DKB Documentation

DKB team

Sep 17, 2020

CONTENTS:

CHAPTER

ONE

PYDKB PACKAGE

Common library for Data Knowledge Base Dataflow stages development.

Dataflow ETL process (extract-transform-load) for populating internal DKB storages and keeping them up to date

Dataflow stage Logical step of ETL process, implemented as standalone executable program (worker)

Dataflow stages are standalone programs, but can be combined into a pipeline by means of Kafka-based supervising program. For details about program compatibility with the supervisor please check documentation for the Metadata Integration Topology Management System (MInT MS) workers[1]. Worker program can be written in any language; `pyDKB` is intended to simplify this process for Python.

Warning: There are three types of stages corresponding three types of ETL operations: *source connector* (data extraction), *processor* (transformation) and *sink connector* (load to internal DKB storage). Currently `pyDKB` library can be used only for *processor* stages, but in future versions *connector* stages will also be supported.

1.1 Quickstart guide

To create simple processor stage application first decide input and output data format. In following examples we will work with data in JSON format (for the full list of supported formats check *pyDKB.dataflow.communication.messages module* section of this documentation).

Now let's start writing example processor `welcome.py` and implement message handler – functional part of the stage (operations to be performed on data flow units):

```python
from pyDKB.dataflow.communication.messages import JSONMessage

def my_process(stage, message):
    """ Single message processing. """
    input_data = message.content()
    name = input_data.get('name')
    if name:
        out_data = {'message': "Welcome, %s!" % name}
        out_message = JSONMessage(out_data)
        stage.output(out_message)
    return True
```

Function must take two arguments: `stage` (stage context object) and `message` (input message, which should be transformed by our stage). Message is a smallest data unit in the data flow running through the processor, and every message is to be processed independently of previous or following ones. `message.content()` and

[1] WIP

`JSONMessage(out_data)` statements are used to decode/encode message to/from Python `dict` object. Message, passed to the function, is taken from the input data flow; to write new message(s) to the output data flow, `stage.output(out_message)` is used. It can be used as many times as many output messages were generated (or once with the list of messages). In our example, messages without key `'name'` will produce no output messages, so `stage.output()` will not be called at all. In terms of data flow it means that the input message is filtered out and will not reach the *sink connector*.

Boolean return value of `my_process()` indicates if the processing was successful or not. If processing failed (`False` is returned), produced output messages will be dropped to avoid loading sketchy information into the DKB storages.

Now as we have processing logic implemented, we need to turn it into fully functional application. Add following lines to `welcome.py`:

```
import sys
from pyDKB.dataflow.stage import ProcessorStage
from pyDKB.dataflow import messageType
from pyDKB.dataflow.communication.messages import JSONMessage

def my_process(stage, message):
    <...function code...>

if __name__ == '__main__':
    stage = ProcessorStage()
    stage.set_input_message_type(messageType.JSON)
    stage.set_output_message_type(messageType.JSON)
    stage.process = my_process
    stage.configure(sys.argv[1:])
    stage.run()
```

First we create stage object: `stage = JSONProcessorStage()`; then indicate that input and output message format is JSON: `stage.set_{input,output}_message_type(messageType.JSON)` (for full list of message types check *pyDKB.dataflow.communication.messages module* section of this documentation); then set stage processing function to our function `my_process()`, parse command line arguments and configure stage instance according to it (`stage.configure(sys.argv[1:])`). Now we ready to start the stage execution.

Easy, right?

It‘s time to run our application. Create input data sample `input.ndjson` with following lines:

```
{"name": "James", "city": "New York"}
{"user": "Jonathan", "role": "support"}
{"name": "John Smith"}
```

and type:

```
$ python welcome.py --dest s input.ndjson
{"message": "Welcome, James!"}
{"message": "Welcome, John Smith!"}
```

`--dest s` indicates that output destination is (s)tdout (default destination is file). For full information about modes in which the stage application can be used, run `python welcome.py -h`.

That‘s it, your first application is ready to be integrated into an ETL process as data processing node. For details about ETL process creation check *MInT Supervisor*[2] documentation.

[2] WIP

1.2 Subpackages

1.2.1 pyDKB.atlas package

Functionality required only for ATLAS metadata processing.

Submodules

pyDKB.atlas.misc module

Miscellaneous functions required to operate with ATLAS metadata objects.

`pyDKB.atlas.misc.`**`dataset_data_format`**`(`*datasetname*`)`

Extract data format from dataset name.

According to dataset naming nomenclature: https://dune.bnl.gov/w/images/9/9e/Gen-int-2007-001_%28NOMENCLATURE%29.pdf for MC datasets:

mcNN_subProject.datasetNumber.physicsShort.prodStep.dataType.Version

for Real Data: DataNN_subProject.runNumber.streamName.prodStep.dataType.Version

In both cases the dataType field is required.

Parameters **`datasetname`** (`str`) – dataset name

Returns data format, None if *datasetname* is None, empty string, etc.

Return type str

`pyDKB.atlas.misc.`**`dataset_scope`**`(`*dsn*`)`

Extract the first field from the dataset name

Example: mc15_13TeV.XXX mc15_13TeV:YYY.XXX

Parameters **`dsn`** (`str`) – full dataset name

Returns dataset scope

Return type str

`pyDKB.atlas.misc.`**`normalize_dataset_name`**`(`*dsn*`)`

Remove an explicitly stated scope from a dataset name.

According to dataset nomenclature, dataset name cannot include a ':' symbol. If a dataset name is in 'A:B' format, then A, probably, is an explicitly stated scope that should be removed.

Parameters **`dsn`** (`str`) – dataset name

Returns dataset name without explicit scope, unchanged dataset name if it was already normal

Return type str

1.2.2 pyDKB.common package

Common modules.

Submodules

pyDKB.common.LoggableObject module

pyDKB.common.LoggableObject

class pyDKB.common.LoggableObject.**LoggableObject**

Bases: `object`

Common ancestor for all classes that need 'log' method.

classmethod log(*message*, *level=3*)

Output log message with given log level.

