filesystem backup Documentation

Release 0.1.2

Miguel Garcia

Nov 09, 2017

CONTENTS

CHAPTER

ONE

OVERVIEW

A command-line script in Python is provided, to manage backups for large filesystems in multiple external disks.

It is intended as a minimalist system, to get the job done but with no GUI or other niceties. At least not yet! I just wanted to sleep well at night.

1.1 Motivation

1.1.1 The Problem

For more than a decade I had being gathering content and storing it in external drives. For backup purposes I used to buy them in pairs, so that one would work as the other's mirror. Of course the solution was far from ideal, there were tv-series, movies, and documentaries in most disks, sparsed pretty much randomly, and when the number of disks reached 15 (plus backups) even finding content was a pain. I had simple text files with the file contents of each disk, which needed to be updated, etc.

1.1.2 An Improvement

A friend talked to me about a NAS he had recently acquired. After little consideration I realized I had been needing one myself for a long time, just did not know such a thing existed. Taking into account the size of the files I already had, plus reasonable mid-term foreseable needs, I bought a 6-slots NAS and put 8GB disks in it (5 of them currently).

Now the content was neatly organised, easy to find and maintain.

I was using RAID5, which is nice, but in several forums I found the clear warning that RAID does not work as backup, so I started worring. I had the need of a real backup, and a bunch of external drives which content was already in the NAS. Obviously they might be used to backup content, but I could not bring myself to even try to micro-manage it. It would be particularly hard because some folders are way bigger that the external drives, so they would have to be split manually.

1.2 Backup System Overview

The idea behind the implementation of **fsbackup** is pretty simple, and everything gets done by the `fsbck` command. Given a list of one or more paths that we want backed-up, the backup system works in three stages.

1.2.1 Stage 1

A command (intended to be scheduled nightly) keeps a collection in a mongoDB database updated with the absolute path, size, last modification timestamp and a hash function (currently SHA-256) of each file in that list of paths. They are interpreted as file-trees, so all the content buried in those paths is included. It can be done with something like:

```
fsbck.py refreshHashes -db=conn_multimedia.json
```

Only new files, or files with a more recent modification timestamp than the one in the database have their hash function recalculated (since it is really time-consuming). As you might have guessed, the `db` argument refers to a json file with information regarding the location of the filesystem, as well as mongoDB collections where the information is stored.

1.2.2 Stage 2

External hard disks work as backup volumes, containing files renamed with their hash function. The folder structure in the original filesystem is not replicated, all files are at root level. Except that, using git-style, they are divided in folders according to the first letters in the hash, to avoid having thousands of files in the same directory.

In order to update the backup, we can mount a disk that works as backup volume (say, it is in G:), and run:

```
fsbck.py processDrive -db=conn_multimedia.json --drive=G
```

This action:

- Removes from the volume files that are not necessary anymore.
- Copies new files that were not backed-up yet.
- Provides a backup status report, with:
  - the number of files/size pending backup (if there was not enough room in that volume).
  - a summary of the number of files/size in each volume.
  - a file per volume is created with the detailed absolute paths of each file backed-up in it.

For this to work properly, another collection in the database stores the hashes backed in each volume.

1.2.3 Stage 3

If/when the time comes information needs to be retrieved from the volumes, the script handles that as well. For instance, the command:

```
fsbck.py checkout -db=conn_multimedia.json --drive=G --sourcepath=//Zeycus/multimedia/
↪movies --destpath=F:\chekouts\movies
```

recovers the relevant information in the actual (G:) volume for a particular folder. In a worst-case scenario, to recover all the files you'd have to do this for every volume.

1.3 So, how do I start?

In a nutshell:

1. Get a mongoDB server connection and create a database there. It could be local, mongoDB hosting (like mlab , just to name one), etc.

