo).

Returns: Void.

dbCursor(*sqlQuery*, *args=()*)
Create a cursor on the given query and return the cursor object.

delSubscrBackLogEntries(*hostId*, *portNo*, *subscrId*)
Delete all entries to be delivered to a subscriber with subscrId

Parameters

- **hostId** (*str*) – Host ID for NGAS host where Data Provider concerned is running.
- **portNo** (*int*) – Port number used by Data Provider concerned
- **subscrId** (*str*) – Subscriber ID

delSubscrBackLogEntry(*hostId*, *portNo*, *subscrId*, *fileId*, *fileVersion*)
Delete an entry in the Subscription Back-Log Table.

hostId: Host ID for NGAS host where Data Provider concerned is running (string).

portNo: Port number used by Data Provider concerned (integer).

subscrId: Subscriber ID (string).

fileId: File ID (string).

fileVersion: File Version (string).

fileName: Filename, i.e., name of file as stored in the Subscription Back-Log Area (string).

Returns: Void.

deleteCacheEntry(*diskId*, *fileId*, *fileVersion*)
Delete an entry from the NGAS Cache Table.

diskId: Disk ID for the cached data object (string).

fileId: File ID for the cached data object (string).

fileVersion: Version of the cached data object (integer).

Returns: Reference to object itself.

deleteDiskInfo(*diskId*, *delFileInfo=1*)
Delete a record for a certain disk in the NGAS DB.

CAUTION: IF THE DB USER WITH WHICH THERE IS LOGGED IN HAS PERMISSION TO EXECUTE DELETE STATEMENTS, THE INFORMATION ABOUT THE DISK IN THE NGAS DB WILL BE DELETED! THIS INFORMATION CANNOT BE RECOVERED!!

diskId: ID of disk for which to delete the entry (string).

delFileInfo: If set to 1, the file information for the files stored on the disk is deleted as well (integer/0|1).

Returns: Reference to object itself.

deleteFileInfo(*hostId*, *diskId*, *fileId*, *fileVersion*, *genSnapshot=1*)
Delete one record for a certain file in the NGAS DB.

CAUTION: IF THE DB USER WITH WHICH THERE IS LOGGED IN HAS PERMISSION TO EXECUTE DELETE STATEMENTS, THE INFORMATION ABOUT THE FILE(S) IN THE NGAS DB WILL BE DELETED! THIS INFORMATION CANNOT BE RECOVERED!!

diskId: ID of disk hosting the file (string).

fileId: ID of file to be deleted. No wildcards accepted (string).

fileVersion: Version of file to delete (integer)

genSnapshot: Generate Db Snapshot (integer/0|1).

Returns: Reference to object itself.

deleteSubscriber(*subscrId*)
Delete the information for one Subscriber from the NGAS DB.

subscrId: Subscriber ID (string).

Returns: Reference to object itself.

destroySingleContainer(*containerId*, *checkForChildren*)
Destroys a single container with id containerId.

If the container contains subcontainers an error is issued.

Before destroying the container, all files associated to the container are removed from it first. If the container has subcontainers, it is not removed though

Parameters

- **containerId** – string
- **checkForChildren** – bool

diskInDb(*diskId*)
Check if disk with the given Disk ID is available in the DB.

diskId: Disk ID (string).

Returns: 1 = Disk ID was found, 0 = Disk ID not found (integer).

dumpFileInfoCluster(*clusterName*, *fileInfoDbmName=None*, *useFileKey=False*, *count=False*)
Dump the info for the files registered in the name space of the referenced cluster.

Note, all files in the cluster are taken, also the ones marked as bad or to be ignored.

clusterName: Name of cluster to consider (string).

fileInfoDbmName: Base name of the DBM in which the file info will be stored. If not given, a name will be generated automatically (string).

useFileKey: Use a file key (<File ID>_<Version>) as key as opposed to just an integer key. NOTE: Multiple occurrences of a given File ID/Version will only appear once (boolean).

count: When useFileKey == True, if count is True, the number of ocurrences of each File ID + Version is counted and an entry added in the DBM:

<File Key>__COUNTER

pointing to a counter indicating the number of occurrences. Note, the usage of '__' in the name of the counter for each file, means it will be skipped when doing a ngamsDbm.getNext(), scanning through the contents of the DBM (boolean).

Returns: Final name of the DBM DB containing the info about the files (string).

dumpMirroringQueue(*instanceId*)
Dump the entire contents of the DB Mirroring Queue into a DBM in raw format.

Returns: Name of DBM hosting the contents of the DB Mirroring Queue (string).

fileInDb(*diskId*, *fileId*, *fileVersion=-1*)
Check if file with the given File ID is registered in NGAS DB in connection with the given Disk ID.

diskId: Disk ID (string)

fileId: File ID (string).

fileVersion: Version of the file. If -1 version is not taken into account (integer).

Returns: 1 = file found, 0 = file no found (integer).

files_in_host (*hostId*, *from_date=None*)
Dump the info of the files defined by the parameters. The file info is dumped into a ngamsDbm DB.

For the parameters check man-page for: ngamsDbBase.getFileSummary1().

Returns: Name of the DBM DB containing the info about the files (string).

fromTimestamp (*timestamp*)
Converts a database timestamp into a number of seconds from the epoch. This is the reverse of *asTimestamp*.

getAvailableVolumes (*hostId*)
Returns a list of rows for all disks that are not marked as completed on `hostId`.

getBestTargetDisk (*diskIds*, *mtRootDir*)
Find the best suitable target disk among a set of disks referred to by their Disk IDs. The condition is:

Get ID for disk that is most full, which is Main Disk, mounted, has a Host ID corresponding to this host ID, which is not completed, which has the lowest Slot ID and which has the Disk ID among the set of disks defined for the given mime-type.

diskIds: List with Disk IDs to probe for (list).

mtRootDir: Base diretory for the NG/AMS Server (string).

Returns: Disk ID or None if no matches were found (string)

getCacheContents (*hostId*)
Execute query by means of a cursor, with which the entire contents of the cache can be downloaded.

hostId: Name of host to consider (string).

Returns: Cursor object with which the contents can be retrieved Cursor object (<NG/AMS DB Cursor Object API>).

getCfgPars (*name*)
Return the list of configuration parameters from the DB associated to the given name.

name: Name of the configuration association (string).

Returns: List with sub-lists with the information. The format is:

[[<Group ID>, <Parameter>, <Value>, <Comment>], . . .]

(list).

getClusterNameFromHostId (*hostId*)
Get the Cluster Name to which a node belongs from its Host ID.

hostId: Host ID (string).

Returns: Cluster Name (string).

getClusterReadyArchivingUnits (*clusterName*)
Return list of NAUs in the local cluster with archiving capability (archiving enabled + have capacity).

The resulting list of nodes will be formatted as:

[<Node>:<Port>, . . .] (list).

clusterName: Name of cluster to consider (string).

Returns: List with ready NAU nodes in the cluster (list/string).

getContainerIdForUniqueName(*containerName*)
Returns the ID of the container that can be uniquely identified by containerName. If no container with such a name exists, or if more than one exists, an error is raised

Parameters **containerName** – string

Returns string

getContainerName(*containerId*)
Returns the name of the container pointed by containerId. If no container with such ID exists, an error is raised

Parameters **containerId** – string

getCreateDbSnapshot()
Return the flag indicating if a DB Snapshot should be created or not.

Returns: Value of the DB Snapshot Creation Flag (boolean).

getDbTime()
Return the time spent for the last DB access.

Returns: Last DB access time in seconds (float).

getDiskCompleted(*diskId*)
Check if a disk is marked in the NGAS DB as completed.

diskId: ID of the disk (string).