Parameters

- **message** (*str*) – message to output
- **level** (`pyDKB.common.types.logLevel` member) – log level of the message

pyDKB.common.Type module

Abstract class for type definitions.

Example

```
>>> myType = Type("Orange", "Apple")
>>> myType.add("Plum")
>>> t = myType.Orange
>>> if t == myType.Orange:
...     print "Orange!"
... elif t == myType.member("Apple"):
...     print "Apple!"
...
Orange!
>>> if not myType.member("Walnut"):
...     print "Wrong type!"
...
Wrong type!
```

class pyDKB.common.Type.**Type**(**args*)

Bases: `object`

Abstract class for type definitions.

Member names (*str*) are passed to the constructor as positional arguments.

add(*name*)

Add member.

Parameters **name** (*str*) – name of the member to be added

hasMember(*val*)

Check if the member exists (by value).

Parameters **val** (*int*) – member to be checked

Returns True/False

Return type bool

member (*name*)
Check if the member exists (by name).

Parameters **name** (*str*) – name to be checked

Returns member value or False if member does not exist

Return type int, bool

memberName (*val*)
Return string name of the member.

Parameters **val** (*int*) – member to retrieve name for

Returns member name of False if member does not exist

Return type str, bool

pyDKB.common.custom_readline module

Implementation of "readline"-like functionality for custom separator.

Todo: make import of `fcntl` (or of this module) optional to avoid errors when library is used under Windows.

`pyDKB.common.custom_readline.`**custom_readline** (*f*, *newline*)
Read lines with custom line separator.

Construct generator with readline-like functionality: with every call of `next()` method it will read data from `f` untill the `newline` separator is found; then yields what was read.

Warning: the last line can be incomplete, if the input data flow is interrupted in the middle of data writing.

To check if iteration is not over without reading next value, one may *send(True)* to the generator: it will return *True* if there is another message to yield or raise *StopIteration* if nothing left.

Parameters

- **f** (*file*) – readable file object
- **newline** (*str*) – delimeter to be used instead of `\n`

Returns iterable object

Return type generator

Todo:

- make last "line" handling more strict: no `newline` == no line;
- rethink function name (as "line" is actually a "message");
- move functionality to `pyDKB.dataflow.communication`[1] submodule)

[1] https://github.com/PanDAWMS/dkb/pull/129

pyDKB.common.exceptions module

Definition of common modules exceptions

exception `pyDKB.common.exceptions.`**`HDFSException`**

Bases: `exceptions.RuntimeError`

Base Exception for HDFS module.

pyDKB.common.hdfs module

Utils to interact with HDFS.

`pyDKB.common.hdfs.`**`File`**(*fname*)

Get and open temporary local copy of HDFS file

Return value: open file object (TemporaryFile).

`pyDKB.common.hdfs.`**`basename`**(*p*)

Return file name without path.

`pyDKB.common.hdfs.`**`check_stderr`**(*proc*, *timeout=None*, *max_lines=1*)

Wait till the end of the subprocess and send its STDERR to STDERR.

Output only MAX_LINES of the STDERR to the current STDERR; if MAX_LINES == None, output all the STDERR.

Return value is the subprocess‘ return code.

`pyDKB.common.hdfs.`**`dirname`**(*p*)

Return dirname without filename.

`pyDKB.common.hdfs.`**`getfile`**(*fname*)

Download file from HDFS.

Return value: file name (without directory)

`pyDKB.common.hdfs.`**`join`**(*p*, **args*)

Join given paths.

`pyDKB.common.hdfs.`**`listdir`**(*dirname*, *mode='a'*)

List files and/or subdirectories of HDFS directory.

Parameters: dirname – directory to list mode – ‘a’: list all objects

‘f’: list files ‘d’: list subdirectories

`pyDKB.common.hdfs.`**`makedirs`**(*dirname*)

Try to create directory (with parents).

`pyDKB.common.hdfs.`**`movefile`**(*fname*, *dest*)

Move local file to HDFS.

`pyDKB.common.hdfs.`**`putfile`**(*fname*, *dest*)

Upload file to HDFS.

pyDKB.common.json_utils module

Utils to work with JSON (dict) objects.

`pyDKB.common.json_utils.`**`nestedKeys`**(*key*)

Transform STRING with nested keys into LIST.

Parameters:

STRING key – dot-separated list of nested keys. If a key contains dot itself, the key must be put between quotation marks.

`pyDKB.common.json_utils.`**`valueByKey`**(*json_data*, *key*)

Return value by a chain (list) of nested keys.

Parameters: DICT json_data – to search in STRING key – dot-separated list of nested keys

pyDKB.common.misc module

pyDKB.common.misc

Miscellanious utility functions.

`pyDKB.common.misc.`**`ensure_argparse_arg_name`**(*args*, *kwargs*)

Ensure cmgline argument name for *argparse.add_argument()*.

Returns the expected argument name and updates `kwargs['dest']` value.

When **:python:func:‘argparse.add_argument()‘** is called, *argparse* decides how to name the new argument basing on the passed parameters (to use this name in *argparse.Namespace* object). This function returns the name, that (supposedly) will be used in *argparse*; but to make sure, `kwargs['dest']` is (re)set to the same value (if applicable).

For parameters description please refer to *argparse* documentation: https://docs.python.org/2.7/library/argparse.html#the-add-argument-method

Parameters

- **`args`** (*`list`*) – list of *argparse.add_argument()* positional parameters
- **`kwargs`** (*`dict`*) – hash of *argparse.add_argument()* keyword parameters

Returns argument name

Return type string

`pyDKB.common.misc.`**`log`**(*message*, *level=3*, **args*)

Output log message with given log level.

In case of multiline messages or list of messages only first line (message) is prepended with provided prefixes and timestamp; in all the next lines (messages) they are replaced with special prefix ‘(==)’, representing that these lines belong to the same log record.

Empty lines and lines containing only whitespace symbols are ignored.

