Exporters Documentation

Release 0.0.1-alpha

Scrapinghub

Jul 20, 2017

Contents

Contents:

CHAPTER 1

Exporters project documentation

Exporters provide a flexible way to export data from multiple sources to multiple destinations, allowing filtering and transforming the data.

This Github repository is used as a central repository.

Full documentation can be found here http://exporters.readthedocs.io/en/latest/

Getting Started

Install exporters

First of all, we recommend to create a virtualenv:

```
virtualenv exporters
source exporters/bin/activate
```

Installing:

```
pip install exporters
```

Creating a configuration

Then, we can create our first configuration object and store it in a file called config.json. This configuration will read from an s3 bucket and store it in our filesystem, exporting only the records which have United States in field country:

```
{
     "reader": {
         "name": "exporters.readers.s3_reader.S3Reader",
         "options": {
             "bucket": "YOUR_BUCKET",
```

```
            "aws_access_key_id": "YOUR_ACCESS_KEY",
            "aws_secret_access_key": "YOUR_SECRET_KEY",
            "prefix": "exporters-tutorial/sample-dataset"
        }
    },
    "filter": {
        "name": "exporters.filters.key_value_regex_filter.KeyValueRegexFilter",
        "options": {
            "keys": [
                {"name": "country", "value": "United States"}
            ]
        }
    },
    "writer":{
        "name": "exporters.writers.fs_writer.FSWriter",
        "options": {
            "filebase": "/tmp/output/"
        }
    }
}
```

Export with script

We can use the provided script to run this export:

```
python bin/export.py --config config.json
```

Use it as a library

The export can be run using exporters as a library:

```
from exporters import BasicExporter

exporter = BasicExporter.from_file_configuration('config.json')
exporter.export()
```

Resuming an export job

Let's suppose we have a pickle file with a previously failed export job. If we want to resume it we must run the export script:

```
python bin/export.py --resume pickle://pickle-file.pickle
```

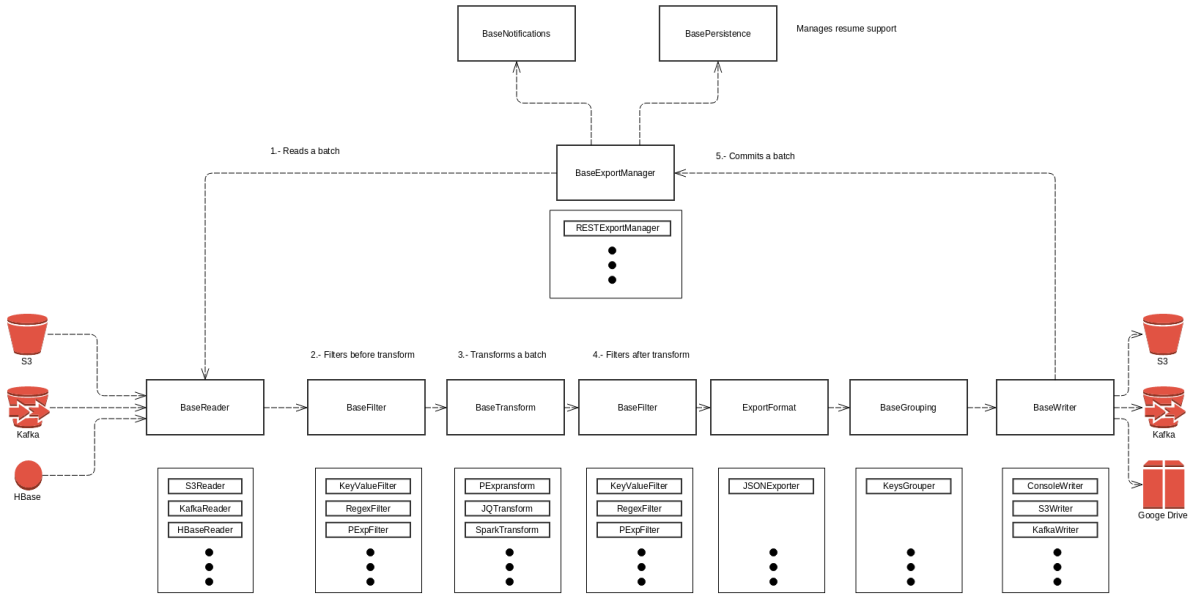
CHAPTER 2

Exporters description

What are exporters?

Exporters aims to provide a flexible and easy to extend infrastructure for exporting data to and from multiple sources, with support for filtering and transformation.

Architecture



Config file

You can define Exporters behaviour with a configuration object. This object has the following attributes:

- reader (mandatory): defines what reader module the export should use and its options.
- writer (mandatory): defines what writer module the export should use and its options.
- exporter_options: it contains general export options, including output format.
- filter_before: defines what filter module should be used before transforming and its options.
- filter_after: defines what filter module should be used after transforming and its options.
- transform: defines what transform module the export should use and its options.
- persistence: defines what persistence module the export should use and its options.
- stats_manager: defines what stats_manager module the export should use and its options.
- grouper: defines what grouper module the export should use and its options.

This is an example of the simplest config file that can be used.

```
 {

     "reader": {
         "name": "exporters.readers.random_reader.RandomReader",
         "options": {
             "number_of_items": 1000,
             "batch_size": 10
         }
     },
     "writer":{
         "name": "exporters.writers.console_writer.ConsoleWriter",
         "options": {
         }
     }
}
```

Features

- Multiple readers and writers.
- Resume support using different technologies as a backend (MySQL, Postgres, Pickle...)
- Support for bypass (direct copying) export pipeline for certain configurations to improve speed exports.
- Easy to extend and flexible architecture.
- Different output formats.
- Notifications and webhooks support.
- Export stats gathering.
- Grouping support.
- Filter and transform exported items.
- Loading options from env.

Going further

Exporters functionality can be exteded by plugins. Please check https://github.com/exporters-plugins to find out about them.

CHAPTER 3

Modules

Every module has a *supported_options* attribute that defines which options are optional or mandatory, and the default values if proceeds. It is a dict with the following shape:

Possible attributes for a supported_option are:

- type - The option type. In addition to the standard python types (`basestring`, `int`, `list`, `dict`...], exporters provide homogeneous list types in `exporters.utils` called `str_list`, `int_list` and `dict_list`, This types indicate that every member of the list needs to be able to be be casted to a string, integer or dictionary respectively.
- default - Default option value if it is not provided by configuration object. If it is present,

the supported_option will be optional instead of mandatory. - env_fallback - If option is not provided by configuration object, it will be loaded from env_fallback environment variable.

Export Manager

This module handles the pipeline iteration. A pipeline iteration usually consists on calling the reader to get a batch, filter it, transform it, filter it again, write it and commit the read batch. It should also be in charge of notifications and retries management.

Provided exporters

BasicExporter

class `exporters.export_managers.basic_exporter.`**`BasicExporter`**(*configuration*)

Bases: `exporters.export_managers.base_exporter.BaseExporter`

Basic export manager reading configuration a json file. It has bypass support.

classmethod **`from_file_configuration`**(*filepath*)

classmethod **`from_persistence_configuration`**(*persistence_uri*)

Bypass support

Exporters architecture provides support for bypassing the pipeline. One example would be in which both reader and writer aim S3 buckets. If no transforms or filtering are needed, keys can be copied directly without downloading them.

All bypass classes are subclasses of BaseBypass class, and must implement two methods:

- **meets_conditions(configuration)** Checks if provided export configuration meets the requirements to use the bypass. If not, it returns False.
- **execute()** Executes the bypass script.
- **close()** Perform all needed actions to leave a clean system after the bypass execution.