Returns: 1 = completed, 0 = not completed. If the disk is not registered None is returned (integer).

getDiskIdFromSlotId(*host*, *slotId*)
Get a Disk ID for corresponding Slot ID and host name.

host: Host name (string).

slotId: ID of slot (string).

Returns: Disk ID or None if no match found (string).

getDiskIds()
Query the Disk IDs contained in the NGAS DB and return these in a list.

Returns: List with Disk IDs (list).

getDiskIdsMountedDisks(*host*, *mtRootDir*)
Get the Disk IDs for the disks mounted on the given host. A list is returned, which contains the Disk IDs of the disks mounted.

host: Name of host where the disk must be mounted (string).

mtRootDir: Base directory for NG/AMS (string).

Returns: List with Disk IDs (list).

getDiskIdsMtPtsMountedDisks(*host*)
Get the mount points for the disks mounted on the given host. A list is returned, which contains the Logical Names of the disks mounted.

host: Name of host where the disk must be mounted (string).

Returns: List with tuples containing Disk IDs and Mount Points (list/tuple).

getDiskInfoForSlotsAndHost(*host*, *slotIdList*)
From a given host and a given list of Slot IDs, the method returns a list with the disk info for the disks matching these.

host: Host name where the disks considered must be mounted (string).

slotIdList: List of Slot IDs for the disk considered (list).

Returns: List with Disk Info objects or [] if no matches were found (list/ngamsDiskInfo).

getDiskInfoFromDiskId(*diskId*)
Query the information for one disk (referred to by its ID), and return this in raw format.

The information about the disk is stored in a list with the the lay-out following the sequence as listed in the overall man-page for the ngamsDbBase class.

diskId: ID of the disk (string).

Returns: Disk information for the disk or [] if disk was not found (list).

getDiskInfoFromDiskIdList(*diskIdList*)
Get disk information from a list of Disk IDs given. The result is returned in a list containing again lists with the field as described in documentation for ngamsDbBase.getDiskInfoFromDiskId().

diskIdList: List with Disk IDs (list/string).

Returns: List with disk information (list/list).

getFileChecksum(*diskId*, *fileId*, *fileVersion*)
Get the checksum for the file.

diskId: ID of disk hosting the file (string).

fileId: ID of file to be deleted. No wildcards accepted (string).

fileVersion: Version of file to delete (integer)

Returns: checksum (string | None).

getFileChecksumValueAndVariant(*diskId*, *fileId*, *fileVersion*)
Get the checksum value and variant for the file.

diskId: ID of disk hosting the file (string).

fileId: ID of file to be deleted. No wildcards accepted (string).

fileVersion: Version of file to delete (integer)

Returns: checksum (string | None).

getFileInfoFromDiskIdFilename(*diskId*, *filename*)
The method queries the file information for a file referred to by the Disk ID for the disk hosting it and the filename as stored in the NGAS DB.

diskId: ID for disk hosting the file (string).

filename: NGAS (relative) filename (string).

Returns: Return ngamsFileInfo object with the information for the file if found or None if the file was not found (ngamsFileInfo|None).

getFileInfoFromFileId(*fileId*, *fileVersion=-1*, *diskId=None*, *ignore=None*, *dbCursor=1*, *order=1*, *order2=0*)
The method queries the file information for the files with the given File ID and returns the information found in a list containing sub-lists each with a list with the information for the file from the ngas_files table, host ID and mount point. The following rules are applied when determining which files to return:

o All files are considered, also files which are Offline.

o Files marked to be ignored are ignored.

o Latest version - first priority.

It is possible to indicate if files marked as being 'bad' in the DB should be taken into account with the 'ignoreBadFiles' flag.

If a specific File Version is specified only that will be taken into account.

The data can be retrieved via the DB Cursor returned by this object. The format of each sub-result is:

[<see getFileInfoFromFileIdHostId()>, <host ID>, <mnt pt>]

fileId: File ID for file to be retrieved (string).

fileVersion: If a File Version is specified only information for files with that version number and File ID are taken into account. The version must be a number in the range [1; oo[(integer).

diskId: ID of disk where file is residing. If specified to None (or empty string) the Disk ID is not taken into account (string).

ignore: If set to 0 or 1, this value of ignore will be queried for. If set to None, ignore is not considered (None|0|1).

dbCursor: If set to 1, a DB cursor is returned from which the files can be retrieved. Otherwise the result is queried and returned in a list (0|1/integer).

order: If set to 0, the list of matching file information will not be order according to the file version (integer/0|1).

Returns: Cursor object or list with results (<NG/AMS DB Cursor Object API>|list).

getFileInfoFromFileIdHostId(*hostId*, *fileId*, *fileVersion=1*, *diskId=None*, *ignore=None*)

Return list with information about a certain file referenced by its File ID. A list is returned with the following elements:

[<Disk ID>, <Filename>, <File ID>, <File Version>, <Format>, <File Size>, <Uncompressed File Size>, <Compression>, <Ingestion Date>, <Ignore>, <Checksum>, <Checksum Plug-In>, <File Status>, <Creation Date>]

hostId: Name of host where the disk is mounted on which the file is stored (string).

fileId: ID for file to acquire information for (string).

fileVersion: Version of the file to query information for (integer).

diskId: Used to refer to a specific disk (string).

ignore: If set to 0 or 1, this value of ignore will be queried for. If set to None, ignore is not considered (None|0|1).

Returns List with information about file, or [] if no file(s) was found (list).

getFileInfoList(*diskId*, *fileId=''*, *fileVersion=-1*, *ignore=None*, *fetch_size=1000*)

The function queries a set of files matching the conditions specified in the input parameters.

diskId: Disk ID of disk hosting the file(s) (string).

fileId: File ID of files to consider. Wildcards can be used (string).

fileVersion: Version of file(s) to consider. If set to -1 this is not taken into account (integer).

ignore: If set to 0 or 1, this value of ignore will be queried for. If set to None, ignore is not considered (None|0|1).

Returns: Cursor object (<NG/AMS DB Cursor Object API>).

getFileSize (*fileId*, *fileVersion*)
Get the ingestion date for the file.

diskId: ID of disk hosting the file (string).

fileId: ID of file to be deleted. No wildcards accepted (string).

fileVersion: Version of file to delete (integer)

Returns: Ingestion date for file or None if file not found (string/ISO 8601 | None).

getFileStatus (*fileId*, *fileVersion*, *diskId*)
Get the file_status string (bit) value in the ngas_files table.

fileId: ID of file (string).

fileVersion: Version of file (integer).

diskId: Disk ID for disk where file is stored (string).

Returns: File Status (8 bits) (string).

getFileSummary1 (*hostId=None, diskIds=[], fileIds=[], ignore=None, fileStatus=['00000000'], lowLimIngestDate=None, order=1*)
Return summary information about files. The information is returned in a list containing again sub-lists with contents as defined by ngamsDbCore.getNgasSummary1Cols() (see general documentation of the ngamsDbBase Class).

hostId: Name of NGAS host on which the files reside (string).

diskIds: Used to limit the query to certain disks (list/string).

fileIds: List of file IDs for which to query information. If not specified, all files of the referenced host will be chosen (list/string|[]).

ignore: If set to 0 or 1, this value of ignore will be queried for. If set to None, ignore is not considered (None|0|1).

fileStatus: With this parameter it is possible to indicate which files to consider according to their File Status (list).

lowLimIngestDate: Lower limit in time for which files are taken into account. Only files with an Ingestion Date after this date, are taken into account (string/ISO 8601).

order: Used to trigger ordering by Slot ID + Ingestion Date (integer/0|1).

