hdfs Documentation

Release 2.1.0

Author

Sep 08, 2017

Contents

API and command line interface for HDFS.

- Project homepage on GitHub
- PyPI entry

CHAPTER 1

Installation

Using pip:

```
$ pip install hdfs
```

By default none of the package requirements for extensions are installed. To do so simply suffix the package name with the desired extensions:

```
$ pip install hdfs[avro,dataframe,kerberos]
```

CHAPTER 2

User guide

Quickstart

This page first goes through the steps required to configure HdfsCLI's command line interface then gives an overview of the python API. If you are only interested in using HdfsCLI as a library, then feel free to jump ahead to the *Python bindings* section.

Configuration

HdfsCLI uses *aliases* to figure out how to connect to different HDFS clusters. These are defined in HdfsCLI's configuration file, located by default at `~/.hdfscli.cfg` (or elsewhere by setting the `HDFSCLI_CONFIG` environment variable correspondingly). See below for a sample configuration defining two aliases, `dev` and `prod`:

```
[global]
default.alias = dev

[dev.alias]
url = http://dev.namenode:port
user = ann

[prod.alias]
url = http://prod.namenode:port
root = /jobs/
```

Each alias is defined as its own `ALIAS.alias` section which must at least contain a `url` option with the URL to the namenode (including protocol and port). All other options can be omitted. If specified, `client` determines which *`hdfs.client.Client`* class to use and the remaining options are passed as keyword arguments to the appropriate constructor. The currently available client classes are:

- *`InsecureClient`* (the default)
- *`TokenClient`*

See the *Kerberos extension* to enable the `KerberosClient` and *Custom client support* to learn how to use other client classes.

The `url` option can be configured to support High Availability namenodes when using WebHDFS, simply add more URLs by delimiting with a semicolon (`;`).

Finally, note the `default.alias` entry in the global configuration section which will be used as default alias if none is specified.

Command line interface

HdfsCLI comes by default with a single entry point `hdfscli` which provides a convenient interface to perform common actions. All its commands accept an `--alias` argument (described above), which defines against which cluster to operate.

Downloading and uploading files

HdfsCLI supports downloading and uploading files and folders transparently from HDFS (we can also specify the degree of parallelism by using the `--threads` option).

```
$ # Write a single file to HDFS.
$ hdfscli upload --alias=dev weights.json models/
$ # Read all files inside a folder from HDFS and store them locally.
$ hdfscli download export/results/ "results-$(date +%F)"
```

If reading (resp. writing) a single file, its contents can also be streamed to standard out (resp. from standard in) by using `-` as path argument:

```
$ # Read a file from HDFS and append its contents to a local log file.
$ hdfscli download logs/1987-03-23.txt - >>logs
```

By default HdfsCLI will throw an error if trying to write to an existing path (either locally or on HDFS). We can force the path to be overwritten with the `--force` option.

Interactive shell

The `interactive` command (used also when no command is specified) will create an HDFS client and expose it inside a python shell (using IPython if available). This makes is convenient to perform file system operations on HDFS and interact with its data. See *Python bindings* below for an overview of the methods available.

```
$ hdfscli --alias=dev

Welcome to the interactive HDFS python shell.
The HDFS client is available as `CLIENT`.

In [1]: CLIENT.list('data/')
Out[1]: ['1.json', '2.json']

In [2]: CLIENT.status('data/2.json')
Out[2]: {
  'accessTime': 1439743128690,
  'blockSize': 134217728,
  'childrenNum': 0,
  'fileId': 16389,
  'group': 'supergroup',
```

```
  'length': 2,
  'modificationTime': 1439743129392,
  'owner': 'drwho',
  'pathSuffix': '',
  'permission': '755',
  'replication': 1,
  'storagePolicy': 0,
  'type': 'FILE'
}

In [3]: CLIENT.delete('data/2.json')
Out[3]: True
```

Using the full power of python lets us easily perform more complex operations such as renaming folder which match some pattern, deleting files which haven't been accessed for some duration, finding all paths owned by a certain user, etc.

More

Cf. `hdfscli --help` for the full list of commands and options.

Python bindings

Instantiating a client

The simplest way of getting a *`hdfs.client.Client`* instance is by using the *Interactive shell* described above, where the client will be automatically available. To instantiate a client programmatically, there are two options:

The first is to import the client class and call its constructor directly. This is the most straightforward and flexible, but doesn't let us reuse our configured aliases:

```
from hdfs import InsecureClient
client = InsecureClient('http://host:port', user='ann')
```

The second leverages the *`hdfs.config.Config`* class to load an existing configuration file (defaulting to the same one as the CLI) and create clients from existing aliases:

```
from hdfs import Config
client = Config().get_client('dev')
```

Reading and writing files

The *`read()`* method provides a file-like interface for reading files from HDFS. It must be used in a `with` block (making sure that connections are always properly closed):

```
# Loading a file in memory.
with client.read('features') as reader:
  features = reader.read()

# Directly deserializing a JSON object.
with client.read('model.json', encoding='utf-8') as reader:
  from json import load
  model = load(reader)
```

If a `chunk_size` argument is passed, the method will return a generator instead, making it sometimes simpler to stream the file's contents.

```
# Stream a file.
with client.read('features', chunk_size=8096) as reader:
  for chunk in reader:
    pass
```

Similarly, if a `delimiter` argument is passed, the method will return a generator of the delimited chunks.

```
with client.read('samples.csv', encoding='utf-8', delimiter='\n') as reader:
  for line in reader:
    pass
```

Writing files to HDFS is done using the *`write()`* method which returns a file-like writable object:

```
# Writing part of a file.
with open('samples') as reader, client.write('samples') as writer:
  for line in reader:
    if line.startswith('-'):
      writer.write(line)

# Writing a serialized JSON object.
with client.write('model.json', encoding='utf-8') as writer:
  from json import dump
  dump(model, writer)
```

For convenience, it is also possible to pass an iterable `data` argument directly to the method.

```
# This is equivalent to the JSON example above.
from json import dumps
client.write('model.json', dumps(model))
```

Exploring the file system

All *`Client`* subclasses expose a variety of methods to interact with HDFS. Most are modeled directly after the WebHDFS operations, a few of these are shown in the snippet below:

```
# Retrieving a file or folder content summary.
content = client.content('dat')

# Listing all files inside a directory.
fnames = client.list('dat')

# Retrieving a file or folder status.
status = client.status('dat/features')

# Renaming ("moving") a file.
client.rename('dat/features', 'features')

# Deleting a file or folder.
client.delete('dat', recursive=True)
```

Other methods build on these to provide more advanced features:

```
# Download a file or folder locally.
client.download('dat', 'dat', n_threads=5)
```

```
# Get all files under a given folder (arbitrary depth).
import posixpath as psp
fpaths = [
  psp.join(dpath, fname)
  for dpath, _, fnames in client.walk('predictions')
  for fname in fnames
]
```

See the *API reference* for the comprehensive list of methods available.

Checking path existence

Most of the methods described above will raise an *HdfsError* if called on a missing path. The recommended way of checking whether a path exists is using the *content()* or *status()* methods with a `strict=False` argument (in which case they will return `None` on a missing path).

More

See the *Advanced usage* section to learn more.

Advanced usage

Path expansion

All *Client* methods provide a path expansion functionality via the *resolve()* method. It enables the use of special markers to identify paths. For example, it currently supports the `#LATEST` marker which expands to the last modified file inside a given folder.

```
# Load the most recent data in the `tracking` folder.
with client.read('tracking/#LATEST') as reader:
  data = reader.read()
```

See the method's documentation for more information.

Custom client support

In order for the CLI to be able to instantiate arbitrary client classes, it has to be able to discover these first. This is done by specifying where they are defined in the `global` section of HdfsCLI's configuration file. For example, here is how we can make the *KerberosClient* class available:

```
[global]
autoload.modules = hdfs.ext.kerberos
```

More precisely, there are two options for telling the CLI where to load the clients from:

- `autoload.modules`, a comma-separated list of modules (which must be on python's path).
- `autoload.paths`, a comma-separated list of paths to python files.

Implementing custom clients can be particularly useful for passing default options (e.g. a custom `session` argument to each client). We describe below a working example implementing a secure client with optional custom certificate support.

We first implement our new client and save it somewhere, for example `/etc/hdfscli.py`.

```
from hdfs import Client
from requests import Session

class SecureClient(Client):

  """A new client subclass for handling HTTPS connections.

  :param url: URL to namenode.
  :param cert: Local certificate. See `requests` documentation for details
    on how to use this.
  :param verify: Whether to check the host's certificate.
  :param \*\*kwargs: Keyword arguments passed to the default `Client`
    constructor.

  """

  def __init__(self, url, cert=None, verify=True, **kwargs):
    session = Session()
    if ',' in cert:
      sessions.cert = [path.strip() for path in cert.split(',')]
    else:
      session.cert = cert
    if isinstance(verify, basestring): # Python 2.
      verify = verify.lower() in ('true', 'yes', 'ok')
    session.verify = verify
    super(SecureClient, self).__init__(url, session=session, **kwargs)
```

We then edit our configuration to tell the CLI how to load this module and define a `prod` alias using our new client:

```
[global]
autoload.paths = /etc/hdfscli.py

[prod.alias]
client = SecureClient
url = https://host:port
cert = /etc/server.crt, /etc/key
```

Note that options used to instantiate clients from the CLI (using *hdfs.client.Client.from_options()* under the hood) are always passed in as strings. This is why we had to implement some parsing logic in the `SecureClient` constructor above.

