Pyloggr Documentation

Release 0.3

Stephane Martin

June 17, 2015

Contents

CHAPTER 1

Overview

pyloggr is a set of tools to

- centralize logs
- parse logs and apply some filters
- store logs in a convenient database
- search logs
- Free software: GPLv3 (or later) license
- Documentation: https://pyloggr.readthedocs.org.
- Github: https://github.com/stephane-martin/pyloggr

CHAPTER 2

Features

- Syslog server: implements RFC 5424 and RFC 3164 formatting, can receive logs over TCP, TCP/TLS or RELP
- Apply some filters to logs. For instance pyloggs supports the grok filter, similar to logstash
- Database storage: currently in PostgreSQL, using JSONB support
- Web frontend: pyloggr monitoring, log exploration

CHAPTER 3

Todo

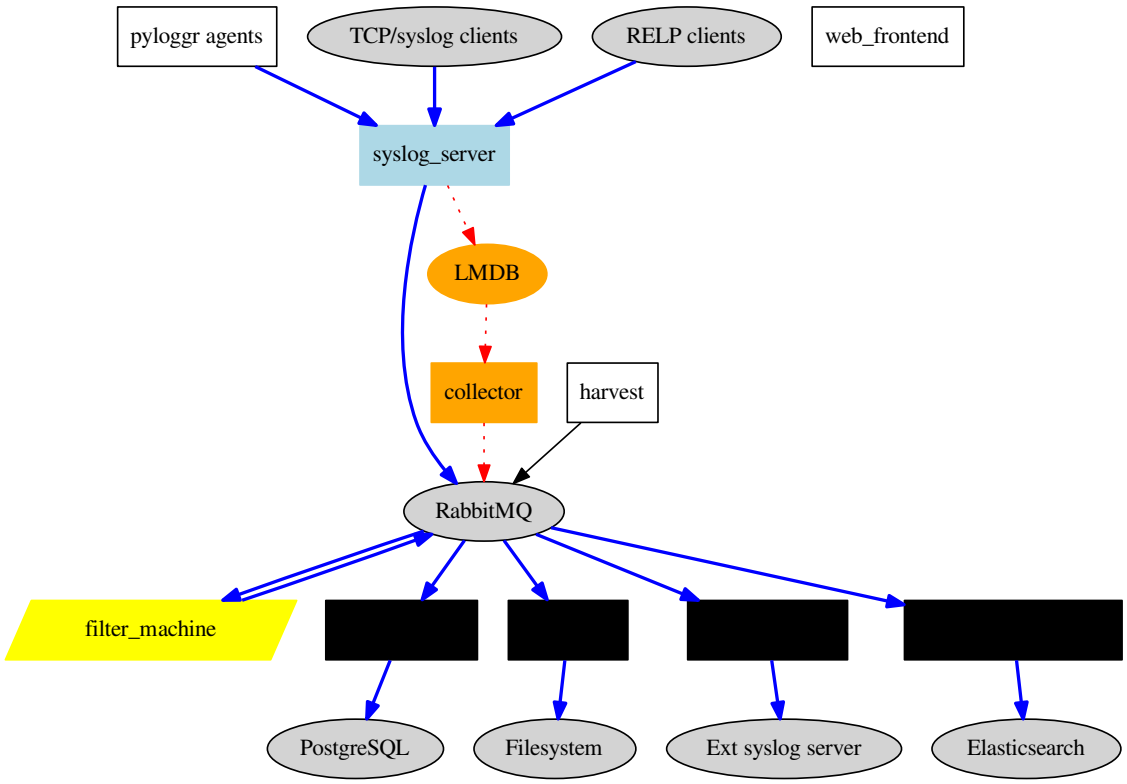
See Github issues

3.1 Architecture

pyloggr architecture is not yet fully stabilized. Nevertheless here are the main characteristics :

- pyloggr has several components. The components are supposed to be ran as independant process
- Components are based on the tornado asynchronous framework
- Communication of syslog messages between components uses RabbitMQ, so that we have good resilience properties

Components can be started/stopped using the *pyloggr_ctl* script.



3.1.1 Components

Components source is located in the *main* directory.

syslog_server

syslog_server is a... syslog server. It can receive syslog messages with TCP or RELP (RELP is a reliable syslog protocol used by Rsyslog).

Received messages can be formatted in traditional syslog format (RFC 3164), modern syslog format (RFC 5424), or as JSON messages.

When using TCP transport, both traditional LF framing, or octet framing can be used.

syslog_server can also pack several messages into a single one, using configurable *Packers*.

harvest

Sometimes you just can't use syslog... For example the paranoid production team could refuse to install rsyslog on some servers (don't laugh, that's real). pyloggr can also receive full log files using FTP, SSH, ... harvest is responsible to monitor the upload directory. When a file has been fully uploaded, it reads it and injects the log lines as syslog messages.

parser

The parser process takes messages from RabbitMQ, parses them, and applies configurable filters to each message. For example, a 'grok' filter similar to logstash is provided.

Filters application is multi-threaded.

shipper2pgsql

The shipper takes messages from RabbitMQ and stores them in the PostgreSQL database.

web_frontend

The frontend is the web interface to ploggr. It can be used to monitor ploggr activity, or to query the log database.

collector

When RabbitMQ is accidently not available, syslog_server will try to save the current messages in Redis (they it will shutdown itself so that no more syslog messages can come in).

collector processes the messages in Redis and transfers them back to RabbitMQ when RabbitMQ is back online.

3.1.2 Never lose a syslog message !

ploggr project was initially started as a prototype after looking at logstash queues. Logstash is a fine and useful piece of software (love it). But currently in the 1.4 branch, messages are stored in internal memory queues. This means that when logstash stops, or crashes, you can actually lose some lines.

Moreover, even if logstash implements the RELP protocol as a possible source, the incoming messages are ACKed very quickly (when they move to the "filter" queue actually). So any problem with filters or with outputs can generate a message loss.

That's why ploggr: - implements RELP for incoming messages - uses RabbitMQ for messages transitions - only ACKs a message from step A when step A+1 has taken responsibility

This way we can ensure that messages won't be lost.

3.2 Installation

3.2.1 Prerequisites

- python 2.7

So far ploggr is not compatible with python 3

- Redis

Redis is used for interprocess communication. It is also used as a "rescue" queue when RabbitMQ becomes unavailable.

- RabbitMQ

RabbitMQ is used for passing syslog messages between ploggr components.