Returns: Cursor object (<NG/AMS DB Cursor Object API>).

getFileSummary1SingleFile (*diskId*, *fileId*, *fileVersion*)
Same as getFileSummary1() but for a single (specific) file.

Returns: List with information from query (list).

getFileSummary2 (*hostId=None*, *fileIds=[]*, *diskId=None*, *ignore=None*, *ing_date=None*, *max_num_records=None*, *upto_ing_date=None*, *fetch_size=1000*)
Return summary information about files. An NG/AMS DB Cursor Object is created, which can be used to query the information sequentially.

The information is returned in a list containing again sub-lists with contents as defined by ngamsDbBase._sum2Cols (see general documentation of the ngamsDbBase Class.

This method returns all the files stored on an NGAS system also the ones with a File Status indicating that it is bad.

hostId: Name of NGAS host on which the files reside. If None is specified, the host is not taken into account (string or a list of string).

fileIds: List of file IDs for which to query information. If not specified, all files of the referenced host will be chosen (list/string|[]).

diskId: Used to refer to all files on a given disk (string|None).

ignore: If set to 0 or 1, this value of ignore will be queried for. If set to None, ignore is not considered (None|0|1).

max_num_records: The maximum number of returned records (if presented) (int)

Returns: Cursor object (<NG/AMS DB Cursor Object API>).

getFileSummary3 (*fileId*, *hostId=None*, *domain=None*, *diskId=None*, *fileVersion=-1*, *cursor=True*, *include_compression=False*)

Return information about files matching the conditions which are not in ignore and which are not marked as bad.

Files are ordered by the File Version (descending).

The resulting file information will be:

<Host ID>, <Ip Address>, <Port>, <Mountpoint>, <Filename>, <File Version>, <format>

fileId: ID of file to retrieve (string).

hostId: Host ID of node hosting file (string|None).

domain: Domain in which the node is residing (string|None).

diskId: Disk ID of disk hosting file (string|None).

fileVersion: Version of file to retrieve (integer).

cursor: Return DB cursor rather than the results (boolean).

Returns: Cursor object (<NG/AMS DB Cursor Object API>).

getFileSummarySpuriousFiles1 (*hostId=None*, *diskId=None*, *fileId=None*, *fileVersion=None*, *fetch_size=1000*)

Return summary information about spurious files, i.e. files registered in the DB as to be ignored and/or having a status indicating that they're not OK. The information is returned in a list containing again sub-lists with contents as defined by ngamsDbBase.getNgasSummary1Cols() (see general documentation of the ngamsDbBase Class.

hostId: Name of NGAS host on which the files reside (string).

diskId: Disk ID of disk to take into account (string|None).

fileId: File ID of file(s) to take into account (string|None).

fileVersion: Version of file(s) to take into account (integer|None).

Returns: Cursor object (<NG/AMS DB Cursor Object API>).

getHostIdsFromClusterName (*clusterName*)

Return the list of host IDs within the context of a given cluster.

clusterName: Name of cluster to consider (string).

Returns: List with nodes in the cluster (list/string).

getHostInfoFromHostIds (*hostList*)

Return a dictionary with the information in connection with each host. If for a host ID given, no information is found in the NGAS Hosts Table, the value for this wil be None.

hostList: List of host IDs (list/string).

Returns: List with sub-lists containing the information about the hosts from the NGAS Hosts Table (list).

getIngDate(*diskId*, *fileId*, *fileVersion*)
Get the ingestion date for the file.

diskId: ID of disk hosting the file (string).

fileId: ID of file to be deleted. No wildcards accepted (string).

fileVersion: Version of file to delete (integer)

Returns: Ingestion date for file or None if file not found (string/ISO 8601 | None).

getIpFromHostId(*hostId*)
Get the IP Address of a host from its Host ID.

hostId: Host ID (string).

Returns: IP Address (string).

getLastDiskCheck(*hostId=''*)
Queries all the Last Check Flags for all disks or for all disks currently mounted in a specific host. A Dictionary is returned containining the Disk IDs as keys, and the time for the last check in seconds. If the value is NULL in the DB, it is set to 0.

hostId: If specified, only disks mounted in this system are taken into account (string).

Returns: Dictionary with entry for each disk. Key is Disk ID (dictionary).

getLatestFileVersion(*fileId*)
The method queries the latest File Version for the file with the given File ID. If a file with the given ID does not exist, -1 is returned.

fileId: File ID (string).

Returns: Latest File Version (integer).

getLogicalNameFromDiskId(*diskId*)
Query the Logical Name of a disk from the DB, based on the Disk ID of the disk.

diskId: Disk ID (string).

Returns: Logical Name or None if not found (string | None).

getLogicalNamesMountedDisks(*host*)
Get the Logical Names for the disks mounted on the given host. A list is returned, which contains the Logical Names of the disks mounted.

host: Name of host where the disk must be mounted (string).

Returns: List with Logical Names (list).

getMaxDiskNumber(*cat=None*)
Get the maximum disk index (number) in connection with the Logical Disk Names in the DB.

cat: 'M' for Main, 'R' for Replication (string).

Returns: The maximum disk number or None if this could not be generated (integer).

getMinLastDiskCheck(*hostId*)
Get the timestamp for the disk that was checked longest time ago for all disks mounted in a specific NGAS Host.

hostId: Host ID of the host to consider (string).

Returns: Time since the 'oldest', last check (seconds since epoch) (integer).

getMtPtFromDiskId(*diskId*)
Get the mount point for the disk referred to.

diskId: ID of the disk (string).

Returns: Mount point of disk or None if not mounted or not found (stringlNone)

getNgasFilesMap()
Return the reference to the map (dictionary) containing the mapping between the column name and index of the ngas_files table.

Returns: Reference to NGAS Files Table name map (dictionary).

getNumberOfFiles(*diskId=''*, *fileId=''*, *fileVersion=-1*, *ignore=None*, *onlyOnlineFiles=0*)
Get the number of files stored on a disk.

diskId: Disk ID of disk to get the number of files for (string).

fileId: File ID for file to be retrieved (string).

fileVersion: If a File Version is specified only information for files with that version number and File ID are taken into account. The version must be a number in the range [1; oo[(integer).

ignore: If set to 0 or 1, this value of ignore will be queried for. If set to None, ignore is not considered (None|0|1).

onlyOnlineFiles: If specified, only files which are Online or on suspended nodes are considered (integer/0|1).

Return: Number of files stored on the disk (integer).

getPortNoFromHostId(*hostId*)
Return the port number corresponding to the host ID.

hostId: Host ID (string).

Return: Port number (integer).

getSlotIdFromDiskId(*diskId*)
Get the Slot ID for a disk, given by the Disk ID for the disk.

diskId: ID of the disk (string).

Returns: Slot ID of disk. If disk is not found None is returned (string | None).

getSlotIdsMountedDisks(*host*)
Get the Slot IDs for the disks mounted on the given host. A list is returned, which contains the Slot IDs of the disks mounted.

host: Name of host where the disk must be mounted (string).

Returns: List with Slot IDs (list).

getSpaceAvailForHost(*hostId*)
Return the amount of free disk space for the given host. This is calculated as the total sum of free space for all non-completed volumes mounted in this node, according to the DB.

hostId: Name of host (string).

Returns: Amount of free disk space in MB bytes (float)

getSrvDataChecking(*hostId*)
Return flag indicating if server is executing a Data Consistency Check.

hostId: Host ID (string).

Returns: Server suspension flag (integer/0|1).

getSrvListFromId(*srvListId*)
Get a server list from its ID. If no list with that ID is found, None is returned.

srvListId: Server list ID (integer).

