Pypeman Documentation

Release 0.1

MHCOMM

Dec 21, 2018

Contents

This is the Pypeman documentation.

Source on github: https://github.com/mhcomm/pypeman

See *Introduction* for more info or *Getting started* for basic functionality.

CHAPTER 1

Introduction

Warning: Pypeman is in Alpha state, do not use in production yet.

The API may change in future versions.

Pypeman is a minimalist but pragmatic EAI / ESB / Middleware / PlumbingThing. You can use it to make the *glue* between your platform components. It's main purpose is to exchange and process data to connect platform in an heterogeneous environment.

Pypeman is developed in py3.5+ using intensively Python asyncio feature.

Main objectives of this project are to make it:

- clean API (put the complexity inside!),
- easy to configure,
- easy to test,
- easy to debug,
- easy to admin,
- highly hackable,
- lightweight,
- handle ~10.000 messages/day on a raspberry

Source on github

CHAPTER 2

Getting started

2.1 Installation

With pip

```
pip install pypeman # or
pip install pypeman[all] # To install with all optional dependencies
```

2.2 Basic usage

Create a fresh project with:

```
pypeman startproject <project_dirname>
```

Above command will create a new directory with a “settings.py” file containing local configs and a “project.py” file with a channel example that you can uncomment to test pypeman. Follow the commented instructions then execute:

```
pypeman start # You can use the --reload option for auto-reloading on changes
```

2.3 Quick command overview

To get command help and more details about commands:

```
pypeman --help
```

To create a fresh project:

```
pypeman startproject <project_name>
```

To start pypeman as daemon:

```
pypeman start [--reload] [--remote-admin]
```

To stop pypeman:

```
pypeman stop
```

To show a channel graph:

```
pypeman graph
```

To launch a remote shell (only if remote-admin is activated):

```
pypeman shell
```

CHAPTER 3

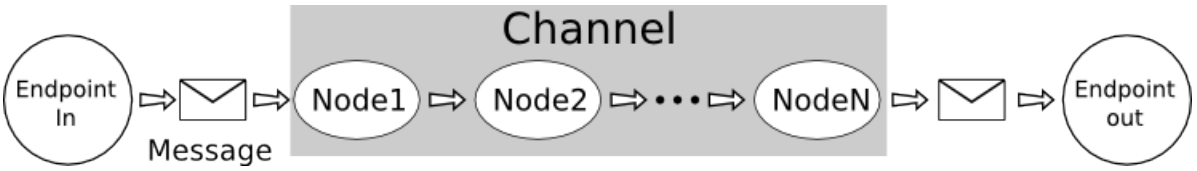
Examples

```
from pypeman import endpoints
from pypeman import channels
from pypeman import nodes

http = endpoints.HTTPEndpoint(address='localhost', port='8080')

main = channels.HttpChannel(endpoint=http, method='*', url='/{name}')
main.append(nodes.Log())

alt = main.fork()

alt.append(nodes.JsonToPython(), nodes.Add1(), nodes.Add1(), nodes.Log())

main.append(nodes.Log(), nodes.JsonToPython(), nodes.Log())
main.append(nodes.Add1(), nodes.Add1(), nodes.Log(), nodes.PythonToJson())
```

CHAPTER 4

User guide

One image is better than 100 words:



4.1 Channels

Channels are chains of processing, the nodes, which use or transform message contents. A channel is specialised for a type of input and can be linked to an endpoint for incoming messages.

A channel receives a message, processes it and returns the response.

Channels are the main components of pypeman. When you want to process a message, you first create a channel then add nodes to process the message.

class pypeman.channels.**BaseChannel**(*name=None*, *parent_channel=None*, *loop=None*, *message_store_factory=None*)

Base channel are generic channels. If you want to create new channel, inherit from the base class and call `self.handle(msg)` method with generated message.

Parameters

- **name** – Channel name is mandatory and must be unique through the whole project. Name gives a way to get channel in test mode.
- **parent_channel** – Used with sub channels. Don't specify yourself.
- **loop** – To specify a custom event loop.
- **message_store_factory** – You can specify a message store (see below) at channel initialisation if you want to save all processed message. Use *message_store_factory* argument with an instance of wanted message store factory.

add(**args*)
Add specified nodes to channel (Shortcut for append).

Parameters **args** – Nodes to add.

append(**args*)
Append specified nodes to channel.

Parameters **args** – Nodes to add.

case(**conditions*, *names=None*, *message_store_factory=None*)
Case between multiple conditions. For each condition specified, a channel is returned by this method in same order as condition are given. When processing a message, conditions are evaluated successively and the first returning true triggers the corresponding channel processing the message. When channel processing is finished, the next node is called.

Parameters

- **conditions** – Multiple conditions, one for each returned channel. Should be a boolean or a function that takes a `msg` argument and should return a boolean.
- **message_store_factory** – Allows you to specify a message store factory for all channel of this *case*.

Returns one channel by condition parameter.

fork(*name=None*, *message_store_factory=None*)
Create a new channel that process a copy of the message at this point. Subchannels are executed in parallel of main process.

Returns The forked channel

get_node(*name*)
Return node with name in argument. Mainly used in tests.

Parameters **name** – The searched node name.

Returns Instance of Node or None if none found.

graph(*prefix=''*, *dot=False*)
Generate a text graph for this channel.

graph_dot(*end=''*)
Generate a compatible dot graph for this channel.

handle(*msg*)
Overload this method only if you know what you are doing but call it from child class to add behaviour.

Parameters **msg** – To be processed msg.

Returns Processed message

handle_and_wait(*msg*)
Handle a message synchronously. Mainly used for testing purpose.

Parameters **msg** – Message to process

Returns Processed message.

is_stopped()

Returns True if channel is in stopped or stopping state.

process(*msg*)
Overload this method only if you know what you are doing. Called by `subhandle` method.

Parameters **msg** – To be processed msg.

Returns Processed message

replay(*msg_id*)
This method allows you to replay a message from channel *message_store*.

Parameters **msg_id** – Message id to replay.

Returns The result of the processing.

start()
Start the channel. Called before starting processus. Can be overloaded to specify specific start procedure.

status
Getter for status

stop()
Stop the channel. Called when pypeman shutdown.

subhandle(*msg*)
Overload this method only if you know what you are doing. Called by `handle` method.

Parameters **msg** – To be processed msg.

Returns Processed message

when(*condition*, *name=None*, *message_store_factory=None*)
New channel bifurcation that is executed only if condition is True. This channel replace further current channel processing.

Parameters **condition** – Can be a value or a function that takes a message argument.

Returns The conditional path channel.

class pypeman.contrib.time.**CronChannel**(**args*, *cron=''*, ***kwargs*)
This channel Launch processing at specified time interval. The first node of the channel receive a payload with the datetime of execution.

Params cron This set the interval. Accept any aiocron compatible string.

start()
Start the channel. Called before starting processus. Can be overloaded to specify specific start procedure.

class pypeman.contrib.http.**HttpChannel**(**args*, *endpoint=None*, *method='*'*, *url='/'*, *encoding=None*, ***kwargs*)
Channel that handles Http connection. The Http message is the message payload and some headers become metadata of message. Needs `aiohttp` python dependency to work.

Parameters

- **endpoint** – HTTP endpoint used to get connections.
- **method** – Method filter.
- **url** – Only matching urls messages will be sent to this channel.
- **encoding** – Encoding of message. Default to 'utf-8'.

start()
Start the channel. Called before starting processus. Can be overloaded to specify specific start procedure.

class pypeman.channels.**FileWatcherChannel**(**args*, *basedir=''*, *regex='.*'*, *interval=1*, *binary_file=False*, *path=''*, ***kwargs*)
Watch for file change or creation. File content becomes message payload. `filepath` is in message meta.

start()
Start the channel. Called before starting processus. Can be overloaded to specify specific start procedure.

class pypeman.contrib.ftp.**FTPWatcherChannel**(**args*, *host=''*, *port=21*, *credentials=''*, *basedir=''*, *regex='.*'*, *interval=60*, *delete_after=False*, *encoding='utf-8'*, *thread_pool=None*, *sort_function=<built-in function sorted>*, ***kwargs*)
Channel that watch ftp for file creation.

download_file(*filename*)
Download a file from ftp asynchronously.

Parameters **filepath** – file path to download.

Returns Content of the downloaded file.

get_file_and_process(*filename*)
Download a file from ftp and launch channel processing on msg with result as payload. Also add a *filepath* header with ftp relative path of downloaded file.

Parameters **filename** – file to download relative to *basedir*.

Returns processed result

start()
Start the channel. Called before starting processus. Can be overloaded to specify specific start procedure.

tick()
One iteration of watching.

watch_for_file()
Watch recursively for ftp new files. If file match regex, it is downloaded then processed in a message.

class pypeman.contrib.hl7.**MLLPChannel**(**args*, *endpoint=None*, *encoding='utf-8'*, ***kwargs*)

start()
Start the channel. Called before starting processus. Can be overloaded to specify specific start procedure.

4.2 Nodes

A node is a processing unit in a channel. To create a node, inherit form *pypeman.nodes.BaseNode* and override *process(msg)* method.

You can use persistence storage to save data between two pypeman executions sush last time a specific channel ran by using the two following methods: *save_data* and *restore_data*

4.2.1 Specific nodes

Base for all node. You must inherit from this class if you want to create a new node for your project.

class pypeman.nodes.**BaseNode**(**args*, *name=None*, *log_output=False*, ***kwargs*)
Base of all Nodes. If you create a new node, you must inherit from this class and implement *process* method.

save_data

Parameters

- **name** – Name of node. Used in log or test.

- **log_output** – To enable output logging for this node.

Store_output_as Store output message in msg.ctx as specified key

Store_input_as Store input message in msg.ctx as specified key

Passthrough If True, node is executed but output message is same as input

async_run(*msg*)
Used to overload behaviour like thread Node without rewriting handle process

fullpath()
Return the channel name and node name dot concatened.

handle(*msg*)
Handle message is called by channel to launch process method on it. Some other structural processing take place here. Please, don't modify unless you know what you are doing.

Parameters **msg** – incoming message

Returns modified message after a process call and some treatment

mock(*input=None*, *output=None*)
Allow to mock input or output of a node for testing purpose.

Parameters

- **input** – A message to replace the input in this node.
- **output** – A return message to replace processing of this mock.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

restore_data(*key*, *default=<object object>*)
Restore previously saved data from configured persistence backend.

Parameters

- **key** – Key of restored data.
- **default** – if key is missing, don't raise exception and return this value instead.

Returns Saved data if exist or default value if specified.

run(*msg*)
Used to overload behaviour like thread Node without rewriting handle process

save_data(*key*, *value*)
Save data in configured persistence backend for next usage.

Parameters

- **key** – Key of saved data.
- **value** – Value saved.

Thread node allow you to create node that execute is process method in another thread to avoid blocking nodes.

class pypeman.nodes.**ThreadNode**(**args*, *thread_pool=None*, ***kwargs*)
Inherit from this class instead of BaseNode to avoid long run node blocking main event loop.

run(*msg*)
Used to overload behaviour like thread Node without rewriting handle process

4.2.2 Other nodes

See *pypeman.nodes* and *pypeman.contrib*.

4.3 Messages

Message contains the core information processed by nodes and carried by channel. The message payload may be: Json, Xml, Soap, Hl7, text, Python object...

Useful attributes:

- payload: the message contents.
- meta: message metadata, should be used to add extra information about the payload.
- context: previous messages can be saved in the context dict for further access.

class pypeman.message.**Message**(*content_type='application/text'*, *payload=None*, *meta=None*)

A message is the unity of informations exchanged between nodes of a channel.

A message have following properties:

attribute payload The message content.

attribute meta The message metadata.

attribute timestamp The creation date of message

attribute uuid uuid to identify message

attribute content_type Used ?

attribute ctx Current context when you want to save a message for later use.

Parameters

- **payload** – You can initialise the payload by setting this param.
- **meta** – Same as payload, you can initialise the meta by setting this param.

add_context(*key*, *msg*)

Add a msg to the *.ctx* property with specified key.

Parameters

- **key** – Key to store message.
- **msg** – Message to store.

copy()

Copy the message. Useful for channel forking purpose.

Returns A copy of current message.

static from_dict(*data*)

Convert the input dict previously converted with *.as_dict()* method in Message object.

Parameters **data** – The input dict.

Returns The message message object correponding to given data.

static from_json(*data*)

Create a message from previously saved json string.

Parameters **data** – Data to read message from.

Returns A new message instance created from json data.

log(*logger=<logging.Logger object>*, *log_level=10*, *payload=True*, *meta=True*, *context=False*)
Log a message.

Parameters

- **logger** – Logger
- **log_level** – log level for all log.
- **payload** – Whether log payload.
- **meta** – Whether log meta.
- **context** – Whether log context.

renew()
Copy the message but update the *timestamp* and *uuid*.

Returns A copy of current message with new *uuid* and *Timestamp*.

timestamp_str()
Return timestamp formated string

to_dict()
Convert the current message object to a dict. Payload is pickled.

Returns A dict with an equivalent of message

to_json()
Create json string for current message.

Returns a json string equivalent for message.

to_print(*payload=True*, *meta=True*, *context=False*)
Return a printable version of message.

Parameters

- **payload** – Whether print payload.
- **meta** – Whether print meta.
- **context** – Whether print context.

4.4 Endpoints

Endpoints are server instances used by channel to get messages from net protocols like HTTP, Soap or HL7, ... They listen to a specific port for a specific protocol.

class pypeman.contrib.http.**HTTPEndpoint**(*adress=None*, *address=None*, *port=None*, *loop=None*, *http_args=None*, *host=None*, *sock=None*, *reuse_port=None*)
Endpoint to receive HTTP connection from outside.

class pypeman.contrib.hl7.**MLLPEndpoint**(*address=None*, *port=None*, *encoding='utf-8'*, *loop=None*, *host=None*, *sock=None*, *reuse_port=None*)

4.5 Message Stores

A Message store is really useful to keep a copy of all messages sent to a channel. It's like a log but with complete message data and metadata. This way you can trace all processing or replay a specific message (Not implemented yet). Each channel can have its message store.

You don't use message stores directly but a MessageStoreFactory instance to allow reuse of a configuration.

4.5.1 Generic classes

class pypeman.msgstore.**MessageStoreFactory**

Message store factory class can generate Message store instance for specific store_id.

get_store(*store_id*)

Parameters **store_id** – identifier of corresponding message store.

Returns A MessageStore corresponding to correct store_id.

class pypeman.msgstore.**MessageStore**

A MessageStore keep an history of processed messages. Mainly used in channels.

change_message_state(*id*, *new_state*)

Change the *id* message state.

Parameters

- **id** – Message specific store id.
- **new_state** – Target state.

get(*id*)

Return one message corresponding to given *id* with his status.

Parameters **id** – Message id. Message store dependant.

Returns A dict *{'id':<message_id>, 'state': <message_state>, 'message': <message_object>}*.

search(*start=0*, *count=10*, *order_by='timestamp'*)

Return a list of message with store specific *id* and processed status.

Parameters

- **start** – First element.
- **count** – Count of elements since first element.
- **order_by** – Message order. Allowed values : ['timestamp', 'status'].

Returns A list of dict *{'id':<message_id>, 'state': <message_state>, 'message': <message_object>}*.

start()

Called at startup to initialize store.

store(*msg*)

Store a message in the store.

Parameters **msg** – The message to store.

Returns Id for this specific message.

total()

Returns total count of messages

4.5.2 Message store factories

class pypeman.msgstore.**NullMessageStoreFactory**
Return an NullMessageStore that do nothing at all.

get_store(*store_id*)

Parameters **store_id** – identifier of corresponding message store.

Returns A MessageStore corresponding to correct store_id.

class pypeman.msgstore.**FakeMessageStoreFactory**
Return an Fake message store

get_store(*store_id*)

Parameters **store_id** – identifier of corresponding message store.

Returns A MessageStore corresponding to correct store_id.

class pypeman.msgstore.**MemoryMessageStoreFactory**
Return a Memory message store. All message are lost at pypeman stop.

get_store(*store_id*)

Parameters **store_id** – identifier of corresponding message store.

Returns A MessageStore corresponding to correct store_id.

class pypeman.msgstore.**FileMessageStoreFactory**(*path*)
Generate a FileMessageStore message store instance. Store a file in *<base_path>/<store_id>/<month>/<day>/* hierachy.

get_store(*store_id*)

Parameters **store_id** – identifier of corresponding message store.

Returns A MessageStore corresponding to correct store_id.

CHAPTER 5

Remote Admin

5.1 Getting started

Pypeman allow you to access remote instance two ways:

- With a python shell that give access to a client module.
- With a custom command shell

You need to add *–remote-admin* at pypeman startup to enable remote admin websocket.

5.2 Python shell

Python shell allow you to programatically administrate a remote pypeman instance through python command. You can start the python shell by executing:

```
pypeman pyshell
```

It starts a ipython instance with a RemoteAdminClient instance named *client*. The client instance API is:

class pypeman.remoteadmin.**RemoteAdminClient**(*loop=None*, *url='ws://localhost:8091'*)
Remote admin client. To be use by ipython shell or pypeman shell.

Params url Pypeman Websocket url.

channels()
Return a list of available channels on remote instance.

exec(*command*)
Execute any valid python code on remote instance and return stdout result.

list_msg(*channel*, *start=0*, *count=10*, *order_by='timestamp'*)
List first 10 messages on specified channel from remote instance.