- PostgreSQL

Pyloggr currently uses a PostgreSQL database to store syslog messages.

- A C compiler may be needed too for the python packages that pyloggr depends on.

3.2.2 Install with pip

When pyloggr will be usable it will be published to pip.

If you'd like to install it now, please use a virtualenv and the setup.py script.

3.3 Configuration

3.3.1 Configuration directory

PYLOGGR_CONFIG_DIR

3.4 Deployment

3.4.1 No daemons

3.4.2 Supervisord

3.5 Credits

3.5.1 Main author

- Stephane Martin <stephane.martin_github@vesperal.eu>

3.5.2 Contributors

None yet. Why not be the first?

3.6 History

0.3 (2015-06-15)

- listen on UDP
- LZ4 compression for RELP
- drop permissions if possible
- more shippers
- packers
- LMDB for inter-process communication

- syslog agent (LMDB as persistent store)

0.1.3 (2015-03-23)

- refactored the configuration handling
- launchers were refactored

0.1.2 (2015-03-17)

- several minor fixes

0.1.1 (2015-03-14)

- More flexible configuration (no more parenthesis, algebra on conditions)
- Code refactoring

0.1.0 (2015-03-12)

- First release on github.

3.7 API Documentation

3.7.1 pyloggr package

Base package for all pyloggr stuff

pyloggr.event

The pyloggr.event module mainly provides the Event class.

Event provides an abstraction of a syslog event.

class **Event** (*procid=u'-'*, *severity=u''*, *facility=u''*, *app_name=u''*, *source=u''*, *programname=u''*, *syslogtag=u''*, *message=u''*, *uuid=None*, *hmac=None*, *timereported=None*, *timegenerated=None*, *timehmac=None*, *custom_fields=None*, *structured_data=None*, *tags=None*, *iut=1*, ***kwargs*)

Bases: `object`

Represents a syslog event, with optional tags, custom fields and structured data

Variables

- **procid** (*int*) –
- ***severity*** (*str*) –
- ***facility*** (*str*) –
- ***app_name*** (*str*) –
- ***source*** (*str*) –
- **programname** (*str*) –
- **syslogtag** (*str*) –
- ***message*** (*str*) –
- ***uuid*** (*str*) –
- ***hmac*** (*str*) –
- ***timereported*** (*Datetime*) –

- **timegenerated** (*Datetime*) –
- **timehmac** (*Datetime*) –
- **custom_fields** (*dictionnary of custom fields*) –
- **structured_data** (*dictionnary representing syslog structured data*) –
- **tags** (*set of str*) –

__contains__(*key*)
Return True if event has the given custom field, and the field is not empty

Parameters **key** (*str*) – custom field key

Return type bool

__delitem__(*key*)
Deletes a custom field

Parameters **key** (*str*) – custom field key

__eq__(*other*)
Two events are equal if they have the same UUID

Return type bool

__ge__(*other*)
x.__ge__(y) <==> x>=y

__getitem__(*key*)
Return a custom field, given its key

Parameters **key** (*str*) – custom field key

__gt__(*other*)
x.__gt__(y) <==> x>y

__le__(*other*)
x.__le__(y) <==> x<=y

__lt__(*other*)
self < other if self.timereported < other.timereported

Return type bool

__setitem__(*key*, *values*)
Sets a custom field

Parameters

- **key** (*str*) – custom field key
- **values** (*iterable*) – custom field values

_parse_trusted()
Parse the "trusted fields" that rsyslog could generate

add_tags(*tags*)
Add some tags to the event

Parameters **tags** – a list of tags

app_name
Name of application that generated the event

apply_filters(*filters*)
Apply some filters to the event

Parameters **filters** – filters to apply

custom_fields
Small helper to access pyloggr specific custom fields

dump(*frmt='JSON'*, *fname=None*)
Dump the event

Explicit format: a string with the following possible placeholders: $DATE, $DATETIME, $MESSAGE, $SOURCE, $APP_NAME, $SEVERITY, $FACILITY, $PROCID, $UUID, $TAGS

Parameters

- **frmt** – dumping format (JSON, MSGPACK, RFC5424, RFC3164, RSYSLOG, ES or an explicit format)
- **fname** – if not None, write the dumped string to fname file

Returns dumped string

Raises OSError if file operation fails (when fname is not None)

dump_dict()
Serialize the event as a native python dict

Return type dict

dump_json()
Dump the event in JSON format

Return type str

dump_msgpack()
Dump the event using msgpack

dump_rfc3164()
Dump the event into a RFC 3164 old-style syslog string

dump_rfc5424()
Dump the event into a RFC 5424 compliant string

dump_rsyslog()
Dump the event as RSYSLOG_FileFormat

see: http://www.rsyslog.com/doc/v8-stable/configuration/templates.html

dump_sql(*cursor*)
Dumps the event as a SQL insert statement

Parameters **cursor** – SQL cursor

Return type str

dumps_elastic()
Dumps in JSON suited for Elasticsearch

Return type str

facility
Event facility

generate_hmac(*self*, *verify_if_exists=True*)
Generate a HMAC from the fields: severity, facility, app_name, source, message, timereported

Parameters **verify_if_exists** (*bool*) – verify event HMAC if it has one

Returns a base 64 encoded HMAC

Return type str

Raises InvalidSignature if HMAC already exists but is invalid

generate_uuid(*new_uuid=None*)
Generate a UUID for the current event

Parameters **new_uuid** – if given, sets the UUID to new_uuid. if not given generate a UUID.

Returns new UUID

Return type str

hmac
Return the event HMAC.

If event doesn't have a HMAC, return empty string If event has a HMAC and is not dirty, return HMAC If event is dirty, compute the new HMAC and return it

classmethod load(*s*)
Try to deserialize an Event from a string or a dictionnary. *load* understands JSON events, RFC 5424 events and RFC 3164 events, or dictionnary events. It automatically detects the type, using regexp tests.

Parameters **s** (*str or dict or bytes*) – string (JSON or RFC 5424 or RFC 3164) or dictionnary

Returns The parsed event

Return type *Event*

Raises ParsingError if deserialization fails

static make_arrow_datetime(*dt*)
Parse a date-time value and return the corresponding Arrow object

Parameters **dt** (*Arrow or datetime or str*) – date-time

Returns Arrow object

static make_facility(*facility*)
Return a normalized facility value

Parameters **facility** (*int or str or unicode*) – syslog facility (integer) or string

static make_severity(*severity*)
Return a normalized severity value

Parameters **severity** (*int or str or unicode*) – syslog priority (integer) or string

message
Event message

classmethod parse_bytes_to_event(*bytes_ev*, *hmac=False*, *swallow_exceptions=False*)
Parse some bytes into an *pyloggr.event.Event* object

Parameters

- **bytes_ev** (*bytes*) – the event as bytes
- **hmac** (*bool*) – generate/verify a HMAC
- **swallow_exceptions** (*bool*) – if True, return None rather than raising validation exceptions

Returns the new Event object

Return type *Event*

Raises

- **ParsingError** – if bytes could not be parsed correctly
- **InvalidSignature** – if *hmac* is True and a HMAC already exists, but is invalid

priority
Return the event computed syslog priority

remove_tags(*tags*)
Remove some tags from the event. If the event does not really have such tag, it is ignored.

Parameters **tags** – a list of tags

severity
Event severity

source
Event source hostname

tags
Access the event tags. Returns a set.

timegenerated
event "first seen" datetime

timehmac
datetime, when the event HMAC was created

timereported
event creation datetime

update_cfield(*key*, *values*)
Append some values to custom field *key*

Parameters

- **key** – custom field key
- **values** – iterable

update_cfields(*d*)
Add some custom fields to the event

Parameters **d** (*dict*) – a dictionnary of new fields

update_uuid_and_hmac()
If event is dirty (core fields have been modified), generate UUID and HMAC

uuid
Return the event UUID. If event is dirty, generate a new UUID and return it.

verify_hmac()
Verify event's HMAC

Throws an InvalidSignature exception if HMAC is invalid

Returns True

Return type bool

Raises InvalidSignature if HMAC is invalid

exception ParsingError(**args*, ***kwargs*)
Bases: exceptions.ValueError

Triggered when a string can't be parsed into an *Event*

pyloggr.cache

This module defines the Cache class and the *cache* singleton. They are used to store and retrieve data from Redis. For example, the syslog server process uses Cache to stores information about currently connected syslog clients, so that the web frontend is able to display that information.

Clients should typically use the *cache* singleton, instead of the *Cache* class.

Cache initialization is done by *initialize* class method. The *initialize* method should be called by launchers, at startup time.

Note: In a development environment, if Redis has not been started by the OS, Redis can be started directly by pyloggr using configuration item `REDIS['try_spawn_redis'] = True`

---

cache
Cache singleton

class Cache
Bases: object

Cache class abstracts storage and retrieval from Redis

Variables **redis_conn** (StrictRedis) – underlying *StrictRedis* connection object (class variable)

classmethod initialize()
Cache initialization.

initialize tries to connect to redis and sets *redis_conn* class variable. If connection fails and `REDIS['try_spawn_redis']` is set, it also tries to spawn the Redis process.

Raises CacheError when redis initialization fails

class SyslogCache(*server_id*)
Bases: object

Stores information about the running pyloggr's syslog processes in a Redis cache

Parameters

- **redis_conn** (StrictRedis) – the Redis connection
- **server_id** (*int*) – The syslog process server_id

clients
Return the list of clients for this syslog process

Returns list of clients or None (Redis not available)

ports
Return the list of ports that the syslog process listens on

Returns list of ports (numeric and socket name)

Return type list

status
Returns syslog process status

Returns Boolean or None (if Redis not available)

class SyslogServerList
Bases: object

Encapsulates the list of syslog processes

__delitem__(*server_id*)
Deletes a SyslogCache object (used when the syslog server shuts down)

Parameters **server_id** – process id

__getitem__(*server_id*)
Return a SyslogCache object based on process id

Parameters **server_id** – process id

Return type *SyslogCache*

__len__()
Returns the number of syslog processes

Returns how many syslog processes, or None (Redis not available)

Return type int

pyloggr.config

Small hack to be able to import configuration from environment variable.

class Config
Bases: object

Config object can be imported and contains configuration parameters

class ConsumerConfig(*queue*, *qos=None*, *binding_key=None*)
Bases: *pyloggr.config.GenericConfig*

Parameters for RabbitMQ consumers

class FilterMachineConfig(*source*, *destination*, *filters*)
Bases: *pyloggr.config.GenericConfig*

Filter machines configuration

class GenericConfig
Bases: object

Base class for configurations

classmethod from_dict(*d*)
Build configuration object from a dictionnary

Parameters **d** – dictionnary

class GlobalConfig(*HMAC_KEY*, *RABBITMQ_HTTP*, *COOKIE_SECRET*, *MAX_WAIT_SECONDS_BEFORE_SHUTDOWN=10*, *PIDS_DIRECTORY='~/pids'*, *SLEEP_TIME=60*, *UID=None*, *GID=None*, *HTTP_PORT=8080*, *EXCHANGE_SPACE='~/lmdb/exchange'*, *RESCUE_QUEUE_DIRNAME='~/lmdb/rescue'*, ***kwargs*)
Bases: object

Placeholder for all configuration parameters

classmethod load(*config_dirname*)

Parameters **config_dirname** – configuration directory path

Return type *GlobalConfig*

class HarvestDirectory(*directory_name*, *app_name=u''*, *packer_group=u''*, *recursive=False*, *facility=u''*, *severity=u''*, *source=u''*)
Bases: *pyloggr.config.GenericConfig*

Directories to harvest file logs from

class LoggingConfig(*level='DEBUG'*, ***kwargs*)
Bases: *pyloggr.config.GenericConfig*

Where to log

class PublisherConfig(*exchange*, *application_id='pyloggr'*, *event_type=''*, *binding_key=''*)
Bases: *pyloggr.config.GenericConfig*

Parameters for RabbitMQ publishers

class RabbitMQConfig(*host*, *user*, *password*, *port=5672*, *vhost='pyloggr'*)
Bases: *pyloggr.config.GenericConfig*

RabbitMQ connection parameters

class SSLConfig(*certfile*, *keyfile*, *ssl_version='PROTOCOL_SSLv23'*, *ca_certs=''*, *cert_reqs=0*)
Bases: *pyloggr.config.GenericConfig*

Syslog servers SSL configuration

class Shipper2FSConfig(*directory*, *filename*, *source_queue*, *seconds_between_flush=10*, *frmt='RSYSLOG'*)
Bases: *pyloggr.config.GenericConfig*

Parameters for filesystem shippers

class Shipper2PGSQL(*host*, *user*, *password*, *source_queue*, *event_stack_size=500*, *port=5432*, *dbname='pyloggr'*, *tablename='events'*, *max_pool=10*, *connect_timeout=10*)
Bases: *pyloggr.config.GenericConfig*

Parameters for PostgreSQL shippers

class Shipper2SyslogConfig(*host*, *port*, *source_queue*, *use_ssl=False*, *protocol='tcp'*, *frmt='RFC5424'*, *source_qos=500*, *verify=True*, *hostname=''*, *ca_certs=None*, *client_cert=None*, *client_key=None*)
Bases: *pyloggr.config.GenericConfig*

Parameters for syslog shippers

class SyslogAgentConfig(*UID=None*, *GID=None*, *destinations=None*, *tcp_ports=None*, *udp_ports=None*, *relp_ports=None*, *pause=5*, *lmdb_db_name='~/lmdb/agent_queue'*, *localhost_only=True*, *server_deadline=120*, *socket_names=None*, *pids_directory='~/pids'*, *logs_directory='~/logs'*, *HMAC_KEY=None*, *logs_level='DEBUG'*)
Bases: *pyloggr.config.GenericConfig*

Parameters for the syslog agent

class SyslogServerConfig(*name*, *ports=None*, *stype='tcp'*, *localhost_only=False*, *socket_names=None*, *ssl=None*, *packer_groups=None*, *compress=False*)
Bases: *pyloggr.config.GenericConfig*

Parameters for syslog servers

set_configuration(*configuration_directory*)
Set up configuration

Parameters **configuration_directory** – configuration parent directory

Returns

set_logging(*filename*, *level='DEBUG'*)
Set logging configuration

Parameters

- **filename** (*str*) – logs file name
- **level** (*str*) – logs verbosity

3.7.2 pyloggr.main package

The *main* subpackage contains code for the different pyloggr processes :

- *pyloggr.main.syslog_server*: Syslog server. Listens for syslog events and stores them in RabbitMQ queues.
- *pyloggr.main.filter_machine*: Event parser. Gets some events from RabbitMQ, apply filters, and stores events back in RabbitMQ.
- *pyloggr.main.shipper2pgsql*: Gets events from RabbitMQ, and ships them to a Postgresql database.
- *pyloggr.main.web_frontend*: Web interface. Provides monitoring of other processes, and log searching.

---

pyloggr.main.agent

Local syslog agent

class Publications(*syslog_agent_config*, *publication_queue*, *published_messages_queue*, *failed_messages_queue*, *syslog_server_is_available*)
Bases: threading.Thread

The Publications thread handles the connection to the remote syslog server to publish the messages

class RetrieveMessagesFromLMDB(*lmdb_db_name*, *publication_queue*, *published_messages_queue*, *failed_messages_queue*, *syslog_server_is_available*, *pause*)
Bases: threading.Thread

The *RetrieveMessagesFromLMDB* thread gets messages from LMDB and pushes them to the Publications thread, via a queue

class StoreMessagesInLMDB(*received_messages_queue*, *lmdb_db_name*)
Bases: threading.Thread

The *StoreMessagesInLMDB* thread gets messages from the TCP, UDP and unix sockets, via a queue, and pushes them to LMDB

class SyslogAgent(*syslog_agent_config*)
Bases: *pyloggr.syslog.server.BaseSyslogServer*

Syslog agent

SyslogServer listens for syslog messages (RELP, RELP/TLS, TCP, TCP/TLS, Unix socket) and sends messages to a remote TCP/Syslog or RELP server.

_start_syslog()
Start to listen for syslog clients

Note: Tornado coroutine

_stop_syslog()
Stop listening for syslog connections

Note: Tornado coroutine

handle_data(*data*, *sockname*, *peername*)
Handle UDP connections

handle_stream(**args*, ***kwargs*)
Handle TCP and RELP clients

launch()
Starts the agent

Note: Tornado coroutine

shutdown(**args*, ***kwargs*)
Authoritarian shutdown

stop_all()
Stops completely the server. Stop listening for syslog clients. Close connection to remote server.

Note: Tornado coroutine

class **SyslogAgentClient**(*stream*, *address*, *syslog_parameters*, *received_messages*)
Bases: *pyloggr.syslog.server.BaseSyslogClientConnection*

Handles TCP connections

_process_event(*bytes_event*, *protocol*, *relp_event_id=None*)
Handle TCP and RELP connections

pyloggr.main.collector

Collect events from the rescue queue and try to forward them to RabbitMQ

pyloggr.main.filter_machine

The Filter Machine process can be used to apply series of filters to events

class **FilterMachine**(*consumer_config*, *publisher_config*, *filters_filename*)
Bases: `object`

Implements an Event parser than retrieves events from RabbitMQ, apply filters, and pushes back events to RabbitMQ.

_apply_filters(*message*)
Apply filters to the event inside the RabbitMQ message.

Note: This method is executed in a separated thread.

Parameters **message** (*pyloggr.consumer.RabbitMQMessage*) – event to apply filters to, as a RabbitMQ message

Returns tuple(message, parsed event). parsed event is None when event couldn't be parsed.

Return type tuple(pyloggr.consumer.RabbitMQMessage, pyloggr.event.Event)

launch(**args*, ***kwargs*)
Starts the parser

Note: Coroutine

shutdown(**args*, ***kwargs*)
Shutdowns (stops definitely) the parser

stop(**args*, ***kwargs*)
Stops the parser

pyloggr.main.harvest

Monitor a directory on the filesystem, and parse new files as logs

pyloggr.main.shipper2fs

Ships events from RabbitMQ to the filesystem

class FSQueue(*filename*, *period*, *frmt*)
Bases: `object`

Store events that have to be exported to a given filename

append(*event*)
Add an event to the queue. The coroutine resolves when the event has been exported.

Parameters **event** (pyloggr.event.Event) – event

flush()
Actually flush the events to the file

stop()
Stop the queue

class FilesystemShipper(*rabbitmq_config*, *export_fs_config*)
Bases: `object`

The FilesystemShipper takes events from a RabbitMQ queue and writes them on filesystem

Parameters

- **rabbitmq_config** (*pyloggr.rabbitmq.Configuration*) – RabbitMQ configuration
- **export_fs_config** (pyloggr.config.Shipper2FSConfig) – Log export configuration

_append(**args*, ***kwargs*)

export(*message*)
Export event to filesystem

Parameters **message** (RabbitMQMessage) – RabbitMQ message

launch()
Start shipper2fs

shutdown()
Shutdowns (stops definitely) the shipper.

stop()
Stops the shipper

pyloggr.main.shipper2pgsql

Ship events to a PostgreSQL database

class PostgresqlShipper(*rabbitmq_config*, *pgsql_config*)
Bases: object

PostgresqlShipper retrieves events from RabbitMQ, and inserts them in PostgreSQL

launch(**args*, ***kwargs*)
Starts the shipper

- Opens a connection to RabbitMQ
- Opens a pool to PostgreSQL
- Consumes messages from RabbitMQ
- Parses messages as regular syslog events
- Periodically ships the events to PostgreSQL

shutdown(**args*, ***kwargs*)
Shutdowns (stops definitely) the shipper.

stop(**args*, ***kwargs*)
Stops the shipper

pyloggr.main.shipper2syslog

Ships events from RabbitMQ to a Syslog server

class SyslogShipper(*rabbitmq_config*, *shipper_config*)
Bases: object

SyslogShipper retrieves events from RabbitMQ, and forwards them to a Syslog server

_forward_message(**args*, ***kwargs*)

launch(**args*, ***kwargs*)
Starts the shipper

- Open a connection to the remote syslog server
- Open a connection to RabbitMQ
- Consume messages from RabbitMQ
- Parse messages as regular syslog events
- Ship events to the remote syslog server

shutdown(**args*, ***kwargs*)
Shutdown the shipper

stop(**args*, ***kwargs*)
Stop the shipper

pyloggr.main.syslog_server

This module provides stuff to implement a the main syslog/RELP server with Tornado

class ListOfClients
Bases: `object`

Stores the current Syslog clients, sends notifications to observers, publishes the list of clients in Redis

__getitem__(*client_id*)

add(*client_id*, *client*)

remove(*client_id*)

classmethod set_server_id(*server_id*)

Parameters **server_id** (*int*) – process number

class MainSyslogServer(*rabbitmq_config*, *syslog_parameters*, *server_id*)
Bases: *pyloggr.syslog.server.BaseSyslogServer*

Tornado syslog server

SyslogServer listens for syslog messages (RELP, RELP/TLS, TCP, TCP/TLS, Unix socket) and sends messages to RabbitMQ.

_start_syslog()
Start to listen for syslog clients

Note: Tornado coroutine

_stop_syslog()
Stop listening for syslog connections

Note: Tornado coroutine

handle_data(*data*, *sockname*, *peername*)
Handle UDP syslog

Parameters

- **data** – data sent
- **sockname** – the server socket info
- **peername** – the client socket info

handle_stream(*stream*, *address*)
Called by tornado when we have a new client.

Parameters

- **stream** (*IOStream*) – IOStream for the new connection
- **address** (*tuple*) – tuple (client IP, client source port)

Note: Tornado coroutine

launch()
Starts the server

•First we try to connect to RabbitMQ

•If successfull, we start to listen for syslog clients

Note: Tornado coroutine

shutdown(**args*, ***kwargs*)
Authoritarian shutdown

stop_all()
Stops completely the server. Stop listening for syslog clients. Close connection to RabbitMQ.

Note: Tornado coroutine

class Notification(*dictionnary*, *routing_key*)
Bases: tuple

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values

_replace(*_self*, ***kwds*)
Return a new Notification object replacing specified fields with new values

dictionnary
Alias for field number 0

routing_key
Alias for field number 1

class Publicator(*syslog_servers_conf*, *rabbitmq_config*)
Bases: object

Publicator manages the RabbitMQ connection and actually makes the publish calls.

Publicator runs in its own thread, and has its own IOLoop.

Parameters

- **syslog_servers_conf** – Syslog configuration (used to initialize packers)
- **rabbitmq_config** – RabbitMQ connection parameters

_do_publish_notification(*message*)

_do_publish_syslog(*message*)

init_thread()
Publicator thread: start a new IOLoop, make it current, call *_do_start* as a callback

notify_observers(*d*, *routing_key=''*)
Send a notification via RabbitMQ

Parameters

- **d** (*dict*) – dictionnary to send as a notification
- **routing_key** (*str*) – RabbitMQ routing key

publish_syslog_message(*protocol*, *server_port*, *client_host*, *bytes_event*, *client_id*, *relp_id=None*)
Ask *publications* to publish a syslog event to RabbitMQ. Can be called by any thread

Parameters

- **protocol** (*str*) – 'tcp' or 'relp'
- **server_port** (*int*) – which syslog server port was used to transmit the event
- **client_host** (*str*) – client hostname that sent the event
- **bytes_event** (*bytes*) – the event as bytes
- **client_id** (*str*) – SyslogClientConnection client_id
- **relp_id** (*int*) – event RELP id

rabbitmq_status()
Return True if we have an established connection to RabbitMQ

shutdown()
Ask Publicator to shutdown. Can be called by any thread.

start()
Start *publications* own thread

class SyslogClientConnection(*stream*, *address*, *syslog_parameters*)
Bases: *pyloggr.syslog.server.BaseSyslogClientConnection*

Encapsulates a connection with a syslog client

_process_event(*bytes_event*, *protocol*, *relp_event_id=None*)
_process_relp_event(bytes_event, relp_event_id) Process a TCP syslog or RELP event.

Parameters

- **bytes_event** – the event as *bytes*
- **protocol** – relp or tcp
- **relp_event_id** – event RELP ID, given by the RELP client

after_published_relp(**args*, ***kwargs*)
Called after an event received by RELP has been published in RabbitMQ

Parameters

- **status** – True if the event was successfully sent
- **event** – the Event object

- **relp_id** – event RELP id

disconnect()
Disconnects the client

on_stream_closed()
Called when a client has been disconnected

props
Return a few properties for this client

Return type dict

class SyslogMessage(*protocol*, *server_port*, *client_host*, *bytes_event*, *client_id*, *relp_id*, *total_messages*)
Bases: tuple

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values

_replace(*_self*, ***kwds*)
Return a new SyslogMessage object replacing specified fields with new values

bytes_event
Alias for field number 3

client_host
Alias for field number 2

client_id
Alias for field number 4

protocol
Alias for field number 0

relp_id
Alias for field number 5

server_port
Alias for field number 1

total_messages
Alias for field number 6

after_published_tcp(**args*, ***kwargs*)
Called after an event received by TCP has been tried to be published in RabbitMQ

Parameters

- **status** – True if the event was successfully sent
- **event** – the Event object
- **bytes_event** – the event as bytes

pyloggr.main.web_frontend

Pyloggr Web interface

class PgSQLStats
Bases: *pyloggr.utils.observable.Observable*

Gather information from the database

class QueryLogs (*application*, *request*, ***kwargs*)
Bases: tornado.web.RequestHandler

Query the log database

class RabbitMQStats
Bases: *pyloggr.utils.observable.Observable*

Gather information from RabbitMQ management API

class Status
Bases: *pyloggr.utils.observable.Observable*

Encapsulates the status of the the various pyloggr components

class StatusPage (*application*, *request*, ***kwargs*)
Bases: tornado.web.RequestHandler

Displays a status page

class SyslogClientsFeed (*application*, *request*, ***kwargs*)
Bases: tornado.websocket.WebSocketHandler, *pyloggr.utils.observable.Observer*

Websocket used to talk with the browser

notified (*d*)
notified is called by observables, when some event is meant to be communicated to the web frontend

Parameters **d** (*dict*) – the event data to transmit to the web frontend

class SyslogServers
Bases: *pyloggr.utils.observable.Observable*, *pyloggr.utils.observable.Observer*

Data about the running Pyloggr's syslog servers

Notified by Rabbitmq Observed by Websocket

class WebServer (*sockets*)
Bases: object

Pyloggr process for the web frontend part

3.7.3 pyloggr.rabbitmq package

The pyloggr.rabbitmq subpackage provides classes for publishing and consuming to/from RabbitMQ. Pika library is used, but the pika callback style has been workarounded in coroutines.

---

exception RabbitMQConnectionError
Bases: socket.error

Exception triggered when connection to RabbitMQ fails

class **RabbitMQMessage**(*delivery_tag*, *props*, *body*, *channel*)
Bases: `object`

Represents a message from RabbitMQ

Consumer.start_consuming returns a queue. Elements in the queue have *RabbitMQMessage* type.

ack()
Acknowledge the message to RabbitMQ

nack()
Acknowledge NOT the message to RabbitMQ

consumer

Provide the Consumer class to manage a consumer connection to RabbitMQ

class **Consumer**(*rabbitmq_config*)
Bases: `object`

A consumer connects to some RabbitMQ instance and eats messages from it

Parameters **rabbitmq_config** (*pyloggr.config.RabbitMQBaseConfig*) – RabbitMQ consumer configuration

_on_consumer_cancelled_by_server(*method_frame=None*)
Invoked by pika when RabbitMQ sends a Basic.Cancel for a consumer.

_on_message(*unused_channel*, *basic_deliver*, *properties*, *body*)
Invoked by pika when a message is delivered from RabbitMQ.

consuming
Returns true if the consumer is actually in consuming state

start()
Opens the connection to RabbitMQ as a consumer

Returns a Toro.Event object that triggers when the connection to RabbitMQ is lost

---

Note: Coroutine

---

start_consuming()
Starts consuming messages from RabbitMQ

Returns a Toro message queue that stores the messages when they arrive

Return type *SimpleToroQueue*

stop()
Shutdowns the connection to RabbitMQ and stops the consumer

Note: Tornado coroutine

---

stop_consuming()
Stops consuming messages from RabbitMQ

Note: Tornado coroutine

---

publisher

The publisher module provides the *Publisher* class to publish messages to RabbitMQ

class **Publisher** (*rabbitmq_config*, *base_routing_key=u''*)
Bases: `object`

Publisher encapsulates the logic for async publishing to RabbitMQ

Parameters

- **rabbitmq_config** (*pyloggr.config.RabbitMQBaseConfig*) – RabbitMQ connection parameters
- **base_routing_key** (*str*) – This routing key will be used if no routing_key is provided to publish methods

publish (*exchange*, *body*, *routing_key=''*, *message_id=None*, *headers=None*, *content_type="application/json"*, *content_encoding="utf-8"*, *persistent=True*, *application_id=None*, *event_type=None*)
Publish a message to RabbitMQ

Parameters

- **exchange** (*str*) – publish to this exchange
- **body** (*str*) – message body
- **routing_key** (*str*) – optional routing key
- **message_id** (*str*) – optional ID for the message
- **headers** (*dict*) – optional message headers
- **content_type** (*str*) – message content type
- **content_encoding** (*str*) – message charset
- **persistent** (*bool*) – if True, message will be persisted in RabbitMQ
- **application_id** (*str*) – optional application ID
- **event_type** (*str*) – optional message type

:param : :type : rtype: bool

Note: Coroutine

---

publish_event (*event*, *routing_key=u''*, *exchange=u''*, *application_id=u''*, *event_type=u''*)
Publish an Event object in RabbitMQ. Always persistent.

Parameters

- **event** (pyloggr.event.Event) – Event object
- **routing_key** (*str or unicode*) – RabbitMQ routing key
- **exchange** (*str or unicode*) – optional exchange (override global config)
- **application_id** (*str or unicode*) – optional application ID (override global config)
- **event_type** (*str or unicode*) – optional event type (override global config)

Note: Tornado coroutine

---

start()
Starts the publisher.

start() raises RabbitMQConnectionError if no connection can be established. If connection succeeds, it returns a toro.Event object that will resolve when connection will be lost

Note: Coroutine

stop()
Stops the publisher

Note: Tornado coroutine

notifications_consumer

class NotificationsConsumer(*rabbitmq_config*)
Bases: *pyloggr.rabbitmq.consumer.Consumer*, *pyloggr.utils.observable.Observable*

Consumes notification that were posted in RabbitMQ

Parameters

- **rabbitmq_config** (*pyloggr.config.RabbitMQBaseConfig*) – RabbitMQ connection parameters (to consume notifications from RabbitMQ)
- **binding_key** (*str*) – Binding key to filter notifications

start_consuming(**args*, ***kwargs*)
Start consuming notifications from RabbitMQ and notify observers.

3.7.4 pyloggr.utils package

The *utils* subpackage provides various tools used by other packages.

ask_question(*question*)
Ask a Y/N question on command line

Parameters **question** – question text

Returns user answer

Return type bool

check_directory(*dname*, *uid=None*, *gid=None*, *create=True*)
Checks that directory *dname* exists (we create it if needed), is really a directory, and is writeable

Parameters **dname** – directory name

Returns absolute directory name

Return type str

Raises OSError when tests fail

chown_r(*path*, *uid*, *gid*)
Recursively chown a directory

Parameters

- **path** – directory path
- **uid** – numeric UID
- **gid** – numeric GID

drop_capabilities(*all=True*)
Drop capabilities on Linux in case pyloggr runs as root. Only keep "net_bind_service" and "syslog".

make_dir_r(*path*)
Recursively create directories

Parameters **path** – directory to create

Returns list of created directories

read_next_token_in_stream(*stream*)
Reads the stream until we get a space delimiter

remove_pid_file(*name*)
Try to remove PID file

Parameters **name** – PID name

sanitize_key(*key*)
Remove unwanted chars (=, spaces, quote marks, [], (), commas) from key. Convert to pure ASCII.

Parameters **key** – key

Returns sanitized key

Return type unicode

sanitize_tag(*tag*)
Remove unwanted chars from tag

Parameters **tag** – tag

Returns sanitized tag

Return type unicode

sleep(*duration*, *wake_event=None*, *threading_event=None*)
Return a Future that just 'sleeps'. The Future can be interrupted in case of process shutdown.

Parameters

- **duration** (*int*) – sleep time in seconds
- **wake_event** (*toro.Event*) – optional event to wake the sleeper
- **threading_event** (*threading.Event*) – optional *threading.Event* to wake up the sleeper

Return type Future

lmdb_wrapper

Small wrapper around lmdb

class LmdbWrapper(*path*, *size=52428800*)
Bases: `object`

Wrapper around lmdb that eases storage and retrieval of JSONable python objects

classmethod get_instance(*path*)

Parameters **path** (*str*) – database directory name

Returns LmdbWrapper object

Return type *LmdbWrapper*

observable

class NotificationProducer

Bases: *pyloggr.utils.observable.Observable*

A NotificationProducer produces some notifications and sends them to RabbitMQ

notify_observers(*d*, *routing_key=None*)

Parameters

- **d** (*dict*) – a message to send to observers
- **routing_key** (*str*) – routing key for the message

Note: Tornado coroutine

class Observable

Bases: object

An observable produces notifications and sends them to observers

notify_observers(*d*, *routing_key=None*)

Notify observers that the observable has a message for them

Parameters

- **d** (*dict*) – message
- **routing_key** – unused

Note: Tornado coroutine

register(*observer*)

Subscribe an observe for future notifications

Parameters **observer** (Observer) – object that implements the Observer interface

unregister(*observer*)

Unsubscribe an observer

Parameters **observer** (Observer) – object that implements the Observer interface

unregister_all()

Unsubscribe all observers

class Observer

Bases: object

Implemented by classes that should be observers

simple_queue

Simplified queues

class SimpleToroQueue(*io_loop=None*)
Bases: `object`

Simplified Toro queue without size limit

empty()
Return `True` if the queue is empty, `False` otherwise

get_all()
Remove ans return all items from the queue, without blocking

get_nowait()
Remove and return an item from the queue without blocking.

Return an item if one is immediately available, else raise `queue.Empty`.

get_wait(*deadline=None*)
Remove and return an item from the queue. Returns a Future.

The Future blocks until an item is available, or raises `toro.Timeout`.

Parameters deadline – Optional timeout, either an absolute timestamp (as returned by `io_loop.time()`) or a

`datetime.timedelta` for a deadline relative to the current time.

put(*item*)
Put an item into the queue (without waiting)

Parameters item – item to add

qsize()
Return number of items in the queue

class ThreadSafeQueue
Bases: `object`

Simplified thread-safe/coroutine queue, without size limit

get()
Pop one item from the queue, without waiting

get_all()
Pop all items from the queue, without waiting

get_wait(*deadline=None*)
Wait for an available item and pop it from the queue

Parameters deadline – optional deadline

put(*item*)
Put an item on the queue

Parameters item – item

TimeoutTask(*func*, *deadline=None*, **args*, ***kwargs*)
Encapsulate a Tornado Task with a deadline

structured_data

Parser for rfc5424 structured data and wrapper classes

class **StructuredData**(*d=None*)
Bases: dict

Encapsulate RFC 5424 structured data

dump()
Dump the structured data as a RFC 5424 string

classmethod **parse**(*s*)
Parse structured data from string into dict of dict

Parameters **s** – string

update(*e=None*, ***f*)
Update structured data using another dict. Values are added to the values sets.

class **StructuredDataNamesValues**(*d=None*)
Bases: dict

Dict subclass. Values are sets of unicode strings. Keys are sanitized before used.

add(*key*, *values*)
Add values to the 'key' set

Parameters

- **key** – key
- **values** – iterable

dump()
Return string representation

update(*e=None*, ***f*)
Update the object using another dict. New values are added to the values set.

split_escape(*s*)
Split string by comma delimiter, excepted escaped commas

Parameters **s** (*str*) – string

Return type str

3.7.5 pyloggr.syslog package

The pyloggr.syslog subpackage provides syslog client and syslog server implementations

syslog.base

Base class for syslog TCP and RELP clients

syslog.relp_client

RELP syslog client

class **RELPClient**(*server*, *port*, *use_ssl=False*, *verify_cert=True*, *hostname=None*, *ca_certs=None*, *client_key=None*, *client_cert=None*, *server_deadline=120*)
Bases: pyloggr.syslog.base.GenericClient

Utility class to send messages or whole files to a RELP server, using an asynchrone TCP client

Parameters

- **server** (*str*) – RELP server hostname or IP
- **port** (*int*) – RELP server port
- **use_ssl** (*bool*) – Should the client connect with SSL

send_events(*events*, *frmt="RFC5424"*)
Send multiple events to the RELP server

Parameters

- **events** (*iterable of Event*) – events to send (iterable of `Event`)
- **frmt** (*str*) – event dumping format
- **compress** (*bool*) – if True, send the events as one LZ4-compressed line

start()
Connect to the RELP server and send 'open' command

Raises socket.error if TCP connection fails

Note: Tornado coroutine

---

stop()
Disconnect from the RELP server

tcp_syslog_client

TCP syslog client

class SyslogClient(*server*, *port*, *use_ssl=False*, *verify_cert=True*, *hostname=None*, *ca_certs=None*, *client_key=None*, *client_cert=None*, *server_deadline=None*)
Bases: `pyloggr.syslog.base.GenericClient`

Utility class to send messages or whole files to a syslog server, using TCP

Parameters

- **server** (*str*) – RELP server hostname or IP
- **port** (*int*) – RELP server port
- **use_ssl** (*bool*) – Should the client connect with SSL

send_events(*events*, *frmt="RFC5424"*)
Send multiple events to the syslog server

Parameters

- **events** – events to send (iterable of `Event`)
- **frmt** – event dumping format

start()
Connect to the syslog server

stop()
Disconnect from the syslog server

syslog.server

This module provides stuff to implement a UDP/TCP/unix socket syslog/RELP server with Tornado

class **BaseSyslogClientConnection**(*stream*, *address*, *syslog_parameters*)

Bases: `object`

Encapsulates a connection with a syslog client

_process_relp_command(*relp_event_id*, *command*, *data*)

RELP client has sent a command. Find the type and make the right answer.

Parameters

- **relp_event_id** – RELP ID, sent by client
- **command** – the RELP command
- **data** – data transmitted after command (can be empty)

_read_next_tokens(**args*, ***kwargs*)

_read_next_token() Reads the stream until we get a space delimiter

Note: Tornado coroutine

disconnect()

Disconnects the client

dispatch_relp_client()

Implements RELP protocol

Note: Tornado coroutine

From http://www.rsyslog.com/doc/relp.html:

```
Request:
RELP-FRAME = RELPID SP COMMAND SP DATALEN [SP DATA] TRAILER

DATA = [SP 1*OCTET] ; command-defined data, if DATALEN is 0, no data is present
COMMAND = 1*32ALPHA
TRAILER = LF

Response:
RSP-HEADER = TXNR SP RSP-CODE [SP HUMANMSG] LF [CMDDATA]

RSP-CODE = 200 / 500 ; 200 is ok, all the rest currently erros
HUAMANMSG = *OCTET ; a human-readble message without LF in it
CMDDATA = *OCTET ; semantics depend on original command
```

dispatch_tcp_client()

Implements Syslog/TCP protocol

Note: Tornado coroutine

From RFC 6587:

```
It can be assumed that octet-counting framing is used if a syslog
frame starts with a digit.

TCP-DATA = *SYSLOG-FRAME
```

```
        SYSLOG-FRAME = MSG-LEN SP SYSLOG-MSG
        MSG-LEN = NONZERO-DIGIT *DIGIT
        NONZERO-DIGIT = %d49-57
        SYSLOG-MSG is defined in the syslog protocol [RFC5424] and may also be considered to be the
        MSG-LEN is the octet count of the SYSLOG-MSG in the SYSLOG-FRAME.

        A transport receiver can assume that non-transparent-framing is used
        if a syslog frame starts with the ASCII character "<" (%d60).

        TCP-DATA = *SYSLOG-FRAME
        SYSLOG-FRAME = SYSLOG-MSG TRAILER
        TRAILER = LF / APP-DEFINED
        APP-DEFINED = 1*2OCTET
        SYSLOG-MSG is defined in the syslog protocol [RFC5424] and may also be considered to be the
```

on_connect()
Called when a client connects to SyslogServer.

We find the protocol by looking at the connecting port. Then run the appropriate dispatch method.

Note: Tornado coroutine

---

on_stream_closed()
Called when a client has been disconnected

class BaseSyslogServer(*syslog_parameters*)
Bases: tornado.tcpserver.TCPServer, pyloggr.syslog.udpserver.UDPServer

Basic Syslog/RELP server

_handle_connection(*connection*, *address*)
Inherits _handle_connection from parent TCPServer to manage SSL connections. Called by Tornado when a client connects.

_start_syslog()
Start to listen for syslog clients

_stop_syslog()
Stop listening for syslog connections

Note: Tornado coroutine

---

handle_data(*data*, *sockname*, *peername*)
Inherit to handle UDP data

handle_stream(*stream*, *address*)
Called by tornado when we have a new client.

Parameters

- **stream** (*IOStream*) – IOStream for the new connection
- **address** (*tuple*) – tuple (client IP, client source port)

Note: Tornado coroutine

---

launch(**args*, ***kwargs*)
Starts the server

•First we try to connect to RabbitMQ

•If successfull, we start to listen for syslog clients

Note: Tornado coroutine

shutdown(**args*, ***kwargs*)
Authoritarian shutdown

stop_all()
Stops completely the server.

Note: Tornado coroutine

class SyslogParameters(*conf*)
Bases: object

Encapsulates the syslog configuration

bind_all_sockets()
Bind the sockets to the current server

Returns list of bound sockets

Return type list

delete_unix_sockets()
Try to delete unix sockets files. Ignore any error and log them as warnings.

wrap_ssl_sock(*sock*, *ssl_options*)
Wrap a socket into a SSL socket

Parameters

- **sock** – socket to wrap
- **ssl_options** (pyloggr.config.SSLConfig) – SSL options

syslog.udpserver

UDP server for tornado

3.7.6 pyloggr.scripts package

The *script* subpackage contains launchers for the pyloggr's processes.

class PyloggrProcess(*fork=True*, *shared_cache=True*)
Bases: object

Boilerplate for starting the different pyloggr processes

_launch(**args*, ***kwargs*)
launch() Abstract method

Note: Tornado coroutine

main()
main method

•Initialize Redis cache

- set up signal handlers
- fork if necessary
- run the *launch* method

shutdown()
Cleanly shutdown the process

Note: Tornado coroutine

scripts.processes

Describe the pyloggr's processes.

class CollectorProcess
Bases: *pyloggr.scripts.PyloggrProcess*

Collect events from the "rescue queue" and inject them back in pyloggr

class FSShipperProcess
Bases: *pyloggr.scripts.PyloggrProcess*

Dumps events to the filesystem

class FilterMachineProcess
Bases: *pyloggr.scripts.PyloggrProcess*

Apply filters to each event found in RabbitMQ, post back into RabbitMQ

Parameters **name** (*str*) – process name

class FrontendProcess
Bases: *pyloggr.scripts.PyloggrProcess*

Web frontend to Pyloggr

class HarvestProcess
Bases: *pyloggr.scripts.PyloggrProcess*

Monitor directories and inject files as logs in Pyloggr

class PgSQLShipperProcess
Bases: *pyloggr.scripts.PyloggrProcess*

Ships events to PostgreSQL

class SyslogAgentProcess
Bases: *pyloggr.scripts.PyloggrProcess*

Implements a syslog agent for end clients

class SyslogProcess
Bases: *pyloggr.scripts.PyloggrProcess*

Implements the syslog server

class SyslogShipperProcess
Bases: *pyloggr.scripts.PyloggrProcess*

Ships events to a remote syslog server

3.8 Indices and tables

- genindex
- modindex
- search

p

Symbols

A

B

C

O

P

Q

R

S

T

U

V

W