Returns: Server list associated with the given ID (string|None).

getSrvListIdFromSrvList(*srvList*)
Get the server list ID associated with the server list. If not defined, a new can be allocated in the NGAS Servers Table automatically.

srvList: Server list ('<host>:<port>,...') (string).

Returns: Server list ID (integer).

getSrvSuspended(*contactAddr*, *ngasHostId=None*)
Return flag indicating if the server is suspended.

contactAddr: Host ID or IP address (string).

ngasHostId: NGAS Host ID, e.g. myhost:8888 (string).

Returns: Server suspension flag (integer/0|1).

getSubscrBackLog(*hostId*, *portNo*, *selectDiskId=False*)
Read all entries in the Subscriber Back-Log Table 'belonging' to a specific Data Provider, and return these in a list with sub-lists.

hostId: Host ID of Data Provider (string).

portNo: Port number used by Data Provider (integer).

Returns: List containing sub-list with the following information:

> **[[<Subscr. ID>, <Subscr. URL>, <File ID>, <Filename>,** <File Version>, <Ingestion Date>, <Format <Mime-Type>], ...]
>
> Note that the part of the list after the Subscriber URL is the same as generated by ngamsDbBase.getFileSummary2() (list/list).

getSubscrBackLogBySubscrId(*subscrId*)
Get all entries in the Susbscriber Back-log Table to be delivered to a specific subscriber

subscrId Subscriber Id

Returns List containing sublist with the following information: [[<file_id>, <file_version>], ...]

getSubscrBackLogCount(*hostId*, *portNo*)
Read the number of entries in the Subscriber Back-Log Table 'belonging' to a specific Data Provider/Mover

hostId: Host ID of Data Provider (string).

portNo: Port number used by Data Provider (integer).

Returns: The number of records (integer)

getSubscrQueue(*subscrId*, *status=None*)
Read all entries in the ngas_subscr_queue table 'belonging' to a specific subscriber, and where the status meets the "status" condition

subscrId: subscriber Id (string) status: the status of current file delivery (int or None)

getSubscriberInfo(*subscrId=None*, *hostId=None*, *portNo=-1*)
Get the information for one or more Subcribers from the ngas_subscribers table and return the contents in a list. The format of this list is formatted as follows:

[<Host ID>, <Port No>, <Priority>, <Subscriber ID>, <Subscriber URL>, <Subscription Start Date>, <Subscription Filter Plug-In>, <Subscription Filter Plug-In Parameters>, <Last File Ingestion Date>]

subscrId: ID of the Subcriber (string).

hostId: Limit the query to Subscribers in connection with one host (Data Provider) (string).

portNo: Limit the query to Subscribers in connection with one host (Data Provider) (integer).

Returns: If a Subscriber ID is specified: List with information about the Subscriber (if found). Otherwise [] is returned (list).

If no Subscriber ID is given: List with sub-lists with information for all Subscribers. Otherwise [] is returned (list/list).

getSubscriberStatus(*subscrIds*, *hostId=''*, *portNo=-1*)
Method to query the information about the Ingestion Date of the last file delivered to the Subscriber. A list is returned, which contains the following:

[(<Subscriber ID>, <Last File Ingestion Date (ISO 8601)>), . . .]

subscrIds: List of Subscriber ID to query (list/string).

hostId: Host name of Subscriber host (string).

portNo: Port number used by Subscriber host (integer).

Returns: List with Subscriber status (list/tuple/string).

getSumBytesStored(*diskId*)
Get the total sum of the sizes of the data files stored on a disk and return this.

diskId: Disk ID of disk to get the sum for (string).

Return: Total sum of bytes stored on the disk (integer).

getWakeUpRequests(*hostId*)
Generates a tuple with suspended NGAS Hosts that have requested to be woken up by this host.

Returns: Tuple containing sub-tuples with information about hosts to be woken up:

(({host id}, {wake-up time (secs since epoch)}), . . .)

(list/tuple)

hasCfgPar(*groupId*, *parName*)
Return 1 if the given configuration parameter (given by its Simplified XPath name) and Configuration Group ID is defined in the configuration table in the DB.

groupId: Group ID for the parameter (string).

parName: Name of parameter (string).

Returns: 1 = parameter defined, 0 = parameter not defined (integer/0|1).

insertCacheEntry(*diskId*, *fileId*, *fileVersion*, *cacheTime*, *delete*)
Insert a new cache entry into the NGAS Cache Table.

diskId: Disk ID of the cache entry (string).

fileId: File ID of the cache entry (string).

fileVersion: File Version of the cache entry (string).

cacheTime: Time the file entered in the cache (= ngas_files.ingestion_time) (float).

delete: Flag indicating if the entry is scheduled for deletion (boolean).

Returns: Reference to object itself.

insertSubscriberEntry(*sub_obj*)
Inserts the new subscription object into the NGAS subscription table. If an object with the same subscription ID exists, its contents are returned; otherwise the given object is returned.

markHostSuspended(*hostId*)
Mark a host as being suspended in the NGAS DB.

hostId: Name of host to mark as suspended. If not given the local host name is used (string).

Returns: Reference to object itself.

mirReqInQueue(*fileId*, *fileVersion*, *instanceId*)
Probe if the a Mirroring Request with the given ID is in the associated Mirroring Queue.

fileId: File ID (string).

fileVersion: File Version (integer).

instanceId: Identification of the NGAS instance taking care of coordinating the mirroring (string).

Returns: Indication if the request is in the queue (boolean).

query2(*sqlQuery*, *args=()*)
Takes an SQL query and a tuple of arguments to bind to the query

read(*containerId*)
Reads a single ngamsContainer object from the database

Parameters **containerId** (*str*) – the id of the container to read

Returns the container object

Return type ngamsContainer.ngamsContainer

readHierarchy(*containerId*, *includeFiles=False*)
Reads an ngamsContainer object from the database and recursively populates it with its children containers.

Parameters **containerId** (*str*) – the id of the container whose hierarchy is to be read

Returns The container object recursively populated

Return type ngamsContainer.ngamsContainer

relGlobalDbSem()
Release acquired access to a critical, global DB interaction.

Returns: Reference to object itself.

removeFileFromContainer(*fileId*, *containerId*)
Removes the file pointed by fileId from the container pointed by containerId. If the file doesn't exist an error will be raised. If the file is currently not associated with the indicated container and error will be raised also.

This method returns the uncompressed size of the file just removed from the container. This can then be used to update the total size of the container

Parameters

- **fileId** (*str*) – the id to remove from the container

- **containerId** (*str*) – the container from which the file needs to be removed

Returns the uncompressed size of the file just removed from the container

Return type integer

remove_file(*file_size*, *disk_id*)
Update the disk information to reflect that a file has been removed from the disk

Parameters

- **file_size** (*int*) – the size of the file on disk
- **disk_id** (*str*) – the ID of the disk

replace_file(*old_file_size*, *old_disk_id*, *new_file_size*, *new_disk_id*)
Update the disk information to reflect a file, potentially of a different size, is being replaced, potentially in a different disk

Parameters

- **old_file_size** (*int*) – the size of the old copy of the file on disk
- **old_disk_id** (*str*) – the ID of the disk with the old copy of the file
- **new_file_size** (*int*) – the size of the new copy of the file on disk, could be the same as `old_file_size`.
- **new_disk_id** (*str*) – the ID of the disk with the old copy of the file, could be the same as `new_disk_id`.

reqWakeUpCall(*localHostId*, *wakeUpHostId*, *wakeUpTime*)
Request a Wake-Up Call via the DB.

wakeUpHostId: Name of host where the NG/AMS Server requested for the Wake-Up Call is running (string).

wakeUpTime: Absolute time for being woken up (seconds since epoch) (integer).