2. Build a JSON config_file for the filesystem you want backed-up. For instance:

```
{
  "connstr": "mongodb://myuser:mypwd@ds21135.mlab.com:34562/fsbackup_tvs761_main",
  "paths": [
    "\\\\ZEYCUS-TVS671\\Multimedia",
    "\\\\ZEYCUS-TVS671\\Resources"
  ],
  "reportpref": "F:\\Dropbox\\fsbackup\\reports\\main_"
}
```

where `connstr` is the conection string to your mongoDB database (in this case, `fsbackup_tvs761_main`). More details in the documentation. Make sure the path in `reportpref` actually exists, reporting files are created there. In this case, `F:\\Dropbox\\fsbackup\\reports`.

3. Create the actual collections in the database with:

```
fsbck.py createDatabase -db=<config_file> --force
```

4. Gather the current filesystem information with:

```
fsbck.py refreshHashes -db=<config_file>
```

The first time hashes are calculated for all files, so this may take **long**.

5. Connect a formated external drive. Assuming it gets mounted in `driveLetter`, execute:

```
fsbck.py processDrive -db=<config_file> --drive=<driveLetter>
```

This fills the volume with backup data. When finished, a message will clarify whether more volumes are needed to go on with the backup.

1.4 Collaboration

You may wish to improve or add features, in that case you are more than welcome, feel free to contact me at zeycus@gmail.com.

CHAPTER

TWO

DATABASE STRUCTURE

Information regarding the filesystem to be backed-up, and the current content of volumes, is stored in a mongoDB database.

2.1 Filesystem

The collection that stores the information about the files currently in the filesystem is (uninspiredly!) named `files`. The entries/documents in it have the form:

```
{
'_id': ObjectId("59e0a71c2afc32cfc4e7fa48"),
'filename': r"\\ZEYCUS-TVS671\Multimedia\video\animePlex\Shin Chan\Season 01\Shin␣
↪Chan - S01E613.mp4",
'hash': "4a7facfe42e8ff8812f9cab058bf79981974d9e2e300d56217d675ec5987cf05",
'timestamp': 1197773340,
'size': 68097104
}
```

where:

- The `filename` field is the absolute path of the file.
- The `hash` field is the SHA-256 hash of the file.
- `timestamp` is the file's last-modified timestamp.
- `size` is the size of the file in bytes, obtained with `os.stat(fn).st_mtime`.

The fields used for look-up are `filename` and `hash`, so the collection should have an index on each of them. The one on `filename` should have `unique=True`, to ensure no filename is added twice[2] .

The class that manages this collection is *`FileDB`*.

2.2 Volumes

On the other hand, the present state of backup volumes is stored in the collection `volumes`, with entries like

```
{
'_id': ObjectId("59e484603e12972bd4209fbe"),
'volume': "3EC0BECC",
```

[2] This is not true for `hash`, because we need to be able to backup systems that contain the same file in different locations. I was surprised to find that I had about a 5% of file redundancy in number of files, it turned out that some tiny files were necessary in many locations.

```
'hash': "0017eef276f4247807fa3f4e565b8c925a2db0f8bfbb020248ad6c3df6a6ea77",
'size': 97092
}
```

where:

- The `volume` is the volume SerialNumber.
- The `hash` field is the SHA-256 hash of the file.
- `size` is the size of the file in bytes.

This entry is saying that volume 3EC0BECC contains a file with the given hash, and filesize 97,092 bytes.

There should be a a unique index on field `hash`[1] .

The methods that add/remove files from a volume (see class *HashVolume*) also update this collection, so that it remains up-to-date.

[1] In fact, this enforces that only one volume may contain a file with a specific hash. If the backup methods are working correctly this should be the case. If the same file is found in different folders in the filesystem, or with different names, no space is wasted and just one copy will be present in backup volumes.

CHAPTER

THREE

VOLUME CONTENT

Volumes contain backups of the files in the filesystem: files with the same content. However, they are renamed with the hash of the content. This means that no information regarding the filename in the real filesystem, or the path where it is located, can be found in the volumes (that information is stored in the `files` collection in the database). All the files in the volume are placed in the root of the filesystem, but classified with their first 3 letters to avoid the problems associated with having too many files in the same folder. An actual volume looks like this:

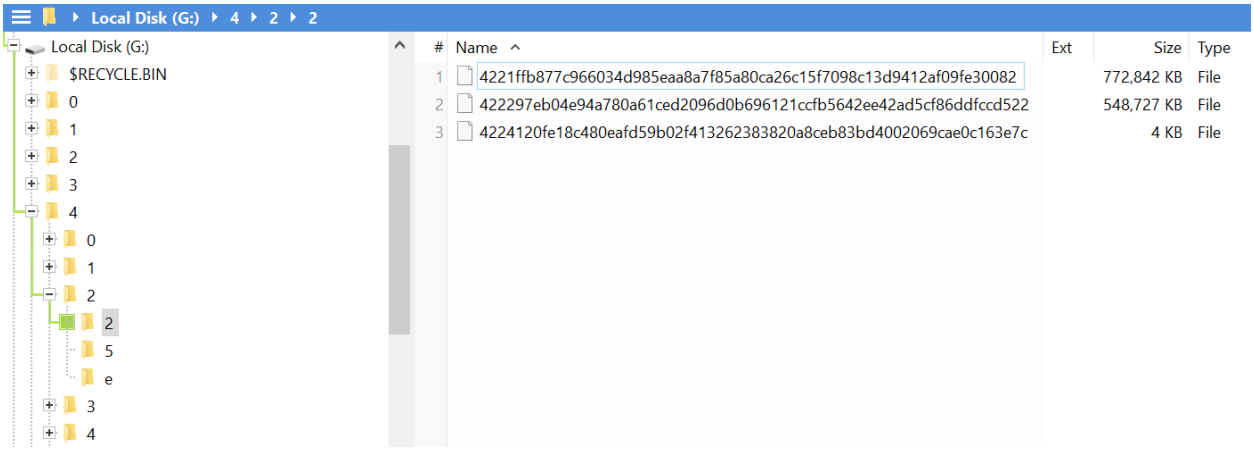


Fig. 3.1: Content of a backup volume.

CHAPTER

FOUR

FILESYSTEM CONFIG FILES

The information about filesystems that we want backed-up is gathered in JSON files, one per filesystem. For instance:

```
{
  "connstr": "mongodb://myuser:mypwd@ds21135.mlab.com:34562/fsbackup_tvs761_main",
  "paths": [
    "\\\\ZEYCUS-TVS671\\Multimedia",
    "\\\\ZEYCUS-TVS671\\Resources"
  ],
  "reportpref": "F:\\Dropbox\\fsbackup\\reports\\main_"
}
```

The information is as follows:

connstr The connection string to the mongoDB database.

paths The list of paths in the filesystem that we want backed-up. So far I've been using absolute paths myself, but I think that paths relative to the location of the config file work as well. But I have not tested it that heavily.

reportpref Prefix for reports. All files created by the `backupStatus` command are created with that prefix.

CHAPTER

FIVE

DETAILED COMMAND USAGE

Everything works via the `fsbck` command. If the installation is correct, it should be available no matter what the active directory is. In this section, the basic usage is shown, but the full detail and optional parameters can be found in *commands* module documentation.

5.1 Database Creation

It is achieved with:

```
fsbck.py createDatabase -db=<config_file>
```

If the database containing the two necessary collections `files` and `volumes` do not exist, they are created. Otherwise the execution fails. If you want it rebuild, add the `--force` flag.

5.2 Create reports for backup status

With:

```
fsbck.py backupStatus -db=<config_file>
```

several text files are created (with different level of detail) regarding the status of the backup:

- size and number of files in each backup volume
- size and number of files not yet backed-up
- size and number of files in the volumes than are no longer necessary
- explicit list of files in each volume

