HashFS Documentation

Release 0.7.1

Derrick Gilland

Oct 14, 2018

Contents

HashFS is a content-addressable file management system. What does that mean? Simply, that HashFS manages a directory where files are saved based on the file's hash.

Typical use cases for this kind of system are ones where:

- Files are written once and never change (e.g. image storage).
- It's desirable to have no duplicate files (e.g. user uploads).
- File metadata is stored elsewhere (e.g. in a database).

CHAPTER 1

Features

- Files are stored once and never duplicated.
- Uses an efficient folder structure optimized for a large number of files. File paths are based on the content hash and are nested based on the first `n` number of characters.
- Can save files from local file paths or readable objects (open file handlers, IO buffers, etc).
- Able to repair the root folder by reindexing all files. Useful if the hashing algorithm or folder structure options change or to initialize existing files.
- Supports any hashing algorithm available via `hashlib.new`.
- Python 2.7+/3.3+ compatible.

CHAPTER 2

Links

- Project: https://github.com/dgilland/hashfs
- Documentation: http://hashfs.readthedocs.org
- PyPI: https://pypi.python.org/pypi/hashfs/
- TravisCI: https://travis-ci.org/dgilland/hashfs

CHAPTER 3

Quickstart

Install using pip:

```
pip install hashfs
```

3.1 Initialization

```
from hashfs import HashFS
```

Designate a root folder for `HashFS`. If the folder doesn't already exist, it will be created.

```
# Set the `depth` to the number of subfolders the file's hash should be split when␣
↪saving.
# Set the `width` to the desired width of each subfolder.
fs = HashFS('temp_hashfs', depth=4, width=1, algorithm='sha256')

# With depth=4 and width=1, files will be saved in the following pattern:
# temp_hashfs/a/b/c/d/efghijklmnopqrstuvwxyz

# With depth=3 and width=2, files will be saved in the following pattern:
# temp_hashfs/ab/cd/ef/ghijklmnopqrstuvwxyz
```

NOTE: The `algorithm` value should be a valid string argument to `hashlib.new()`.

CHAPTER 4

Basic Usage

`HashFS` supports basic file storage, retrieval, and removal as well as some more advanced features like file repair.

4.1 Storing Content

Add content to the folder using either readable objects (e.g. `StringIO`) or file paths (e.g. `'a/path/to/some/file'`).

```
from io import StringIO

some_content = StringIO('some content')

address = fs.put(some_content)

# Or if you'd like to save the file with an extension...
address = fs.put(some_content, '.txt')

# The id of the file (i.e. the hexdigest of its contents).
address.id

# The absolute path where the file was saved.
address.abspath

# The path relative to fs.root.
address.relpath

# Whether the file previously existed.
address.is_duplicate
```

4.2 Retrieving File Address

Get a file's `HashAddress` by address ID or path. This address would be identical to the address returned by `put()`.

```
assert fs.get(address.id) == address
assert fs.get(address.relpath) == address
assert fs.get(address.abspath) == address
assert fs.get('invalid') is None
```

4.3 Retrieving Content

Get a `BufferedReader` handler for an existing file by address ID or path.

```
fileio = fs.open(address.id)

# Or using the full path...
fileio = fs.open(address.abspath)

# Or using a path relative to fs.root
fileio = fs.open(address.relpath)
```

NOTE: When getting a file that was saved with an extension, it's not necessary to supply the extension. Extensions are ignored when looking for a file based on the ID or path.

4.4 Removing Content

Delete a file by address ID or path.

```
fs.delete(address.id)
fs.delete(address.abspath)
fs.delete(address.relpath)
```

NOTE: When a file is deleted, any parent directories above the file will also be deleted if they are empty directories.

CHAPTER 5

Advanced Usage

Below are some of the more advanced features of `HashFS`.

5.1 Repairing Files

The `HashFS` files may not always be in sync with it's `depth`, `width`, or `algorithm` settings (e.g. if `HashFS` takes ownership of a directory that wasn't previously stored using content hashes or if the `HashFS` settings change). These files can be easily reindexed using `repair()`.

```
repaired = fs.repair()

# Or if you want to drop file extensions...
repaired = fs.repair(extensions=False)
```

WARNING: It's recommended that a backup of the directory be made before repairing just in case something goes wrong.

5.2 Walking Corrupted Files

Instead of actually repairing the files, you can iterate over them for custom processing.

```
for corrupted_path, expected_address in fs.corrupted():
    # do something
```

WARNING: `HashFS.corrupted()` is a generator so be aware that modifying the file system while iterating could have unexpected results.

5.3 Walking All Files

Iterate over files.

```
for file in fs.files():
    # do something

# Or using the class' iter method...
for file in fs:
    # do something
```

Iterate over folders that contain files (i.e. ignore the nested subfolders that only contain folders).

```
for folder in fs.folders():
    # do something
```

5.4 Computing Size

Compute the size in bytes of all files in the `root` directory.

```
total_bytes = fs.size()
```

Count the total number of files.

```
total_files = fs.count()

# Or via len()...
total_files = len(fs)
```

For more details, please see the full documentation at http://hashfs.readthedocs.org.

CHAPTER 6

Guide

6.1 Installation

hashfs requires Python >= 2.7 or >= 3.3.

To install from PyPI:

```
pip install hashfs
```

6.2 API Reference

HashFS is a content-addressable file management system. What does that mean? Simply, that HashFS manages a directory where files are saved based on the file's hash.

Typical use cases for this kind of system are ones where:

- Files are written once and never change (e.g. image storage).
- It's desirable to have no duplicate files (e.g. user uploads).
- File metadata is stored elsewhere (e.g. in a database).

class hashfs.**HashFS**(*root*, *depth=4*, *width=1*, *algorithm='sha256'*, *fmode=436*, *dmode=493*)

Content addressable file manager.

root
str – Directory path used as root of storage space.

depth
int, optional – Depth of subfolders to create when saving a file.

width
int, optional – Width of each subfolder to create when saving a file.

algorithm
str – Hash algorithm to use when computing file hash. Algorithm should be available in `hashlib` module. Defaults to `'sha256'`.

fmode
int, optional – File mode permission to set when adding files to directory. Defaults to `0o664` which allows owner/group to read/write and everyone else to read.

dmode
int, optional – Directory mode permission to set for subdirectories. Defaults to `0o755` which allows owner/group to read/write and everyone else to read and everyone to execute.

computehash(*stream*)
Compute hash of file using *algorithm*.

corrupted(*extensions=True*)
Return generator that yields corrupted files as `(path, address)` where `path` is the path of the corrupted file and `address` is the *HashAddress* of the expected location.

count()
Return count of the number of files in the *root* directory.

delete(*file*)
Delete file using id or path. Remove any empty directories after deleting. No exception is raised if file doesn't exist.

Parameters **file** (*str*) – Address ID or path of file.

exists(*file*)
Check whether a given file id or path exists on disk.

files()
Return generator that yields all files in the *root* directory.

folders()
Return generator that yields all folders in the *root* directory that contain files.

get(*file*)
Return `HashAdress` from given id or path. If *file* does not refer to a valid file, then `None` is returned.

Parameters **file** (*str*) – Address ID or path of file.

Returns File's hash address.

Return type *HashAddress*

haspath(*path*)
Return whether *path* is a subdirectory of the *root* directory.

idpath(*id*, *extension=''*)
Build the file path for a given hash id. Optionally, append a file extension.

makepath(*path*)
Physically create the folder path on disk.

open(*file*, *mode='rb'*)
Return open buffer object from given id or path.

Parameters

- **file** (*str*) – Address ID or path of file.
- **mode** (*str, optional*) – Mode to open file in. Defaults to `'rb'`.

Returns An `io` buffer dependent on the *mode*.

Return type Buffer

Raises `IOError` – If file doesn't exist.

put(*file*, *extension=None*)
Store contents of *file* on disk using its content hash for the address.

Parameters

- **file** (*mixed*) – Readable object or path to file.
- **extension** (*str, optional*) – Optional extension to append to file when saving.

Returns File's hash address.

Return type *HashAddress*

realpath(*file*)
Attempt to determine the real path of a file id or path through successive checking of candidate paths. If the real path is stored with an extension, the path is considered a match if the basename matches the expected file path of the id.

relpath(*path*)
Return *path* relative to the *root* directory.

remove_empty(*subpath*)
Successively remove all empty folders starting with *subpath* and proceeding "up" through directory tree until reaching the *root* folder.

repair(*extensions=True*)
Repair any file locations whose content address doesn't match it's file path.

shard(*id*)
Shard content ID into subfolders.

size()
Return the total size in bytes of all files in the *root* directory.

unshard(*path*)
Unshard path to determine hash value.

class hashfs.**HashAddress**
File address containing file's path on disk and it's content hash ID.

id
str – Hash ID (hexdigest) of file contents.

relpath
str – Relative path location to *HashFS.root*.

abspath
str – Absoluate path location of file on disk.

is_duplicate
boolean, optional – Whether the hash address created was a duplicate of a previously existing file. Can only be `True` after a put operation. Defaults to `False`.

CHAPTER 7

Project Info

7.1 License

7.2 Versioning

This project follows Semantic Versioning with the following caveats:

- Only the public API (i.e. the objects imported into the hashfs module) will maintain backwards compatibility between MINOR version bumps.
- Objects within any other parts of the library are not guaranteed to not break between MINOR version bumps.

With that in mind, it is recommended to only use or import objects from the main module, hashfs.

7.3 Changelog

7.3.1 v0.7.1 (2018-10-13)

- Replace usage of `distutils.dir_util.mkpath` with `os.path.makedirs`.

7.3.2 v0.7.0 (2016-04-19)

- Use `shutil.move` instead of `shutil.copy` to move temporary file created during `put` operation to `HashFS` directory.

7.3.3 v0.6.0 (2015-10-19)

- Add faster `scandir` package for iterating over files/folders when platform is Python < 3.5. Scandir implementation was added to `os` module starting with Python 3.5.

7.3.4 v0.5.0 (2015-07-02)

- Rename private method `HashFS.copy` to `HashFS._copy`.
- Add `is_duplicate` attribute to `HashAddress`.
- Make `HashFS.put()` return `HashAddress` with `is_duplicate=True` when file with same hash already exists on disk.

7.3.5 v0.4.0 (2015-06-03)

- Add `HashFS.size()` method that returns the size of all files in bytes.
- Add `HashFS.count()`/`HashFS.__len__()` methods that return the count of all files.
- Add `HashFS.__iter__()` method to support iteration. Proxies to `HashFS.files()`.
- Add `HashFS.__contains__()` method to support `in` operator. Proxies to `HashFS.exists()`.
- Don't create the root directory (if it doesn't exist) until at least one file has been added.
- Fix `HashFS.repair()` not using `extensions` argument properly.

7.3.6 v0.3.0 (2015-06-02)

- Rename `HashFS.length` parameter/property to `width`. (**breaking change**)

7.3.7 v0.2.0 (2015-05-29)

- Rename `HashFS.get` to `HashFS.open`. (**breaking change**)
- Add `HashFS.get()` method that returns a `HashAddress` or `None` given a file ID or path.

7.3.8 v0.1.0 (2015-05-28)

- Add `HashFS.get()` method that retrieves a reader object given a file ID or path.
- Add `HashFS.delete()` method that deletes a file ID or path.
- Add `HashFS.folders()` method that returns the folder paths that directly contain files (i.e. subpaths that only contain folders are ignored).
- Add `HashFS.detokenize()` method that returns the file ID contained in a file path.
- Add `HashFS.repair()` method that reindexes any files under root directory whose file path doesn't not match its tokenized file ID.
- Rename `Address` classs to `HashAddress`. (**breaking change**)
- Rename `HashAddress.digest` to `HashAddress.id`. (**breaking change**)
- Rename `HashAddress.path` to `HashAddress.abspath`. (**breaking change**)
- Add `HashAddress.relpath` which represents path relative to `HashFS.root`.

7.3.9 v0.0.1 (2015-05-27)

- First release.
- Add `HashFS` class.
- Add `HashFS.put()` method that saves a file path or file-like object by content hash.
- Add `HashFS.files()` method that returns all files under root directory.
- Add `HashFS.exists()` which checks either a file hash or file path for existence.

7.4 Authors

7.4.1 Lead

- Derrick Gilland, dgilland@gmail.com, dgilland@github

7.4.2 Contributors

None

7.5 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

7.5.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/dgilland/hashfs/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" or "help wanted" is open to whoever wants to implement it.

Write Documentation

HashFS could always use more documentation, whether as part of the official HashFS docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/dgilland/hashfs/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

7.5.2 Get Started!

Ready to contribute? Here's how to set up `hashfs` for local development.

1. Fork the `hashfs` repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/hashfs.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenv installed, this is how you set up your fork for local development:

```
$ cd hashfs
$ make build
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass linting (PEP8 and pylint) and the tests, including testing other Python versions with tox:

```
$ make test-full
```

6. Add yourself to `AUTHORS.rst`.

7. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

8. Submit a pull request through the GitHub website.

7.5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the README.rst.
3. The pull request should work for Python 2.7, 3.3, and 3.4. Check https://travis-ci.org/dgilland/hashfs/pull_requests and make sure that the tests pass for all supported Python versions.

7.5.4 Project CLI

Some useful CLI commands when working on the project are below. **NOTE:** All commands are run from the root of the project and require `make`.

make build

Run the `clean` and `install` commands.

```
make build
```

make install

Install Python dependencies into virtualenv located at `env/`.

```
make install
```

make clean

Remove build/test related temporary files like `env/`, `.tox`, `.coverage`, and `__pycache__`.

```
make clean
```

make test

Run unittests under the virtualenv's default Python version. Does not test all support Python versions. To test all supported versions, see *make test-full*.

```
make test
```

make test-full

Run unittest and linting for all supported Python versions. **NOTE:** This will fail if you do not have all Python versions installed on your system. If you are on an Ubuntu based system, the Dead Snakes PPA is a good resource for easily installing multiple Python versions. If for whatever reason you're unable to have all Python versions on your development machine, note that Travis-CI will run full integration tests on all pull requests.

```
make test-full
```

make lint

Run `make pylint` and `make pep8` commands.

```
make lint
```

make pylint

Run `pylint` compliance check on code base.

```
make pylint
```

make pep8

Run PEP8 compliance check on code base.

```
make pep8
```

make docs

Build documentation to `docs/_build/`.

```
make docs
```

CHAPTER 8

Indices and Tables

- genindex
- modindex
- search

Python Module Index

h

Index