Returns: Reference to object itself.

resetDbTime()
Reset the Db timer.

Returns: Reference to object itself.

resetWakeUpCall(*hostId*, *resetSrvSusp=0*)
Cancel/reset the Wake-Up Call parameters.

hostId: If specified, another host ID than the one where this NG/AMS Server is running can be indicated (string).

Returns: Reference to object itself.

setContainerSize(*containerId*, *containerSize*)
Updates the size of the indicated container

setDbTmpDir(*tmpDir*)
Set the DB temporary directory.

tmpDir: Temporary directory (string).

Returns: Reference to object itself.

setFileChecksum(*hostId*, *fileId*, *fileVersion*, *diskId*, *checksum*, *checksumPlugIn*)
Set the checksum value in the ngas_files table.

hostId: ID of this NGAS host

fileId: ID of file (string).

fileVersion: Version of file (integer).

diskId: ID of disk where file is stored (string).

checksum: Checksum of file (string).

checksumPlugIn: Name of plug-in used to generate the checksum (string).

Returns: Reference to object itself.

setLastCheckDisk(*diskId*, *timeSecs*)
Update the Last Check Flag for a disk.

diskId: ID of disk for which to update record (string).

timeSecs: Time in seconds to set for the disk (integer).

Returns: Reference to object itself.

setLogicalNameForDiskId(*diskId*, *logicalName*)
Change the Logical Name of the disk with the given Disk ID.

diskId: Disk ID (string).

logicalName: New Logical Name (string).

Returns: Void.

subscrBackLogEntryInDb(*hostId*, *portNo*, *subscrId*, *fileId*, *fileVersion*)
Check if there is an entry in the Subscription Back-Log for that file/Subscriber.

hostId: Host ID for NGAS host where Data Provider concerned is running (string).

portNo: Port number used by Data Provider concerned (integer).

subscrId: Subscriber ID (string).

fileId: File ID (string).

fileVersion: File Version (string).

Returns: 1 = file found, 0 = file no found (integer).

subscriberInDb(*subscrId*)
Check if the Subscriber with the given ID is registered in the DB.

subscrId: Subscriber ID (string).

Returns: 1 = Subscriber registered, 0 = Subscriber not registered (integer).

takeGlobalDbSem()
Acquire access to a critical, global DB interaction.

Returns: Reference to object itself.

transaction()
Creates a new transaction object and return it

triggerEvents(*eventInfo=None*)
Set the Event Objects to inform other threads about DB changes.

eventInfo: Piece of information to be transferred from one thread to another (free format).

Returns: Reference to object itself.

unpackMirReqSqlResult (*sqlResult*)
Unpack a SQL result for one row in the DB Mirroring Table. The columns in the result must be ordered according to the sequence given by ngamsDbCore.getNgasMirQueueCols().

sqlResult: List with elements resulting from the query for one row (list).

Returns: Mirroring Request Object (ngamsMirroringRequest).

updateCacheEntry (*diskId*, *fileId*, *fileVersion*, *delete*)
Update the online status of this cached data object.

diskId: Disk ID for the cached data object (string).

fileId: File ID for the cached data object (string).

fileVersion: Version of the cached data object (integer).

delete: Entry scheduled for deletion (integer/0|1).

Returns: Reference to object itself.

updateDataCheckStat (*hostId*, *start*, *remain*, *estimTime*, *rate*, *checkMb*, *checkedMb*, *checkFiles*, *checkedFiles*)
Update the statistics for the Data Checking Thread.

hostId: ID of NGAS Host to update statistics for (string).

start: Start of checking in seconds since epoch (integer).

remain: Estimated remaining time in seconds (integer).

estimTime: Estimated total time in seconds to complete the check cycle (integer)

rate: Rate of checking in MB/s (float).

checkMb: Amount of data to check in MB (float).

checkedMb: Amount checked in MB (float).

checkFiles: Number of files to check (integer).

checkedFiles: Number of files checked (integer).

Returns: Reference to object itself.

updateDbTime (*dbAccessTime*)
Update the DB access timer.

dbAccessTime: DB access time to add in seconds (float).

Returns: Reference to object itself.

updateDiskFileStatus (*diskId*, *fileSize*)
Update the NGAS Disks Table according to a new file archived.

diskId: Disk ID (string).

fileSize: Size of file as stored on disk (integer).

Returns: Reference to object itself.

updateDiskInfo (*fileSize*, *diskId*)
Update the row for the volume `diskId` hosting the new file of size `fileSize`.

updateMirReq (*mirReqObj*)
Update the referenced Mirroring Request in the DB. The request is defined by a the set of File ID and File Version.

mirReqObj: Instance of the Mirroring Request Object containing the information about the Mirroring Request (ngamsMirroringRequest).

Returns: Reference to object itself.

updateSrvHostInfo(*hostId*, *srvInfo*)
Update the information in the DB, which is managed by the server itself. All columns starting with `srv_` in the ngas_hosts tables are defined. The values can be taken from an instance of the ngamsHostInfo class.

srvInfo: List containing all information about the host. These are all fields starting with `srv_` from `srv_version` to `srv_state` (list).

ignoreErr: If set to 1, a possible exception thrown will be caught, and this error ignored. Otherwise the method will throw an exception itself (integer/0|1).

Returns: Void.

updateStatusMirReq(*fileId*, *fileVersion*, *newStatus*)
Update the status of the Mirroring Request.

fileId: File ID (string).

fileVersion: File Version (integer).

newStatus: New status for the Mirroring Request to write to the DB
(ngamsMirroringRequest.NGAMS_MIR_REQ_STAT_SCHED, ...)

Returns: Reference to object itself.

updateSubscrQueueEntry(*subscrId*, *fileId*, *fileVersion*, *diskId*, *status*, *status_date*, *comment=None*)
Update the status (and comment) of a file in the persistent queue given its primary key

updateSubscrQueueEntryStatus(*subscrId*, *oldStatus*, *newStatus*)
change the status from old to new for files belonging to a subscriber

updateSubscrStatus(*subscrId*, *fileIngDate*)
Update the Subscriber Status so that it reflects the File Ingestion Date of the last file ingested.

subscrId: Subscriber ID (string).

fileIngDate: File Ingestion Date (string/ISO 8601).

Returns: Void.

updateSubscriberEntry(*sub_obj*)
The method writes the information in connection with a Subscriber in the NGAS DB. If an entry already exists for that disk, it is updated with the information given as input parameters. Otherwise, a new entry is created.

hostId: ... filterPlugInPars: Parameters for the Subscriber (string).

priority: Priority of Subscriber (integer).

startDate: Date the subscription should start from (string/ISO 8601).

lastFileIngDate: Ingestion dtae of last file delivered (string/ISO 8601).

Returns: Returns 1 if a new entry was created in the DB and 0 if an existing entry was updated (integer/0|1).

writeCfgPar(*groupId*, *parName*, *value*, *comment*)
Write a configuration parameter to the NGAS DB. If the parameter is already defined, the value/comment are updated.

groupId: Configuration Group ID (string).

parName: Name of parameter (string).

value: Value of parameter (string).

comment: Comment for parameter (string).

Returns: Reference to object itself.

writeDiskEntry(*diskId*, *archive*, *installationDate*, *type*, *manufacturer*, *logicalName*, *hostId*, *slotId*, *mounted*, *mountPoint*, *numberOfFiles*, *availableMb*, *bytesStored*, *completed*, *completionDate*, *checksum*, *totalDiskWriteTime*, *lastCheck*, *lastHostId*)

The method writes the information in connection with a disk in the NGAS DB. If an entry already exists for that disk, it is updated with the information contained in the Disk Info Object. Otherwise, a new entry is created.

Returns 1 if a new entry was created in the DB, 0 if an existing entry was updated.

writeFileEntry(*hostId*, *diskId*, *filename*, *fileId*, *fileVersion*, *format*, *fileSize*, *uncompressedFileSize*, *compression*, *ingestionDate*, *ignore*, *checksum*, *checksumPlugIn*, *fileStatus*, *creationDate*, *iotime*, *ingestionRate*, *genSnapshot=1*, *updateDiskInfo=0*, *prev_disk_id=None*)

The method writes the information in connection with a file in the NGAS DB. If an entry already exists for that file, it is updated with the information contained in the File Info Object. Otherwise, a new entry is created.

diskId Values for the columns in the ngas_disks ... table (use values returned from ngamsFileInfo).

genSnapshot: Generate a snapshot file (integer/0|1).

updateDiskInfo: Update automatically the disk info for the disk hosting this file (integer/0|1).

Returns: Void.

writeHostInfo(*hostInfoObj*)

Create an entry in the NGAS Hosts Table

hostInfoObj: ngamsHostInfo object containing the information for the new entry (ngamsHostInfo).

Returns: Reference to object itself.

writeMirReq(*mirReqObj*, *check=True*)

Write the referenced Mirroring Request in the DB. The request is defined by a the set of File ID and File Version.

mirReqObj: Instance of the Mirroring Request Object containing the information about the Mirroring Request (ngamsMirroringRequest).

check: Check if the entry is already in the queue. In case yes, just update it (boolean).

Returns: Reference to object itself.

11.2 Configuration

class ngamsLib.ngamsConfig.**ngamsConfig**(*filename=''*, *dbObj=None*)

Class to handle the information in the NG/AMS Configuration.

addAuthUser(*user*, *password*)

Add a user in the object.

user: User name (string).

password: Encrypted password (string).

Returns: Reference to object itself.

addAuthUserCommands(*user*, *commands*)
Add a user in the object.

user: User name (string).

commands: comma separated commands (string). e.g. RETRIEVE,STATUS,QARCHIVE a "*" means all commands

Returns: Reference to object itself.

addMimeTypeMap(*mimeType*, *extension*)
Add a mime-type map to the object.

mimeType: Mime-type (string).

extension: Extension corresponding to mime-type (string).

Returns: Reference to object itself.

addMirroringSrcObj(*mirSrcObj*)
Add a new Mirroring Source Object in the internal list.

mirSrcObj: Mirroring Source Object (ngamsMirroringSource).

Returns: Reference to object itself.

addStorageSetObj(*storageSetObj*)
Add a Storage Set object to the configuration.

storageSetObj: Instance of Storage Set class (ngamsStorageSet).

Returns: Reference to object itself.

addStreamObj(*streamObj*)
Add an ngamsStream object.

streamObj: Stream object (ngamsStream).

Returns: Reference to object itself.

clear()
Clear the object.

Returns: Reference to object itself.

dumpXmlDic()
Dump the contents of the XML Dictionary in a buffer in the format:

<Key> = <Value> <Key> = <Value> . . .

Returns: Reference to string buffer with the XML Dictionary dump (string).

genXml(*hideCritInfo=1*)
Generate an XML DOM Node object from the contents of the ngamsConfig object.

Returns: XML DOM Node (Node).

genXmlDoc(*hideCritInfo=1*)
Generate an XML Document from the contents loaded in a string buffer and return this.

hideCritInfo: Hide critical information (integer/0|1).

Returns: XML document (string).

getAlertNotifList()
Get reference to tuple with recipients of Alert Notification Events.

Returns: Tuple with recipients (tuple).

getAllowArchiveReq()
Get the Allow Archive Request Flag.

Returns: Allow Archive Request Flag (integer).

getAllowProcessingReq()
Get the Allow Processing Request Flag.

Returns: Allow Processing Request Flag (integer).

getAllowRemoveReq()
Get the Allow Remove Request Flag.

Returns: Allow Remove Request Flag (integer).

getAllowRetrieveReq()
Get the Allow Retrieve Request Flag.

Returns: Allow Retrieve Request Flag (integer).

getArchiveName()
Get name of the archive.

Returns: Name of archive (string).

getArchiveRotatedLogfiles()
Whether rotated logfiles are automatically archived locally or not

getArchiveUnits()
Get Archive Units.

Returns: Archive Units (string).

getAssocSlotId(*slotId*)
Get the Slot ID of the disk associated to the disk with the given Slot ID.

slotId: Slot ID (string).

Returns: Slot ID of associated disk - "" if not found (string).

getAuthExcludeCommandList()
Return the list of commands excluded from authorization

Returns: comma separated commands (string). e.g. ARCHIVE,CHECKFILE,RETRIEVE,STATUS
A "*" is a wildcard for all commands

getAuthHttpHdrVal(*user*)
Generate the value to be sent with the HTTP Autorization Header. If no specific user is given, an arbitrary user is chosen.

user: Name of registered user (string|None).

Returns: Authorization HTTP Header value (string).

getAuthUserCommands(*user*)
Returns the info (password) for a user.

user: User name (string).

Returns: Password or None (string).

getAuthUserInfo(*user*)
Returns the info (password) for a user.

user: User name (string).

Returns: Password or None (string).

getAuthorize()
Return the authorization flag.

Returns: 1 = authorization on (integer/0|1).

getAutoUnsubscribe()
Return the Auto-Unsubscribe Flag.

Returns: Auto Un-Subscribe Flag (integer/0|1).

getBackLogBufferDirectory()
Get the Back Log Buffer Directory.

Returns: Back Log Buffer Directory (string).

getBackLogBuffering()
Get the enable/disable Back Log Buffering Flag.

Returns: Back Log Buffering on/off (0/1) (integer).

getBackLogDir()
Return the exact (complete) name of the Back-Log Buffer Directory.

Returns: Name of Back-Log Buffer Directory (string).

getBackLogExpTime()
Return the expiration time for directories and files in the Subscription Back-Log Area.

Returns: Expiration time (string/ISO 8601).

getBlockSize()
Get HTTP data read/write block size.

Returns: HTTP data read/write block size (integer).

getCRCVariant()
Defines the CRC Variant to use.

Returns

- -1: Don't perform any CRC calculation at all
- 0: `crc32` (using python's binascii implementation w/o masking)
- 1: `crc32c` (using Intel's SSE 4.2 implementation via the `crc32c` module)
- 2: `crc32z` (using python's binascii implementation w/ masking)

getCachingEnabled()
Whether the server is configured to operate in caching mode (default: False)

getCachingPeriod()
Return the period for checking the cache holding.

Returns: Value of caching period (integer).

getCfg()
Get the name of the configuration file loaded into the object.

Returns: Name of configuration file (string).

getDataCheckActive()
Return the Data Check Service enable/disable flag.

Returns: Data Check Service enabled/disabled (integer).

getDataCheckForceNotif()
Return the Force Data Check Notification Flag.

Returns: Force notification = 1 (integer/0|1).

getDataCheckMaxProcs()
Return the maximum number of parallel Data Check sub-processes.

Returns: Maximum number of sub-processes (integer).

getDataCheckMinCycle()
Return the Data Check Service Minimum Cycle Time.

Returns: Data Check Minimum Cycle Time (string).

getDataCheckNotifList()
Get reference to tuple with recipients of Data Check Notification Messages.

Returns: Tuple with recipients (tuple).

getDataCheckScan()
Return the Data Check Scan Flag.

Returns: Data Check Scan Flag (integer/0|1).

getDataMoverHostIds()

getDataMoverSuspenstionTime()
Return the Data Mover (Subscription) Thread Suspension Time.

Returns: Suspension time (string/ISO 8601).

getDbInterface()
Get the name of the NG/AMS DB Interface Plug-In in use.

Returns: DB Interface Plug-In (string).

getDbMaxPoolCons()
Max number of DB Pool Connections.

NOTICE: 7 connections was chosen as a default to allow for all the NGAS background services to have a long running db connection while allowing user requests to be serviced. Anything less than 7 with all the services enabled might cause user requests to block waiting for a db connection to be placed back in the pool from a long running service.

Returns: Max number of DB Pool Connections.

getDbParameters()
Return DB connection parameters.

Returns: DB connection parameters (string).

getDbSessionSql()
SQL commands to run whenever a connection is established

getDbSnapshot()
Return the DB Snapshot Feature on/off.

Returns: DB Snapshot Feature state (integer/0|1).

getDbUseFileIgnore()
Indicates whether to use "file_ignore" as the column name on the "ngas_files" table as opposed to "ignore". For historical reasons the same column has been referenced using two different names.

getDiskChangeNotifList()
Get reference to tuple with recipients of Disk Change Notification Events.

Returns: Tuple with recipients (tuple).

getDiskSpaceNotifList()
Get reference to tuple with recipients of Disk Space Notification Events.

Returns: Tuple with recipients (tuple).

getDiskSyncPlugIn()
Get name of the Disk Sync Plug-In.

Returns: Name of Disk Sync Plug-In (string).

getErrorNotifList()
Get reference to tuple with recipients of Error Notification Events.

Returns: Tuple with recipients (tuple).

getExtFromMimeType(*mimeType*)
Get the file extension corresponding to the given mime-type.

mimeType: Mime-type (string).

Returns: Extension corresponding to mime-type (string).

getFileStagingEnable()
Return if the file staging flag is set to Enabled / Disable

Returns: 1 = enabled, 0 = disabled (integer/0|1)

getFileStagingPlugIn()
Return the name of the FileStagingPlugIn, which takes file online if it is offline.

The plugin must contain these two functions:

def isFileOffline(filename) PAR: filename: string RETURN: 1 - yes, 0 - no, Exception - error

def stageFiles(filenameList) PAR: filenameList: List of file names (string) RETURN: the number of files staged. Exception, if any errors

getFreeSpaceDiskChangeMb()
Get the limit for the minimum free disk space before changing disk.

Returns: MB limit for changing disk (integer).

getIdleSuspension()
Return the NGAS Idle Suspension Flag.

Returns: Idle Suspension Flag (integer/0|1).

getIdleSuspensionTime()
Return the Idle Suspension Time.

Returns: Idle Suspension Time in seconds (integer).

getIpAddress()
Get socket port number.

Returns: Reference to object itself.

getJanitorPlugins()
Get the list of Janitor Plug-in names.

Returns: Janitor Service Suspension Time (string).

getJanitorSuspensionTime()
Get Janitor Service Suspension Time.

Returns: Janitor Service Suspension Time (string).

getLabelPrinterPlugIn()
Get name of Label Printer Plug-In.

Returns: Name of Printer Plug-In (string).

getLabelPrinterPlugInPars()
Get input parameters for Label Printer Plug-In.

Returns: Input parameters for Label Printer Plug-In (string).

getLocalLogFile()
Return the Local Log File.

Returns: Name of Local Log File (string).

getLocalLogLevel()
Return the Local Log Level.

Returns: Local Log Level (integer).

getLogRotateCache()
Return the size of the internal log rotation cache.

Returns: Size of internal log buffer (integer).

getLogRotateInt()
Return the Log Rotation Interval given as an ISO 8601 timestamp.

Returns: Log Rotation Interval as ISO 8601 format (string).

getMaxRetentionSize()
Get the Maximum Retention Size, which is the maximum number of Email Notification Messages, which is kept before sending these out.

Returns: Maximum retention buffer size (integer).

getMaxRetentionTime()
Return the Maximum Retention Time, which is the maximum time an Email Notification Message should be retained before it is send out.

Returns: Maximum Retention Time as ISO 8601 format (string).

getMaxSimReqs()
Get the maximum number of simultaneous requests.

Returns: Maximum number of simultaneous requests (integer).

getMimeTypeMappings()
Return list containing sub-lists, one for each mime-type/extension mapping. The format is: [[mime-type, ext], [mime-type, ext], . . .].

Returns: List with mime-type mappings ([[mt, ext], [mt, ext], . . .]).

getMinFreeSpaceWarningMb()
Get the limit for the minimum free space available, before a warning is issued to change the disk.

Returns: Minimum free space before issuing warning (integer).

getMinSpaceSysDirMb()
Get the minimum amount of free disk space required on the NG/AMS System Directories.

Returns: Minimum space (integer).

getMirroringActive()
Return the flag indicating if the Mirroring Service is activated.

Returns: Value of mirroring activated flag (integer).

getMirroringErrorRetryPeriod()
Return the period for retrying to mirroring failing requests.

Returns: Error retry timeout (integer).

getMirroringErrorRetryTimeOut()
Return the timeout for retrying to mirror a failing request.

Returns: Error retry timeout (integer).

getMirroringReportRecipients()
Return the report recipients list.

Returns: The list of report recipients.

getMirroringSrcList()
Get reference to list with Mirroring Source Objects.

Returns: List with Mirroring Source Objects ([ngamsMirroringSource, . . .]).

getMirroringSrcObj(*id*)
Find the Mirroring Source Object with the given ID.

id: ID associated to the Mirroring Source Object (string).

Returns: Reference to Mirroring Source Object in question (ngamsMirroringSource).

getMirroringSrcObjFromSrvList(*srvList*)
Return the Mirroring Source Object associated to the given Server List.

srvList: Server list, common separated list of

'<Node>:<Port,. . . ' (string).

Returns: Reference to Mirroring Source Object associated to the given server list (ngamsMirroring-Source).

getMirroringThreads()
Return the number of mirroring threads to use.

Returns: Number of mirroring threads (integer).

getNGASJobMANHost()

getNoDiskSpaceNotifList()
Get reference to tuple with recipients of No Free Disks Notification Events.

Returns: Tuple with recipients (tuple).

getNotifActive()
Return the Email Notification Active Flag.

Returns: Notification Active Flag (integer)

getNotifSmtpHost()
Return the SMTP Host for sending Notification e-mails.

Returns: SMTP Host (string).

getNotifSmtpPort()
Return the SMTP port for sending Notification e-mails.

Returns: SMTP Port (int).

getOfflinePlugIn()
Get name of Offline Plug-In.

Returns: Name of Offline Plug-In (string).

getOfflinePlugInPars()
Get input parameters for Offline Plug-In.

Returns: Input parameters for Offline Plug-In (string).

getOnlinePlugIn()
Get name of Online Plug-In.

Returns: Name of Online Plug-In (string).

getOnlinePlugInPars()
Get input parameters for Online Plug-In.

Returns: Input parameters for Online Plug-In (string).

getPathPrefix()
Return Path Prefix.

Returns: Path Prefix (string).

getPluginsPath()
Get the directory where plug-ins are placed.

getPortNo()
Get socket port number.

Returns: Reference to object itself.

getProcessingDirectory()
Get NG/AMS Processing Directory.

Returns: Processing directory (string).

getProxyCRC()
If the proxy archive server check CRC as well

By default, 0 (do not check CRC)

getProxyMode()
Get Proxy Mode Flag.