Parameters

- **`message`** (*`object`*) – message to output (string, list of strings or any other object)
- **`level`** (`pyDKB.common.types.logLevel` member) – log level of the message
- **`*args`** – additional prefixes (will be output between log level prefix and message body)

pyDKB.common.types module

pyDKB.common.types

Definitions of types used across all the library modules.

1.2.3 pyDKB.dataflow package

Dataflow organization utils.

Subpackages

pyDKB.dataflow.communication package

pyDKB.dataflow.communication

`pyDKB.dataflow.communication.`**`Message`**(*msg_type*)
Return class XXXMessage, where XXX is the passed type.

Subpackages

pyDKB.dataflow.communication.consumer package

Consumer submodule init file.

`class` `pyDKB.dataflow.communication.consumer.`**`ConsumerBuilder`**(*config={}*)
Bases: `object`

Constructor for Consumer instance.

`build`(*config={}*)
Return constructed consumer.

`consumerClass = None`

`setSource`(*source*)
Set data source for the consumer.

`setType`(*Type*)
Set message type for the consumer.

Submodules

pyDKB.dataflow.communication.consumer.Consumer module

pyDKB.dataflow.communication.consumer.Consumer

`class` `pyDKB.dataflow.communication.consumer.Consumer.`**`Consumer`**(*config={}*)
Bases: *`pyDKB.common.LoggableObject.LoggableObject`*

Data consumer implementation.

`close`()
Close opened data stream and data source.

config = None

get_message()

Get new message from current source.

Return values: Message object False (failed to parse message) None (all input sources are empty)

get_source_info()

Return current source info.

get_stream()

Get input stream linked to the current source.

Return value: InputStream None (no sources left to read from)

init_stream()

Init input stream.

message_class()

Return message class.

message_type = None

next()

Return new Message, read from input stream.

reconfigure(*config={}*)

(Re)initialize consumer with stage config arguments.

reset_stream()

Reset input stream to the current source.

set_message_type(*Type*)

Set input message type.

stream_is_readable()

Check if input data stream is readable.

Returns True – stream is initialized and not empty, False – stream is empty, None – stream is not initialized

Return type bool, NoneType

exception pyDKB.dataflow.communication.consumer.Consumer.**ConsumerException**(*message=”*, *reason=None*)

Bases: *pyDKB.dataflow.exceptions.DataflowException*

Dataflow Consumer exception.

pyDKB.dataflow.communication.consumer.FileConsumer module

pyDKB.dataflow.communication.consumer.FileConsumer

Data consumer implementation for common (static) files.

TODO: think about:

- updatable files
- pipes (better, from the point of StreamConsumer)
- round-robin (for updatable sources)
- ...

class pyDKB.dataflow.communication.consumer.FileConsumer.**FileConsumer**(*config={}*)
Bases: *pyDKB.dataflow.communication.consumer.Consumer.Consumer*

Data consumer implementation for HDFS data source.

current_file = None

get_source()
Get nearest non-empty source (current or next).

get_source_info()
Return current source info.

init_sources()
Initialize sources iterator if not initialized yet.

next_source()
Reset $current_file to the next non-empty file.

Return value: File descriptor of the new $current_file None (no files left)

reconfigure(*config={}*)
(Re)initialize consumer with Stage configuration.

source_is_readable()
Check if current source is readable.

Returns None – no source, False – source is empty / fully read, True – source is defined and is not empty

Return type bool, NoneType

pyDKB.dataflow.communication.consumer.HDFSConsumer module

pyDKB.dataflow.communication.consumer.HDFSConsumer

class pyDKB.dataflow.communication.consumer.HDFSConsumer.**HDFSConsumer**(*config={}*)
Bases: *pyDKB.dataflow.communication.consumer.FileConsumer.FileConsumer*

Data consumer implementation for HDFS data source.

reconfigure(*config={}*)
Configure HDFS Consumer according to the config parameters.

pyDKB.dataflow.communication.consumer.StreamConsumer module

pyDKB.dataflow.communication.consumer.StreamConsumer

Data consumer implementation for a single stream.

TODO: think about multiple streams (like a number of named pipes, etc). Prehaps, even merge this class with FileConsumer.

class pyDKB.dataflow.communication.consumer.StreamConsumer.**StreamConsumer**(*config={}*)
Bases: *pyDKB.dataflow.communication.consumer.Consumer.Consumer*

Data consumer implementation for Stream data source.

fd = None

get_source()
Get Stream file descriptor.

get_source_info()
Return current source info.

next_source()
Return None.

As currenty we believe that there is only one input stream

reconfigure(*config={}*)
(Re)configure Stream consumer.

pyDKB.dataflow.communication.producer package

Producer submodule init file.

class pyDKB.dataflow.communication.producer.**ProducerBuilder**(*config={}*)
Bases: `object`

Constructor for Producer instance.

build(*config={}*)
Return constructed producer.

message_type = None

producerClass = None

setDest(*dest*)
Set data destination for the producer.

setSourceInfoMethod(*src_info*)
Set method to get current source info.

setType(*Type*)
Set message type for the producer.

src_info = None

Submodules

pyDKB.dataflow.communication.producer.FileProducer module

pyDKB.dataflow.communication.producer.FileProducer

Data producer implementation for common (static) files.

TODO: think about:

- pipes (better, from the point of StreamProducer)
- multiple parallel dests
- . . .

class pyDKB.dataflow.communication.producer.FileProducer.**FileProducer**(*config={}*)
Bases: *pyDKB.dataflow.communication.producer.Producer.Producer*

Data producer implementation for local file data dest.

close()
Close opened files and remove temporary one.

`close_file`()
: Close current file.

`config_dir`(*config={}*)
: Configure output directory.

`current_file = None`

`default_dir`()
: Get default directory name.

`dirname`(*dirname=None*)
: Set/get preferable directory name.

`ensure_dir`()
: Ensure that current directory for output files exists.

`file_info`()
: Return output file metadata (name, directory, . . .).

`get_dest`()
: Get destination file descriptor.

`get_dest_info`()
: Get current destination info.

`get_dir`()
: Get current directory for output files.

`get_filename`()
: Return filename, corresponding the source, or timestamp-based.

`get_source_info`()
: Set current data source, if any.

`reconfigure`(*config={}*)
: (Re)configure producer according to the config hash.

`reset_file`()
: Resets current file according to the current source info.

Metadata include:

- fd – open file descriptor
- name – file name
- dir – directory name
- local_path – local path to the file

`set_default_dir`()
: Set default directory name.

`subdir`(*base_dir*, *sub_dir=''*)
: Construct full path for $subdir of $base_dir.

pyDKB.dataflow.communication.producer.HDFSProducer module

pyDKB.dataflow.communication.producer.HDFSProducer

Data producer implementation for common (static) files in HDFS.

TODO: think about:

- pipes (better, from the point of StreamProducer)
- multiple parallel dests
- …

class `pyDKB.dataflow.communication.producer.HDFSProducer.`**`HDFSProducer`**(*config={}*)

Bases: *`pyDKB.dataflow.communication.producer.FileProducer.FileProducer`*

Data producer implementation for HDFS data dest.

`close_file`()
: Close current file and move it to HDFS.

`config_dir`(*config={}*)
: Configure output directory.

`ensure_dir`()
: Ensure that current directory for output files exists.

`file_info`()
: Return output file metadata (name, directory, . . .).

`set_default_dir`()
: Set default directory name.

`subdir`(*base_dir*, *sub_dir=”*)
: Construct full path for $sub_dir of $base_dir.

pyDKB.dataflow.communication.producer.Producer module

pyDKB.dataflow.communication.producer.Producer

class `pyDKB.dataflow.communication.producer.Producer.`**`Producer`**(*config={}*)

Bases: *`pyDKB.common.LoggableObject.LoggableObject`*

Data producer implementation.

`close`()
: Close opened data stream and data dest.

`config = None`

`drop`()
: Drop buffered messages.

`eop`()
: Write EOP marker to the current dest.

`flush`()
: Flush buffered messages to the current dest.

`get_dest`()
: Return current destination.

`get_dest_info`()
: Return current dest info.

`get_stream`(*actualize=True*)
: Get output stream linked to the current dest.

  If $actualize parameter set to True, will try to reset current stream destination; else will use last known destination or None.

init_stream()
: Init output stream (without real destination).

message_class()
: Return message class.

message_type = None

reconfigure(*config={}*)
: (Re)initialize producer with stage config arguments.

reset_stream()
: Reset input stream to the current dest.

set_message_type(*Type*)
: Set input message type.

write(*msg*)
: Put new message to the current dest (buffer).

exception pyDKB.dataflow.communication.producer.Producer.**ProducerException**(*message=”*, *reason=None*)

Bases: *pyDKB.dataflow.exceptions.DataflowException*

Dataflow Producer exception.

pyDKB.dataflow.communication.producer.StreamProducer module

pyDKB.dataflow.communication.producer.StreamProducer

Data producer implementation for a single stream.

TODO: think about multiple streams (like a number of named pipes, etc). Prehaps, even merge this class with FileProducer.

class pyDKB.dataflow.communication.producer.StreamProducer.**StreamProducer**(*config={}*)

Bases: *pyDKB.dataflow.communication.producer.Producer.Producer*

Data producer implementation for Stream data dest.

fd = None

get_dest()
: Get Stream file descriptor.

get_dest_info()
: Return current dest info.

reconfigure(*config={}*)
: (Re)configure Stream producer.

pyDKB.dataflow.communication.stream package

pyDKB.dataflow.communication.stream

class pyDKB.dataflow.communication.stream.**StreamBuilder**(*fd*, *config={}*)

Bases: object

Constructor for Stream object.

build(*config={}*)
Create instance of Stream.

message_type = None

setStream(*stream*)
Set stream type: 'input' or 'output'.

setType(*Type*)
Set message type for the Stream.

streamClass = None

class pyDKB.dataflow.communication.stream.**Stream**(*fd=None*, *config={}*)
Bases: `pyDKB.common.LoggableObject.LoggableObject`

Abstract class for input/output streams.

EOM = None

close()
Close open file descriptors etc.

configure(*config*)
Stream configuration.

get_fd()
Return open file descriptor or raise exception.

message_type()
Get type of the messages in the stream.

reset(*fd*, *close=True*)
Reset file descriptor in operation.

Parameters **fd** – open file descriptor TODO: IOBase objects

Returns previous file descriptor (or None)

set_message_type(*msg_type*)
Set type of the messages in the stream.

class pyDKB.dataflow.communication.stream.**InputStream**(*fd=None*, *config={}*)
Bases: `pyDKB.dataflow.communication.stream.Stream.Stream`

Implementation of the input stream.

get_message()
Get next message from the input stream.

Returns parsed next message, False – parsing failed, None – no messages left

Return type *pyDKB.dataflow.communication.messages.AbstractMessage*, bool, NoneType

is_readable()
Check if current input stream is readable.

Returns None – not initialized, False – empty, True – not empty

Return type bool, NoneType

next()
Get next message from the input stream.

Returns parsed next message, False – parsing failed or unexpected end of stream occurred

Return type *pyDKB.dataflow.communication.messages.AbstractMessage*, bool

parse_message (*message*)
Verify and parse input message.

Parameters **message** (pyDKB.dataflow.communication.messages.AbstractMessage) – message to parse

Returns decoded message or False if parsing failed

Return type *pyDKB.dataflow.communication.messages.AbstractMessage*, bool

reset (*fd*, *close=True*, *force=False*)
Reset current stream with new file descriptor.

Overrides parent method to reset __iterator property.

Parameters

- **fd** (*file*) – new file descriptor
- **close** (*bool*) – if True, close the old file descriptor
- **force** (*bool*) – if True, force the reset of iterator (normally, iterator is not reset if the new file descriptor is the same as the old one)

Returns old file descriptor

Return type file

class pyDKB.dataflow.communication.stream.**OutputStream** (*fd=None*, *config={}*)
Bases: *pyDKB.dataflow.communication.stream.Stream.Stream*

Implementation of the output stream.

configure (*config={}*)
Configure instance.

drop ()
Drop buffer without sending messages anywhere.

eop ()
Signalize Supervisor about end of process.

flush ()
Flush buffer to the output stream.

msg_buffer = []

write (*message*)
Add message to the buffer.

Submodules

pyDKB.dataflow.communication.stream.InputStream module

pyDKB.dataflow.communication.stream.InputStream

class pyDKB.dataflow.communication.stream.InputStream.**InputStream** (*fd=None*, *config={}*)
Bases: *pyDKB.dataflow.communication.stream.Stream.Stream*

Implementation of the input stream.

get_message ()
Get next message from the input stream.

Returns parsed next message, False – parsing failed, None – no messages left

Return type *pyDKB.dataflow.communication.messages.AbstractMessage*, bool, NoneType

is_readable()
Check if current input stream is readable.

Returns None – not initialized, False – empty, True – not empty

Return type bool, NoneType

next()
Get next message from the input stream.

Returns parsed next message, False – parsing failed or unexpected end of stream occurred

Return type *pyDKB.dataflow.communication.messages.AbstractMessage*, bool

parse_message(*message*)
Verify and parse input message.

Parameters message (`pyDKB.dataflow.communication.messages.AbstractMessage`) – message to parse

Returns decoded message or False if parsing failed

Return type *pyDKB.dataflow.communication.messages.AbstractMessage*, bool

reset(*fd*, *close=True*, *force=False*)
Reset current stream with new file descriptor.

Overrides parent method to reset __iterator property.

Parameters

- **fd** (*file*) – new file descriptor
- **close** (*bool*) – if True, close the old file descriptor
- **force** (*bool*) – if True, force the reset of iterator (normally, iterator is not reset if the new file descriptor is the same as the old one)

Returns old file descriptor

Return type file

pyDKB.dataflow.communication.stream.OutputStream module

pyDKB.dataflow.communication.stream.OutputStream

class pyDKB.dataflow.communication.stream.OutputStream.**OutputStream**(*fd=None*, *config={}*)

Bases: *pyDKB.dataflow.communication.stream.Stream.Stream*

Implementation of the output stream.

configure(*config={}*)
Configure instance.

drop()
Drop buffer without sending messages anywhere.

eop()
Signalize Supervisor about end of process.

flush()
Flush buffer to the output stream.

msg_buffer = []

write(*message*)
Add message to the buffer.

pyDKB.dataflow.communication.stream.Stream module

pyDKB.dataflow.commuication.stream.Stream

class pyDKB.dataflow.communication.stream.Stream.**Stream**(*fd=None*, *config={}*)
Bases: *pyDKB.common.LoggableObject.LoggableObject*

Abstract class for input/output streams.

EOM = None

close()
Close open file descriptors etc.

configure(*config*)
Stream configuration.

get_fd()
Return open file descriptor or raise exception.

message_type()
Get type of the messages in the stream.

reset(*fd*, *close=True*)
Reset file descriptor in operation.

Parameters **fd** – open file descriptor TODO: IOBase objects

Returns previous file descriptor (or None)

set_message_type(*msg_type*)
Set type of the messages in the stream.

pyDKB.dataflow.communication.stream.exceptions module

pyDKB.dataflow.communication.stream.exceptions

exception pyDKB.dataflow.communication.stream.exceptions.**StreamException**(*message=''*, *reason=None*)
Bases: *pyDKB.dataflow.exceptions.DataflowException*

Exception for Stream operations.

Submodules

pyDKB.dataflow.communication.messages module

pyDKB.dataflow.communication.messages

Definition of abstract message class and specific message classes

class pyDKB.dataflow.communication.messages.**AbstractMessage**(*message=None*)

Bases: `object`

Abstract message

content()

Return message content.

decode(*code*)

Decode original from CODE to TYPE-specific format.

Raises ValueError

decoded = None

encode(*code*)

Encode original message from TYPE-specific format to CODE.

Raises ValueError

encoded = None

classmethod extension()

Return file extension corresponding to this message type.

getOriginal()

Return original message.

incompl = None

incomplete(*status=None*)

Set message incomplete marker and/or get previous/current value.

Parameters status (*bool, NoneType*) – new status (if not passed, current status is returned)

Returns incomplete marker status (previous value, if reset)

Return type bool

msg_type = None

native_types = []

classmethod typeName()

Return message type name as string.

exception pyDKB.dataflow.communication.messages.**DecodeUnknownType**(*code, cls*)

Bases: `exceptions.NotImplementedError`

Exception to be thrown when message type is not decodable.

exception pyDKB.dataflow.communication.messages.**EncodeUnknownType**(*code, cls*)

Bases: `exceptions.NotImplementedError`

Exception to be thrown when message type is not encodable.

class pyDKB.dataflow.communication.messages.**JSONMessage**(*message=None*)

Bases: *pyDKB.dataflow.communication.messages.AbstractMessage*

Message in JSON format.

decode(*code=1*)

Decode original data as JSON.

encode(*code=1*)

Encode JSON as CODE.

incompl_key = '_incomplete'

msg_type = 2

native_types = [<type 'dict'>, <type 'list'>, <type 'int'>, <type 'float'>]

pyDKB.dataflow.communication.messages.**Message**(*msg_type*)
Return class XXXMessage, where XXX is the passed type.

class pyDKB.dataflow.communication.messages.**TTLMessage**(*message=None*)
Bases: *pyDKB.dataflow.communication.messages.AbstractMessage*

Messages in TTL format

Single message = single TTL statement

decode(*code=1*)
Decode original data as TTL.

Currently takes text as it is. TODO: check some formal matter to confirm the string is TTL.

encode(*code=1*)
Encode TTL as CODE.

msg_type = 3

native_types = [<type 'str'>, <type 'unicode'>]

pyDKB.dataflow.stage package

Stage submodule init file.

class pyDKB.dataflow.stage.**ProcessorStage**(*description='DKB Dataflow data processing stage.'*)
Bases: *pyDKB.dataflow.stage.AbstractStage.AbstractStage*

Abstract class to implement Processor stages

Processor stage – is a stage for data processing/transformation.

Class/instance variable description:

- **communication.consumer.Consumer instance** __input
- Generator object for output file descriptor OR file descriptor (for (s)tream mode)

  __output
- **List of objects to be "stopped"** __stoppable

clear_buffer()
Drop buffered output messages.

configure(*args=None*)
Configure stage according to the config parameters.

If $args specified, arguments will be parsed anew.

defaultArguments()
Default parser configuration.

flush_buffer()
Flush message buffer to the output.

forward()
Send EOPMarker to the output stream.

get_source_info()
Get information about current source.

input()
Generator for input messages.

Returns iterable object. Every iteration returns single input message to be processed.

input_message_class()
Get input message class.

output(*message*)
Put the (list of) message(s) to the output buffer.

output_message_class()
Get output message class.

static process(*stage*, *input_message*)
Transform input_message -> output_message.

To be implemented individually for every stage. Takes the stage as first argument to allow calling output()

from inside the function.

Return value: True – processing successfully finished False – processing failed (skip the input message)

run()
Run process() for every input() message.

set_default_arguments(*ignore_on_skip=False*, ***kwargs*)
Set (or overwrite) default values for arguments.

Parameters

- **ignore_on_skip** (`bool`) – ignore new default value when the stage is to be skipped
- **<arg_name>** (`object`) – default value for argument <arg_name>

set_input_message_type(*Type=None*)
Set input message type.

set_output_message_type(*Type=None*)
Set output message class.

static skip_process(*stage*, *input_message*)
Process input_message in “skip” processing mode.

Skip mode is turned on with command line parameter *–skip*; in this mode stage is expected to skip its “semantic” part of the message transformation (like changes in data fields format, definition of additional data fields, etc) but perform actions required to keep the dataflow seamless.

The simpliest way to achieve this is to send the input message to the output without changes (marking it as “incomplete”), and this is the default implementation of the method.

In some cases it may be necessary to re-implement it (just like *process()*) to keep the dataflow unbroken.

NOTE: the output messages in “skip” mode MUST be marked as “incomplete”.

Parameters **input_message** (*pyDKB.messages.AbstractMessage*) – message to process

Returns True/False (success/failure)

Return type bool

stop()
Finalize all the processes and prepare to exit.

Submodules

pyDKB.dataflow.stage.AbstractStage module

Definition of an abstract class for Dataflow Stages.

class `pyDKB.dataflow.stage.AbstractStage.`**AbstractStage**(*description='DKB Dataflow stage'*)
Bases: *`pyDKB.common.LoggableObject.LoggableObject`*

Class/instance variable description: * Argument parser (argparse.ArgumentParser)

__parser

- Original default argument values (dict)
- **(filled by `add_argument()` method)** _default_args
- **Parsed arguments (argparse.Namespace)** ARGS
- **Stage config parser (ConfigParser.SafeConfigParser)** __config
- **Stage custom config (defaultdict(defaultdict(str)))** CONFIG

add_argument(**args*, ***kwargs*)
Add specific (not common) arguments.

args_error(*message*)
Output USAGE, error message and exit with code 2.

config_error(*message='Failed to read config file:'*)
Output error message and exit with code 3.

defaultArguments()
Config argument parser with parameters common for all stages.

log_configuration()
Log stage configuration.

output_error(*message=None*, *exc_info=None*)
Output traceback of the passed (or last) error with *message*.

parse_args(*args*)
Parse arguments and set dependant arguments if needed.

Exits in case of error with code: 2 – failed to parse arguments 3 – failed to read config file

print_usage(*fd=<open file '<stderr>', mode 'w'>*)
Print usage message.

read_config()
Read stage custom config file.

Returns (True|False)

reset_default_arguments(*args=None*)
Reset default argument values to the original ones.

Original default value is a value passed to the `add_argument()` method.

Parameters **args** (*list, NoneType*) – list of arguments to be reset. If not specified or set to None, all known arguments will be reset

run()
Run the stage.

set_default_arguments(***kwargs*)
Set (or overwrite) default values for arguments.

Parameters **<arg_name>** (*object*) – default value for argument <arg_name>

set_error(*err_type*, *err_val*, *err_trace*)
Set object *_err* variable from the last error info.

stop()
Stop running processes and output error information.

pyDKB.dataflow.stage.ProcessorStage module

Definition of an abstract class for Dataflow Data Processing Stages.

USAGE: ProcessorStage [<options>] [<input files>]

OPTIONS:

-s, --source	{f|s|h} - where to get data from: local (f)iles, (s)tdin, (h)dfs
-i, --input-dir	DIR - base directory for relative input file names (for local and HDFS sources). If <input files> not specified, all files from the directory will be taken as the input.
-d, --dest	{f|s|h} - where to send data to: local (f)iles, (s)tdout, (h)dfs
-o, --output-dir	DIR - base directory for output files (for local and HDFS sources)
--hdfs	• equivalent to "–source h –dest h"
-m, --mode	MODE - MODE: (f)ile = –source f **–dest f (can be** **rewritten with 's'** or 'h') **(s)tream = –source s (can be** rewritten with 'h') –dest s **(m)apreduce = –source s (can be** rewritten with 'h') –dest s

class pyDKB.dataflow.stage.ProcessorStage.**ProcessorStage**(*description='DKB Dataflow data processing stage.'*)

Bases: *pyDKB.dataflow.stage.AbstractStage.AbstractStage*

Abstract class to implement Processor stages

Processor stage – is a stage for data processing/transformation.

Class/instance variable description:

- **communication.consumer.Consumer instance** __input
- Generator object for output file descriptor OR file descriptor (for (s)tream mode)

  __output
- **List of objects to be "stopped"** __stoppable

`clear_buffer`()
Drop buffered output messages.

`configure`(*args=None*)
Configure stage according to the config parameters.

If $args specified, arguments will be parsed anew.

`defaultArguments`()
Default parser configuration.

`flush_buffer`()
Flush message buffer to the output.

`forward`()
Send EOPMarker to the output stream.

`get_source_info`()
Get information about current source.

`input`()
Generator for input messages.

Returns iterable object. Every iteration returns single input message to be processed.

`input_message_class`()
Get input message class.

`output`(*message*)
Put the (list of) message(s) to the output buffer.

`output_message_class`()
Get output message class.

`static process`(*stage*, *input_message*)
Transform input_message -> output_message.

To be implemented individually for every stage. Takes the stage as first argument to allow calling output()

from inside the function.

Return value: True – processing successfully finished False – processing failed (skip the input message)

`run`()
Run process() for every input() message.

`set_default_arguments`(*ignore_on_skip=False*, ***kwargs*)
Set (or overwrite) default values for arguments.

Parameters

- **`ignore_on_skip`** (`bool`) – ignore new default value when the stage is to be skipped
- **`<arg_name>`** (`object`) – default value for argument <arg_name>

`set_input_message_type`(*Type=None*)
Set input message type.

set_output_message_type(*Type=None*)
Set output message class.

static skip_process(*stage*, *input_message*)
Process input_message in "skip" processing mode.

Skip mode is turned on with command line parameter *–skip*; in this mode stage is expected to skip its "semantic" part of the message transformation (like changes in data fields format, definition of additional data fields, etc) but perform actions required to keep the dataflow seamless.

The simpliest way to achieve this is to send the input message to the output without changes (marking it as "incomplete"), and this is the default implementation of the method.

In some cases it may be necessary to re-implement it (just like *process()*) to keep the dataflow unbroken.

NOTE: the output messages in "skip" mode MUST be marked as "incomplete".

Parameters **input_message** (*pyDKB.messages.AbstractMessage*) – message to process

Returns True/False (success/failure)

Return type bool

stop()
Finalize all the processes and prepare to exit.

Submodules

pyDKB.dataflow.cds module

Extended CDSInvenioConnector allowing us to login via Kerberos

pyDKB.dataflow.dkbID module

Utils to generate unique yet meaningful identifier for DKB objects.

pyDKB.dataflow.dkbID.**dkbID**(*json_data*, *data_type*)
Return unique identifier for object of TYPE based on DATA.

pyDKB.dataflow.exceptions module

Definition of DKB Dataflow exceptions

exception pyDKB.dataflow.exceptions.**DataflowException**(*message=''*, *reason=None*)
Bases: exceptions.Exception

Base Exception for Dataflow modules.

reason = None

pyDKB.dataflow.types module

Type definitions for library objects.

CHAPTER

TWO

STAGES

2.1 Stage 055

Stage for converting JSON documents (output of stage 015) into TTL documents (input for stage 060).

Initial JSON document should have the following structure:

```
{
  "GLANCE": {},
  "CDS" : {},
  "dkbID" : ...,
  "supporting_notes": [
        {
            "GLANCE": {},
            "CDS": {},
            "dkbID": ...,
        },
        {
            ...
        }
  ]
}
```

Some functions accept specific parts of this JSON - for example, if ‘data’ variable contains the initial JSON then “‘CDS’ part of the initial JSON” means “data.get(‘CDS’)”.

Resulting TTL file has the following structure:

```
PAPER a atlas:Paper .
PAPER atlas:hasGLANCE_ID __ .
PAPER atlas:hasShortTitle __ .
PAPER atlas:hasFullTitle __ .
PAPER atlas:hasRefCode __ .
PAPER atlas:hasCreationDate __ .
PAPER atlas:hasCDSReportNumber __ .
PAPER atlas:hasCDSInternal __ .
PAPER atlas:hasCDS_ID __ .
PAPER atlas:hasAbstract __ .
PAPER atlas:hasArXivCode __ .
PAPER atlas:hasFullTitle __ .
PAPER atlas:hasDOI __ .
PAPER atlas:hasKeyword __ .
JOURNAL_ISSUE a atlas:JournalIssue .
JOURNAL_ISSUE atlas:hasTitle __ .
```

(continues on next page)

(continued from previous page)

```
JOURNAL_ISSUE atlas:hasVolume __ .
JOURNAL_ISSUE atlas:hasYear __ .
JOURNAL_ISSUE atlas:containsPublication> PAPER .
SUPPORTING_DOCUMENT a atlas:SupportingDocument .
SUPPORTING_DOCUMENT atlas:hasGLANCE_ID __ .
SUPPORTING_DOCUMENT atlas:hasLabel __ .
SUPPORTING_DOCUMENT atlas:hasURL __ .
SUPPORTING_DOCUMENT atlas:hasCreationDate __ .
SUPPORTING_DOCUMENT atlas:hasCDSInternal __ .
SUPPORTING_DOCUMENT atlas:hasCDS_ID __ .
SUPPORTING_DOCUMENT atlas:hasAbstract __ .
SUPPORTING_DOCUMENT atlas:hasKeyword __ .
PAPER atlas:isBasedOn SUPPORTING_DOCUMENT .
```

TODO: This module doesn't convert authors metadata. This task is still under consideration.

`055_documents2TTL.documents2ttl.`**`abstract_extraction`**(*data*)
Extract abstract from JSON.

Parameters **`data`** (*`dict`*) – 'CDS' part of the initial JSON

Returns abstract or None if it was not found

Return type str or NoneType

`055_documents2TTL.documents2ttl.`**`arxiv_extraction`**(*data*)
Extract arXiv code from JSON.

Parameters **`data`** (*`dict`*) – 'CDS' part of the initial JSON

Returns arXiv code or None if it was not found

Return type str or NoneType

`055_documents2TTL.documents2ttl.`**`cds_id_extraction`**(*data*)
Extract CDS id from JSON.

Parameters **`data`** (*`dict`*) – 'CDS' part of the initial JSON

Returns CDS id or None if it was not found

Return type int or NoneType

`055_documents2TTL.documents2ttl.`**`cds_internal_extraction`**(*data*)
Extract CDS internal report number parameter from JSON.

Parameters **`data`** (*`dict`*) – 'CDS' part of the initial JSON

Returns CDS internal report number or None if it was not found

Return type unicode or NoneType

`055_documents2TTL.documents2ttl.`**`cds_parameter_extraction`**(*param_name*, *json_data*)
Extract CDS parameter value from JSON.

Parameters

- **`param_name`** (*`str`*) – name of the parameter, defined in *_CDS_ATTRS dict
- **`json_data`** (*`dict`*) – 'CDS' part of the initial JSON

Returns parameter value or None if it was not found

Return type int, str, NoneType

`055_documents2TTL.documents2ttl.`**`creation_date_extraction`**(*data*)
Extract creation date from JSON.

Parameters **`data`** (*`dict`*) – 'CDS' part of the initial JSON

Returns creation date or None if it was not found

Return type str or NoneType

`055_documents2TTL.documents2ttl.`**`define_globals`**(*args*)
Define global variables for further usage in other functions.

Global variables GRAPH and ONTOLOGY are defined, their values are received from the command line arguments via argparse.

Parameters **`args`** (*`argparse.Namespace`*) – stage arguments

`055_documents2TTL.documents2ttl.`**`document_cds`**(*data*, *doc_iri*, *cds_attrs*)
Convert CDS metadata from JSON to TTL.

Parameters

- **`data`** (*`dict`*) – 'CDS' part of the initial JSON
- **`doc_iri`** (*`str`*) – document IRI for current graph
- **`cds_attrs`** (*`list`*) – PAPER_CDS_ATTRS | NOTE_CDS_ATTRS

Returns TTL string with CDS metadata

Return type str

`055_documents2TTL.documents2ttl.`**`document_glance`**(*data*, *doc_iri*, *glance_attrs*)
Convert GLANCE metadata from JSON to TTL.

Parameters

- **`data`** (*`dict`*) – 'GLANCE' part of the initial JSON
- **`doc_iri`** (*`str`*) – document IRI for current graph
- **`glance_attrs`** (*`list`*) – PAPER_GLANCE_ATTRS | NOTE_GLANCE_ATTRS

Returns TTL string with GLANCE metadata

Return type str

`055_documents2TTL.documents2ttl.`**`document_links`**(*data*)
Construct TTL sentences to link paper to its supporting documents.

The result looks as following:

```
PAPER atlas:isBasedOn SUPPORTING_DOCUMENT
```

Parameters **`data`** (*`dict`*) – initial JSON

Returns TTL string with links

Return type str

`055_documents2TTL.documents2ttl.`**`doi2ttl`**(*doi*, *doc_iri*)
Convert DOI parameter from JSON to TTL.

Parameters

- **`doi`** (*`str, unicode or list`*) – 'doi' part of the initial JSON
- **`doc_iri`** (*`str`*) – document IRI for current graph

Returns TTL string with DOI

Return type str

`055_documents2TTL.documents2ttl.`**`fix_list_values`**(*list_vals*)

Apply fix_string to each item in a list.

Parameters **`list_vals`** (*`list`*) – list with strings to be fixed

Returns list with fixed strings

Return type list

`055_documents2TTL.documents2ttl.`**`fix_string`**(*wrong_string*)

Fix escape sequences in a string.

Parameters **`wrong_string`** (*`object`*) – string to be fixed, or any non-string object.

Returns fixed string, or unchanged non-string object

Return type object

`055_documents2TTL.documents2ttl.`**`generate_journal_id`**(*journal_dict*)

Generate a journal issue ID based on title, volume and year.

Parameters **`journal_dict`** (*`dict`*) – journal parameters

Returns journal ID

Return type str

`055_documents2TTL.documents2ttl.`**`get_document_iri`**(*doc_id*)

Construct an IRI for a document.

Parameters **`doc_id`** (*`str`*) – document id

Returns IRI

Return type str

`055_documents2TTL.documents2ttl.`**`glance_parameter_extraction`**(*param_name*, *json_data*)

Extract a GLANCE parameter value from JSON.

Parameters

- **`param_name`** (*`str`*) – name of the parameter
- **`json_data`** (*`dict`*) – 'GLANCE' part of the initial JSON

Returns parameter value or None if it was not found

Return type str, unicode, NoneType

`055_documents2TTL.documents2ttl.`**`keywords2ttl`**(*keywords*, *doc_iri*)

Convert keywords from JSON to TTL.

Parameters

- **`keywords`** (*`dict or list of dicts`*) – 'keywords' part of the initial JSON
- **`doc_iri`** (*`str`*) – document IRI for current graph

Returns TTL string with keywords

Return type str

`055_documents2TTL.documents2ttl.`**`main`**(*argv*)

Parse command line arguments and run the stage.

Parameters **`argv`** (*`list`*) – arguments

`055_documents2TTL.documents2ttl.`**`process`**(*stage*, *msg*)

Transform input JSON message into TTL documents.

Implementation of *`ProcessorStage.process()`* for hooking the stage into DKB workflow. Output message containing the TTL result is generated in this function.

Parameters

- **`stage`** (*`pyDKB.dataflow.stage.ProcessorStage`*) – stage instance
- **`msg`** (`pyDKB.dataflow.communication.messages.JSONMessage`) – input message with initial JSON

Returns True

Return type bool

`055_documents2TTL.documents2ttl.`**`process_journals`**(*data*, *doc_iri*)

Convert journal data from JSON to TTL.

Parameters

- **`data`** (*`list, dict`*) – 'CDS' part of the initial JSON
- **`doc_iri`** (*`str`*) – document IRI for current graph

Returns TTL string with journal issue with connection to paper

Return type str

`055_documents2TTL.documents2ttl.`**`report_number_extraction`**(*data*)

Extract report number from JSON.

Parameters **`data`** (*`dict`*) – 'CDS' part of the initial JSON

Returns report number or None if it was not found

Return type unicode or NoneType

`055_documents2TTL.documents2ttl.`**`title_extraction`**(*data*)

Extracting title from JSON.

Parameters **`data`** (*`dict`*) – 'CDS' part of the initial JSON

Returns title or None if it was not found

Return type str or NoneType

CHAPTER THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX