# DKB Documentation

DKB team

Dec 06, 2018

# 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.messages 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):

```
from pyDKB.dataflow.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 JSONProcessorStage
from pyDKB.dataflow.messages import JSONMessage

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

if __name__ == '__main__':
    stage = JSONProcessorStage()
    stage.process = my_process
    stage.parse_args(sys.argv[1:])
    stage.run()
```

First we create stage object and indicate that input and output message format is JSON: `stage = JSONProcessorStage()` (for full list of processors check *pyDKB.dataflow.stage package* section of this documentation); then set stage processing function to our function `my_process()`, parse command line arguments (`stage.parse_args(sys.argv[1:])`) and 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.common package

Common modules.

#### Submodules

#### 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.

**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)

### 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.

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

pyDKB.common.hdfs.**File**(*fname*)
Get and open temporary local copy of HDFS file

Return value: open file object (TemporaryFile).

pyDKB.common.hdfs.**basename**(*path*)
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**(*path*)
Return dirname without filename.

pyDKB.common.hdfs.**getfile**(*fname*)
Download file from HDFS.

Return value: file name (without directory)

pyDKB.common.hdfs.**join**(*path*, *filename*)
Join path and filename.

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

## 1.2.2 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: object

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.

**log**(*message*, *level=3*)
Output log message with given log level.

**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.

**exception** pyDKB.dataflow.communication.consumer.Consumer.**ConsumerException**
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_empty**()
Check if current source is empty.

**Return value:** True (empty) False (not empty) None (no source)

### 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: object

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).

**log**(*message*, *level=3*)
Output log message with given log level.

**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**
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: object

Abstract class for input/output streams.

**close**()
Close open file descriptors etc.

**configure**(*config*)
Stream configuration.

**get_fd**()
Return open file descriptor or raise exception.

**log**(*message*, *level=3*)
Output log message with given log level.

**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

**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.

**Return values:** Message object False (failed to parse message) None (no messages left)

**next**()
Get next message from the input stream.

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

**Retrun value:** Message object False (failed to parse)

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

Overrides parent method to reset __iterator property.

**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.

**Return values:** Message object False (failed to parse message) None (no messages left)

**next**()
Get next message from the input stream.

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

**Retrun value:** Message object False (failed to parse)

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

Overrides parent method to reset __iterator property.

#### 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: object

Abstract class for input/output streams.

**close**()
Close open file descriptors etc.

**configure**(*config*)
Stream configuration.

**get_fd**()
Return open file descriptor or raise exception.

**log**(*message*, *level=3*)
Output log message with given log level.

**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

**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**
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 this message type.

**getOriginal**()
Return original message.

**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.

**msg_type = 2**

**native_types = [<type 'dict'>]**

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/transfornation.

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_out_stream**()
Get current 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_input_message_type**(*Type=None*)
Set input message type.

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

**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: object

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

__parser

- **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**(*message*, *level=3*)
Output log message with given log level.

**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**()
Reads stage custom config file.

**Returns** (TruelFalse)

**run**()
Run the stage.

**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/transfornation.

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_out_stream**()
Get current 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_input_message_type**(*Type=None*)
Set input message type.

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

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

### Submodules

### pyDKB.dataflow.cds module

Extended CDSInvenioConnector allowing us to login via Kerberos

**class** pyDKB.dataflow.cds.**CDSInvenioConnector**(**args*)
Bases: invenio_client.contrib.cds.CDSInvenioConnector

CDSInvenioConnector which closes the browser in most cases.

**delete**(*restore_handlers=True*)

**handlers = False**

**kill**(*signum*, *frame*)
Run del and propagate signal.

**orig_handlers = {}**

**class** pyDKB.dataflow.cds.**KerberizedCDSInvenioConnector**(*login='user'*, *password='password'*)
Bases: *pyDKB.dataflow.cds.CDSInvenioConnector*

Represents same CDSInvenioConnector, but this one is aware about SPNEGO: Simple and Protected GSSAPI Negotiation Mechanism

### 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**
Bases: exceptions.Exception

Base Exception for Dataflow modules.

### pyDKB.dataflow.types module

Type definitions for library objects.

CHAPTER

TWO

# Stage 055

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

JSON file should have the following structure:

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

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 __ .
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*) – JSON string

**Returns** abstract

**Return type** str

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

**Parameters** **data** (*dict*) – JSON string

**Returns** arXiv code

**Return type** str

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

**Parameters** **data** (*dict*) – JSON string

**Returns** CDS id

**Return type** int

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

**Parameters** **data** (*dict*) – JSON data from file or stream

**Returns** CDS internal report number

**Return type** unicode

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

**Parameters**

- **param_name** (*str*) – name of the parameter, defined in *_CDS_ATTRS dict
- **json_data** – JSON with CDS parameters

**Returns** parameter value

**Return type** int, str

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

**Parameters** **data** (*dict*) – JSON string

**Returns** creation date

**Return type** str

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

**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*) – JSON data from file or stream
- **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*) – JSON data from file or stream
- **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.**documents_links**(*data*)
Convert links from JSON to TTL.

The result looks as following: PAPER atlas:isBasedOn SUPPORTING_DOCUMENT

**Parameters** **data** (*dict*) – JSON data from file or stream

**Returns** TTL string with links

**Return type** str

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

**Parameters**

- **doi** (*str, unicode or list*) – doi from JSON string
- **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** (*str, unicode*) – string to be fixed

**Returns** fixed string

**Return type** str

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 parameter value from GLANCE JSON.

**Parameters**

- **param_name** (*str*) – name of the parameter
- **json_data** (*dict*) – JSON with GLANCE metadata

**Returns** parameter value

**Return type** str, unicode

055_documents2TTL.documents2ttl.**keywords2ttl**(*keywords*, *doc_iri*)
Convert keywords from JSON string to TTL.

**Parameters**

- **keywords** (*dict or list of dicts*) – keywords parameters from JSON string
- **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*)
Process a message. Convert the message's contents from JSON to TTL.

**Parameters**

- **stage** (*pyDKB.dataflow.stage.ProcessorStage*) – stage instance
- **msg** (*pyDKB.dataflow.Message*) – input message with JSON data

055_documents2TTL.documents2ttl.**process_journals**(*data*, *doc_iri*)
Convert journal data from JSON to TTL.

**Parameters**

- **data** (*list, dict*) – 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 string.

**Parameters** **data** – JSON data from file or stream

**Returns** report number

**Return type** unicode

055_documents2TTL.documents2ttl.**title_extraction**(*data*)
Extracting title from JSON.

**Parameters** **data** (*dict*) – JSON string

**Returns** title

**Return type** str

CHAPTER

THREE

# Indices and tables

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## 0



## p

# INDEX

## Symbols



## A



## B



## C

## D



## E



## F

## G



## H



## I



## J

## R