rejected Documentation

Release 3.12.3

Gavin M. Roy

Nov 06, 2018

Contents

Rejected is a AMQP consumer daemon and message processing framework. It allows for rapid development of message processing consumers by handling all of the core functionality of communicating with RabbitMQ and management of consumer processes.

Rejected runs as a master process with multiple consumer configurations that are each run it an isolated process. It has the ability to collect statistical data from the consumer processes and report on it.

CHAPTER 1

Features

- Dynamic QoS
- Automatic exception handling including connection management and consumer restarting
- Smart consumer classes that can automatically decode and deserialize message bodies based upon message headers
- Metrics logging and submission to statsd
- Built-in profiling of consumer code

CHAPTER 2

Installation

rejected is available from the Python Package Index and can be installed by running **`easy_install rejected`** or **`pip install rejected`**

CHAPTER 3

Getting Started

3.1 Consumer Examples

The following example illustrates a very simple consumer that simply logs each message body as it's received.

```
from rejected import consumer
import logging

__version__ = '1.0.0'

LOGGER = logging.getLogger(__name__)


class ExampleConsumer(consumer.Consumer):

    def process(self):
        LOGGER.info(self.body)
```

All interaction with RabbitMQ with regard to connection management and message handling, including acknowledgements and rejections are automatically handled for you.

The `__version__` variable provides context in the rejected log files when consumers are started and can be useful for investigating consumer behaviors in production.

In this next example, a contrived `ExampleConsumer._connect_to_database` method is added that will return `False`. When `ExampleConsumer.process` evaluates if it could connect to the database and finds it can not, it will raise a *`rejected.consumer.ConsumerException`* which will requeue the message in RabbitMQ and increment an error counter. When too many errors occur, rejected will automatically restart the consumer after a brief quiet period. For more information on these exceptions, check out the *consumer API documentation*.

```
from rejected import consumer
import logging

__version__ = '1.0.0'
```

(continues on next page)

(continued from previous page)

```
LOGGER = logging.getLogger(__name__)


class ExampleConsumer(consumer.Consumer):

    def _connect_to_database(self):
        return False

    def process(self):
        if not self._connect_to_database:
            raise consumer.ConsumerException('Database error')

        LOGGER.info(self.body)
```

Some consumers are also publishers. In this next example, the message body will be republished to a new exchange on the same RabbitMQ connection:

```
from rejected import consumer
import logging

__version__ = '1.0.0'

LOGGER = logging.getLogger(__name__)


class ExampleConsumer(consumer.PublishingConsumer):

    def process(self):
        LOGGER.info(self.body)
        self.publish('new-exchange', 'routing-key', {}, self.body)
```

Note that the previous example extends *rejected.consumer.PublishingConsumer* instead of *rejected.consumer.Consumer*. For more information about what base consumer classes exist, be sure to check out the *consumer API documentation*.

3.2 Configuration File Syntax

The rejected configuration uses YAML as the markup language. YAML's format, like Python code is whitespace dependent for control structure in blocks. If you're having problems with your rejected configuration, the first thing you should do is ensure that the YAML syntax is correct. yamllint.com is a good resource for validating that your configuration file can be parsed.

The configuration file is split into three main sections: Application, Daemon, and Logging.

The example configuration file provides a good starting point for creating your own configuration file.

3.2.1 Application

The application section of the configuration is broken down into multiple top-level options:

poll_interval	How often rejected should poll consumer processes for stats data in seconds (int/float)
log_stats	Should stats data be logged via Python's standard logging mechanism (bool)
statsd	Enable and configure statsd metric submission (obj)
Connections	A subsection with RabbitMQ connection information for consumers (obj)
Consumers	Where each consumer type is configured (obj)

statsd

enabled	Enable the per consumer statsd metric reporting (bool)
host	The hostname or ip address of the statsd server (str)
port	The port of the statsd server (int)

Connections

Each RabbitMQ connection entry should be a nested object with a unique name with connection attributes.

Connection Name		
	host	The hostname or ip address of the RabbitMQ server (str)
	port	The port of the RabbitMQ server (int)
	vhost	The virtual host to connect to (str)
	user	The username to connect as (str)
	pass	The password to use (str)
	heartbeat_interval	Optional: the AMQP heartbeat interval (int) default: 300 sec

Consumers

Each consumer entry should be a nested object with a unique name with consumer attributes.

Consumer Name		
	consumer	The package.module.Class path to the consumer code (str)
	connections	The connections, by name, to connect to from the Connections section (list)
	qty	The number of consumers per connection to run (int)
	queue	The RabbitMQ queue name to consume from (int)
	ack	Explicitly acknowledge messages (no_ack = not ack) (bool)
	dynamic_qos	Automatically figure out channel prefetch-count QoS settings by message velocity
	max_errors	Number of errors encountered before restarting a consumer
	config | Free-form key-value configuration section for the consumer (obj)	

3.2.2 Daemon

This section contains the settings required to run the application as a daemon. They are as follows:

user	The username to run as when the process is daemonized (bool)
group	Optional The group name to switch to when the process is daemonized (str)
pidfile	The pidfile to write when the process is daemonized (str)

3.2.3 Logging

rejected uses `logging.config.dictConfig` to create a flexible method for configuring the python standard logging module. If rejected is being run in Python 2.6, logutils.dictconfig.dictConfig is used instead.

The following basic example illustrates all of the required sections in the dictConfig format, implemented in YAML:

```
version: 1
formatters: []
verbose:
  format: '%(levelname) -10s %(asctime)s %(process)-6d %(processName) -15s %(name) -
↪10s %(funcName) -20s: %(message)s'
  datefmt: '%Y-%m-%d %H:%M:%S'
handlers:
  console:
    class: logging.StreamHandler
    formatter: verbose
    debug_only: True
loggers:
  rejected:
    handlers: [console]
    level: INFO
    propagate: true
  myconsumer:
    handlers: [console]
    level: DEBUG
    propagate: true
disable_existing_loggers: true
incremental: false
```

Note: The debug_only node of the Logging > handlers > console section is not part of the standard dictConfig format. Please see the *Logging Caveats* section below for more information.

Logging Caveats

In order to allow for customizable console output when running in the foreground and no console output when daemonized, a `debug_only` node has been added to the standard dictConfig format in the handler section. This method is evaluated when logging is configured and if present, it is removed prior to passing the dictionary to dictConfig if present.

If the value is set to true and the application is not running in the foreground, the configuration for the handler and references to it will be removed from the configuration dictionary.

Troubleshooting

If you find that your application is not logging anything or sending output to the terminal, ensure that you have created a logger section in your configuration for your consumer package. For example if your Consumer instance is named `myconsumer.MyConsumer` make sure there is a `myconsumer` logger in the logging configuration.

3.3 Command-Line Options

The rejected command line application allows you to spawn the rejected process as a daemon. Additionally it has options for running interactively (`-f`), which along with the `-o` switch for specifying a single consumer to run and `-q` to specify quantity, makes for easier debugging.

If you specify `-P /path/to/write/data/to`, rejected will automatically enable `cProfile`, writing the profiling data to the path specified. This can be used in conjunction with graphviz to diagram code execution and hotspots.

3.3.1 Help

```
usage: rejected [-h] [-c CONFIG] [-f] [-P PROFILE] [-o CONSUMER]
                [-p PREPEND_PATH] [-q QUANTITY]

RabbitMQ consumer framework

optional arguments:
  -h, --help            show this help message and exit
  -c CONFIG, --config CONFIG
                        Path to the configuration file
  -f, --foreground      Run the application interactively
  -P PROFILE, --profile PROFILE
                        Profile the consumer modules, specifying the output
                        directory.
  -o CONSUMER, --only CONSUMER
                        Only run the consumer specified
  -p PREPEND_PATH, --prepend-path PREPEND_PATH
                        Prepend the python path with the value.
  -q QUANTITY, --qty QUANTITY
                        Run the specified quanty of consumer processes when
                        used in conjunction with -o
```

CHAPTER 4

API Documentation

4.1 Consumer API

The *Consumer*, *PublishingConsumer*, *SmartConsumer*, and *SmartPublishingConsumer* provide base classes to extend for consumer applications.

While the *Consumer* class provides all the structure required for implementing a rejected consumer, the *SmartConsumer* adds functionality designed to make writing consumers even easier. When messages are received by consumers extending *SmartConsumer*, if the message's `content_type` property contains one of the supported mime-types, the message body will automatically be deserialized, making the deserialized message body available via the `body` attribute. Additionally, should one of the supported `content_encoding` types (`gzip` or `bzip2`) be specified in the message's property, it will automatically be decoded.

4.1.1 Message Type Validation

In any of the consumer base classes, if the `MESSAGE_TYPE` attribute is set, the `type` property of incoming messages will be validated against when a message is received, checking for string equality against the `MESSAGE_TYPE` attribute. If they are not matched, the consumer will not process the message and will drop the message without an exception if the `DROP_INVALID_MESSAGES` attribute is set to `True`. If it is `False`, a *ConsumerException* is raised.

4.1.2 Consumer Classes

Consumer

class rejected.consumer.**Consumer**(*settings*, *process*)

Base consumer class that defines the contract between rejected and consumer applications.

In any of the consumer base classes, if the `MESSAGE_TYPE` attribute is set, the `type` property of incoming messages will be validated against when a message is received, checking for string equality against the `MESSAGE_TYPE` attribute. If they are not matched, the consumer will not process the message and will drop

the message without an exception if the DROP_INVALID_MESSAGES attribute is set to True. If it is False, a *MessageException* is raised.

If a consumer raises a *ProcessingException*, the message that was being processed will be republished to the exchange specified by the ERROR_EXCHANGE attribute of the consumer's class using the routing key that was last used for the message. The original message body and properties will be used and an additional header X-Processing-Exceptions will be added that will contain the number of times the message has had a ProcessingException raised for it. In combination with a queue that has x-message-ttl set and x-dead-letter-exchange that points to the original exchange for the queue the consumer is consuming off of, you can implement a delayed retry cycle for messages that are failing to process due to external resource or service issues.

If ERROR_MAX_RETRY is set on the class, the headers for each method will be inspected and if the value of X-Processing-Exceptions is greater than or equal to the ERROR_MAX_RETRY value, the message will be dropped.

DROP_INVALID_MESSAGES = False
Drop a message if its type property doesn't match MESSAGE_TYPE

ERROR_EXCHANGE = 'errors'
The exchange to publish ProcessingErrors to

ERROR_MAX_RETRY = None
The number of ''ProcessingErrors' before a message is dropped

MESSAGE_TYPE = None
Used to validate the message type of a message before processing

app_id
Access the current message's app-id property as an attribute of the consumer class.

Return type str

body
Access the opaque body from the current message.

Return type str

configuration
Access the configuration stanza for the consumer as specified by the config section for the consumer in the rejected configuration.

Deprecated since version 3.1: Use *settings* instead.

Return type dict

content_encoding
Access the current message's content-encoding property as an attribute of the consumer class.

Return type str

content_type
Access the current message's content-type property as an attribute of the consumer class. :rtype: str

correlation_id
Access the current message's correlation-id property as an attribute of the consumer class.

Return type str

exchange
Access the exchange the message was published to as an attribute of the consumer class.

Return type str

expiration
Access the current message's `expiration` property as an attribute of the consumer class.

Return type str

finish()
Finishes message processing for the current message.

headers
Access the current message's `headers` property as an attribute of the consumer class.

Return type dict

initialize()
Extend this method for any initialization tasks that occur only when the *Consumer* class is created.

log_exception(*msg_format*, **args*, ***kwargs*)
Customize the logging of uncaught exceptions.

Parameters

- **msg_format** (*str*) – format of msg to log with `self.logger.error`
- **args** – positional arguments to pass to `self.logger.error`
- **kwargs** – keyword args to pass into `self.logger.error`
- **send_to_sentry** (*bool*) – if omitted or *truthy*, this keyword will send the captured exception to Sentry (if enabled).

By default, this method will log the message using `logging.Logger.error()` and send the exception to Sentry. If an exception is currently active, then the traceback will be logged at the debug level.

message_id
Access the current message's `message-id` property as an attribute of the consumer class. :rtype: str

message_type
Access the current message's `type` property as an attribute of the consumer class.

Return type str

name
Property returning the name of the consumer class.

Return type str

on_finish()
Called after the end of a request. Override this method to perform cleanup, logging, etc. This method is a counterpart to *prepare*. `on_finish` may not produce any output, as it is called after the response has been sent to the client.

prepare()
Called when a message is received before *process*.

Asynchronous support: Decorate this method with *.gen.coroutine* or *.return_future* to make it asynchronous (the *asynchronous* decorator cannot be used on *prepare*).

If this method returns a *.Future* execution will not proceed until the *.Future* is done.

priority
Access the current message's `priority` property as an attribute of the consumer class.

Return type int

process()
Extend this method for implementing your Consumer logic.

If the message can not be processed and the Consumer should stop after n failures to process messages, raise the ConsumerException.

Raises ConsumerException

Raises NotImplementedError

properties
Access the current message's properties in dict form as an attribute of the consumer class.

Return type dict

redelivered
Indicates if the current message has been redelivered.

Return type bool

reply_to
Access the current message's `reply-to` property as an attribute of the consumer class.

Return type str

require_setting(*name*, *feature='this feature'*)
Raises an exception if the given app setting is not defined.

Parameters

- **name** (*str*) – The parameter name
- **feature** (*str*) – A friendly name for the setting feature

routing_key
Access the routing key for the current message.

Return type str

send_exception_to_sentry(*exc_info*)
Send an exception to Sentry if enabled.

Parameters **exc_info** (*tuple*) – exception information as returned from `sys.exc_info()`

sentry_client
Access the Sentry raven `Client` instance or `None`

Use this object to add tags or additional context to Sentry error reports (see `raven.base.Client.tags_context()`) or to report messages (via `raven.base.Client.captureMessage()`) directly to Sentry.

Return type `raven.base.Client`

set_sentry_context(*tag*, *value*)
Set a context tag in Sentry for the given key and value.

Parameters

- **tag** (*str*) – The context tag name
- **value** (*str*) – The context value

settings
Access the consumer settings as specified by the `config` section for the consumer in the rejected configuration.

Return type dict

shutdown()
Override to cleanly shutdown when rejected is stopping

statsd_add_timing(*key*, *duration*)
Add a timing to statsd

Parameters

- **key** (*str*) – The key to add the timing to
- **duration** (*int* | *float*) – The timing value

statsd_incr(*key*, *value=1*)
Increment the specified key in statsd if statsd is enabled.

Parameters

- **key** (*str*) – The key to increment
- **value** (*int*) – The value to increment the key by

statsd_track_duration(*key*)
Time around a context and emit a statsd metric.

Parameters **key** (*str*) – The key for the timing to track

timestamp
Access the unix epoch timestamp value from the properties of the current message.

Return type int

unset_sentry_context(*tag*)
Remove a context tag from sentry

Parameters **tag** (*str*) – The context tag to remove

user_id
Access the user-id from the current message's properties.

Return type str

yield_to_ioloop()
Function that will allow Rejected to process IOLoop events while in a tight-loop inside an asynchronous consumer.

PublishingConsumer

class rejected.consumer.**PublishingConsumer**(*settings*, *process*)
The PublishingConsumer extends the Consumer class, adding two methods, one that allows for *publishing* of messages back on the same channel that the consumer is communicating on and another for *replying to messages*, adding RPC reply semantics to the outbound message.

In any of the consumer base classes, if the `MESSAGE_TYPE` attribute is set, the `type` property of incoming messages will be validated against when a message is received, checking for string equality against the `MESSAGE_TYPE` attribute. If they are not matched, the consumer will not process the message and will drop the message without an exception if the `DROP_INVALID_MESSAGES` attribute is set to `True`. If it is `False`, a *ConsumerException* is raised.

app_id
Access the current message's `app-id` property as an attribute of the consumer class.

Return type str

body
Access the opaque body from the current message.

Return type str

configuration
Access the configuration stanza for the consumer as specified by the `config` section for the consumer in the rejected configuration.

Deprecated since version 3.1: Use *settings* instead.

Return type dict

content_encoding
Access the current message's `content-encoding` property as an attribute of the consumer class.

Return type str

content_type
Access the current message's `content-type` property as an attribute of the consumer class. :rtype: str

correlation_id
Access the current message's `correlation-id` property as an attribute of the consumer class.

Return type str

exchange
Access the exchange the message was published to as an attribute of the consumer class.

Return type str

expiration
Access the current message's `expiration` property as an attribute of the consumer class.

Return type str

finish()
Finishes message processing for the current message.

headers
Access the current message's `headers` property as an attribute of the consumer class.

Return type dict

initialize()
Extend this method for any initialization tasks that occur only when the *Consumer* class is created.

log_exception(*msg_format*, **args*, ***kwargs*)
Customize the logging of uncaught exceptions.

Parameters

- **msg_format** (*str*) – format of msg to log with `self.logger.error`
- **args** – positional arguments to pass to `self.logger.error`
- **kwargs** – keyword args to pass into `self.logger.error`
- **send_to_sentry** (*bool*) – if omitted or *truthy*, this keyword will send the captured exception to Sentry (if enabled).

By default, this method will log the message using `logging.Logger.error()` and send the exception to Sentry. If an exception is currently active, then the traceback will be logged at the debug level.

message_id
Access the current message's `message-id` property as an attribute of the consumer class. :rtype: str

message_type
Access the current message's `type` property as an attribute of the consumer class.

Return type str

name
Property returning the name of the consumer class.

Return type str

on_finish()
Called after the end of a request. Override this method to perform cleanup, logging, etc. This method is a counterpart to *prepare*. `on_finish` may not produce any output, as it is called after the response has been sent to the client.

prepare()
Called when a message is received before *process*.

Asynchronous support: Decorate this method with *.gen.coroutine* or *.return_future* to make it asynchronous (the *asynchronous* decorator cannot be used on *prepare*).

If this method returns a *.Future* execution will not proceed until the *.Future* is done.

priority
Access the current message's `priority` property as an attribute of the consumer class.

Return type int

process()
Extend this method for implementing your Consumer logic.

If the message can not be processed and the Consumer should stop after n failures to process messages, raise the ConsumerException.

Raises ConsumerException

Raises NotImplementedError

properties
Access the current message's properties in dict form as an attribute of the consumer class.

Return type dict

publish_message(*exchange*, *routing_key*, *properties*, *body*)
Publish a message to RabbitMQ on the same channel the original message was received on.

Parameters

- **exchange** (*str*) – The exchange to publish to
- **routing_key** (*str*) – The routing key to publish with
- **properties** (*dict*) – The message properties
- **body** (*str*) – The message body

redelivered
Indicates if the current message has been redelivered.

Return type bool

reply(*response_body*, *properties*, *auto_id=True*, *exchange=None*, *reply_to=None*)
Reply to the received message.

If auto_id is True, a new uuid4 value will be generated for the message_id and correlation_id will be set to the message_id of the original message. In addition, the timestamp will be assigned the current time

of the message. If auto_id is False, neither the message_id or the correlation_id will be changed in the properties.

If exchange is not set, the exchange the message was received on will be used.

If reply_to is set in the original properties, it will be used as the routing key. If the reply_to is not set in the properties and it is not passed in, a ValueException will be raised. If reply to is set in the properties, it will be cleared out prior to the message being republished.

Parameters

- **response_body** (*any*) – The message body to send
- **properties** (*rejected.data.Properties*) – Message properties to use
- **auto_id** (*bool*) – Automatically shuffle message_id & correlation_id
- **exchange** (*str*) – Override the exchange to publish to
- **reply_to** (*str*) – Override the reply_to in the properties

Raises ValueError

reply_to
Access the current message's `reply-to` property as an attribute of the consumer class.

Return type str

require_setting(*name*, *feature='this feature'*)
Raises an exception if the given app setting is not defined.

Parameters

- **name** (*str*) – The parameter name
- **feature** (*str*) – A friendly name for the setting feature

routing_key
Access the routing key for the current message.

Return type str

send_exception_to_sentry(*exc_info*)
Send an exception to Sentry if enabled.

Parameters **exc_info** (*tuple*) – exception information as returned from `sys.exc_info()`

sentry_client
Access the Sentry raven `Client` instance or `None`

Use this object to add tags or additional context to Sentry error reports (see `raven.base.Client.tags_context()`) or to report messages (via `raven.base.Client.captureMessage()`) directly to Sentry.

Return type `raven.base.Client`

set_sentry_context(*tag*, *value*)
Set a context tag in Sentry for the given key and value.

Parameters

- **tag** (*str*) – The context tag name
- **value** (*str*) – The context value

settings
Access the consumer settings as specified by the `config` section for the consumer in the rejected configuration.

Return type dict

shutdown()
Override to cleanly shutdown when rejected is stopping

statsd_add_timing(*key*, *duration*)
Add a timing to statsd

Parameters

- **key** (*str*) – The key to add the timing to
- **duration** (*int* | *float*) – The timing value

statsd_incr(*key*, *value=1*)
Increment the specified key in statsd if statsd is enabled.

Parameters

- **key** (*str*) – The key to increment
- **value** (*int*) – The value to increment the key by

statsd_track_duration(*key*)
Time around a context and emit a statsd metric.

Parameters **key** (*str*) – The key for the timing to track

timestamp
Access the unix epoch timestamp value from the properties of the current message.

Return type int

unset_sentry_context(*tag*)
Remove a context tag from sentry

Parameters **tag** (*str*) – The context tag to remove

user_id
Access the user-id from the current message's properties.

Return type str

yield_to_ioloop()
Function that will allow Rejected to process IOLoop events while in a tight-loop inside an asynchronous consumer.

SmartConsumer

class rejected.consumer.**SmartConsumer**(*settings*, *process*)
Base class to ease the implementation of strongly typed message consumers that validate and automatically decode and deserialize the inbound message body based upon the message properties. Additionally, should one of the supported `content_encoding` types (`gzip` or `bzip2`) be specified in the message's property, it will automatically be decoded.

Supported MIME types for automatic deserialization are:

- application/json
- application/pickle

- application/x-pickle
- application/x-plist
- application/x-vnd.python.pickle
- application/vnd.python.pickle
- text/csv
- text/html (with beautifulsoup4 installed)
- text/xml (with beautifulsoup4 installed)
- text/yaml
- text/x-yaml

In any of the consumer base classes, if the `MESSAGE_TYPE` attribute is set, the `type` property of incoming messages will be validated against when a message is received, checking for string equality against the `MESSAGE_TYPE` attribute. If they are not matched, the consumer will not process the message and will drop the message without an exception if the `DROP_INVALID_MESSAGES` attribute is set to `True`. If it is `False`, a *`ConsumerException`* is raised.

`app_id`
Access the current message's `app-id` property as an attribute of the consumer class.

Return type str

`body`
Return the message body, unencoded if needed, deserialized if possible.

Return type any

`configuration`
Access the configuration stanza for the consumer as specified by the `config` section for the consumer in the rejected configuration.

Deprecated since version 3.1: Use *`settings`* instead.

Return type dict

`content_encoding`
Access the current message's `content-encoding` property as an attribute of the consumer class.

Return type str

`content_type`
Access the current message's `content-type` property as an attribute of the consumer class. :rtype: str

`correlation_id`
Access the current message's `correlation-id` property as an attribute of the consumer class.

Return type str

`exchange`
Access the exchange the message was published to as an attribute of the consumer class.

Return type str

`expiration`
Access the current message's `expiration` property as an attribute of the consumer class.

Return type str

`finish``()`
Finishes message processing for the current message.

headers
Access the current message's `headers` property as an attribute of the consumer class.

Return type dict

initialize()
Extend this method for any initialization tasks that occur only when the *Consumer* class is created.

log_exception(*msg_format*, **args*, ***kwargs*)
Customize the logging of uncaught exceptions.

Parameters

- **msg_format** (*str*) – format of msg to log with `self.logger.error`
- **args** – positional arguments to pass to `self.logger.error`
- **kwargs** – keyword args to pass into `self.logger.error`
- **send_to_sentry** (*bool*) – if omitted or *truthy*, this keyword will send the captured exception to Sentry (if enabled).

By default, this method will log the message using `logging.Logger.error()` and send the exception to Sentry. If an exception is currently active, then the traceback will be logged at the debug level.

message_id
Access the current message's `message-id` property as an attribute of the consumer class. :rtype: str

message_type
Access the current message's `type` property as an attribute of the consumer class.

Return type str

name
Property returning the name of the consumer class.

Return type str

on_finish()
Called after the end of a request. Override this method to perform cleanup, logging, etc. This method is a counterpart to *prepare*. `on_finish` may not produce any output, as it is called after the response has been sent to the client.

prepare()
Called when a message is received before *process*.

Asynchronous support: Decorate this method with *.gen.coroutine* or *.return_future* to make it asynchronous (the *asynchronous* decorator cannot be used on *prepare*).

If this method returns a *.Future* execution will not proceed until the *.Future* is done.

priority
Access the current message's `priority` property as an attribute of the consumer class.

Return type int

process()
Extend this method for implementing your Consumer logic.

If the message can not be processed and the Consumer should stop after n failures to process messages, raise the ConsumerException.

Raises ConsumerException

Raises NotImplementedError

properties
Access the current message's properties in dict form as an attribute of the consumer class.

Return type dict

redelivered
Indicates if the current message has been redelivered.

Return type bool

reply_to
Access the current message's `reply-to` property as an attribute of the consumer class.

Return type str

require_setting(*name*, *feature='this feature'*)
Raises an exception if the given app setting is not defined.

Parameters

- **name** (*str*) – The parameter name
- **feature** (*str*) – A friendly name for the setting feature

routing_key
Access the routing key for the current message.

Return type str

send_exception_to_sentry(*exc_info*)
Send an exception to Sentry if enabled.

Parameters **exc_info** (*tuple*) – exception information as returned from `sys.exc_info()`

sentry_client
Access the Sentry raven `Client` instance or `None`

Use this object to add tags or additional context to Sentry error reports (see `raven.base.Client.tags_context()`) or to report messages (via `raven.base.Client.captureMessage()`) directly to Sentry.

Return type `raven.base.Client`

set_sentry_context(*tag*, *value*)
Set a context tag in Sentry for the given key and value.

Parameters

- **tag** (*str*) – The context tag name
- **value** (*str*) – The context value

settings
Access the consumer settings as specified by the `config` section for the consumer in the rejected configuration.

Return type dict

shutdown()
Override to cleanly shutdown when rejected is stopping

statsd_add_timing(*key*, *duration*)
Add a timing to statsd

Parameters

- **key** (*str*) – The key to add the timing to
- **duration** (*int* / *float*) – The timing value

statsd_incr(*key*, *value=1*)
Increment the specified key in statsd if statsd is enabled.

Parameters

- **key** (*str*) – The key to increment
- **value** (*int*) – The value to increment the key by

statsd_track_duration(*key*)
Time around a context and emit a statsd metric.

Parameters **key** (*str*) – The key for the timing to track

timestamp
Access the unix epoch timestamp value from the properties of the current message.

Return type int

unset_sentry_context(*tag*)
Remove a context tag from sentry

Parameters **tag** (*str*) – The context tag to remove

user_id
Access the user-id from the current message's properties.

Return type str

yield_to_ioloop()
Function that will allow Rejected to process IOLoop events while in a tight-loop inside an asynchronous consumer.

SmartPublishingConsumer

class rejected.consumer.**SmartPublishingConsumer**(*settings*, *process*)
PublishingConsumer with serialization built in

app_id
Access the current message's `app-id` property as an attribute of the consumer class.

Return type str

body
Return the message body, unencoded if needed, deserialized if possible.

Return type any

configuration
Access the configuration stanza for the consumer as specified by the `config` section for the consumer in the rejected configuration.

Deprecated since version 3.1: Use *settings* instead.

Return type dict

content_encoding
Access the current message's `content-encoding` property as an attribute of the consumer class.

Return type str

content_type
Access the current message's `content-type` property as an attribute of the consumer class. :rtype: str

correlation_id
Access the current message's `correlation-id` property as an attribute of the consumer class.

Return type str

exchange
Access the exchange the message was published to as an attribute of the consumer class.

Return type str

expiration
Access the current message's `expiration` property as an attribute of the consumer class.

Return type str

finish()
Finishes message processing for the current message.

headers
Access the current message's `headers` property as an attribute of the consumer class.

Return type dict

initialize()
Extend this method for any initialization tasks that occur only when the *Consumer* class is created.

log_exception(*msg_format*, **args*, ***kwargs*)
Customize the logging of uncaught exceptions.

Parameters

- **msg_format** (*str*) – format of msg to log with `self.logger.error`
- **args** – positional arguments to pass to `self.logger.error`
- **kwargs** – keyword args to pass into `self.logger.error`
- **send_to_sentry** (*bool*) – if omitted or *truthy*, this keyword will send the captured exception to Sentry (if enabled).

By default, this method will log the message using `logging.Logger.error()` and send the exception to Sentry. If an exception is currently active, then the traceback will be logged at the debug level.

message_id
Access the current message's `message-id` property as an attribute of the consumer class. :rtype: str

message_type
Access the current message's `type` property as an attribute of the consumer class.

Return type str

name
Property returning the name of the consumer class.

Return type str

on_finish()
Called after the end of a request. Override this method to perform cleanup, logging, etc. This method is a counterpart to *prepare*. `on_finish` may not produce any output, as it is called after the response has been sent to the client.

prepare()
Called when a message is received before *process*.

Asynchronous support: Decorate this method with *.gen.coroutine* or *.return_future* to make it asynchronous (the *asynchronous* decorator cannot be used on *prepare*).

If this method returns a *.Future* execution will not proceed until the *.Future* is done.

priority
Access the current message's `priority` property as an attribute of the consumer class.

Return type int

process()
Extend this method for implementing your Consumer logic.

If the message can not be processed and the Consumer should stop after n failures to process messages, raise the ConsumerException.

Raises ConsumerException

Raises NotImplementedError

properties
Access the current message's properties in dict form as an attribute of the consumer class.

Return type dict

publish_message(*exchange*, *routing_key*, *properties*, *body*, *no_serialization=False*, *no_encoding=False*)
Publish a message to RabbitMQ on the same channel the original message was received on.

By default, if you pass a non-string object to the body and the properties have a supported content-type set, the body will be auto-serialized in the specified content-type.

If the properties do not have a timestamp set, it will be set to the current time.

If you specify a content-encoding in the properties and the encoding is supported, the body will be auto-encoded.

Both of these behaviors can be disabled by setting no_serialization or no_encoding to True.

Parameters

- **exchange** (*str*) – The exchange to publish to
- **routing_key** (*str*) – The routing key to publish with
- **properties** (*dict*) – The message properties
- **body** (*mixed*) – The message body to publish
- **no_serialization** – Turn off auto-serialization of the body
- **no_encoding** – Turn off auto-encoding of the body

redelivered
Indicates if the current message has been redelivered.

Return type bool

reply(*response_body*, *properties*, *auto_id=True*, *exchange=None*, *reply_to=None*)
Reply to the received message.

If auto_id is True, a new uuid4 value will be generated for the message_id and correlation_id will be set to the message_id of the original message. In addition, the timestamp will be assigned the current time

of the message. If auto_id is False, neither the message_id or the correlation_id will be changed in the properties.

If exchange is not set, the exchange the message was received on will be used.

If reply_to is set in the original properties, it will be used as the routing key. If the reply_to is not set in the properties and it is not passed in, a ValueException will be raised. If reply to is set in the properties, it will be cleared out prior to the message being republished.

Parameters

- **response_body** (*any*) – The message body to send
- **properties** (*rejected.data.Properties*) – Message properties to use
- **auto_id** (*bool*) – Automatically shuffle message_id & correlation_id
- **exchange** (*str*) – Override the exchange to publish to
- **reply_to** (*str*) – Override the reply_to in the properties

Raises ValueError

reply_to
Access the current message's `reply-to` property as an attribute of the consumer class.

Return type str

require_setting(*name*, *feature='this feature'*)
Raises an exception if the given app setting is not defined.

Parameters

- **name** (*str*) – The parameter name
- **feature** (*str*) – A friendly name for the setting feature

routing_key
Access the routing key for the current message.

Return type str

send_exception_to_sentry(*exc_info*)
Send an exception to Sentry if enabled.

Parameters **exc_info** (*tuple*) – exception information as returned from `sys.exc_info()`

sentry_client
Access the Sentry raven `Client` instance or `None`

Use this object to add tags or additional context to Sentry error reports (see `raven.base.Client.tags_context()`) or to report messages (via `raven.base.Client.captureMessage()`) directly to Sentry.

Return type `raven.base.Client`

set_sentry_context(*tag*, *value*)
Set a context tag in Sentry for the given key and value.

Parameters

- **tag** (*str*) – The context tag name
- **value** (*str*) – The context value

settings
Access the consumer settings as specified by the `config` section for the consumer in the rejected configuration.

Return type dict

shutdown()
Override to cleanly shutdown when rejected is stopping

statsd_add_timing(*key*, *duration*)
Add a timing to statsd

Parameters

- **key** (*str*) – The key to add the timing to
- **duration** (*int* | *float*) – The timing value

statsd_incr(*key*, *value=1*)
Increment the specified key in statsd if statsd is enabled.

Parameters

- **key** (*str*) – The key to increment
- **value** (*int*) – The value to increment the key by

statsd_track_duration(*key*)
Time around a context and emit a statsd metric.

Parameters **key** (*str*) – The key for the timing to track

timestamp
Access the unix epoch timestamp value from the properties of the current message.

Return type int

unset_sentry_context(*tag*)
Remove a context tag from sentry

Parameters **tag** (*str*) – The context tag to remove

user_id
Access the user-id from the current message's properties.

Return type str

yield_to_ioloop()
Function that will allow Rejected to process IOLoop events while in a tight-loop inside an asynchronous consumer.

4.1.3 Exceptions

There are two exception types that consumer applications should raise to handle problems that may arise when processing a message. When these exceptions are raised, rejected will reject the message delivery, letting RabbitMQ know that there was a failure.

The *ConsumerException* should be raised when there is a problem in the consumer itself, such as inability to contact a database server or other resources. When a *ConsumerException* is raised, the message will be rejected *and* requeued, adding it back to the RabbitMQ it was delivered back to. Additionally, rejected keeps track of consumer exceptions and will shutdown the consumer process and start a new one once a consumer has exceeded its configured maximum error count within a `60` second window. The default maximum error count is `5`.

The *MessageException* should be raised when there is a problem with the message. When this exception is raised, the message will be rejected on the RabbitMQ server *without* requeue, discarding the message. This should be done when there is a problem with the message itself, such as a malformed payload or non-supported properties like `content-type` or `type`.

If a consumer raises a *ProcessingException*, the message that was being processed will be republished to the exchange specified by the `ERROR_EXCHANGE` attribute of the *consumer's class* using the routing key that was last used for the message. The original message body and properties will be used and an additional header `X-Processing-Exceptions` will be added that will contain the number of times the message has had a `ProcessingException` raised for it. In combination with a queue that has `x-message-ttl` set and `x-dead-letter-exchange` that points to the original exchange for the queue the consumer is consuming off of, you can implement a delayed retry cycle for messages that are failing to process due to external resource or service issues.

If `ERROR_MAX_RETRY` is set on the class, the headers for each method will be inspected and if the value of `X-Processing-Exceptions` is greater than or equal to the `ERROR_MAX_RETRY` value, the message will be dropped.

Note: If unhandled exceptions are raised by a consumer, they will be caught by rejected, logged, and turned into a *ConsumerException*.

---

class rejected.consumer.**ConsumerException**
: May be called when processing a message to indicate a problem that the Consumer may be experiencing that should cause it to stop.

class rejected.consumer.**MessageException**
: Invoke when a message should be rejected and not requeued, but not due to a processing error that should cause the consumer to stop.

class rejected.consumer.**ProcessingException**
: Invoke when a message should be rejected and not requeued, but not due to a processing error that should cause the consumer to stop. This should be used for when you want to reject a message which will be republished to a retry queue, without anything being stated about the exception.

CHAPTER 5

Issues

Please report any issues to the Github repo at https://github.com/gmr/rejected/issues

CHAPTER 6

Source

rejected source is available on Github at https://github.com/gmr/rejected

CHAPTER 7

Version History

See history

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index

S

T

U

Y