Params channel The channel name.

Params start Start index of listing.

Params count Count from index.

Params order_by Message order. only 'timestamp' and '-timestamp' handled for now.

Returns list of message with status.

push_msg(*channel*, *text*)
Push a new message from text param to the channel.

Params channel The channel name.

Params text This text will be the payload of the message.

replay_msg(*channel*, *msg_ids*)
Replay specified message from id list of specified channel on remote instance.

Params channel The channel name.

Params msg_ids Message id list to replay

Returns List of result for each message. Result can be {'error': <msg_error>} for one id if error occurs.

send_command(*command*, *args=None*)
Send a command to remote instance

start(*channel*)
Start the specified channel on remote instance.

Params channel The channel name.

stop(*channel*)
Stop the specified channel on remote instance.

Params channel The channel name.

5.3 Custom command shell

The custom command shell has simple commands to ease administration for rapid tasks but with less possibility.

To launch remote shell, execute:

```
pypeman shell
```

You can show command help this way:

```
pypeman > help # For command list
pypeman > help <command name> # For help on a specific command
```

CHAPTER 6

Channel testing

6.1 Getting started

Testing framework uses the standard unittest python package. So you can write your tests as always in a *tests* package or a module visible from your project. To use pypeman specifics helpers, your test case classes must inherit from *test.PypeTestCase*.

To launch tests, just execute :

```
pypeman test
```

6.2 Specific helpers

PypeTestCase.get_channel(channel_name) To get a channel in test mode from your project file by name.

Channel_instance.get_node(node_name) To get a specific node by name.

Node_instance.mock(input=None, output=None)

> Allows to mock node inputs or the output msg. If you mock output, the original process() is completely bypassed. You can also use a function that takes a *msg* argument as input or output mock to use or modify a message.

Node_instance.last_input() Returns last input message of a node.

Node_instance.processed Keeps a processed message count. Reset between each test.

CHAPTER 7

API

7.1 Channels

class pypeman.channels.**BaseChannel**(*name=None*, *parent_channel=None*, *loop=None*, *message_store_factory=None*)

Base channel are generic channels. If you want to create new channel, inherit from the base class and call `self.handle(msg)` method with generated message.

Parameters

- **name** – Channel name is mandatory and must be unique through the whole project. Name gives a way to get channel in test mode.
- **parent_channel** – Used with sub channels. Don't specify yourself.
- **loop** – To specify a custom event loop.
- **message_store_factory** – You can specify a message store (see below) at channel initialisation if you want to save all processed message. Use *message_store_factory* argument with an instance of wanted message store factory.

add(**args*)

Add specified nodes to channel (Shortcut for append).

Parameters **args** – Nodes to add.

append(**args*)

Append specified nodes to channel.

Parameters **args** – Nodes to add.

case(**conditions*, *names=None*, *message_store_factory=None*)

Case between multiple conditions. For each condition specified, a channel is returned by this method in same order as condition are given. When processing a message, conditions are evaluated successively and the first returning true triggers the corresponding channel processing the message. When channel processing is finished, the next node is called.

Parameters

- **conditions** – Multiple conditions, one for each returned channel. Should be a boolean or a function that takes a `msg` argument and should return a boolean.
- **message_store_factory** – Allows you to specify a message store factory for all channel of this *case*.

Returns one channel by condition parameter.

fork(*name=None*, *message_store_factory=None*)
Create a new channel that process a copy of the message at this point. Subchannels are executed in parallel of main process.

Returns The forked channel

get_node(*name*)
Return node with name in argument. Mainly used in tests.

Parameters **name** – The searched node name.

Returns Instance of Node or None if none found.

graph(*prefix=''*, *dot=False*)
Generate a text graph for this channel.

graph_dot(*end=''*)
Generate a compatible dot graph for this channel.

handle(*msg*)
Overload this method only if you know what you are doing but call it from child class to add behaviour.

Parameters **msg** – To be processed msg.

Returns Processed message

handle_and_wait(*msg*)
Handle a message synchronously. Mainly used for testing purpose.

Parameters **msg** – Message to process

Returns Processed message.

is_stopped()

Returns True if channel is in stopped or stopping state.

process(*msg*)
Overload this method only if you know what you are doing. Called by `subhandle` method.

Parameters **msg** – To be processed msg.

Returns Processed message

replay(*msg_id*)
This method allows you to replay a message from channel *message_store*.

Parameters **msg_id** – Message id to replay.

Returns The result of the processing.

start()
Start the channel. Called before starting processus. Can be overloaded to specify specific start procedure.

status
Getter for status

stop()
Stop the channel. Called when pypeman shutdown.

subhandle(*msg*)
Overload this method only if you know what you are doing. Called by `handle` method.

Parameters **msg** – To be processed msg.

Returns Processed message

when(*condition*, *name=None*, *message_store_factory=None*)
New channel bifurcation that is executed only if condition is True. This channel replace further current channel processing.

Parameters **condition** – Can be a value or a function that takes a message argument.

Returns The conditional path channel.

class pypeman.channels.**Case**(**args*, *names=None*, *parent_channel=None*, *message_store_factory=None*, *loop=None*)
Case node internally used for *.case()* BaseChannel method. Don't use it.

exception pypeman.channels.**ChannelStopped**
The channel is stopped and can't process message.

class pypeman.channels.**ConditionSubChannel**(*condition=<function ConditionSubChannel.<lambda>>*, ***kwargs*)
ConditionSubchannel used for make alternative path. This processing replace all further channel processing.

subhandle(*msg*)
Overload this method only if you know what you are doing. Called by `handle` method.

Parameters **msg** – To be processed msg.

Returns Processed message

exception pypeman.channels.**Dropped**
Used to stop process as message is processed. Default success should be returned.

class pypeman.channels.**FileWatcherChannel**(**args*, *basedir=''*, *regex='.*'*, *interval=1*, *binary_file=False*, *path=''*, ***kwargs*)
Watch for file change or creation. File content becomes message payload. `filepath` is in message meta.

start()
Start the channel. Called before starting processus. Can be overloaded to specify specific start procedure.

exception pypeman.channels.**Rejected**
Used to tell caller the message is invalid with a error return.

class pypeman.channels.**SubChannel**(*name=None*, *parent_channel=None*, *loop=None*, *message_store_factory=None*)
Subchannel used for forking channel processing.

process(*msg*)
Overload this method only if you know what you are doing. Called by `subhandle` method.

Parameters **msg** – To be processed msg.

Returns Processed message

7.2 Endpoint

7.3 Node

class pypeman.nodes.**B64Decode**(**args*, *altchars=None*, ***kwargs*)
Decode payload from byte to specified encoding

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.nodes.**B64Encode**(**args*, *altchars=None*, ***kwargs*)
Encode payload in specified encoding to byte.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.nodes.**BaseNode**(**args*, *name=None*, *log_output=False*, ***kwargs*)
Base of all Nodes. If you create a new node, you must inherit from this class and implement *process* method.

save_data

Parameters

- **name** – Name of node. Used in log or test.
- **log_output** – To enable output logging for this node.

Store_output_as Store output message in msg.ctx as specified key

Store_input_as Store input message in msg.ctx as specified key

Passthrough If True, node is executed but output message is same as input

async_run(*msg*)
Used to overload behaviour like thread Node without rewriting handle process

fullpath()
Return the channel name and node name dot concatened.

handle(*msg*)
Handle message is called by channel to launch process method on it. Some other structural processing take place here. Please, don't modify unless you know what you are doing.

Parameters **msg** – incoming message

Returns modified message after a process call and some treatment

mock(*input=None*, *output=None*)
Allow to mock input or output of a node for testing purpose.

Parameters

- **input** – A message to replace the input in this node.
- **output** – A return message to replace processing of this mock.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

restore_data(*key*, *default=<object object>*)
Restore previously saved data from configured persistence backend.

Parameters

- **key** – Key of restored data.
- **default** – if key is missing, don't raise exception and return this value instead.

Returns Saved data if exist or default value if specified.

run(*msg*)
Used to overload behaviour like thread Node without rewriting handle process

save_data(*key*, *value*)
Save data in configured persistence backend for next usage.

Parameters

- **key** – Key of saved data.
- **value** – Value saved.

class pypeman.nodes.**Decode**(**args*, ***kwargs*)
Decode payload from byte to specified encoding

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.nodes.**Drop**(*message=None*, **args*, ***kwargs*)
Use this node to tell the channel the message is Dropped.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.nodes.**DropNode**(**args*, ***kwargs*)

class pypeman.nodes.**Email**(**args*, *host=None*, *port=None*, *user=None*, *password=None*, *ssl=False*, *start_tls=False*, *subject=None*, *sender=None*, *recipients=None*, *content=None*, ***kwargs*)
Node that send Email.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.nodes.**Empty**(**args*, *name=None*, *log_output=False*, ***kwargs*)
Return an empty new message.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.nodes.**Encode**(**args*, ***kwargs*)
Encode payload in specified encoding to byte.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.nodes.**FileReader**(*filename=None*, *filepath=None*, *binary_file=False*, **args*, ***kwargs*)
Reads a file and sets payload to the file's contents.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.nodes.**FileWriter**(*filepath=None*, *binary_mode=False*, *safe_file=True*, **args*, ***kwargs*)
Write a file with the message content.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.nodes.**JsonToPython**(**args*, *encoding='utf-8'*, ***kwargs*)
Convert json message payload to python dict.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.nodes.**Log**(**args*, ***kwargs*)
Node to show some information about node, channel and message. Use for debug.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.nodes.**Map**(**args*, ***kwargs*)
Used to map input message keys->values to another keys->values

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.nodes.**MappingNode**(**args*, ***kwargs*)

class pypeman.nodes.**MessageStore**(**args*, ***kwargs*)

class pypeman.nodes.**PythonToJson**(**args*, *encoding='utf-8'*, *indent=None*, ***kwargs*)
Convert python payload to json.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.nodes.**RaiseError**(**args*, *name=None*, *log_output=False*, ***kwargs*)

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.nodes.**Save**(**args*, *uri=None*, ***kwargs*)
Save a message in specified uri

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.nodes.**SaveFileBackend**(*path*, *filename*, *channel*)
Backend used to store message with `Save` node.

class pypeman.nodes.**SaveNullBackend**
For testing purpose

class pypeman.nodes.**SetCtx**(*ctx_name*, **args*, ***kwargs*)
Push the message in the context with the key *ctx_name*

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.nodes.**Sleep**(**args*, *duration=1*, ***kwargs*)
Wait *duration* seconds before returning message.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.nodes.**ThreadNode**(**args*, *thread_pool=None*, ***kwargs*)
Inherit from this class instead of BaseNode to avoid long run node blocking main event loop.

run(*msg*)
Used to overload behaviour like thread Node without rewriting handle process

class pypeman.nodes.**ToOrderedDict**(**args*, ***kwargs*)
this node yields an ordered dict with the keys 'keys' and the values from the payload if the payload does not contain certain values defaults can be specified with defaults

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

pypeman.nodes.**callable_or_value**(*val*, *msg*)
Return *val(msg)* if value is a callable else *val*.

pypeman.nodes.**choose_first_not_none**(**args*)
Choose first non None alternative in args. :param args: alternative list :return: the first non None alternative.

7.4 Contrib

7.4.1 FTP

class pypeman.contrib.ftp.**FTPConnection**(*host*, *port*, *credentials*)
FTP connection manager.

class pypeman.contrib.ftp.**FTPFileDeleter**(*host=''*, *port=21*, *credentials=None*, *filepath=None*, ***kwargs*)
Node to delete a file from FTP.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.contrib.ftp.**FTPFileReader**(*host=''*, *port=21*, *credentials=None*, *filepath=None*, ***kwargs*)
Node to read a file from FTP.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.contrib.ftp.**FTPFileWriter**(*host=''*, *port=21*, *credentials=None*, *filepath=None*, ***kwargs*)
Node to write content to FTP. File is first written with *.part* concatenated to its name then renamed to avoid partial upload.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.contrib.ftp.**FTPHelper**(*host*, *port*, *credentials*)
FTP helper to abstract ftp access.

delete(*filepath*)
Delete an FTP file. :param filepath: File to delete.

download_file(*filepath*)
Download a file from ftp asynchronously. :param filepath: file path to download. :return: content of the file.

rename(*fromfilepath*, *tofilepath*)
Rename a file from path to another path in ftp. :param fromfilepath: original file to rename. :param tofilepath: destination file.

upload_file(*filepath*, *content*)
Upload an file to ftp. :param filepath: Path of file to create. :param content: Content to upload.

class pypeman.contrib.ftp.**FTPWatcherChannel**(**args*, *host=''*, *port=21*, *credentials=''*, *basedir=''*, *regex='.*'*, *interval=60*, *delete_after=False*, *encoding='utf-8'*, *thread_pool=None*, *sort_function=<built-in function sorted>*, ***kwargs*)
Channel that watch ftp for file creation.

download_file(*filename*)
Download a file from ftp asynchronously.

Parameters **filepath** – file path to download.

Returns Content of the downloaded file.

get_file_and_process(*filename*)
Download a file from ftp and launch channel processing on msg with result as payload. Also add a *filepath* header with ftp relative path of downloaded file.

Parameters **filename** – file to download relative to *basedir*.

Returns processed result

start()
Start the channel. Called before starting processus. Can be overloaded to specify specific start procedure.

tick()
One iteration of watching.

watch_for_file()
Watch recursively for ftp new files. If file match regex, it is downloaded then processed in a message.

7.4.2 HL7

class pypeman.contrib.hl7.**HL7ToPython**(**args*, ***kwargs*)
Convert hl7 payload to python struct.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.contrib.hl7.**MLLPChannel**(**args*, *endpoint=None*, *encoding='utf-8'*, ***kwargs*)

start()
Start the channel. Called before starting processus. Can be overloaded to specify specific start procedure.

class pypeman.contrib.hl7.**MLLPProtocol**(*handler*, *loop=None*)

Minimal Lower-Layer Protocol (MLLP) takes the form: <VT>[HL7 Message]<FS><CR>

References:

- http://www.hl7standards.com/blog/2007/05/02/hl7-mlp-minimum-layer-protocol-defined/
- http://www.hl7standards.com/blog/2007/02/01/ack-message-original-mode-acknowledgement/

connection_lost(*exc*)
Called when the connection is lost or closed. The argument is an exception object or None (the latter meaning a regular EOF is received or the connection was aborted or closed).

connection_made(*transport*)
Called when a connection is made. The argument is the transport representing the pipe connection. To receive data, wait for data_received() calls. When the connection is closed, connection_lost() is called.

data_received(*data*)
Called when some data is received. The argument is a bytes object.

class pypeman.contrib.hl7.**PythonToHL7**(**args*, ***kwargs*)
Convert python payload to HL7. Must be HL7 structure.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

7.4.3 HTTP

class pypeman.contrib.http.**HTTPEndpoint**(*adress=None*, *address=None*, *port=None*, *loop=None*, *http_args=None*, *host=None*, *sock=None*, *reuse_port=None*)
Endpoint to receive HTTP connection from outside.

class pypeman.contrib.http.**HttpChannel**(**args*, *endpoint=None*, *method='*'*, *url='/'*, *encoding=None*, ***kwargs*)
Channel that handles Http connection. The Http message is the message payload and some headers become metadata of message. Needs `aiohttp` python dependency to work.

Parameters

- **endpoint** – HTTP endpoint used to get connections.
- **method** – Method filter.
- **url** – Only matching urls messages will be sent to this channel.
- **encoding** – Encoding of message. Default to 'utf-8'.

start()
Start the channel. Called before starting processus. Can be overloaded to specify specific start procedure.

class pypeman.contrib.http.**HttpRequest**(*url*, **args*, *method=None*, *headers=None*, *auth=None*, *verify=True*, *params=None*, *client_cert=None*, ***kwargs*)
Http request node :param url: url to send. :param method: 'get', 'put' or 'post', use meta['method'] if None, Default to 'get'. :param headers: headers for request, use meta['headers'] if None. :param auth: tuple or aiohttp.BasicAuth object. :param verify: verify ssl. Default True. :param params: get params in dict. List for multiple elements, ex :

{'param1': 'omega', param2: ['alpha', 'beta']}

Parameters **client_cert** – tuple with .crt and .key path

handle_request(*msg*)
generate url and handle request

process(*msg*)
handles request

class pypeman.contrib.http.**RequestNode**(**args*, ***kwargs*)

7.4.4 Time

class pypeman.contrib.time.**CronChannel**(**args*, *cron=''*, ***kwargs*)
This channel Launch processing at specified time interval. The first node of the channel receive a payload with the datetime of execution.

Params cron This set the interval. Accept any aiocron compatible string.

start()
Start the channel. Called before starting processus. Can be overloaded to specify specific start procedure.

7.4.5 XML

class pypeman.contrib.xml.**PythonToXML**(**args*, ***kwargs*)
Convert python payload to XML.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

class pypeman.contrib.xml.**XMLToPython**(**args*, ***kwargs*)
Convert XML message payload to python dict.

process(*msg*)
Implement this function in child classes to create a new Node.

Parameters **msg** – The incoming message

Returns The processed message

7.5 Message

class pypeman.message.**Message**(*content_type='application/text'*, *payload=None*, *meta=None*)
A message is the unity of informations exchanged between nodes of a channel.

A message have following properties:

attribute payload The message content.

attribute meta The message metadata.

attribute timestamp The creation date of message

attribute uuid uuid to identify message

attribute content_type Used ?

attribute ctx Current context when you want to save a message for later use.

Parameters

- **payload** – You can initialise the payload by setting this param.
- **meta** – Same as payload, you can initialise the meta by setting this param.

add_context(*key*, *msg*)
Add a msg to the *.ctx* property with specified key.

Parameters

- **key** – Key to store message.
- **msg** – Message to store.

copy()
Copy the message. Useful for channel forking purpose.

Returns A copy of current message.

static from_dict(*data*)
Convert the input dict previously converted with *.as_dict()* method in Message object.

Parameters **data** – The input dict.

Returns The message message object correponding to given data.

static from_json(*data*)
Create a message from previously saved json string.

Parameters **data** – Data to read message from.

Returns A new message instance created from json data.

log(*logger=<logging.Logger object>*, *log_level=10*, *payload=True*, *meta=True*, *context=False*)
Log a message.

Parameters

- **logger** – Logger
- **log_level** – log level for all log.
- **payload** – Whether log payload.
- **meta** – Whether log meta.
- **context** – Whether log context.

renew()
Copy the message but update the *timestamp* and *uuid*.

Returns A copy of current message with new *uuid* and *Timestamp*.

timestamp_str()
Return timestamp formated string

to_dict()
Convert the current message object to a dict. Payload is pickled.

Returns A dict with an equivalent of message

to_json()
Create json string for current message.

Returns a json string equivalent for message.

to_print(*payload=True*, *meta=True*, *context=False*)
Return a printable version of message.

Parameters

- **payload** – Whether print payload.
- **meta** – Whether print meta.
- **context** – Whether print context.

7.6 Message store

class pypeman.msgstore.**FakeMessageStore**
For testing purpose

get(*id*)
Return one message corresponding to given *id* with his status.

Parameters **id** – Message id. Message store dependant.

Returns A dict *{'id':<message_id>, 'state': <message_state>, 'message': <message_object>}*.

search(***kwargs*)
Return a list of message with store specific *id* and processed status.

Parameters

- **start** – First element.
- **count** – Count of elements since first element.
- **order_by** – Message order. Allowed values : ['timestamp', 'status'].

Returns A list of dict *{'id':<message_id>, 'state': <message_state>, 'message': <message_object>}*.

store(*msg*)
Store a message in the store.

Parameters **msg** – The message to store.

Returns Id for this specific message.

total()

Returns total count of messages

class pypeman.msgstore.**FakeMessageStoreFactory**
Return an Fake message store

get_store(*store_id*)

Parameters **store_id** – identifier of corresponding message store.

Returns A MessageStore corresponding to correct store_id.

class pypeman.msgstore.**FileMessageStore**(*path*, *store_id*)
Store a file in *<base_path>/<store_id>/<month>/<day>/* hierachy.

change_message_state(*id*, *new_state*)
Change the *id* message state.

Parameters

- **id** – Message specific store id.
- **new_state** – Target state.

count_msgs()
Count message by listing all directories. To be used at startup.

get(*id*)
Return one message corresponding to given *id* with his status.

Parameters **id** – Message id. Message store dependant.

Returns A dict *{'id':<message_id>, 'state': <message_state>, 'message': <message_object>}*.

search(*start=0*, *count=10*, *order_by='timestamp'*)
Return a list of message with store specific *id* and processed status.

Parameters

- **start** – First element.
- **count** – Count of elements since first element.
- **order_by** – Message order. Allowed values : ['timestamp', 'status'].

Returns A list of dict *{'id':<message_id>, 'state': <message_state>, 'message': <message_object>}*.

sorted_list_directories(*path*, *reverse=True*)

Parameters

- **path** – Base path
- **reverse** – reverse order

Returns List of directories in specified path ordered

start()
Called at startup to initialize store.

store(*msg*)
Store a file in *<base_path>/<store_id>/<month>/<day>/* hierachy.

total()

Returns total count of messages

class pypeman.msgstore.**FileMessageStoreFactory**(*path*)
Generate a FileMessageStore message store instance. Store a file in *<base_path>/<store_id>/<month>/<day>/* hierachy.

get_store(*store_id*)

Parameters **store_id** – identifier of corresponding message store.

Returns A MessageStore corresponding to correct store_id.

class pypeman.msgstore.**MemoryMessageStore**(*base_dict*, *store_id*)
Store messages in memory

change_message_state(*id*, *new_state*)
Change the *id* message state.

Parameters

- **id** – Message specific store id.
- **new_state** – Target state.

get(*id*)
Return one message corresponding to given *id* with his status.

Parameters **id** – Message id. Message store dependant.

Returns A dict *{'id':<message_id>, 'state': <message_state>, 'message': <message_object>}*.

search(*start=0*, *count=10*, *order_by='timestamp'*)
Return a list of message with store specific *id* and processed status.

Parameters

- **start** – First element.
- **count** – Count of elements since first element.
- **order_by** – Message order. Allowed values : ['timestamp', 'status'].

Returns A list of dict *{'id':<message_id>, 'state': <message_state>, 'message': <message_object>}*.

store(*msg*)
Store a message in the store.

Parameters **msg** – The message to store.

Returns Id for this specific message.

total()

Returns total count of messages

class pypeman.msgstore.**MemoryMessageStoreFactory**
Return a Memory message store. All message are lost at pypeman stop.

get_store(*store_id*)

Parameters **store_id** – identifier of corresponding message store.

Returns A MessageStore corresponding to correct store_id.

class pypeman.msgstore.**MessageStore**
A MessageStore keep an history of processed messages. Mainly used in channels.

change_message_state(*id*, *new_state*)
Change the *id* message state.

Parameters

- **id** – Message specific store id.
- **new_state** – Target state.

get(*id*)
Return one message corresponding to given *id* with his status.

Parameters **id** – Message id. Message store dependant.

Returns A dict *{'id':<message_id>, 'state': <message_state>, 'message': <message_object>}*.

search(*start=0*, *count=10*, *order_by='timestamp'*)
Return a list of message with store specific *id* and processed status.

Parameters

- **start** – First element.
- **count** – Count of elements since first element.
- **order_by** – Message order. Allowed values : ['timestamp', 'status'].

Returns A list of dict *{'id':<message_id>, 'state': <message_state>, 'message': <message_object>}*.

start()
Called at startup to initialize store.

store(*msg*)
Store a message in the store.

Parameters **msg** – The message to store.

Returns Id for this specific message.

total()

Returns total count of messages

class pypeman.msgstore.**MessageStoreFactory**
Message store factory class can generate Message store instance for specific store_id.

get_store(*store_id*)

Parameters **store_id** – identifier of corresponding message store.

Returns A MessageStore corresponding to correct store_id.

class pypeman.msgstore.**NullMessageStore**
For testing purpose

get(*id*)
Return one message corresponding to given *id* with his status.

Parameters **id** – Message id. Message store dependant.

Returns A dict *{'id':<message_id>, 'state': <message_state>, 'message': <message_object>}*.

search(***kwargs*)
Return a list of message with store specific *id* and processed status.

Parameters

- **start** – First element.
- **count** – Count of elements since first element.
- **order_by** – Message order. Allowed values : ['timestamp', 'status'].

Returns A list of dict *{'id':<message_id>, 'state': <message_state>, 'message': <message_object>}*.

store(*msg*)
Store a message in the store.

Parameters **msg** – The message to store.

Returns Id for this specific message.

total()

Returns total count of messages

class pypeman.msgstore.**NullMessageStoreFactory**
Return an NullMessageStore that do nothing at all.

get_store(*store_id*)

Parameters **store_id** – identifier of corresponding message store.

Returns A MessageStore corresponding to correct store_id.

7.7 Events

class pypeman.events.**Event**
Asyncio Event class.

add_handler(*handler*)
Add a new handler for this event.

fire(**args*, ***kargs*)
Fire current event. All handler are going to be executed.

getHandlerCount()
Return declared handler count.

receiver(*handler*)
Function decorator to add an handler.

remove_handler(*handler*)
Remove a previously defined handler for this event.

7.8 Remote admin

class pypeman.remoteadmin.**PypemanShell**(*url*)

do_channels(*arg*)
List avaible channels

do_exit(*arg*)
Exit program

do_list(*channel*, *arg*)
List messages of selected channel. You can specify start, end and order_by arguments

do_push(*channel*, *arg*)
Inject message with text as payload for selected channel

do_replay(*channel*, *arg*)
Replay a message list by their ids

do_select(*arg*)
Select a channel by is name. Mandatory for channel oriented command

do_start(*arg*)
Start a channel by his name

do_stop(*arg*)
Stop a channel by his name

class pypeman.remoteadmin.**RemoteAdminClient**(*loop=None*, *url='ws://localhost:8091'*)
Remote admin client. To be use by ipython shell or pypeman shell.

Params url Pypeman Websocket url.

channels()
Return a list of available channels on remote instance.

exec(*command*)
Execute any valid python code on remote instance and return stdout result.

list_msg(*channel*, *start=0*, *count=10*, *order_by='timestamp'*)
List first 10 messages on specified channel from remote instance.

Params channel The channel name.

Params start Start index of listing.

Params count Count from index.

Params order_by Message order. only 'timestamp' and '-timestamp' handled for now.

Returns list of message with status.

push_msg(*channel*, *text*)
Push a new message from text param to the channel.

Params channel The channel name.

Params text This text will be the payload of the message.

replay_msg(*channel*, *msg_ids*)
Replay specified message from id list of specified channel on remote instance.

Params channel The channel name.

Params msg_ids Message id list to replay

Returns List of result for each message. Result can be {'error': <msg_error>} for one id if error occurs.

send_command(*command*, *args=None*)
Send a command to remote instance

start(*channel*)
Start the specified channel on remote instance.

Params channel The channel name.

stop(*channel*)
Stop the specified channel on remote instance.

Params channel The channel name.

class pypeman.remoteadmin.**RemoteAdminServer**(*loop=None*, *host='localhost'*, *port='8091'*, *ssl=None*, *url=None*)
Expose json/rpc function to a client by a websocket.

channels()
Return a list of available channels.

command(*websocket*, *path*)
Generic function to handle a command from client.

exec(*command*)
Execute a python command on this instance and return the stdout result.

Parameters **command** – The python command to execute. Can be multiline.

Returns Command stdout result.

get_channel(*name*)
return channel by is name.all

list_msg(*channel*, *start=0*, *count=10*, *order_by='timestamp'*)
List first *count* messages from message store of specified channel.

Params channel The channel name.

push_msg(*channel*, *text*)
Push a message in the channel.

Params channel The channel name.

Params msg_ids The text added to the payload.

replay_msg(*channel*, *msg_ids*)
Replay messages from message store.

Params channel The channel name.

Params msg_ids The message ids list to replay.

start()
Start remote admin server

start_channel(*channel*)
Start the specified channel

Params channel The channel name to start.

stop_channel(*channel*)
Stop the specified channel

Params channel The channel name to stop.

CHAPTER 8

About

8.1 Licence

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/
```

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as

a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted

for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

```
To apply the Apache License to your work, attach the following
boilerplate notice, with the fields enclosed by brackets "{}"
replaced with your own identifying information. (Don't include
the brackets!)  The text should be enclosed in the appropriate
comment syntax for the file format. We also recommend that a
file or class name and description of purpose be included on the
same "printed page" as the copyright notice for easier
identification within third-party archives.
```

Copyright {yyyy} {name of copyright owner}

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

```
http://www.apache.org/licenses/LICENSE-2.0
```

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Python Module Index

p

Index

R

S

T

U

W

X