Returns: Proxy Mode Flag (integer).

getReplication()
Return File Replication on/off flag.

Returns: File Replication on/off flag (integer).

getRequestDbBackend()
Returns whether the server should keep a request database or not.

getRootDirectory()
Get NGAS Root Directory.

NOTE: THE NGAMS_PREFIX environment variable overrides the one in the Config-file.

Returns: NGAS Root Directory (string).

getSender()
Return the senders email address. This is important in cases where the smtp server is setup to allow emails only from known domains and the NGAS server sits on a private network.

Returns: email address for the 'from' field (string).

getSlotIdDefined(*slotId*)
Returns 1 if the Slot ID indicated is used in one of the Storage Sets defined, otherwise 0 is returned.

slotId: Slot ID (string).

Returns: 1 if Slot ID is defined, otherwise 0 (integer/0|1).

getSlotIds()
Return tuple with Slot IDs. The format is:

[<Main Slot ID 1>,[<Rep. Slot ID 1>,] <Main Slot ID 2>, . . .]

Returns: Tuple with Slot IDs (tuple).

getStorageSetFromId(*storageSetId*)
Return a Storage Set object from a given Storage Set ID.

storageSetId: Storage Set ID (string).

Returns: Instance of ngamsStorageSet or None (ngamsStorageSet | None).

getStorageSetFromSlotId(*slotId*)
Get a Storage Set object from a given Slot ID.

slotId: Slot ID (string).

Returns: Instance of ngamsStorageSet or None (ngamsStorageSet | None).

getStorageSetList()
Get reference to list with Storage Set objects.

Returns: List with storage set objects ([ngamsStorageSet, . . .]).

getStreamFromMimeType(*mimeType*)
Get an ngamsStream object from its mime-type.

mimeType: Mime-type for Stream (string).

Returns: Stream object or None (ngamsStream|None).

getStreamList()
Get list containing the Stream objects

Returns: List containing Stream objects ([ngamsStream, . . .]).

getSubscrEnable()
Return the Susbcription Enable/Disable Flag to switch on/off the subscription for data from data providers.

Returns: 1 = enabled, 0 = disabled (integer/0|1).

getSubscrSuspTime()
Return the Subscription Thread Suspension Time.

Returns: Suspension time (string/ISO 8601).

getSubscriptionsDic()
Get reference to list with Subscriptions Objects.

Returns: Subscriber List (list/ngamsSubscriber).

getSuspensionPlugIn()
Return the name of the Suspension Plug-In.

Returns: Name of plug-in (string).

getSuspensionPlugInPars()
Return the Suspension Plug-In parameters.

Returns: Plug-in parameters (string).

getSysLog()
Return the syslog on/off flag.

Returns: Syslog on/off flag (integer).

getSysLogAddress()
Return the address where syslog is listening for incoming messages. If no address is given, a platform-dependent default is used

Returns: Syslog address (string).

getSysLogPrefix()
Return the syslog prefix.

Returns: Syslog prefix (string).

getTimeOut()
Gets the timeout that applies to HTTP requests.

getVal(*parName*)
Return the value of a parameter.

parName: Name of the parameter in the 'Simplified XPath Syntax', e.g.:

NgamsCfg.Server[1].ArchiveName (string).

Returns: Value of parameter or None (<Value>|None).

getVolumeDirectory()
Return value of the Volume Directory attribute in the Server Element.

Returns: Value of VolumeDirectory (string).

getWakeUpCallTimeOut()
Return the Wake-Up Call Time-Out for waiting for an NGAS host being woken up to be up and running.

Returns: Time-out in seconds (integer).

getWakeUpPlugIn()
Return the name of the Wake-Up Plug-In.

Returns: Name of plug-in (string).

getWakeUpPlugInPars()
Return the parameters to the Wake-Up Plug-In.

Returns: Plug-in parameters (string).

getWakeUpServerHost()
Return the Wake-Up Server host name.

Returns: Name of Wake-up Server Host (string).

hasAuthUser(*user*)
Check if a user with the given ID is defined.

user: User name (string).

Returns: 1 = user defined (integer/0|1).

load(*filename*, *check=0*)
Load an NG/AMS Configuration File into the object.

filename: Name of configuration file (string).

check: If set to 1 the semantics is checked after loading (integer/0|1).

Returns: Reference to object itself.

loadFromDb(*name*, *dbObj=None*)
Load a configuration from the DB via the given ID.

name: Name of the configuration in the DB (string).

dbObj: DB connection object (ngamsDb).

Returns: Reference to object itself.

save(*targetFilename*, *hideCritInfo=1*)
Save the configuration in the object into a XML document with the given name.

targetFilename: Name of target file (string).

hideCritInfo: If set to 1 passwords and other 'confidential' information appearing in the log file, will be hidden (integer/0|1).

Returns: Reference to object itself.

setDbObj(*dbObj*)
Set the DB connection object of this instance.

dbObj: DB connection object (ngamsDb).

Returns: Reference to object itself.

setDiskChangeNotifList(*subscriberList*)
Set the list of Disk Change Notification Subscribers.

subscriberList: List of subscribers of the Disk Change Notification (list).

Returns: Reference to object itself.

setDiskSpaceNotifList(*subscriberList*)
Set the list of Disk Space Notification Subscribers.

subscriberList: List of subscribers of the Disk Change Notification (list).

Returns: Reference to object itself.

storeVal(*parName*, *value*, *dbCfgGroupId=None*)
Set the value of the given parameter.

parName: Name of parameter e.g.:

NgamsCfg.Server[1].RootDirectory (string).

value: Value of the parameter (string).

dbCfgGroupId: DB configuration group ID (string|None).

Returns: Reference to object itself.

writeToDb (*dbObj=None*)
Write the configuration loaded into the DB.

dbObj: DB connection object (ngamsDb).

Returns: Reference to object itself.

11.3 Server classes

class ngamsServer.ngamsServer.**ngamsHttpRequestHandler** (*request*, *client_address*, *server*)
Class used to handle an HTTP request. The various `send_*` methods should make it easy for the rest of the code to send different kind of replies to users

proxy_request (*host_id*, *host*, *port*, *timeout=300*)
Proxy the current request to `host:port`

redirect (*host*, *port*)
Redirects the client to the requested path, but on `host:port`

send_data (*data*, *mime_type*, *code=200*, *message=None*, *fname=None*, *hdrs={}*)
Sends back `data`, which is of type `mime_type`. If `fname` is given then the data is sent as an attachment.

send_file (*f*, *mime_type*, *start_byte=0*, *fname=None*, *hdrs={}*)
Sends file `f` of type `mime_type` to the client. Optionally a different starting byte to start the transmission from, and a different name for the file to present the data to the user can be given.

send_file_headers (*fname*, *mime_type*, *size*, *start_byte=0*, *hdrs={}*)
Sends the headers advertising file `fname`, but without its data. Headers set by this method take precedence over values given by the caller via the `hdrs` optional argument

send_ingest_status (*msg*, *disk_info*)
Reply to the client with a standard ingest status XML document

send_response (*code*, *message=None*, *hdrs={}*)
Sends the initial status line plus headers to the client, can't be called twice

send_status (*message*, *status='SUCCESS'*, *code=None*, *http_message=None*, *hdrs={}*)
Creates and sends an NGAS status XML document back to the client

class ngamsServer.ngamsServer.**ngamsServer** (*cfg_fname*, *_cert=None*)
Class providing the functionality of the NG/AMS Server.

cfg = None
The underlying configuration object of type *ngamsConfig*

db = None
A reference to the underlying database of type *ngamsDb*

Index

H

I

L

M

N

P

Q

R

S

T

U

W