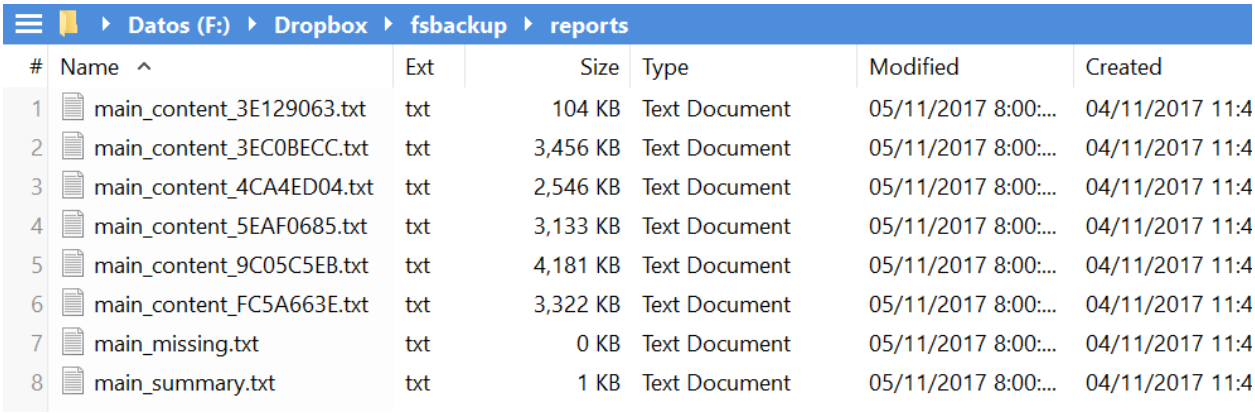
An example of the files created:

Contrary to what it might seem, this operation is fairly quick.

5.3 Database `files` update

This command updates the database information to match the current state of the filesystem. If files are modified their hash is recalculated, if files were removed their entries are eliminated from the database, and new files require new entries.

This is achieved with:



Datos (F:) ▸ Dropbox ▸ fsbackup ▸ reports

#	Name	Ext	Size	Type	Modified	Created
1	main_content_3E129063.txt	txt	104 KB	Text Document	05/11/2017 8:00:...	04/11/2017 11:4
2	main_content_3EC0BECC.txt	txt	3,456 KB	Text Document	05/11/2017 8:00:...	04/11/2017 11:4
3	main_content_4CA4ED04.txt	txt	2,546 KB	Text Document	05/11/2017 8:00:...	04/11/2017 11:4
4	main_content_5EAF0685.txt	txt	3,133 KB	Text Document	05/11/2017 8:00:...	04/11/2017 11:4
5	main_content_9C05C5EB.txt	txt	4,181 KB	Text Document	05/11/2017 8:00:...	04/11/2017 11:4
6	main_content_FC5A663E.txt	txt	3,322 KB	Text Document	05/11/2017 8:00:...	04/11/2017 11:4
7	main_missing.txt	txt	0 KB	Text Document	05/11/2017 8:00:...	04/11/2017 11:4
8	main_summary.txt	txt	1 KB	Text Document	05/11/2017 8:00:...	04/11/2017 11:4

Fig. 5.1: Files created by `backupStatus`.

```
fsbck.py refreshHashes -db=<config_file>
```

For large filesystems the calculation of hashes is time-consuming. The first calculation for my NAS took nearly a whole week, so I prefer to perform this process dayly, in scheduled task at night, and a `backupStatus` immediatly after it.

5.4 Volume update

This is the way content gets actually backed-up. Suppose you have a volume with available space on it, or if you are going to create a new volume, just a formated external drive. When connected, it is assigned a drive letter, say J:[3] . Then to perform the update use:

```
fsbck.py updateVolume -db=<config_file> --drive=J
```

New files are added to the volume, until it is full or all of them are processed, a text message tells which of the two.

Warning: Be sure that the `files` information is updated (via command `refreshHashes`) before invoking a volume update. Otherwise, when the script tries to copy a file that the database is mentioning, it might not be physically there anymore, and thus exceptions would arise. There is no problem, however, if the only difference is that new files were created.