Provided Bypass scripts

S3Bypass

exception exporters.bypasses.s3_to_s3_bypass.**InvalidKeyIntegrityCheck**
Bases: exceptions.Exception

Exception thrown when two s3 keys have different md5 checksums

class exporters.bypasses.s3_to_s3_bypass.**S3Bypass**(*config*, *metadata*)
Bases: exporters.bypasses.base_s3_bypass.BaseS3Bypass

Bypass executed by default when data source and data destination are S3 buckets. It should be transparent to user. Conditions are:

•S3Reader and S3Writer are used on configuration.

•No filter modules are set up.

•No transform module is set up.

•No grouper module is set up.

•S3 Writer has not a items_limit set in configuration.

•S3 Writer has default items_per_buffer_write and size_per_buffer_write per default.

•S3 Writer has default write_buffer.

This bypass tries to directly copy the S3 keys between the read and write buckets. If is is not possible due to permission issues, it will download the key from the read bucket and directly upload it to the write bucket.

close()

execute()

get_dest_key_name(*name*)

classmethod meets_conditions(*config*)

exporters.bypasses.s3_to_s3_bypass.**key_permissions**(**args*, ***kwds*)

S3ToAzureBlobBypass

class exporters.bypasses.s3_to_azure_blob_bypass.**S3AzureBlobBypass**(*config*, *metadata*)
Bases: exporters.bypasses.base_s3_bypass.BaseS3Bypass

Bypass executed by default when data source is an S3 bucket and data destination is an Azure blob container. It should be transparent to user. Conditions are:

- S3Reader and AzureBlobWriter are used on configuration.
- No filter modules are set up.
- No transform module is set up.
- No grouper module is set up.
- AzureBlobWriter has not a items_limit set in configuration.
- AzureBlobWriter has default items_per_buffer_write and size_per_buffer_write per default.
- AzureBlobWriter has default write_buffer.

classmethod **meets_conditions**(*config*)

S3ToAzureFileBypass

class exporters.bypasses.s3_to_azure_file_bypass.**S3AzureFileBypass**(*config*, *metadata*)

Bases: exporters.bypasses.base_s3_bypass.BaseS3Bypass

Bypass executed by default when data source is an S3 bucket and data destination is an Azure share. It should be transparent to user. Conditions are:

- S3Reader and AzureFileWriter are used on configuration.
- No filter modules are set up.
- No transform module is set up.
- No grouper module is set up.
- AzureFileWriter has not a items_limit set in configuration.
- AzureFileWriter has default items_per_buffer_write and size_per_buffer_write per default.
- AzureFileWriter has default write_buffer.

classmethod **meets_conditions**(*config*)

StreamBypass

class exporters.bypasses.stream_bypass.**Stream**(*filename*, *size*, *meta*)

Bases: tuple

filename
Alias for field number 0

meta
Alias for field number 2

size
Alias for field number 1

class exporters.bypasses.stream_bypass.**StreamBypass**(*config*, *metadata*)

Bases: exporters.bypasses.base.BaseBypass

Generic Bypass that streams the contents of the files instead of running them through the export pipeline.

It should be transparent to user. Conditions are:

- Reader module supports get_read_streams
- Writer module supports write_stream
- No filter modules are set up.
- No transform module is set up.
- No grouper module is set up.
- writer has no option items_limit set in configuration.
- writer has default items_per_buffer_write and size_per_buffer_write per default.
- writer has default write_buffer.

close()

execute()

classmethod meets_conditions(*config*)

class exporters.bypasses.stream_bypass.**StreamBypassState**(*config*, *metadata*)
Bases: object

commit_copied(*stream*)

delete()

increment_bytes(*cnt*)

Reader

There are two kinds of readers, stream readers and structured readers. Stream readers read a stream of data from a binary backend, while in structured readers the underlying storage stores structured items.

Stream readers have to be combined with a decompressor and a deserializer to read the items from them (by default, ZLibDecompressor and JsonDeserializer are used).

Readers are in charge of providing batches of items to the pipeline. All readers are subclasses of BaseReader class.

Structured readers must implement:

- **get_next_batch()** This method is called from the manager. It must return a list or a generator of BaseRecord objects. When it has nothing else to read, it must set class variable "finished" to True.

Stream readers must implement:

- **get_read_streams()** This should be an iterator that yields streams (file-like objects).

class exporters.readers.base_reader.**BaseReader**(*options*, *metadata*)
Bases: exporters.pipeline.base_pipeline_item.BasePipelineItem

This module reads and creates a batch to pass them to the pipeline

close()

get_all_metadata(*module='reader'*)

get_last_position()
Returns the last read position.

get_metadata(*key*, *module='reader'*)

get_next_batch()
This method is called from the manager. It must return a list or a generator of BaseRecord objects. When it has nothing else to read, it must set class variable "finished" to True.

increase_read()

is_finished()
Returns whether if there are items left to be read or not.

set_last_position(*last_position*)
Called from the manager, it is in charge of updating the last position of data commited by the writer, in order to have resume support

set_metadata(*key*, *value*, *module='reader'*)

supported_options = {}

update_metadata(*data*, *module='reader'*)

class exporters.readers.base_stream_reader.**StreamBasedReader**(**args*, ***kwargs*)
Bases: exporters.readers.base_reader.BaseReader

Abstract readers for storage backends that operate in bytes instead of json objects.

The bytes are first decompressed using a decompressor and then deserialized using a deserializer.

Avaliable Options:

- **batch_size (int)** Number of items to be returned in each batch

decompressor = <exporters.decompressors.ZLibDecompressor object>

deserializer = <exporters.deserializers.JsonLinesDeserializer object>

get_next_batch()
This method is called from the manager. It must return a list or a generator of BaseRecord objects. When it has nothing else to read, it must set class variable "finished" to True.

get_read_streams()
To be subclassed

iteritems()

iteritems_retrying(*stream_data*)

set_last_position(*last_position*)
Called from the manager, it is in charge of updating the last position of data commited by the writer, in order to have resume support

supported_options = {'batch_size': {'default': 10000, 'type': (<type 'int'>, <type 'long'>)}}

exporters.readers.base_stream_reader.**is_stream_reader**(*reader*)

Provided readers

RandomReader

Random items generator, just for testing purposes

class exporters.readers.random_reader.**RandomReader**(**args*, ***kwargs*)
Bases: exporters.readers.base_reader.BaseReader

It is just a reader with testing purposes. It generates random data in a quantity that is set in its config section.

- **number_of_items (int)** Number of total items that must be returned by the reader before finishing.
- **batch_size (int)** Number of items to be returned in each batch.

get_next_batch()
This method is called from the manager. It must return a list or a generator of BaseRecord objects. When it has nothing else to read, it must set class variable "finished" to True.

set_last_position(*last_position*)
Called from the manager, it is in charge of updating the last position of data commited by the writer, in order to have resume support

supported_options = {'batch_size': {'default': 100, 'type': (<type 'int'>, <type 'long'>)}, 'number_of_items': {'defau

FSReader (Stream)

class exporters.readers.fs_reader.**FSReader**(**args*, ***kwargs*)
Bases: *exporters.readers.base_stream_reader.StreamBasedReader*

Reads items from files located in filesystem and compressed with gzip with a common path.

- **input (str/dict or a list of str/dict)** Specification of files to be read.

  Accepts either one "input_unit" or many of them in a list. "input_unit" is defined as follows:

  If a string, it indicates a filename, e.g. "/path/to/filename".

  If a dictionary, it indicates a directory to be read with the following elements:

  - "dir": path to directory, e.g. "/path/to/dir".
  - "dir_pointer": path to file containing path to directory, e.g. "/path/to/pointer/file" which contains "/path/to/dir". Cannot be used together with "dir".

    For example:

    We have a weekly export set with CRON. If we wanted to point to a new data path every week, we should keep updating the export configuration. With a pointer, we can set the reader to read from that file, which contains one line with a valid path to datasets, so only that pointer file should be updated.
  - "pattern": (optional) regular expression to filter filenames, e.g. "output.*.jl.gz$"

get_read_streams()

open_stream(*stream*)

supported_options = {'input': {'default': {'dir': ''}, 'type': (<type 'str'>, <type 'dict'>, <type 'list'>)}, 'batch_size':

KafkaScannerReader

Kafka reader

class exporters.readers.kafka_scanner_reader.**KafkaScannerReader**(**args*, ***kwargs*)
Bases: exporters.readers.base_reader.BaseReader

This reader retrieves items from kafka brokers.

- **batch_size (int)** Number of items to be returned in each batch
- **brokers (list)** List of brokers uris.
- **topic (str)** Topic to read from.

- **partitions (list)** Partitions to read from.

get_from_kafka(**args*, ***kw*)

get_next_batch()
This method is called from the manager. It must return a list or a generator of BaseRecord objects. When it has nothing else to read, it must set class variable "finished" to True.

set_last_position(*last_position*)
Called from the manager, it is in charge of updating the last position of data commited by the writer, in order to have resume support

supported_options = {'batch_size': {'default': 10000, 'type': (<type 'int'>, <type 'long'>)}, 'topic': {'type': (<type 'b

KafkaRandomReader

Kafka random reader

class exporters.readers.kafka_random_reader.**KafkaRandomReader**(**args*, ***kwargs*)
Bases: exporters.readers.base_reader.BaseReader

This reader retrieves a random subset of items from kafka brokers.

- **record_count (int)** Number of items to be returned in total
- **batch_size (int)** Number of items to be returned in each batch
- **brokers (list)** List of brokers uris.
- **topic (str)** Topic to read from.
- **group (str)** Reading group for kafka client.

consume_messages(*batchsize*)
Get messages batch from the reservoir

create_processor()

decompress_messages(*offmsgs*)
Decompress pre-defined compressed fields for each message. Msgs should be unpacked before this step.

fill_reservoir()

get_from_kafka(**args*, ***kw*)
Method called to get and process a batch

get_next_batch()
This method is called from the manager. It must return a list or a generator of BaseRecord objects. When it has nothing else to read, it must set class variable "finished" to True.

set_last_position(*last_position*)
Called from the manager, it is in charge of updating the last position of data commited by the writer, in order to have resume support

supported_options = {'record_count': {'type': (<type 'int'>, <type 'long'>)}, 'group': {'type': (<type 'basestring'>,)

static unpack_messages(*msgs*)

S3Reader (Stream)

class exporters.readers.s3_reader.**S3BucketKeysFetcher**(*reader_options*, *aws_access_key_id*, *aws_secret_access_key*)

Bases: object

pending_keys()

class exporters.readers.s3_reader.**S3Reader**(**args*, ***kwargs*)

Bases: *exporters.readers.base_stream_reader.StreamBasedReader*

Reads items from keys located in S3 buckets and compressed with gzip with a common path.

- **bucket (str)** Name of the bucket to retrieve items from.
- **aws_access_key_id (str)** Public access key to the s3 bucket.
- **aws_secret_access_key (str)** Secret access key to the s3 bucket.
- **prefix (str)** Prefix of s3 keys to be read.
- **prefix_pointer (str)** Prefix pointing to the last version of dataset. This adds support for regular exports. For example:

  We have a weekly export set with CRON. If we wanted to point to a new data prefix every week, we should keep updating the export configuration. With a pointer, we can set the reader to read from that key, which contains one or several lines with valid prefixes to datasets, so only that pointer file should be updated.
- **pattern (str)** S3 key name pattern (REGEX). All files that don't match this regex string will be discarded by the reader.

get_read_streams()

open_stream(*stream*)

supported_options = {'pattern': {'default': None, 'type': (<type 'basestring'>,)}, 'bucket': {'type': (<type 'basestrin

exporters.readers.s3_reader.**format_prefixes**(*prefixes*, *start*, *end*)

exporters.readers.s3_reader.**get_bucket**(*bucket*, *aws_access_key_id*, *aws_secret_access_key*, ***kwargs*)

exporters.readers.s3_reader.**patch_http_response_read**(*func*)

HubstorageReader

class exporters.readers.hubstorage_reader.**HubstorageReader**(**args*, ***kwargs*)

Bases: exporters.readers.base_reader.BaseReader

This reader retrieves items from Scrapinghub Hubstorage collections.

- **batch_size (int)** Number of items to be returned in each batch
- **apikey (str)** API key with access to the project where the items are being generated.
- **project_id (int or str)** Id of the project.
- **collection_name (str)** Name of the collection of items.
- **count (int)** Number of records to read from collection.
- **prefixes (list)** Only include records with given key prefixes.
- **exclude_prefixes (list)** Exclude records with given key prefixes.

- **secondary_collections (list)** A list of secondary collections to merge from.
- **startts (int or str)** Either milliseconds since epoch, or date string.
- **endts (int or str)** Either milliseconds since epoch, or date string.

get_next_batch()
This method is called from the manager. It must return a list or a generator of BaseRecord objects. When it has nothing else to read, it must set class variable "finished" to True.

set_last_position(*last_position*)
Called from the manager, it is in charge of updating the last position of data commited by the writer, in order to have resume support

supported_options = {'count': {'default': 0, 'type': (<type 'int'>, <type 'long'>)}, 'secondary_collections': {'default'

Writer

Writers are in charge of writing batches of items to final destination. All writers are subclasses of BaseWriter class, and must implement:

- **write(dump_path, group_key=None)** The manager calls this method. It consumes a dump_path, which is the path of an items buffer file compressed with gzip. It also has an optional group_key, which provides information regarding the group membership of the items contained in that file.

All writers have also the following common options:

- **items_per_buffer_write** Number of items to be written before a buffer flush takes place.
- **size_per_buffer_write** Size of buffer files before being flushed.
- **items_limit** Number of items to be written before ending the export process. This is useful for testing exports.

class exporters.writers.base_writer.**BaseWriter**(*options*, *metadata*, **args*, ***kwargs*)
Bases: exporters.pipeline.base_pipeline_item.BasePipelineItem

This module receives a batch and writes it where needed.

close()
Close all buffers, cleaning all temporary files.

finish_writing()
This method is hook for operations to be done after everything has been written (e.g. consistency checks, write a checkpoint, etc).

The default implementation calls self._check_write_consistency if option check_consistency is True.

flush()
Ensure all remaining buffers are written.

get_all_metadata(*module='writer'*)

get_metadata(*key*, *module='writer'*)

grouping_info

hash_algorithm = None

increment_written_items()

set_metadata(*key*, *value*, *module='writer'*)

supported_options = {'write_buffer': {'default': 'exporters.write_buffers.base.WriteBuffer', 'type': (<type 'basestri

update_metadata(*data*, *module='writer'*)

write(*path*, *key*)
Receive path to buffer file and group key and write its contents to the configured destination.

Should be implemented in derived classes.

It's called when it's time to flush a buffer, usually by either write_batch() or flush() methods.

write_batch(*batch*)
Buffer a batch of items to be written and update internal counters.

Calling this method doesn't guarantee that all items have been written. To ensure everything has been written you need to call flush().

exception exporters.writers.base_writer.**InconsistentWriteState**
Bases: exceptions.Exception

This exception is thrown when write state is inconsistent with expected final state

exception exporters.writers.base_writer.**ItemsLimitReached**
Bases: exceptions.Exception

This exception is thrown when the desired items number has been reached

Provided writers

ConsoleWriter

class exporters.writers.console_writer.**ConsoleWriter**(*options*, **args*, ***kwargs*)
Bases: exporters.writers.base_writer.BaseWriter

It is just a writer with testing purposes. It prints every item in console.

It has no other options.

close()

supported_options = {'write_buffer': {'default': 'exporters.write_buffers.base.WriteBuffer', 'type': (<type 'basestri

write_batch(*batch*)

S3Writer

class exporters.writers.s3_writer.**S3Writer**(*options*, **args*, ***kwargs*)
Bases: exporters.writers.filebase_base_writer.FilebaseBaseWriter

Writes items to S3 bucket. It is a File Based writer, so it has filebase option available

- **bucket (str)** Name of the bucket to write the items to.
- **aws_access_key_id (str)** Public acces key to the s3 bucket.
- **aws_secret_access_key (str)** Secret access key to the s3 bucket.
- **filebase (str)** Base path to store the items in the bucket.
- **aws_region (str)** AWS region to connect to.
- **save_metadata (bool)** Save key's items count as metadata. Default: True
- **filebase** Path to store the exported files

close()
Called to clean all possible tmp files created during the process.

get_file_suffix(*path*, *prefix*)

supported_options = {'filebase': {'type': (<type 'basestring'>,)}, 'write_buffer': {'default': 'exporters.write_buffers.

write(*dump_path*, *group_key=None*, *file_name=None*)

exporters.writers.s3_writer.**multipart_upload**(**args*, ***kwds*)

exporters.writers.s3_writer.**should_use_multipart_upload**(*path*, *bucket*)

FTPWriter

class exporters.writers.ftp_writer.**FTPWriter**(*options*, **args*, ***kwargs*)
Bases: exporters.writers.filebase_base_writer.FilebaseBaseWriter

Writes items to FTP server. It is a File Based writer, so it has filebase option available

- **host (str)** Ftp server ip
- **port (int)** Ftp port
- **ftp_user (str)** Ftp user
- **ftp_password (str)** Ftp password
- **filebase (str)** Path to store the exported files

build_ftp_instance()

supported_options = {'filebase': {'type': (<type 'basestring'>,)}, 'write_buffer': {'default': 'exporters.write_buffers.

write(**args*, ***kw*)

exception exporters.writers.ftp_writer.**FtpCreateDirsException**
Bases: exceptions.Exception

SFTPWriter

class exporters.writers.sftp_writer.**SFTPWriter**(*options*, **args*, ***kwargs*)
Bases: exporters.writers.filebase_base_writer.FilebaseBaseWriter

Writes items to SFTP server. It is a File Based writer, so it has filebase option available

- **host (str)** SFtp server ip
- **port (int)** SFtp port
- **sftp_user (str)** SFtp user
- **sftp_password (str)** SFtp password
- **filebase (str)** Path to store the exported files

supported_options = {'filebase': {'type': (<type 'basestring'>,)}, 'write_buffer': {'default': 'exporters.write_buffers.

write(**args*, ***kw*)

FSWriter

class exporters.writers.fs_writer.**FSWriter**(*options*, **args*, ***kwargs*)
Bases: exporters.writers.filebase_base_writer.FilebaseBaseWriter

Writes items to local file system files. It is a File Based writer, so it has filebase option available

- **filebase (str)** Path to store the exported files

get_file_suffix(*path*, *prefix*)
Gets a valid filename

supported_options = {'filebase': {'type': (<type 'basestring'>,)}, 'write_buffer': {'default': 'exporters.write_buffers.

write(*dump_path*, *group_key=None*, *file_name=None*)

MailWriter

class exporters.writers.mail_writer.**MailWriter**(*options*, **args*, ***kwargs*)
Bases: exporters.writers.base_writer.BaseWriter

Send emails with items files attached

- **email (str)** Email address where data will be sent
- **subject (str)** Subject of the email
- **from (str)** Sender of the email
- **max_mails_sent (str)** maximum amount of emails that will be sent

send_mail(**args*, ***kw*)

supported_options = {'access_key': {'type': (<type 'basestring'>,), 'env_fallback': 'EXPORTERS_MAIL_AWS_AC

write(*dump_path*, *group_key=None*, *file_name=None*)

exception exporters.writers.mail_writer.**MaxMailsSent**
Bases: exceptions.Exception

This exception is thrown when we try to send more than allowed mails number

AggregationWriter

CloudsearchWriter

class exporters.writers.cloudsearch_writer.**CloudSearchWriter**(*options*, **args*, ***kwargs*)
Bases: exporters.writers.base_writer.BaseWriter

This writer stores items in CloudSearch Amazon Web Services service (https://aws.amazon.com/es/cloudsearch/)

- **endpoint_url** Document Endpoint (e.g.: http://doc-movies-123456789012.us-east-1.cloudsearch.amazonaws.com)
- **id_field** Field to use as identifier
- **access_key** Public acces key to the s3 bucket.
- **secret_key** Secret access key to the s3 bucket.

supported_options = {'access_key': {'default': None, 'type': (<type 'basestring'>,), 'env_fallback': 'EXPORTERS_C

write(**args*, ***kw*)

exporters.writers.cloudsearch_writer.**create_document_batches**(*jsonlines*, *id_field*, *max_batch_size=5000000*)

Create batches in expected AWS Cloudsearch format, limiting the byte size per batch according to given max_batch_size

See: http://docs.aws.amazon.com/cloudsearch/latest/developerguide/preparing-data.html

GDriveWriter

class exporters.writers.gdrive_writer.**GDriveWriter**(**args*, ***kwargs*)

Bases: exporters.writers.filebase_base_writer.FilebaseBaseWriter

Writes items to Google Drive account. It is a File Based writer, so it has filebase

- **client_secret (object)** JSON object containing client secrets (client-secret.json) file obtained when creating the google drive API key.
- **credentials (object)** JSON object containing credentials, obtained by authenticating the application using the bin/get_gdrive_credentials.py ds script
- **filebase (str)** Path to store the exported files

get_file_suffix(*path*, *prefix*)

Gets a valid filename

supported_options = {'filebase': {'type': (<type 'basestring'>,)}, 'write_buffer': {'default': 'exporters.write_buffers.

write(**args*, ***kw*)

GStorageWriter

class exporters.writers.gstorage_writer.**GStorageWriter**(*options*, **args*, ***kwargs*)

Bases: exporters.writers.filebase_base_writer.FilebaseBaseWriter

Writes items to Google Storage buckets. It is a File Based writer, so it has filebase option available

- **filebase (str)** Path to store the exported files
- **project (str)** Valid project name
- **bucket (str)** Google Storage bucket name
- **credentials (str or dict)** Object with valid Google credentials, could be set using env variable EXPORTERS_GSTORAGE_CREDS_RESOURCE which should include reference to credentials JSON file installed with setuptools. This reference should have form "package_name:file_path"

supported_options = {'filebase': {'type': (<type 'basestring'>,)}, 'write_buffer': {'default': 'exporters.write_buffers.

write(*dump_path*, *group_key=None*, *file_name=None*)

write_stream(*stream*, *file_obj*)

HubstorageReduceWriter

class exporters.writers.hs_reduce_writer.**HubstorageReduceWriter**(**args*, ***kwargs*)

Bases: *exporters.writers.reduce_writer.ReduceWriter*

This writer allow exporters to make aggregation of items data and push results into Scrapinghub Hubstorage collections

- **code (str)** Python code defining a reduce_function(item, accumulator=None)
- **collection_url (str)** Hubstorage Collection URL
- **key (str)** Element key where to push the accumulated result
- **apikey (dict)** Hubstorage API key

finish_writing()

get_result(***extra*)

supported_options = {'write_buffer': {'default': 'exporters.write_buffers.base.WriteBuffer', 'type': (<type 'basestri

write_batch(*batch*)

ReduceWriter

class exporters.writers.reduce_writer.**ReduceWriter**(**args*, ***kwargs*)

Bases: exporters.writers.base_writer.BaseWriter

This writer allow exporters to make aggregation of items data and print the results

- **code (str)** Python code defining a reduce_function(item, accumulator=None)

reduced_result

supported_options = {'write_buffer': {'default': 'exporters.write_buffers.base.WriteBuffer', 'type': (<type 'basestri

write_batch(*batch*)

exporters.writers.reduce_writer.**compile_reduce_function**(*reduce_code*, *source_path=None*)

OdoWriter

HubstorageWriter

class exporters.writers.hubstorage_writer.**HubstorageWriter**(**args*, ***kwargs*)

Bases: exporters.writers.base_writer.BaseWriter

This writer sends items into Scrapinghub Hubstorage collection.

- **apikey (str)** API key with access to the project where the items are being generated.
- **project_id (str)** Id of the project.
- **collection_name (str)** Name of the collection of items.
- **key_field (str)** Record field which should be used as Hubstorage item key

flush()

supported_options = {'write_buffer': {'default': 'exporters.write_buffers.base.WriteBuffer', 'type': (<type 'basestri

write_batch(*batch*)

Transform

You can apply some item transformations as a part of an export job. Using this module, read items can be modified or cleaned before being written. To add a new transform module, you must overwrite the following method:

- **transform_batch(batch)** Receives the batch, transforms its items and yields them,

class exporters.transform.base_transform.**BaseTransform**(*options*, *metadata=None*)
Bases: exporters.pipeline.base_pipeline_item.BasePipelineItem

This module receives a batch and writes it where needed. It can implement the following methods:

get_all_metadata(*module='transform'*)

get_metadata(*key*, *module='transform'*)

set_metadata(*key*, *value*, *module='transform'*)

supported_options = {}

transform_batch(*batch*)
Receives the batch, transforms it, and returns it.

update_metadata(*data*, *module='transform'*)

Provided transform

NoTransform

class exporters.transform.no_transform.**NoTransform**(**args*, ***kwargs*)
Bases: exporters.transform.base_transform.BaseTransform

It leaves the batch as is. This is provided for the cases where no transformations are needed on the original items.

supported_options = {}

transform_batch(*batch*)

JqTransform

class exporters.transform.jq_transform.**JQTransform**(**args*, ***kwargs*)
Bases: exporters.transform.base_transform.BaseTransform

It applies jq transformations to items. To see documentation about possible jq transformations please refer to its official documentation.

- **jq_filter (str)** Valid jq filter

supported_options = {'jq_filter': {'type': (<type 'basestring'>,)}}

transform_batch(*batch*)

PythonexpTransform

class exporters.transform.pythonexp_transform.**PythonexpTransform**(**args*, ***kwargs*)
Bases: exporters.transform.base_transform.BaseTransform

It applies python expressions to items.

- **python_expression (str)** Valid python expression

is_valid_python_expression(*python_expressions*)

supported_options = {'python_expressions': {'type': <class 'exporters.utils.list[unicode]'>}}

transform_batch(*batch*)

PythonmapTransform

class exporters.transform.pythonmap.**PythonMapTransform**(**args*, ***kwargs*)
Bases: exporters.transform.base_transform.BaseTransform

Transform implementation that maps items using Python expressions

supported_options = {'map': {'type': (<type 'basestring'>,)}}

transform_batch(*batch*)

FlatsonTransform

class exporters.transform.flatson_transform.**FlatsonTransform**(**args*, ***kwargs*)
Bases: exporters.transform.base_transform.BaseTransform

It flatten a JSON-like dataset into flat CSV-like tables using the Flatson library, please refer to Flatson official documentation.

- **flatson_schema (dict)** Valid Flatson schema

supported_options = {'flatson_schema': {'type': <type 'dict'>}}

transform_batch(*batch*)

Filter

This module receives a batch, filters it according to some parameters, and returns it. It must implement the following method:

- **filter(item)** It receives an item and returns True if the item must be included, or False otherwise

class exporters.filters.base_filter.**BaseFilter**(*options*, *metadata*)
Bases: exporters.pipeline.base_pipeline_item.BasePipelineItem

This module receives a batch, filter it according to some parameters, and returns it.

filter(*item*)
It receives an item and returns True if the filter must be included, or False if not

filter_batch(*batch*)
Receives the batch, filters it, and returns it.

get_all_metadata(*module='filter'*)

get_metadata(*key*, *module='filter'*)

log_at_every = 1000

set_metadata(*key*, *value*, *module='filter'*)

supported_options = {}

update_metadata(*data*, *module='filter'*)

Provided filters

NoFilter

class exporters.filters.no_filter.**NoFilter**(**args*, ***kwargs*)
Bases: exporters.filters.base_filter.BaseFilter

It leaves the batch as is. This is provided for the cases where no filters are needed on the original items.

filter_batch(*batch*)

supported_options = {}

KeyValueFilter

KeyValueRegex

PythonExpeRegex

class exporters.filters.pythonexp_filter.**PythonexpFilter**(**args*, ***kwargs*)
Bases: exporters.filters.base_filter.BaseFilter

Filter items depending on python expression. This is NOT sure, so make sure you only use it in contained environments

- **python_expression (str)** Python expression to filter by
- **imports(dict)** An object with neede imports for expressions

filter(*item*)

supported_options = {'imports': {'default': {}, 'type': <type 'dict'>}, 'python_expression': {'type': (<type 'basestrin

exporters.filters.pythonexp_filter.**load_imports**(*imports*)

Persistence

This module is in charge of resuming support. It persists the current state of the read and written items, and inform of that state on demand. It's usually called from an export manager, and must implement the following methods:

- **get_last_position()** Returns the last commited position
- **commit_position(last_position)** Commits a position that has been through all the pipeline. Position can be any serializable object. This supports both usual position abstractions (number of batch) or specific abstractions such as offsets in Kafka (which are dicts)
- **generate_new_job()** Creates and instantiates all that is needed to keep persistence (tmp files, remote connections...)
- **close()** Cleans tmp files, closes remote connections...
- **configuration_from_uri(uri, regex)** returns a configuration object

It must also define a *uri_regex* to help the module find a previously created resume abstraction.

class exporters.persistence.base_persistence.**BasePersistence**(*options*, *metadata*)
Bases: exporters.pipeline.base_pipeline_item.BasePipelineItem

Base module for persistence modules

close()
Cleans tmp files, close remote connections...

commit_position(*last_position*)
Commits a position that has been through all the pipeline. Position can be any serializable object. This support both usual position abstractions (number of batch) of specific abstractions such as offsets in Kafka (which are a dict).

static configuration_from_uri(*uri*, *regex*)
returns a configuration object.

delete()

generate_new_job()
Creates and instantiates all that is needed to keep persistence (tmp files, remote connections...).

get_all_metadata(*module='persistence'*)

get_last_position()
Returns the last commited position.

get_metadata(*key*, *module='persistence'*)

set_metadata(*key*, *value*, *module='persistence'*)

supported_options = {}

update_metadata(*data*, *module='persistence'*)

Provided persistence

PicklePersistence

class exporters.persistence.pickle_persistence.**PicklePersistence**(**args*, ***kwargs*)
Bases: exporters.persistence.base_persistence.BasePersistence

Manages persistence using pickle module loading and dumping as a backend.

- **file_path (str)** Path to store the pickle file

close()

commit_position(*last_position=None*)

static configuration_from_uri(*uri*, *uri_regex*)
returns a configuration object.

delete()

generate_new_job()

get_last_position()

supported_options = {'file_path': {'default': '.', 'type': (<type 'basestring'>,)}}

uri_regex = 'pickle:(([a-zA-Z\\d-]|\\/)+)'

AlchemyPersistence

class exporters.persistence.alchemy_persistence.**BaseAlchemyPersistence**(**args*, ***kwargs*)
Bases: exporters.persistence.base_persistence.BasePersistence

PROTOCOL = None

classmethod build_db_conn_uri(***kwargs*)
Build the database connection URI from the given keyword arguments

close()

commit_position(*last_position=None*)

classmethod configuration_from_uri(*persistence_uri*)
Return a configuration object.

generate_new_job()

get_last_position()

classmethod parse_persistence_uri(*persistence_uri*)
Parse a database URI and the persistence state ID from the given persistence URI

persistence_uri_re = '(?P<proto>[a-z]+)://(?P<user>.+):(?P<password>.+)@(?P<host>.+):(?P<port>\\d+)/(?P<data

supported_options = {'database': {'type': (<type 'basestring'>,)}, 'host': {'type': (<type 'basestring'>,)}, 'user': {'ty

class exporters.persistence.alchemy_persistence.**Job**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

configuration

id

job_finished

last_committed

last_position

class exporters.persistence.alchemy_persistence.**MysqlPersistence**(**args*, ***kwargs*)
Bases: exporters.persistence.alchemy_persistence.BaseAlchemyPersistence

Manage export persistence using a mysql database as a backend. It will add a row for every job in a table called Jobs.

•user (str)

Username with access to mysql database

•password (str)

Password string

•host (str)

DB server host ip

•port (int)

DB server port

•database (str)

Name of the database in which store jobs persistence

PROTOCOL = 'mysql'

supported_options = {'database': {'type': (<type 'basestring'>,)}, 'host': {'type': (<type 'basestring'>,)}, 'user': {'ty

class exporters.persistence.alchemy_persistence.**PostgresqlPersistence**(**args*, ***kwargs*)

Bases: exporters.persistence.alchemy_persistence.BaseAlchemyPersistence

Manage export persistence using a postgresql database as a backend. It will add a row for every job in a table called Jobs.

•user (str)

Username with access to postgresql database

•password (str)

Password string

•host (str)

DB server host ip

•port (int)

DB server port

•database (str)

Name of the database in which store jobs persistence

PROTOCOL = 'postgresql'

supported_options = {'database': {'type': (<type 'basestring'>,)}, 'host': {'type': (<type 'basestring'>,)}, 'user': {'ty

class exporters.persistence.alchemy_persistence.**SqlitePersistence**(**args*, ***kwargs*)

Bases: exporters.persistence.alchemy_persistence.BaseAlchemyPersistence

Manage export persistence using a postgresql database as a backend. It will add a row for every job in a table called Jobs.

•user (str)

Username with access to postgresql database

•password (str)

Password string

•host (str)

DB server host ip

•port (int)

DB server port

•database (str)

Name of the database in which store jobs persistence

PROTOCOL = 'postgresql'

classmethod build_db_conn_uri(***kwargs*)

persistence_uri_re = '(?P<proto>sqlite)://(?P<database>.+):(?P<job_id>\\d+)'

supported_options = {'database': {'type': (<type 'basestring'>,)}, 'host': {'default': '', 'type': (<type 'basestring'>,)

Notifications

You can define notifications for main export job events such as starting an export, ending or failing. These events can be sent to multiple destinations by adding proper modules to an export configuration.

class exporters.notifications.base_notifier.**BaseNotifier**(*options*, *metadata*, **args*, ***kwargs*)

Bases: exporters.pipeline.base_pipeline_item.BasePipelineItem

This module takes care of notifications delivery. It has a slightly different architecture than the others due to the support of multiple notification endpoints to be loaded at the same time. As you can see in the provided example, the notifications parameter is an array of notification objects. To extend and add notification endpoints, they can implement the following methods:

get_all_metadata(*module='notifier'*)

get_metadata(*key*, *module='notifier'*)

notify_complete_dump(*receivers=None*)
Notifies the end of a dump to the receivers

notify_failed_job(*mgs*, *stack_trace*, *receivers=None*)
Notifies the failure of a dump to the receivers

notify_start_dump(*receivers=None*)
Notifies the start of a dump to the receivers

set_metadata(*key*, *value*, *module='notifier'*)

supported_options = {}

update_metadata(*data*, *module='notifier'*)

Provided notifications

SESMailNotifier

WebhookNotifier

class exporters.notifications.webhook_notifier.**WebhookNotifier**(**args*, ***kwargs*)

Bases: exporters.notifications.base_notifier.BaseNotifier

Performs a POST request to provided endpoints

- **endpoints (list)** Endpoints waiting for a start notification

notify_complete_dump(*receivers=None*, *info=None*)

notify_failed_job(*msg*, *stack_trace*, *receivers=None*, *info=None*)

notify_start_dump(*receivers=None*, *info=None*)

supported_options = {'endpoints': {'default': [], 'type': <class 'exporters.utils.list[unicode]'>}}

Grouping

This module adds support for grouping items. It must implement the following methods:

- **group_batch(batch)** It adds grouping info to all the items from a batch. Every item, which is a BaseRecord, has a group_membership attribute that should be updated by this method before yielding it

class exporters.groupers.base_grouper.**BaseGrouper**(*options*, *metadata=None*)
Bases: exporters.pipeline.base_pipeline_item.BasePipelineItem

Base class fro groupers

get_all_metadata(*module='grouper'*)

get_metadata(*key*, *module='grouper'*)

group_batch(*batch*)
Yields items with group_membership attribute filled

set_metadata(*key*, *value*, *module='grouper'*)

supported_options = {}

update_metadata(*data*, *module='grouper'*)

Provided Groupers

KeyFileGrouper

class exporters.groupers.file_key_grouper.**FileKeyGrouper**(**args*, ***kwargs*)
Bases: exporters.groupers.base_grouper.BaseGrouper

Groups items depending on their keys. It adds the group membership information to items.

•**keys (list)** A list of keys to group by

group_batch(*batch*)

supported_options = {'keys': {'type': <class 'exporters.utils.list[unicode]'>}}

NoGrouper

class exporters.groupers.no_grouper.**NoGrouper**(**args*, ***kwargs*)
Bases: exporters.groupers.base_grouper.BaseGrouper

Default group module, used when no grouping strategies are needed.

group_batch(*batch*)

supported_options = {}

PythonExpGrouper

class exporters.groupers.python_exp_grouper.**PythonExpGrouper**(**args*, ***kwargs*)
Bases: exporters.groupers.base_grouper.BaseGrouper

Groups items depending on python expressions. It adds the group membership information to items.

•**python_expressions (list)** A list of python expressions to group by

group_batch(*batch*)

supported_options = {'python_expressions': {'type': <class 'exporters.utils.list[unicode]'>}}

Stats Managers

This module provides support for keeping track of export statistics. A Stats Manager must implement the following methods:

- **iteration_report(times, stats)** It recieves the times spent in every step of the export pipeline iteration, and the aggregated stats
- **final_report(stats)** Usually called at the end of an export job

Provided Stats Managers

BasicStatsManager

class exporters.stats_managers.basic_stats_manager.**BasicStatsManager**(**args*, ***kwargs*)

Bases: exporters.stats_managers.base_stats_manager.BaseStatsManager

Module to be used when no stats tracking is needed. It does nothing for iteration reports, and only prints a debug log message with final stats

final_report()

iteration_report(*times*)

supported_options = {}

LoggingStatsManager

class exporters.stats_managers.logging_stats_manager.**LoggingStatsManager**(**args*, ***kwargs*)

Bases: exporters.stats_managers.basic_stats_manager.BasicStatsManager

This stats manager prints a log message with useful stats and times for every pipeline iteration.

iteration_report(*times*)

supported_options = {}

Export Formatters

Exporters use formatter modules to export in different formats. An export formatter must implement the following method:

- **format(batch)** It adds formatting info to all the items from a batch. Every item, which is a BaseRecord, has a formatted attribute that should be updated by this method before yielding it

Provided Export Formatters

JsonExportFormatter

class exporters.export_formatter.json_export_formatter.**JsonExportFormatter**(**args*, ***kwargs*)

Bases: exporters.export_formatter.base_export_formatter.BaseExportFormatter

This export formatter provides a way of exporting items in JSON format. This one is the default formatter.

- **pretty_print(bool)** If set to True, items will be exported with an ident of 2 and keys sorted, they will exported with a text line otherwise.

file_extension = 'jl'

format(*item*)

format_footer()

format_header()

supported_options = {'jsonlines': {'default': True, 'type': <type 'bool'>}, 'pretty_print': {'default': False, 'type': <t

exporters.export_formatter.json_export_formatter.**default**(*o*)

CSVExportFormatter

class exporters.export_formatter.csv_export_formatter.**CSVExportFormatter**(**args*, ***kwargs*)

Bases: exporters.export_formatter.base_export_formatter.BaseExportFormatter

This export formatter provides a way of exporting items in CSV format. This are the supported options:

- **show_titles(bool)** If set to True, first lines of exported files will have a row of column names
- **fields(list)** List of item fields to be exported
- **schema(dict)** Valid json schema of dataset items
- **delimiter(str)** field delimiter character for csv rows

file_extension = 'csv'

format(*item*)

format_header()

supported_options = {'fields': {'default': [], 'type': <class 'exporters.utils.list[unicode]'>}, 'show_titles': {'default':

XMLExportFormatter

class exporters.export_formatter.xml_export_formatter.**XMLExportFormatter**(**args*, ***kwargs*)

Bases: exporters.export_formatter.base_export_formatter.BaseExportFormatter

This export formatter provides a way of exporting items in XML format

file_extension = 'xml'

format(*item*)

format_footer()

format_header()

supported_options = {'fields_order': {'default': [], 'type': <class 'exporters.utils.list[unicode]'>}, 'item_name': {'def

Decompressors

Decompressors take a compressed stream from a stream reader and return an uncompressed stream. They have to implement one function:

- **decompress(stream)** Decompress the input stream (returns an uncompressed stream)

class exporters.decompressors.**BaseDecompressor**(*options*, *metadata*, **args*, ***kwargs*)
Bases: exporters.pipeline.base_pipeline_item.BasePipelineItem

decompress()

supported_options = {}

class exporters.decompressors.**ZLibDecompressor**(*options*, *metadata*, **args*, ***kwargs*)
Bases: *exporters.decompressors.BaseDecompressor*

decompress(*stream*)

supported_options = {}

class exporters.decompressors.**NoDecompressor**(*options*, *metadata*, **args*, ***kwargs*)
Bases: *exporters.decompressors.BaseDecompressor*

decompress(*stream*)

supported_options = {}

Deserializers

Deserializers take a stream of uncompressed bytes and returns an iterator of records They have to implement one function:

- **deserialize(stream)** Deserialize the input stream (return an iterator of records)

class exporters.deserializers.**BaseDeserializer**(*options*, *metadata*, **args*, ***kwargs*)
Bases: exporters.pipeline.base_pipeline_item.BasePipelineItem

deserialize(*stream*)

supported_options = {}

class exporters.deserializers.**JsonLinesDeserializer**(*options*, *metadata*, **args*, ***kwargs*)
Bases: *exporters.deserializers.BaseDeserializer*

deserialize(*stream*)

supported_options = {}

class exporters.deserializers.**CSVDeserializer**(*options*, *metadata*, **args*, ***kwargs*)
Bases: *exporters.deserializers.BaseDeserializer*

deserialize(*stream*)

supported_options = {}

CHAPTER 4

Contributing

There are many ways to contribute to Exporters project. Here are some of them:

- Report bugs and request features in the issue tracker, trying to follow the guidelines detailed in Reporting bugs below.
- Submit patches for new functionality and/or bug fixes. Please read *Writing patches* and *Submitting patches* below for details on how to write and submit a patch.

Reporting bugs

Well-written bug reports are very helpful, so keep in mind the following guidelines when reporting a new bug.

- check the open issues to see if it has already been reported. If it has, don't dismiss the report but check the ticket history and comments, you may find additional useful information to contribute.
- write complete, reproducible, specific bug reports. The smaller the test case, the better. Remember that other developers won't have your project to reproduce the bug, so please include all relevant files required to reproduce it.

Writing patches

The better written a patch is, the higher chance that it'll get accepted and the sooner that will be merged.

Well-written patches should:

- contain the minimum amount of code required for the specific change. Small patches are easier to review and merge. So, if you're doing more than one change (or bug fix), please consider submitting one patch per change. Do not collapse multiple changes into a single patch. For big changes consider using a patch queue.
- pass all unit-tests. See *Running tests* below.
- include one (or more) test cases that check the bug fixed or the new functionality added. See *Writing tests* below.

- if you're adding or changing a public (documented) API, please include the documentation changes in the same patch. See Documentation policies below.

Submitting patches

The best way to submit a patch is to issue a pull request on Github, optionally creating a new issue first.

Remember to explain what was fixed or the new functionality (what it is, why it's needed, etc). The more info you include, the easier will be for core developers to understand and accept your patch.

You can also discuss the new functionality (or bug fix) before creating the patch, but it's always good to have a patch ready to illustrate your arguments and show that you have put some additional thought into the subject. A good starting point is to send a pull request on Github. It can be simple enough to illustrate your idea, and leave documentation/tests for later, after the idea has been validated and proven useful.

Finally, try to keep aesthetic changes (PEP 8 compliance, unused imports removal, etc) in separate commits than functional changes. This will make pull requests easier to review and more likely to get merged.

Coding style

Please follow these coding conventions when writing code for inclusion in Exporters:

- Unless otherwise specified, follow PEP 8.
- It's OK to use lines longer than 80 chars if it improves the code readability.
- Don't put your name in the code you contribute. Our policy is to keep the contributor's name in the AUTHORS file distributed with Exporters.

Writing modules

All modules contributions must be placed inside contrib folder. Please read *Modules* documentation to know more about every module type specifics.

Tests

Running tests requires tox.

Running tests

Make sure you have a recent enough tox installation:

```
tox --version
```

If your version is older than 1.7.0, please update it first:

```
pip install -U tox
```

To run all tests go to the root directory of Scrapy source code and run:

```
tox
```

To run a specific test (say tests/test_filters.py) use:

```
tox -- tests/test_filters.py
```

To see coverage report install coverage (pip install coverage) and run:

```
coverage report
```

see output of coverage –help for more options like html or xml report.

Writing tests

All functionality (including new features and bug fixes) must include a test case to check that it works as expected, so please include tests for your patches if you want them to get accepted sooner.

Exporters uses unit-tests, which are located in the tests/ directory. Their module name typically resembles the full path of the module they're testing.

Writing contrib tests

Tests for contributed modules will only be executed if TEST_CONTRIB env variable is not set. So add this decorator to them:

```
@unittest.skipUnless(os.getenv('TEST_CONTRIB'), 'disabled contrib test')
```

CHAPTER 5

Documentation

Documentation uses Sphinx.

Policies

Don't use docstrings for documenting classes, or methods which are already documented in the official documentation. Do use docstrings for documenting functions not present in the official documentation.

Editing the docs

1. Install requirements (`watchdog` and `sphinx`):

```
pip install -r requirements/docs.txt
```

2. Compile the docs:

```
make docs
```

You can then open the compiled HTML docs in your browser. You will need to compile whenever you make changes to the .rst files.

3. Edit the doc files.

Read more about reStructuredText syntax.

CHAPTER 6

Exporters Tutorial

In this tutorial, we are going to learn how the new exporters work. The purpose of this project is to have a tool to allow us to export from a wide range of sources to a wide range of targets, allowing us to perform complex filtering and transformations to items.

Let's make a simple export

With this tutorial, we are going to read a topic in a Kafka cluster into an S3 bucket, filtering out non-American comapnies. We need to create a proper configuration file. To do it we create a plain JSON file with the proper data.

Config file

The configuration file should look like this:

```
{
    "exporter_options": {
        "log_level": "DEBUG",
        "logger_name": "export-pipeline",
        "formatter": {
            "name": "exporters.export_formatter.json_export_formatter.
↪JsonExportFormatter",
            "options": {}
        },
        "notifications":[
        ]
    },
    "reader": {
        "name": "exporters.readers.kafka_scanner_reader.KafkaScannerReader",
        "options": {
            "brokers": [LIST OF BROKERS URLS],
            "topic": "your-topic-name",
            "group": "exporters-test"
```

```
        }
    },
    "filter": {
        "name": "exporters.filters.key_value_regex_filter.KeyValueRegexFilter",
        "options": {
            "keys": [
                {"name": "country", "value": "United States"}
            ]
        }
    },
    "writer": {
        "name": "exporters.writers.s3_writer.S3Writer",
        "options": {
            "bucket": "your-bucket",
            "filebase": "tests/export_tutorial_{:%d-%b-%Y}",
            "aws_access_key_id": "YOUR-ACCESS-KEY",
            "aws_secret_access_key": "YOUR-ACCESS-SECRET"
        }
    }
}
```

Save this file as ~/config.json, and you can launch the export job with the following command:

```
python bin/export.py --config ~/config.json
```

GDrive export tutorial

Step by Step GDrvie export

First two steps will only have to be done once:

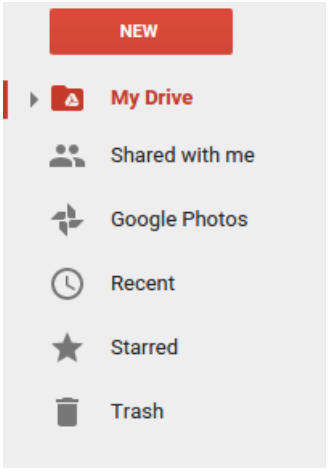
1. Make sure you have your client-secret.json file. If not, follow the steps described here. More info about this file can be found here.
2. Get your credentials file. We have added a script that helps you with this process. Usage:

   ```
   python bin/get_gdrive_credentials.py PATH_TO_CLIENT_SECRET_FILE
   ```

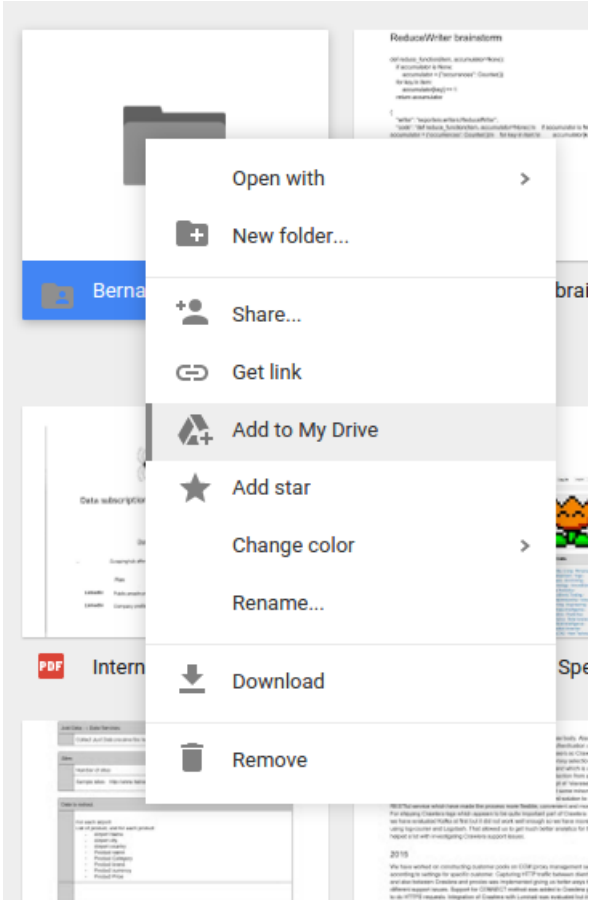
   It will open a tab in your browser where you can login with your Google account. When you do that, the script will print where the credentials file has been created.

   Now for every delivery:

3. Ask the destination owner to create a folder and share it with your Google user.
4. The folder will appear under Shared with me section.



Go there, right click on the shared folder and click on "Add to my drive". This will add the folder the client shared with you in your My Drive. section, which can be seen by exporters.



5. Configure writer filepath to point the client's folder. For example, if client shared with you a folder called "export-data", and you have added to your drive, writer configuration could look like:

```
"writer":{
    "name": "exporters.writers.gdrive_writer.GDriveWriter",
    "options": {
        "filebase": "export-data/gwriter-test_",
        "client_secret": {client-secret.json OBJECT},
```

```
        "credentials": {credentials OBJECT}
    }
}
```

6. To run the export, you could use the bin/export.py:

```
python export.py --config CONFIGPATH
```

Resume export tutorial

Let's assume we have a failed export job, that was using this configuration:

```
{
    "reader": {
        "name": "exporters.readers.random_reader.RandomReader",
        "options": {
        }
    },
    "writer": {
        "name": "exporters.writers.console_writer.ConsoleWriter",
        "options": {

        }
    },
    "persistence":{
        "name": "exporters.persistence.pickle_persistence.PicklePersistence",
        "options": {
            "file_path": "job_state.pickle"
        }
    }
}
```

To resume the export, you must run:

```
python export.py --resume pickle://job_state.pickle
```

Dropbox export tutorial

To get the needed access_token please follow this steps.

1. Go to your dropbox apps section
2. Press *Create app* button.
3. Select *Dropbox Api* and *Full Dropbox* permissions, and set a proper name for your app.
4. Press *Generate* button under *Generated access token*.
5. Use the generated token for your configuration.

```
"writer":{
    "name": "exporters.writers.dropbox_writer.DropboxWriter",
    "options": {
        "access_token": "YOUR_ACCESS_TOKEN",
```

```
        "filebase": "/export/exported_file"
    }
}
```

CHAPTER 7

Credits

This project is maintained by Scrapinghub, you can reach us at info@scrapinghub.com.

Committers

- Bernardo Botella
- Elias Dorneles
- David Bengoa
- Kirill Zaborsky
- Wilfredo Bardales Roncalla
- Martin Olveyra
- Tom Russell
- Łukasz Pawluczuk
- Dennis Again
- Raphael Passini
- Artur Gaspar
- José Ricardo
- Victoria Terenina
- Serhiy Yasko
- Richard Dowinton
- Rocio Aramberri

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

e

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

W

X

Z