Tracking transfer progress

The *read()*, *upload()*, *download()* client methods accept a `progress` callback argument which can be used to track transfers. The passed function will be called every `chunk_size` bytes with two arguments:

- The source path of the file currently being transferred.
- The number of bytes currently transferred for this file or `-1` to signal that this file's transfer has just finished.

Below is an implementation of a toy tracker which simply outputs to standard error the total number of transferred bytes each time a file transfer completes (we must still take care to ensure correct behavior even during multi-threaded

transfers).

```
from sys import stderr
from threading import Lock

class Progress(object):

  """Basic progress tracker callback."""

  def __init__(self):
    self._data = {}
    self._lock = Lock()

  def __call__(self, hdfs_path, nbytes):
    with self._lock:
      if nbytes >= 0:
          self._data[hdfs_path] = nbytes
      else:
        stderr.write('%s\n' % (sum(self._data.values()), ))
```

Finally, note that the *write()* method doesn't expose a `progress` argument since this functionality can be replicated by passing a custom `data` generator (or within the context manager).

Logging configuration

It is possible to configure and disable where the CLI logs are written for each entry point. To do this, we can set the following options in its corresponding section (the entry point's name suffixed with `.command`). For example:

```
[hdfscli-avro.command]
log.level = INFO
log.path = /tmp/hdfscli/avro.log
```

The following options are available:

- `log.level`, handler log level (defaults to `DEBUG`).
- `log.path`, path to log file. The log is rotated every day (keeping a single copy). The default is a file named `COMMAND.log` in your current temporary directory. It is possible to view the currently active log file at any time by using the `--log` option at the command line.
- `log.disable`, disable logging to a file entirely (defaults to `False`).

Renaming entry points

By default the command line entry point will be named `hdfscli`. You can choose another name by specifying the `HDFSCLI_ENTRY_POINT` environment variable at installation time:

```
$ HDFSCLI_ENTRY_POINT=hdfs pip install hdfs
```

Extension prefixes will be adjusted similarly (e.g. in the previous example, `hdfscli-avro` would become `hdfs-avro`).

API reference

Client

WebHDFS API clients.

class `hdfs.client.`**`Client`**(*url*, *root=None*, *proxy=None*, *timeout=None*, *session=None*)
Bases: `object`

Base HDFS web client.

Parameters

- **`url`** – Hostname or IP address of HDFS namenode, prefixed with protocol, followed by WebHDFS port on namenode. You may also specify multiple URLs separated by semicolons for High Availability support.
- **`proxy`** – User to proxy as.
- **`root`** – Root path, this will be prefixed to all HDFS paths passed to the client. If the root is relative, the path will be assumed relative to the user's home directory.
- **`timeout`** – Connection timeouts, forwarded to the request handler. How long to wait for the server to send data before giving up, as a float, or a `(connect_timeout, read_timeout)` tuple. If the timeout is reached, an appropriate exception will be raised. See the requests documentation for details.
- **`session`** – `requests.Session` instance, used to emit all requests.

In general, this client should only be used directly when its subclasses (e.g. *`InsecureClient`*, *`TokenClient`*, and others provided by extensions) do not provide enough flexibility.

`acl_status`(*hdfs_path*, *strict=True*)
Get AclStatus for a file or folder on HDFS.

Parameters

- **`hdfs_path`** – Remote path.
- **`strict`** – If `False`, return `None` rather than raise an exception if the path doesn't exist.

`checksum`(*hdfs_path*)
Get a remote file's checksum.

Parameters **`hdfs_path`** – Remote path. Must point to a file.

`content`(*hdfs_path*, *strict=True*)
Get ContentSummary for a file or folder on HDFS.

Parameters

- **`hdfs_path`** – Remote path.
- **`strict`** – If `False`, return `None` rather than raise an exception if the path doesn't exist.

`delete`(*hdfs_path*, *recursive=False*)
Remove a file or directory from HDFS.

Parameters

- **`hdfs_path`** – HDFS path.
- **`recursive`** – Recursively delete files and directories. By default, this method will raise an `HdfsError` if trying to delete a non-empty directory.

This function returns `True` if the deletion was successful and `False` if no file or directory previously existed at `hdfs_path`.

download(*hdfs_path*, *local_path*, *overwrite=False*, *n_threads=1*, *temp_dir=None*, ***kwargs*)
Download a file or folder from HDFS and save it locally.

Parameters

- **hdfs_path** – Path on HDFS of the file or folder to download. If a folder, all the files under it will be downloaded.
- **local_path** – Local path. If it already exists and is a directory, the files will be downloaded inside of it.
- **overwrite** – Overwrite any existing file or directory.
- **n_threads** – Number of threads to use for parallelization. A value of `0` (or negative) uses as many threads as there are files.
- **temp_dir** – Directory under which the files will first be downloaded when `overwrite=True` and the final destination path already exists. Once the download successfully completes, it will be swapped in.
- ****kwargs** – Keyword arguments forwarded to *read()*. If no `chunk_size` argument is passed, a default value of 64 kB will be used. If a `progress` argument is passed and threading is used, care must be taken to ensure correct behavior.

On success, this method returns the local download path.

classmethod from_options(*options*, *class_name='Client'*)
Load client from options.

Parameters

- **options** – Options dictionary.
- **class_name** – Client class name. Defaults to the base *Client* class.

This method provides a single entry point to instantiate any registered *Client* subclass. To register a subclass, simply load its containing module. If using the CLI, you can use the `autoload.modules` and `autoload.paths` options.

list(*hdfs_path*, *status=False*)
Return names of files contained in a remote folder.

Parameters

- **hdfs_path** – Remote path to a directory. If `hdfs_path` doesn't exist or points to a normal file, an `HdfsError` will be raised.
- **status** – Also return each file's corresponding FileStatus.

makedirs(*hdfs_path*, *permission=None*)
Create a remote directory, recursively if necessary.

Parameters

- **hdfs_path** – Remote path. Intermediate directories will be created appropriately.
- **permission** – Octal permission to set on the newly created directory. These permissions will only be set on directories that do not already exist.

This function currently has no return value as WebHDFS doesn't return a meaningful flag.

parts(*hdfs_path*, *parts=None*, *status=False*)
Returns a dictionary of part-files corresponding to a path.

Parameters

- **hdfs_path** – Remote path. This directory should contain at most one part file per partition (otherwise one will be picked arbitrarily).
- **parts** – List of part-files numbers or total number of part-files to select. If a number, that many partitions will be chosen at random. By default all part-files are returned. If `parts` is a list and one of the parts is not found or too many samples are demanded, an *`HdfsError`* is raised.
- **status** – Also return each file's corresponding FileStatus.

read(**args*, ***kwds*)

Read a file from HDFS.

Parameters

- **hdfs_path** – HDFS path.
- **offset** – Starting byte position.
- **length** – Number of bytes to be processed. `None` will read the entire file.
- **buffer_size** – Size of the buffer in bytes used for transferring the data. Defaults the the value set in the HDFS configuration.
- **encoding** – Encoding used to decode the request. By default the raw data is returned. This is mostly helpful in python 3, for example to deserialize JSON data (as the decoder expects unicode).
- **chunk_size** – If set to a positive number, the context manager will return a generator yielding every `chunk_size` bytes instead of a file-like object (unless `delimiter` is also set, see below).
- **delimiter** – If set, the context manager will return a generator yielding each time the delimiter is encountered. This parameter requires the `encoding` to be specified.
- **progress** – Callback function to track progress, called every `chunk_size` bytes (not available if the chunk size isn't specified). It will be passed two arguments, the path to the file being uploaded and the number of bytes transferred so far. On completion, it will be called once with `-1` as second argument.

This method must be called using a `with` block:

```
with client.read('foo') as reader:
  content = reader.read()
```

This ensures that connections are always properly closed.

Note: The raw file-like object returned by this method (when called without an encoding, chunk size, or delimiter) can have a very different performance profile than local files. In particular, line-oriented methods are often slower. The recommended workaround is to specify an encoding when possible or read the entire file before splitting it.

---

rename(*hdfs_src_path*, *hdfs_dst_path*)

Move a file or folder.

Parameters

- **hdfs_src_path** – Source path.

- **hdfs_dst_path** – Destination path. If the path already exists and is a directory, the source will be moved into it. If the path exists and is a file, or if a parent destination directory is missing, this method will raise an `HdfsError`.

resolve(*hdfs_path*)
Return absolute, normalized path, with special markers expanded.

Parameters **hdfs_path** – Remote path.

Currently supported markers:

- `'#LATEST'`: this marker gets expanded to the most recently updated file or folder. They can be combined using the `'{N}'` suffix. For example, `'foo/#LATEST{2}'` is equivalent to `'foo/#LATEST/#LATEST'`.

set_acl(*hdfs_path*, *acl_spec*, *clear=True*)
SetAcl_ or ModifyAcl_ for a file or folder on HDFS.

Parameters

- **hdfs_path** – Path to an existing remote file or directory. An `HdfsError` will be raised if the path doesn't exist.
- **acl_spec** – String representation of an ACL spec. Must be a valid string with entries for user, group and other. For example: `"user::rwx,user:foo:rw-,group::r--,other::---"`.
- **clear** – Clear existing ACL entries. If set to false, all existing ACL entries that are not specified in this call are retained without changes, behaving like ModifyAcl_. For example: `"user:foo:rwx"`.

.. _SetAcl: SETACL_

set_owner(*hdfs_path*, *owner=None*, *group=None*)
Change the owner of file.

Parameters

- **hdfs_path** – HDFS path.
- **owner** – Optional, new owner for file.
- **group** – Optional, new group for file.

At least one of `owner` and `group` must be specified.

set_permission(*hdfs_path*, *permission*)
Change the permissions of file.

Parameters

- **hdfs_path** – HDFS path.
- **permission** – New octal permissions string of file.

set_replication(*hdfs_path*, *replication*)
Set file replication.

Parameters

- **hdfs_path** – Path to an existing remote file. An `HdfsError` will be raised if the path doesn't exist or points to a directory.
- **replication** – Replication factor.

set_times(*hdfs_path*, *access_time=None*, *modification_time=None*)
Change remote timestamps.

Parameters

- **hdfs_path** – HDFS path.
- **access_time** – Timestamp of last file access.
- **modification_time** – Timestamps of last file access.

status(*hdfs_path*, *strict=True*)
Get FileStatus for a file or folder on HDFS.

Parameters

- **hdfs_path** – Remote path.
- **strict** – If `False`, return `None` rather than raise an exception if the path doesn't exist.

upload(*hdfs_path*, *local_path*, *overwrite=False*, *n_threads=1*, *temp_dir=None*, *chunk_size=65536*, *progress=None*, *cleanup=True*, ***kwargs*)
Upload a file or directory to HDFS.

Parameters

- **hdfs_path** – Target HDFS path. If it already exists and is a directory, files will be uploaded inside.
- **local_path** – Local path to file or folder. If a folder, all the files inside of it will be uploaded (note that this implies that folders empty of files will not be created remotely).
- **overwrite** – Overwrite any existing file or directory.
- **n_threads** – Number of threads to use for parallelization. A value of `0` (or negative) uses as many threads as there are files.
- **temp_dir** – Directory under which the files will first be uploaded when `overwrite=True` and the final remote path already exists. Once the upload successfully completes, it will be swapped in.
- **chunk_size** – Interval in bytes by which the files will be uploaded.
- **progress** – Callback function to track progress, called every `chunk_size` bytes. It will be passed two arguments, the path to the file being uploaded and the number of bytes transferred so far. On completion, it will be called once with `-1` as second argument.
- **cleanup** – Delete any uploaded files if an error occurs during the upload.
- ****kwargs** – Keyword arguments forwarded to *write()*.

On success, this method returns the remote upload path.

walk(*hdfs_path*, *depth=0*, *status=False*)
Depth-first walk of remote filesystem.

Parameters

- **hdfs_path** – Starting path. If the path doesn't exist, an `HdfsError` will be raised. If it points to a file, the returned generator will be empty.
- **depth** – Maximum depth to explore. `0` for no limit.
- **status** – Also return each file or folder's corresponding FileStatus.

This method returns a generator yielding tuples `(path, dirs, files)` where `path` is the absolute path to the current directory, `dirs` is the list of directory names it contains, and `files` is the list of file names it contains.

write(*hdfs_path*, *data=None*, *overwrite=False*, *permission=None*, *blocksize=None*, *replication=None*, *buffersize=None*, *append=False*, *encoding=None*)
Create a file on HDFS.

Parameters

- **hdfs_path** – Path where to create file. The necessary directories will be created appropriately.
- **data** – Contents of file to write. Can be a string, a generator or a file object. The last two options will allow streaming upload (i.e. without having to load the entire contents into memory). If `None`, this method will return a file-like object and should be called using a `with` block (see below for examples).
- **overwrite** – Overwrite any existing file or directory.
- **permission** – Octal permission to set on the newly created file. Leading zeros may be omitted.
- **blocksize** – Block size of the file.
- **replication** – Number of replications of the file.
- **buffersize** – Size of upload buffer.
- **append** – Append to a file rather than create a new one.
- **encoding** – Encoding used to serialize data written.

Sample usages:

```
from json import dump, dumps

records = [
  {'name': 'foo', 'weight': 1},
  {'name': 'bar', 'weight': 2},
]

# As a context manager:
with client.write('data/records.jsonl', encoding='utf-8') as writer:
  dump(records, writer)

# Or, passing in a generator directly:
client.write('data/records.jsonl', data=dumps(records), encoding='utf-8')
```

class hdfs.client.**InsecureClient**(*url*, *user=None*, ***kwargs*)
Bases: *hdfs.client.Client*

HDFS web client to use when security is off.

Parameters

- **url** – Hostname or IP address of HDFS namenode, prefixed with protocol, followed by WebHDFS port on namenode
- **user** – User default. Defaults to the current user's (as determined by `whoami`).
- ****kwargs** – Keyword arguments passed to the base class' constructor.

Note that if a session argument is passed in, it will be modified in-place to support authentication.

class hdfs.client.**TokenClient**(*url*, *token*, ***kwargs*)

Bases: *hdfs.client.Client*

HDFS web client using Hadoop token delegation security.

Parameters

- **url** – Hostname or IP address of HDFS namenode, prefixed with protocol, followed by WebHDFS port on namenode
- **token** – Hadoop delegation token.
- ****kwargs** – Keyword arguments passed to the base class' constructor.

Note that if a session argument is passed in, it will be modified in-place to support authentication.

Extensions

The following extensions are currently available:

Kerberos

Support for clusters using Kerberos authentication.

This extension adds a new *hdfs.client.Client* subclass, *KerberosClient*, which handles authentication appropriately with Kerberized clusters:

```
from hdfs.ext.kerberos import KerberosClient
client = KerberosClient('http://host:port')
```

To expose this class to the command line interface (so that it can be used by aliases), we add the following line inside the `global` section of `~/.hdfscli.cfg` (or wherever our configuration file is located):

```
autoload.modules = hdfs.ext.kerberos
```

Here is what our earlier configuration would look like if we updated it to support a Kerberized production grid:

```
[global]
default.alias = dev
autoload.modules = hdfs.ext.kerberos

[dev.alias]
url = http://dev.namenode:port

[prod.alias]
url = http://prod.namenode:port
client = KerberosClient
```

class hdfs.ext.kerberos.**KerberosClient**(*url*, *mutual_auth='OPTIONAL'*, *max_concurrency=1*, *root=None*, *proxy=None*, *timeout=None*, *session=None*, ***kwargs*)

Bases: *hdfs.client.Client*

HDFS web client using Kerberos authentication.

Parameters

- **url** – Hostname or IP address of HDFS namenode, prefixed with protocol, followed by WebHDFS port on namenode.

- **mutual_auth** – Whether to enforce mutual authentication or not (possible values: `'REQUIRED'`, `'OPTIONAL'`, `'DISABLED'`).
- **max_concurrency** – Maximum number of allowed concurrent authentication requests. This is required since requests exceeding the threshold allowed by the server will be unable to authenticate.
- **proxy** – User to proxy as.
- **root** – Root path, this will be prefixed to all HDFS paths passed to the client. If the root is relative, the path will be assumed relative to the user's home directory.
- **timeout** – Connection timeouts, forwarded to the request handler. How long to wait for the server to send data before giving up, as a float, or a `(connect_timeout, read_timeout)` tuple. If the timeout is reached, an appropriate exception will be raised. See the requests documentation for details.
- **session** – `requests.Session` instance, used to emit all requests.
- ****kwargs** – Additional arguments passed to the underlying `HTTPKerberosAuth` class.

To avoid replay errors, a timeout of 1 ms is enforced between requests. If a session argument is passed in, it will be modified in-place to support authentication.

Avro

Read and write Avro files directly from HDFS.

This extension enables streaming decoding and encoding of files from and to HDFS. It requires the `fastavro` library.

- *AvroWriter* writes Avro files on HDFS from python objects.
- *AvroReader* reads Avro files from HDFS into an iterable of records.

Sample usage:

```
#!/usr/bin/env python
# encoding: utf-8

"""Avro extension example."""

from hdfs import Config
from hdfs.ext.avro import AvroReader, AvroWriter


# Get the default alias' client.
client = Config().get_client()

# Some sample data.
records = [
  {'name': 'Ann', 'age': 23},
  {'name': 'Bob', 'age': 22},
]

# Write an Avro File to HDFS (since our records' schema is very simple, we let
# the writer infer it automatically, otherwise we would pass it as argument).
with AvroWriter(client, 'names.avro', overwrite=True) as writer:
  for record in records:
    writer.write(record)

# Read it back.
```

```
with AvroReader(client, 'names.avro') as reader:
  schema = reader.schema # The inferred schema.
  content = reader.content # The remote file's HDFS content object.
  assert list(reader) == records # The records match!
```

It also features an entry point (named `hdfscli-avro` by default) which provides access to the above functionality from the shell. For usage examples and more information:

```
$ hdfscli-avro --help
```

class hdfs.ext.avro.**AvroReader**(*client*, *hdfs_path*, *parts=None*)

Bases: object

HDFS Avro file reader.

Parameters

- **client** – *hdfs.client.Client* instance.
- **hdfs_path** – Remote path.
- **parts** – Part-files to read, when reading a distributed file. The default is to read all part-files in order. See *hdfs.client.Client.parts()* for details.

The contents of the file will be decoded in a streaming manner, as the data is transferred. This makes it possible to use on files of arbitrary size. As a convenience, the content summary object of the remote file is available on the reader's `content` attribute.

Usage:

```
with AvroReader(client, 'foo.avro') as reader:
  schema = reader.schema # The remote file's Avro schema.
  content = reader.content # Content metadata (e.g. size).
  for record in reader:
    pass # and its records
```

schema

Get the underlying file's schema.

The schema will only be available after entering the reader's corresponding `with` block.

class hdfs.ext.avro.**AvroWriter**(*client*, *hdfs_path*, *schema=None*, *codec=None*, *sync_interval=None*, *sync_marker=None*, *metadata=None*, ***kwargs*)

Bases: object

Write an Avro file on HDFS from python dictionaries.

Parameters

- **client** – *hdfs.client.Client* instance.
- **hdfs_path** – Remote path.
- **schema** – Avro schema. If not specified, the writer will try to infer it from the first record sent. There are however limitations regarding what can be inferred.
- **codec** – Compression codec. The default is `'null'` (no compression).
- **sync_interval** – Number of bytes after which a block will be written.
- **sync_marker** – 16 byte tag used for synchronization. If not specified, one will be generated at random.

- **metadata** – Additional metadata to include in the container file's header. Keys starting with `'avro.'` are reserved.
- ****kwargs** – Keyword arguments forwarded to *hdfs.client.Client.write()*.

Usage:

```
with AvroWriter(client, 'data.avro') as writer:
  for record in records:
    writer.write(record)
```

schema
Avro schema.

write(*record*)
Store a record.

Parameters **record** – Record object to store.

Only available inside the `with` block.

Dataframe

Read and write Pandas dataframes directly from HDFS.

```
#!/usr/bin/env python
# encoding: utf-8

"""Dataframe extension example."""

from hdfs import Config
from hdfs.ext.dataframe import read_dataframe, write_dataframe
import pandas as pd


# Get the default alias' client.
client = Config().get_client()

# A sample dataframe.
df = pd.DataFrame.from_records([
  {'A': 1, 'B': 2},
  {'A': 11, 'B': 23}
])

# Write dataframe to HDFS using Avro serialization.
write_dataframe(client, 'data.avro', df, overwrite=True)

# Read the Avro file back from HDFS.
_df = read_dataframe(client, 'data.avro')

# The frames match!
pd.util.testing.assert_frame_equal(df, _df)
```

This extension requires both the `avro` extension and `pandas` to be installed. Currently only Avro serialization is supported.

`hdfs.ext.dataframe.`**read_dataframe**(*client*, *hdfs_path*)
Read dataframe from HDFS Avro file.

Parameters

- **client** – *hdfs.client.Client* instance.
- **hdfs_path** – Remote path to an Avro file (potentially distributed).

hdfs.ext.dataframe.**write_dataframe**(*client*, *hdfs_path*, *df*, ***kwargs*)
Save dataframe to HDFS as Avro.

Parameters

- **client** – *hdfs.client.Client* instance.
- **hdfs_path** – Remote path where the dataframe will be stored.
- **df** – Dataframe to store.
- ****kwargs** – Keyword arguments passed through to *hdfs.ext.avro.AvroWriter*.

Configuration

Command line interface configuration module.

This module provides programmatic access to HdfsCLI's configuration settings. In particular it exposes the ability to instantiate clients from aliases (see *Config.get_client()*).

class hdfs.config.**Config**(*path=None*, *stream_log_level=None*)
Bases: ConfigParser.RawConfigParser

Configuration class.

Parameters

- **path** – path to configuration file. If no file exists at that location, the configuration parser will be empty. If not specified, the value of the HDFSCLI_CONFIG environment variable is used if it exists, otherwise it defaults to ~/.hdfscli.cfg.
- **stream_log_level** – Stream handler log level, attached to the root logger. A false-ish value will disable this handler. This is particularly useful with the *catch()* function which reports exceptions as log messages.

On instantiation, the configuration object will attempt to load modules defined in the autoload global options (see *Custom client support* for more information).

get_client(*alias=None*)
Load HDFS client.

Parameters **alias** – The client to look up. If not specified, the default alias be used (default.alias option in the global section) if available and an error will be raised otherwise.

Further calls to this method for the same alias will return the same client instance (in particular, any option changes to this alias will not be taken into account).

get_log_handler(*command*)
Configure and return log handler.

Parameters **command** – The command to load the configuration for. All options will be looked up in the [COMMAND.command] section. This is currently only used for configuring the file handler for logging. If logging is disabled for the command, a *NullHandler* will be returned, else a TimedRotatingFileHandler.

class hdfs.config.**NullHandler**(*level=0*)
Bases: logging.Handler

Pass-through logging handler.

This is required for python <2.7.

Initializes the instance - basically setting the formatter to None and the filter list to empty.

emit(*record*)
Do nothing.

hdfs.config.**catch**(**error_classes*)
Returns a decorator that catches errors and prints messages to stderr.

Parameters ***error_classes** – Error classes.

Also exits with status 1 if any errors are caught.

Utilities

Common utilities.

class hdfs.util.**AsyncWriter**(*consumer*)
Bases: object

Asynchronous publisher-consumer.

Parameters **consumer** – Function which takes a single generator as argument.

This class can be used to transform functions which expect a generator into file-like writer objects. This can make it possible to combine different APIs together more easily. For example, to send streaming requests:

```
import requests as rq

with AsyncWriter(lambda data: rq.post(URL, data=data)) as writer:
  writer.write('Hello, world!')
```

flush()
Pass-through implementation.

write(*chunk*)
Stream data to the underlying consumer.

Parameters **chunk** – Bytes to write. These will be buffered in memory until the consumer reads them.

exception hdfs.util.**HdfsError**(*message*, **args*, ***kwargs*)
Bases: exceptions.Exception

Base error class.

Parameters

- **message** – Error message.
- **args** – optional Message formatting arguments.

hdfs.util.**temppath**(**args*, ***kwds*)
Create a temporary path.

Parameters **dpath** – Explicit directory name where to create the temporary path. A system dependent default will be used otherwise (cf. tempfile.mkstemp).

Usage:

```
with temppath() as path:
  pass # do stuff
```

Any file or directory corresponding to the path will be automatically deleted afterwards.

CHAPTER 3

Sample script

```python
#!/usr/bin/env python
# encoding: utf-8

"""Sample HdfsCLI script.

This example shows how to write files to HDFS, read them back, and perform a
few other simple filesystem operations.

"""

from hdfs import Config
from json import dump, load


# Get the default alias' client. (See the quickstart section in the
# documentation to learn more about this.)
client = Config().get_client()

# Some fake data that we are interested in uploading to HDFS.
model = {
  '(intercept)': 48.,
  'first_feature': 2.,
  'second_feature': 12.,
}

# First, we delete any existing `models/` folder on HDFS.
client.delete('models', recursive=True)

# We can now upload the data, first as CSV.
with client.write('models/1.csv', encoding='utf-8') as writer:
  for item in model.items():
    writer.write(u'%s,%s\n' % item)

# We can also serialize it to JSON and directly upload it.
with client.write('models/1.json', encoding='utf-8') as writer:
```

```
  dump(model, writer)

# We can check that the files exist and get their properties.
assert client.list('models') == ['1.csv', '1.json']
status = client.status('models/1.csv')
content = client.content('models/1.json')

# Later, we can download the files back. The `delimiter` option makes it
# convenient to read CSV files.
with client.read('models/1.csv', delimiter='\n', encoding='utf-8') as reader:
  items = (line.split(',') for line in reader if line)
  assert dict((name, float(value)) for name, value in items) == model

# Loading JSON directly from HDFS is even simpler.
with client.read('models/1.json', encoding='utf-8') as reader:
  assert load(reader) == model
```

More examples can be found in the `examples/` folder on GitHub.

Python Module Index

h

Index

W