5.5 Volume clensing

When you remove files from your backed-up filesystem, copies of them remain in backup volumes. There is no harm in it, just the waste of space. As time passes, the wasted space in volumes could amount to something. With:

```
fsbck.py cleanVolume -db=<config_file> --drive=<driveLetter>
```

the files in the volume than are not shown as necessary by the database are removed.

[3] I realize this is terribly Windows-oriented. For linux systems it would be rather similar, if/when Linux support is provided this documentation should be improved.

5.6 Volume processing

In the first days, when I wanted to update a volume I found myself always performing:

1. volume clensing
2. volume update
3. backuptatus reports regeneration

I created a batch, but after a while I decided an additional command was in order to do it all: `processDrive`. With:

```
fsbck.py processDrive -db=<config_file> --drive=<driveLetter>
```

those three tasks are performed. This keeps the volumes clean of old files, the system fully updated and the status reports reflecting the current backup status.

In a day-to-day basis this is almost the only command you need (if the `refreshHashes` is taken care of by an scheduled task). Of course, you could manually run `refreshHashes` before processing a drive, just to make sure the database is up-to-date.

5.7 Information recovery from volumes

All the burden of keeping the filesystem updated has a single purpose: to be able to recover content from the backup volumes when necessary. This operation may be infrequent, but it is arguably the most important. It is currently performed with the `checkout` command:

```
fsbck.py updateVolume -db=<config_file> --drive=<driveLetter> --sourcepath=\\ZEYCUS-
↪TVS671\Multimedia\video\seriesPlex\Monk --destpath=F:\temp\Monk
```

This process finds all the files in the volume that are a backup of a file in the given `sourcepath` (or in a subfolder), and copies them recreating the folder structure within the path `destpath`.

Needless to say, to recover the whole folder content you need to process all the volumes containing at least one relevant file. It is possible to see which volumes are involved by searching the backup-status report files. Or just process them all, it takes very little time if no content is necessary.

5.8 Recalculation of Volume Information

The operations that add and remove files from the volume in same time update the database. So, theoretically, the database is always up-to-date. I have not found a single case in which this was not the case, but nevertheless implemented:

```
fsbck.py extractVolumeInfo -db=<config_file> --drive=<driveLetter>
```

What this does is to remove from the `volumes` collection all the entries associated to the present volume, then it is traversed and an entry is created for each actual file found.

5.9 Volume Integrity Check

In case we want to make sure that a backup volume is OK, we can perform an integrity check with:

```
fsbck.py integrityCheck -db=<config_file> --drive=<driveLetter>
```

This is a time consuming operation that actually compares each file of the volume with its counterpart in the actual filesystem (if it was not deleted). For 3TB disks it is taking me over a day.

Warning: This is supposed to be done after a `refreshHashes`. Otherwise the information in the DDBB might not reflect the actual state of the filesystem.

CHAPTER

SIX

OBSERVATIONS

6.1 Volume identification

Volumes are not numbered, instead they are identified by a unique identifier. For now it is their filesystem volume serial number. This means you never need to process the volumes in any order, nor when you update them.

For instance, suppose you remove some huge files from your filesystem (who would want to see **THAT** tv-show again!?). As a consequence the backupstatus report shows that a volume contains now 300GB of removable files. You could choose this volume for your next `processDrive`: useless content will be dropped, making room and using it for fresh file backups.

6.2 Volume content

Files are not backed-up in any order. The system just aims to have each file backed-up in a (single) volume. This means content is more or less randomly divided among volumes.

CHAPTER

SEVEN

PLEASE, BE AWARE!

Warning: To be able to use mongoDB, we must have a connection to a mongoDB server. It could be our own machine, a hosting service, etc.

If you are new to mongoDB, several tutorials are available, this is one of them. There are also many mongoDB-hosting services that provide free sandboxes with a decent size, no need to spend a dime just to experiment.

If you have mongoDB installed, to serve it locally (in Windows) just run:

```
mongod.exe --dbpath=<database_path>
```

7.1 Regarding tests

Warning: To be able to run tests, we need a mongoDB server to connect to (I know of no better way. If there is, please let me know). The tests are written assuming that a local server is running.

Then, a client is created that connects to it, creates testing databases/collections, fills them, accesses information stored, and wipes them all in the end.

7.2 Information safety

The mongoDBs created are essential to be able to recover contents from the backup.

Warning: If they were lost, in the volumes you won't see proper filenames or extensions. Therefore although the content is indeed there, finding what you need would be, at the very least, awefully painful, if not utterly infeasable.

For that reason it is reasonable to make sure the mongoDB databases are safe, and backed-up as frequently and redundantly as possible. I am using mongoDB hosting, and keep a local copy as well. Even periodically storing a copy with its timestamp might be interesting, if you want to play it safe.

7.3 License

This software is released under MIT license, with no warranty implied or otherwise. That said, on the sunny side a unittest is included that performs the complete backup cycle and makes sure that the checkout is identical to the original filesystem. And `integrityCheck` command is available, which actually compares each backed-up file with its counterpart in the filesystem.

CHAPTER

EIGHT

TODO

1. Currently only Windows is supported.

There are several aspects in this process than are very OS-dependent. For instance:

- Copying files
- The systax for absolute paths
- Extraction of volume id

So far I had only Windows in mind, and even had to implement at least an ugly hack (to handle +260 chars absolute paths, which surprisingly causes problems in Windows). I wish `fsbackup` worked for Linux as well, at least, that is the very first thing I'd like to do.

2. It seems not all filesystems have volume serialNumber. For that reason it seems that using disk serial numbers instead might be an improvement. I chose volume serialnumbers because it was easy to extract, while the drive serial number containing a volume seemed hard to get (Googled for a while, found no easy path).

3. For now, the only way to retrieve information from the volumes is the `checkout` command, which rebuilds a folder/subfolder recursively. However, it would be easy to add filters to recover only files that match a given regular expression, or filter them for timestamp or other features, etc.

   Truth be told, this kind of operation is what I implemented for the case in which something *goes wrong*: content was deleted unwantingly, or the disk just crashed. Fortunately those events happen pretty rarely, so little effort was dedicated to the recovery of information.

CHAPTER

NINE

RELEASE HISTORY

9.1 0.1.2 (2017-11-09)

Improvements

- New safe file copy: deletes target file if the writting process failed.
- New "How do I start?" section in README.
- New "Release History".
- Replace deprecated pymongo collections `remove` with `delete_many`.

Bugfixes

- Fixed typo in setup `tests_require` argument.

9.2 0.1.1 (2017-11-05)

- First version made available

CHAPTER

TEN

CODE DOCUMENTATION

10.1 Main Commands Module

fsbackup.commands.**backupStatus**(*fDB*, *volDB*, *reportPref*)
Generates the status report.

Several files are created:

- summary.txt: global summary.
- missing.txt: list of files not yet backed-up.
- content_<vol>.txt: the list of files backed-up in each volume.

Parameters

- **fDB** (FileDB) – the information regarding files
- **volDB** (*permanent-dict class*) – the informating regarading volumes
- **reportPref** (*str*) – prefix that tells where to create reporting

fsbackup.commands.**extractVolumeInfo**(*hashVol*)
Regenerates the DDBB information regarding the files contained in the present volume.

Parameters **hashVol** (HashVolume) – the information regarding volumes

fsbackup.commands.**cleanVolume**(*fDB*, *hashVol*)
Removes files from the volume that are not necessary anymore.

Returns the number of deleted files.

Parameters

- **fDB** (FileDB) – the information regarding files
- **hashVol** (HashVolume) – the information regarding volumes

Return type int

fsbackup.commands.**updateVolume**(*fDB*, *hashVol*)
Deletes useless files in the volume, and copies new files that need to be backed-up.

Parameters

- **fDB** (FileDB) – the information regarding files
- **hashVol** (HashVolume) – the information regarding volumes

fsbackup.commands.**refreshFileInfo**(*fDB*, *forceRecalc*)
Updates the filename collection in the database, reflecting changes in the filesystem.

Parameters

- **fDB** (FileDB) – the information regarding files
- **forceRecalc** (*bool*) – flag that tells if hashes & timestamps should be recalculated from the file always. If False (the default), recalculation happens always when the timestamp of the file is more recent than that in the database, or for new files. If True, we recalculate for every file.

fsbackup.commands.**createDatabase**(*database*, *forceFlag*, *logger*)

Creates database collections from scratch.

Parameters

- **fDB** (FileDB) – the information regarding files
- **forceFlag** (*bool*) – tells whether to remove info, if collections already exist

fsbackup.commands.**integrityCheck**(*fDB*, *hashVol*)

Performs an integrity check for the volume.

Parameters

- **fDB** (FileDB) – the information regarding files
- **hashVol** (HashVolume) – the information regarding volumes

10.2 Auxiliary Modules

10.2.1 Module miscTools

fsbackup.miscTools.**buildVolumeInfoList**(*container*)

Returns, for each volume, the association {file-hash: file-size}.

Parameters **container** (*MongoAsDict*) – a MongoAsDict with the volume information

Return type list of pairs (volId, {sha:size})

10.2.2 Module fileTools

fsbackup.fileTools.**sizeof_fmt**(*num*, *suffix='B'*)

Returns a human-readable string for a file size.

Parameters

- **num** (*int*) – size of the file, in units
- **suffix** (*str*) – the unit. Use 'B' for bytes, the default.

Return type str

Stolen from:

http://stackoverflow.com/questions/1094841/reusable-library-to-get-human-readable-version-of-file-size

fsbackup.fileTools.**abspath2longabspath**(*abspath*)

Returns an absolute filepath than works for longer than 260 chars in Windows.

In Windows there is seems to be no support for paths longer than 260 chrs. Files that exist are not found, cannot be open, etc. However, using this trick I seem to be able to access them.

Post with the trick description:

https://msdn.microsoft.com/en-us/library/aa365247.aspx#maxpath

10.2.3 Module `diskTools`

`fsbackup.diskTools.`**`genDrivesInfo`**`()`
Generator for drives information.

`fsbackup.diskTools.`**`genVolumesInfo`**`()`
Generator for volumes information.

`fsbackup.diskTools.`**`getVolumeInfo`**`(`*driveLetter*`)`
Returns volume info for the given driveLetter.

Parameters **`driveLetter`** (*`str`*) – the drive letter, for instance 'C'

Return type dict

`fsbackup.diskTools.`**`getAvailableLetter`**`()`
Returns the first drive letter available.

10.3 Class `HashVolume`

class `fsbackup.hashVolume.`**`HashVolume`**`(`*logger*, *locationPath*, *container*, *volId=None*`)`
Class that handles a backup volume.

`allVolumesHashes``()`
Returns the set of all hashes in any volume, according to the DDBB.

Return type set

`augmentWithFiles``(`*fDB*`)`
Include in the volume backup for the files that need it.

It is done until all files are backed-up, on until the volume is full.

Parameters **`fDB`** (`FileDB`) – filesystem information in DDBB.

Return type

a pair (isFinished, hashList)

- isFinished tells whether the backup is complete. It is False if there are still files that are not backed-up in any volume.
- hashList is the list of hashes of the created files.

Note: The strategy to choose which file to backup next is the following, but there are no strong reasons for this, it should be changed if another is found better.

- While there is plenty of room in the volume (threshold currently set to 20GB) and there is room for the biggest file that requires backup, files are chosen randomly. The reason is that usually there are folders with huge files, others with only tiny files. If files were processed by their folder order, a volume could end up with millions of small files, while another could contain just hundreds of heavy files. Not that it would be a problem in principle, but I thought it might be better to balance volumes, and a simple strategy is the random choice.
- When the previous condition fails, choose the biggest file that fits, until none does.

---

checkout(*fDB*, *sourcePath*, *destPath*)
Rebuilds the filesystem, or a subfolder, from the backup content.

Returns a list of the filenames (in the original filesystem) that were restored.

Parameters

- **fDB** (FileDB) – filesystem information in DDBB.
- **sourcePath** (*str*) – path in the filesystem that you want restored
- **destPath** (*str*) – location where you want the files created

Return type list of str

cleanOldHashes(*totalHashesNeeded*)
Removes files that are no longer necessary.

Returns the number of files removed.

Parameters **totalHashesNeeded** (*set*) – hashes of files that need to be backed-up.

Return type int

fnForHash(*sha*)
Returns the absolute path of the file for a given hash.

The first three letters in the hash are used to create a 3-levels folder system, for instance hash `4c07766937a4d241fafd3104426766f07c3ce9de7e577a76ad61eba512433cea` corresponds to file

```
self.locationPath/4/c/0/4c07766937a4d241fafd3104426766f07c3ce9de7e577a76ad61eba5
```

Parameters **sha** (*str*) – any valid SHA

Return type str

getAvailableSpace()
Returns the available free space in the volume drive, in bytes.

Return type int

recalculateContainer()
Rebuilds the DDBB volume information, traversing the files in the volume.

Note: This is something ordinarily you don't need to do, because the DDBB is kept synchronized with the files in the volume. This method is to be used in case for some reason the synchronization was broken.

---

remove(*sha*)
Deletes the file with a given hash.

Parameters **sha** (*str*) – the given hash

retrieveFilename(*sha*, *filename*)
Extracts a file from the volume, given its hash.

Parameters

- **sha** (*str*) – the given hash
- **filename** (*str*) – the filename of the file to be created

storeFilename (*filename*, *size*, *sha=None*)
Creates a file in the volume.

The filename in the volume is the sha, not the original filename.

Parameters

- **filename** (*str*) – location of the original file
- **size** (*int*) – size in bytes of the original file
- **sha** – the hash for the file. If not provided, it is calculated now

traverseFiles ()
Iterator over pairs (hash, size) for the present volume, checking which actual files are stored in it.

10.4 Class FileDB

class fsbackup.fileDB.**FileDB** (*logger*, *fsPaths*, *container*)
Class that handles the DDBB filesystem information.

Specifically, which files need to be backed-up, their location, size and hash.

checkout (*vol*, *sourcePath*, *destPath*)
Rebuilds the filesystem, or a subfolder, from the backup content.

We just invoke the chekout method of the volume.

Parameters

- **vol** (HashVolume) – the volume from which information is to be restored.
- **sourcePath** (*str*) – path in the filesystem that you want restored
- **destPath** (*str*) – location where you want the files created

Return type list of str

hashesSet ()
Returns the set of hashes in the DDBB.

Return type set

reportStatusToFile (*volHashesInfo*, *fnBase*)
Creates backup-status report files.

Parameters

- **volHashesInfo** (*dict {vol: {hash: size}}*) – for each volume, associates the hash of each file with its size.
- **fnBase** (*str*) – prefix of the report files to be created

update (*forceRecalc=False*)
Updates the DDBB info traversing the actual filesystem.

After execution, the DDBB reflects exactly the files currently in the filesystem, with their correct hash and size.

Parameters **forceRecalc** (*bool*) – flag that tells if hashes & timestamps should be recalculated from the file always. If False (the default), recalculation happens only when the timestamp of the file is more recent than that in the database, or for new files. If True, recalculation takes place for every file.

volumeIntegrityCheck(*vol*)
Performs a volume integrity check.

For each file that according to the DDBB is in this volume, a full comparison is performed between the file in the filesystem and the file in the backup volume. Of course, only when the file exists yet in the filesystem.

A final report with errors is generated, a list of errors returned.

Parameters **vol** (HashVolume) – the volume from which information is to be restored.

Return type list of str

10.5 Class MountPathInDrive

class fsbackup.mountPathInDrive.**MountPathInDrive**(*path*, *driveLetter*)
Simple context manager for temporaly mounting a path in a Windows drive.

Usage example:

```
with MountPathInDrive(path=r"F:\sources", driveLetter='J'):
    print(os.listdir("J:"))
```

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX