RabbitLeap Documentation

Release 0.1.1

Ahmed AlSahaf

Mar 20, 2021

Contents:

1 Get Started **1**
- 1.1 Overview . . . 1
- 1.2 Installation . . . 1
- 1.3 Hello, World! . . . 2

2 Key Topics **7**
- 2.1 Consumer . . . 7
- 2.2 Routing . . . 11
- 2.3 Handling . . . 13
- 2.4 Retry Policies . . . 15

3 Contributing **19**
- 3.1 Types of Contributions . . . 19
- 3.2 Get Started! . . . 20
- 3.3 Pull Request Guidelines . . . 20
- 3.4 Add a New Test . . . 21

4 Credits **23**
- 4.1 Development Lead . . . 23
- 4.2 Contributors . . . 23

5 History **25**
- 5.1 0.1.1 (09/18/2018) . . . 25

6 Reference **27**
- 6.1 rabbitleap . . . 27

7 Indices and tables **41**

Python Module Index **43**

Index **45**

CHAPTER 1

Get Started

1.1 Overview

RabbitLeap is a simple RabbitMQ consuming framework. It's built on top of Pika, a RabbitMQ client library for Python.

1.1.1 Features

- Automatically recovers from connection failures
- Configurable retry policy for handing failures
- Automatically route messages to handlers, based on custom logic and different message properties

1.2 Installation

1.2.1 Stable release

To install RabbitLeap, run this command in your terminal:

```
$ pip install rabbitleap
```

This is the preferred method to install RabbitLeap, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

1.2.2 From sources

The sources for RabbitLeap can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/asahaf/rabbitleap
```

Or download the tarball:

```
$ curl  -OL https://github.com/asahaf/rabbitleap/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

1.3 Hello, World!

In this example, we're going to create a consumer that consumes 4 types of messages:

1. Messages of type "rabbit.eat"
2. Messages of type "rabbit.leap" or "rabbit.jump"
3. Messages of type "dog.eat"

1.3.1 1. Defining Handlers

We start with defining handlers for these 4 types of messages.

```
from rabbitleap.consumer import Consumer
from rabbitleap.handling import MessageHandler


class RabbitEatHandler(MessageHandler):

    def handle(self):
        print('rabbit eat: {}'.format(self.envelope.payload.decode('utf-8')))


# Handling both "leap" and "jump"
class RabbitLeapJumpHandler(MessageHandler):

    def handle(self):
        print('{}: {}'.format(self.envelope.type,
                              self.envelope.payload.decode('utf-8')))


class DogEatHandler(MessageHandler):

    def handle(self):
        print('dog eat: {}'.format(self.envelope.payload.decode('utf-8')))
```

Notice inside `handle` methods we access `self.envelope`; The consumer creates an envelope for each message it receives from RabbitMQ and it's available to the handler. The *Envelope* contains message properties, payload, and delivery information.

1.3.2 2. Creating a Consumer

Now, after we have all handlers defined, it's time to create a consumer and add the handlers to it.

```
consumer_queue_name = 'consumer_queue'
amqp_url = r'amqp://guest:guest@localhost:5672/%2f'

consumer = Consumer(amqp_url=amqp_url, queue_name=consumer_queue_name)
# route message of type `rabbit.eat` to RabbitEatHandler
consumer.add_handler(r'rabbit\.eat', RabbitEatHandler)
# route message of types rabbit.leap or rabbit.jump to RabbitLeapJumpHandler
consumer.add_handler(r'rabbit\.(leap|jump)', RabbitLeapJumpHandler)
consumer.add_handler(r'dog\.eat', DogEatHandler)
```

add_handler() accepts a regular expression pattern which is for message type matching, and a Handler class which handles the message envelope.

1.3.3 3. Starting The Consumer

Everything now is set and ready, lets start the consumer.

```
try:
    consumer.start()
except KeyboardInterrupt:
    consumer.stop()
```

1.3.4 4. Putting Everything Together

```
from rabbitleap.consumer import Consumer
from rabbitleap.handling import MessageHandler


class RabbitEatHandler(MessageHandler):

    def handle(self):
        print('rabbit eat: {}'.format(self.envelope.payload.decode('utf-8')))


# Handling both "leap" and "jump"
class RabbitLeapJumpHandler(MessageHandler):

    def handle(self):
        print('{}: {}'.format(self.envelope.type,
                              self.envelope.payload.decode('utf-8')))


class DogEatHandler(MessageHandler):

    def handle(self):
        print('dog eat: {}'.format(self.envelope.payload.decode('utf-8')))


consumer_queue_name = 'consumer_queue'
amqp_url = r'amqp://guest:guest@localhost:5672/%2f'

consumer = Consumer(amqp_url=amqp_url, queue_name=consumer_queue_name)
# route message of type `rabbit.eat` to RabbitEatHandler
consumer.add_handler(r'rabbit\.eat', RabbitEatHandler)
```

(continues on next page)

(continued from previous page)

```
# route message of types rabbit.leap or rabbit.jump to RabbitLeapJumpHandler
consumer.add_handler(r'rabbit\.(leap|jump)', RabbitLeapJumpHandler)
consumer.add_handler(r'dog\.eat', DogEatHandler)

try:
    consumer.start()
except KeyboardInterrupt:
    consumer.stop()
```

Save the file as `consumer.py`

1.3.5 5. Running The Consumer Script

```
$ python consumer.py
```

1.3.6 6. Testing The Consumer

To test the consumer, we need to send some messages to its queue. To do that, we need to create a small python program that connects to RabbitMQ and sends messages to the consumer queue.

```
import pika
from pika.spec import BasicProperties

amqp_url = r'amqp://guest:guest@localhost:5672/%2f'
connection = pika.BlockingConnection(pika.URLParameters(amqp_url))

channel = connection.channel()
queue_name = 'consumer_queue'
channel.queue_declare(queue=queue_name, durable=True)

channel.basic_publish(
    exchange='',
    routing_key=queue_name,
    properties=BasicProperties(type='rabbit.eat'),
    body='carrot')

channel.basic_publish(
    exchange='',
    routing_key=queue_name,
    properties=BasicProperties(type='rabbit.leap'),
    body='over the dog')

channel.basic_publish(
    exchange='',
    routing_key=queue_name,
    properties=BasicProperties(type='rabbit.jump'),
    body='over the dog')

channel.basic_publish(
    exchange='',
    routing_key=queue_name,
    properties=BasicProperties(type='dog.eat'),
    body='meat')
```

Save the file as `publisher.py`

1.3.7 7. Running The Publisher Script

```
$ python publisher.py
```

Congratulations! you've created the first consumer. Next, navigate to *Key Topics* to understand the concepts.

CHAPTER 2

Key Topics

2.1 Consumer

The *Consumer* is the main part of RabbitLeap framework. It's responsible for connecting to RabbitMQ, besides, receiving, acknowledging, and rejecting messages. It handles connection failures and automatically reconnects to RabbitMQ. The consumer consumes a share or exclusive RabbitMQ queue.

When the *Consumer* receives a message from RabbitMQ, it packages the message properties, payload, and delivery information in an *Envelope* object, preparing it for handling. The envelope then, is routed to its handler by the *Router*, which the consumer relies on to route envelopes to their handlers.

The consumer can be configured with a retry policy. The *RetryPolicy* defines how to do handling retries incase a timeout or an error happens.

2.1.1 Consumer

class rabbitleap.consumer.**Consumer**(*amqp_url*, *queue_name*, *durable=True*, *exclusive=False*, *dlx_name=None*, *auto_reconnect=True*, *auto_reconnect_delay=3*)

Bases: *rabbitleap.routing.RuleRouter*

RabbitMQ Consumer.

Public Methods:

start(): start consumer.

stop(): stop consumer.

restart(): restart consumer.

abort(): reject message.

skip(): skip message handling.

error(): raise *HandlingError* exception to report a handling error.

add_exchange_bind(): add exchange bind to the bindings list.

add_rule(): add a routing rule to the routing rules list.

set_default_rule(): set default routing rule to catch all unmatched messages

add_handler(): add handler, it creates a routing rule then add it to the routing rules list.

set_default_handler(): set default handler, it creates a rule then set it as default routing rule

set_retry_policy(): set a retry policy

unset_retry_policy(): un-set retry policy

Consumer.**start**()
Start consumer.

Consumer.**stop**()
Stop consumer.

Consumer.**restart**()
Restart consumer.

Close the connection and reconnect again.

Consumer.**abort**(*reason=None*)
Abort handling the message.

This can be called during *pre_handle()* or handle() to abort handling. It raises *AbortHandling* exception. The exception is handled by the consumer, and causes the consumer to reject the message.

Raise *AbortHandling* when called

NOTE: when called inside the handler, the handler MUST re-raise *AbortHandling* exception to the consumer, in case the exception is handled inside.

Parameters **reason** (*str*) – Reason for aborting handling the message

Consumer.**skip**(*reason=None*)
Skip handling the message.

This can be called during *pre_handle()* or *handle()* to skip handling. It raises *SkipHandling* exception. The exception is handled by the consumer, and causes the consumer to ack and skip the message.

Raise *SkipHandling* when called

NOTE: when called inside the handler, the handler should re-raise *SkipHandling* exception to the consumer, in case the exception is handled inside.

Parameters **reason** (*str*) – Reason for skipping handling the message

Consumer.**error**(*error_msg=None*)
Raise *HandlingError* exception.

This method can be called inside the handler or invoked by the consumer in case of an error happened while handling the message. This method raises *HandlingError* which is handled by the consumer. The consumer will retry handling the message if a retry policy is set or reject the message incase no retry policy is set. When calling this method inside the handler, it should be called during *pre_handle()* or *handle()*.

Raise *HandlingError* when called

NOTE: when called inside the handler, the handler should re-raise *HandlingError* exception to the consumer, in case the exception is handled inside.

Parameters **error_msg** (*str*) – Error message

Consumer.**add_exchange_bind**(*exchange_name*, *routing_key*, *declare_exchange=False*, *declare_kwargs=None*)

Add exchange binding to the bindings list

NOTE: Actual exchange binding is happening during the consumer start up, when the connection with RabbitMQ established. Invoking this method after the connection is already established won't take affect until the consumer re-establishes the connection. The method *restart()* can be called to force the consumer to disconnect and reconnect again, doing exchange binding as part of that process.

Raise AssertionError if *declare_exchange* is True and *declare_kwargs* is None or 'type' is not in *declare_kwargs*

Parameters

- **exchange_name** (*str*) – name of the exchange to bind to.
- **routing_key** (*str*) – binding routing key.
- **declare_exchange** (*bool*) – should declare exchange before binding.
- **declare_kwargs** (*dict*) – is *exchange_declare()* arguments.

declare_kwargs dict:

```
'exchange_type': `required`,
'durable': `optional`, default to ``False``.
'auto_delete': `optional`, default to ``False``
'internal': `optional`, default to ``False``
`arguments`: `optional` additional exchange declaration arguments
```

Consumer.**add_rule**(*rule*)

Add a routing rule to the routing rules list.

A routing rule is added to the end of routing rules list, just before the default rule which is always the last one.

A routing rule is an instance of the class Rule. It has 3 fields, a Matcher, target, and target arguments field. The matcher is used to determines whether the target can handler the envelope. The target can be a *Handler* class or a Router instance (sub-router). The target arguments is a dictionary passed to the newly created handler for initialization.

Since the rules of the *Rule Router* are checked sequentially in the order they added, more specific rules should be added first, then generic ones later.

Parameters **rule** (Rule) – routing rule instance.

Consumer.**set_default_rule**(*rule*)

Set default rule.

Default rule, when set, it catches all unroutable envelopes.

Parameters **rule** (Rule) – default routing rule which catches all unmatched messages. None will unset default rule

Consumer.**add_handler**(*matcher_or_pattern*, *target*, *target_kwargs=None*)

Construct and add a routing rule for the provided handler.

Handlers are added as routing rules. A routing rule (*Rule*) is an object that contains a matcher (*Matcher*) instance, target (Handler subclass or a Router instance), and target kwargs. A routing rule is constructed and added routing rules list.

matcher is an instance of *Matcher* (can be a *str* explained later) which calling its *match()* method, passing *Envelope* instance, returns True or False indicating match or no match.

target can be a *Handler* subclass or a *Router* instance. When a *Router* instance is provided as a target, it will act as a sub-router which has its own routing logic. This way, a chain of routers can be constructed.

target_kwargs is a dict passed to the handler *initialize()* hook method.

When finding a handler, the *Rule Router* (which the consumer is based on) goes through the list of rules sequentially in the order they were added, invoking the matcher's match method of the each rule.

In case the router finds a match whose target is a router (sub-router) instance, its *find_handler()* is invoked it find handler

In case, the match target is a subclass of *Handler*, the router creates a handler instance, invokes its *initialize()* method passing the handler kwargs, then returns it.

The router stops upon first match, returning the handler to the consumer.

If a *default_handler* or *default_rule* is set, a default rule is added to the end of the routing rules list which will catch all unmatched messages.

The router returns `None` when there is no match and no *default_rule* is set.

Since the rules of the *Rule Router* (which the consumer is based on) are checked sequentially, more specific handlers should be added first, then generic ones later.

When passed matcher is a string, the default matcher *MessageTypeMatches* is constructed and the passed string is its message type regular expression string

Parameters

- **matcher** (*Matcher|str*) – a *Matcher* instance used to determin the match or a pattern string used for the default matcher type *MessageTypeMatches* as its message type regx pattern.
- **target** (*Type[Handler]|Router*) – a subclass of *Handler* or a *Router* instance.
- **target_kwargs** (*dict*) – a dict of kwargs that are passed to handler *initialize()* method. Only used when the target is a *Handler* subclass

`Consumer.`**set_default_handler**(*default_target*, *default_target_kwargs=None*)

Construct and add default routing rule for the given target

This method constructs a routing rule for the given target and pass it to *set_default_rule()*.

Parameters

- **default_target** (*Type[Handler]*) – Default target, which will catch all unmatched message. "None" means unset default target, unmached messages will be sent to dlx.
- **default_target_kwargs** (*dict*) – a dict of kwargs that are passed to handler *initialize()* method. Only used when the target is a *Handler* subclass

`Consumer.`**set_retry_policy**(*retry_policy*)

Set retry policy.

Retry policy can be an instance of any *RetryPolicy* subclass.

Parameters **retry_policy** (RetryPolicy) – an instance of a retry policy

`Consumer.`**unset_retry_policy**()

Unset retry policy.

2.2 Routing

When the *Consumer* receives a message from RabbitMQ, it prepares an *Envelope* object of that message, for the handler. However, the prepared envelope somehow needs to be routed to its handler, where it's actually consumed. The envelope may be routed based on the message type, payload, or any other criteria; It depends on the routing logic. For this reason, the consumer doesn't make the routing decisions itself, it delegates the routing to a router. The router sits between the consumer and handlers. Its responsibility is, to route each incoming envelope to its handler, returning the handler to the consumer for execution.

2.2.1 Router

class rabbitleap.routing.**Router**(***kwargs*)
Base class for routers.

Subclasses MUST implement *find_handler()*

Router.**find_handler**(*envelope*)
Find a handler

This method expects an *Envelope* object as an argument and returns a *Handler* instance to its caller or None, indicating the given envelope is unroutable.

Subclasses routers MUST implement method.

2.2.2 Rule Router

class rabbitleap.routing.**RuleRouter**(*consumer*, *default_rule=None*)
Bases: *rabbitleap.routing.Router*

Rule router.

Rule Router is an implementation of the base class *Router*. It uses routing rules to route envelopes to handlers. The rule router maintains a list routing rules, through which it goes sequentially to find a handler for a given envelope. Rules added to the router are appended to the end of its rules list, and since the router goes through the routing rules sequentially in the order they're added, more specific rules should be added first, then the general ones later.

A routing rule is an instance of the class *Rule*. It has 3 fields, a *Matcher*, target, and target arguments field. The matcher is used to determines whether the target can handle the envelope. The target can be a *Handler* class or a *Router* instance (sub-router). The target arguments is a dictionary passed to the newly created handler for initialization.

When the target is a *Handler* class, the router creates an instance of it, then returns the instance to the caller. However, if the target is a *Router* instance, it would act as a sub-router (child router). The parent router delegates finding the handler to the child router. The sub-router doesn't have to be of the same type, it can be any *Router* implementation. Chained routers let one router delegates the routing to the next one.

The rule router returns None when the given envelope is unroutable (no handler can handle it). However, the rule router may be configured with a default routing rule which catches all unroutable envelopes, check *set_default_rule()*.

Rule router always creates a new handler instance when its *find_handler()* is called, even for the same envelope, except when the rule's target is a *Router* instance, which may have different implementation and may not return a new instance.

Actually, the *Consumer* itself is a Rule Router. It implements extra stuff to communicate with RabbitMQ.

RuleRouter.**add_rule**(*rule*)
Add a routing rule to the routing rules list.

A routing rule is added to the end of routing rules list, just before the default rule which is always the last one.

A routing rule is an instance of the class *Rule*. It has 3 fields, a *Matcher*, target, and target arguments field. The matcher is used to determines whether the target can handler the envelope. The target can be a *Handler* class or a *Router* instance (sub-router). The target arguments is a dictionary passed to the newly created handler for initialization.

Since the rules of the *Rule Router* are checked sequentially in the order they added, more specific rules should be added first, then generic ones later.

Parameters **rule** (Rule) – routing rule instance.

RuleRouter.**set_default_rule**(*rule*)
Set default rule.

Default rule, when set, it catches all unroutable envelopes.

Parameters **rule** (Rule) – default routing rule which catches all unmatched messages. None will unset default rule

RuleRouter.**find_handler**(*envelope*)
Find and return a handler

The router goes through the routing rules list sequentially to find a handler for the given envelope.

When the target, in matched *Rule*, is a *Handler* class, an instance of it is created and returned. However, if the target is a *Router* instance, it would act as a sub-router. The sub-router's *find_handler()* is invoked to get a *Handler* instance.

None is returned when the given envelope is unroutable (no handler can handle it). However, if a default rule is set, its handler instance will be returned

NOTE: The router always creates a new handler instance for each find handler call, even for the same message.

Parameters **envelope** (Envelope) – Message envelope

Return Handler *Handler* instance

Routing Rule

class rabbitleap.routing.**Rule**(*matcher*, *target*, *target_kwargs=None*)
A matching rule.

A rule (routing rule) is an object that links a matcher (*Matcher*) instance, to a target.

The matcher is used to determines whether the target can handler the envelope. The target can be a *Handler* class or a *Router* instance (sub-router). The target arguments is a dictionary passed to the newly created handler for initialization.

Matchers

class rabbitleap.routing.**Matcher**
Base class for matchers

This is the base class for matcher. Matcher is used by *Rule* to check if its target can handle the given *Envelope* or not

Subclasses MUST implement *match()* method.

Matcher.**match**(*envelope*)
Return True or False indicating the target can handle the message.

This method accepts an *Envelope* object as an argument and returns boolean, indicating whether the target can handle the envelope or not.

Subclasses MUST implement this method

Parameters **envelope** (Envelope) – Message envelope

Returns can handle or not

Return type bool

class rabbitleap.routing.**AnyMatches**
Bases: *rabbitleap.routing.Matcher*

Match all messages macher

This matcher matches nothing. It always returns False. It's used in *RuleRouter* when a default rule is set to catch all unroutable envelopes

AnyMatches.**match**(*envelope*)
Always return True

class rabbitleap.routing.**NoneMatches**
Bases: *rabbitleap.routing.Matcher*

Match noting macher

This matcher matches nothing. It always returns False.

NoneMatches.**match**(*envelope*)
Always return False

class rabbitleap.routing.**MessageTypeMatches**(*message_type_pattern*)
Bases: *rabbitleap.routing.Matcher*

Match messages based on message type macher

This matcher does match based on the message type.

The message type is provided as a regular expression string or a compiled re.Pattern

The matcher returns True when find a match in the message type, or False otherwise.

MessageTypeMatches.**match**(*envelope*)
Return True or False indicating the target can handle the message.

This method accepts an *Envelope* object as an argument and returns boolean, indicating whether the target can handle the envelope or not.

Subclasses MUST implement this method

Parameters **envelope** (Envelope) – Message envelope

Returns can handle or not

Return type bool

2.3 Handling

Handlers what actually consume message envelopes. When the router routes an envelope and returns a handler instance to the *Consumer*, the *Consumer* executes the handler by invoking its *pre_handle()*, *handle()*, and

post_handle() methods respectively.

2.3.1 Handler

class rabbitleap.handling.**Handler**(*envelope*, ***kwargs*)
Base class for envelope handlers.

envelope is a reference to the given message envelope.

Subclasses MUST implement *handle()* method.

Methods:

initialize(): initialization hook used to initialize the handler with kwargs.

pre_handle(): pre handling method invoked before *handle()*.

handle(): to be implemented by the subclasses for the actual handling logic.

post_handle(): post handling method invoked after *handle()*.

Handler.**initialize**(***kwargs*)
Initialize handler.

This method is an initialization hook for the handler.

Handler.**pre_handle**()
Pre handle message envelope.

This method is invoked by the *Consumer* before invoking *handle()*. It's meant for validation and preparation before the actual handling.

Handler subclasses can override this method to add pre handling logic.

Handler.**handle**()
Handle message envelope.

This method is invoked by the *Consumer* after invoking *pre_handle()*. The actual envelope handling should happen in this method.

Handler subclasses MUST implement this method.

Handler.**post_handle**()
Post handle message envelope.

This method is invoked by the consumer after invoking *handle()*. It's meant for clean up and logging after the actual handling

Handler subclasses can override this method to add post handling logic.

Message Handler

class rabbitleap.handling.**MessageHandler**(*consumer*, *envelope*, ***kwargs*)
Bases: *rabbitleap.handling.Handler*

Message handler.

This class extens the *Handler* class with methods used to reject and skip envelopes, also report handling error. It hold a reference to:class:.*Consumer* instance which does the execution.

Methods:

initialize(): initialization method, a hook initialize the handler with kwargs.

pre_handle(): pre handling method invoked before *handle()*.

handle(): overridden by the subclass implementing the handling logic.

post_handle(): post handling method invoked after *handle()*.

error(): a shortcut for *Consumer.error()* to raise *HandlingError* exception.

abort(): a shortcut for *Consumer.abort()* to reject message.

skip(): a shortcut for *Consumer.skip()* to skip message handling.

MessageHandler.**abort**(*reason=None*)
Abort handling the message.

This method is a shortcut for *Consumer.abort()*.

NOTE: when called inside the handler, the handler should re-raise *AbortHandling* exception to the consumer if the exception is handled inside it.

Parameters **reason** (*str*) – Reason for aborting handling the message.

MessageHandler.**error**(*error_msg=None*)
Raise *HandlingError* exception.

This method is a shortcut for *Consumer.error()*.

Raise *HandlingError* when called.

NOTE: when called inside the handler, the handler should re-raise *HandlingError* exception to the consumer, in case the exception is handled inside.

Parameters **error_msg** (*str*) – Error message.

MessageHandler.**skip**(*reason=None*)
Skip handling the message.

This method is a shortcut for *Consumer.skip()*.

NOTE: when called inside the handler, the handler should re-raise *SkipHandling* exception to the consumer if the exception is handled inside it.

Parameters **reason** (*str*) – Reason for skipping handling the message.

2.4 Retry Policies

Noting is perfect, errors and timeouts may happen, and when such failures happen, the consumer has to decide what to do with that. By default, the consumer would reject the envelope (RabbitMQ message) when a failure happens. However, errors and timeouts issues, unless there is a software bug, usually solved with retries. Just like the routing, the consumer doesn't make the retry decision itself, the consumer delegates it to a retry policy. Retry policy defines how the retry is performed. Retries usually happens with back-offs to avoid worsening the situation by hammering other services with more requests, especially if it was a timeout issue. The consumer can be configured to use a retry policy by calling *Consumer.set_retry_policy()*, passing an instance of *RetryPolicy*. When a retry policy is set, the consumer won't reject messages, but rather, it send them to the retry policy to deal with the situation by invoking *RetryPolicy.retry()* method. Based on it's implementation, The retry policy decides how to do retries.

There are 4 different retry policies available:

1. *UnlimitedRetriesPolicy*, Unlimited retries policy
2. *LimitedRetriesPolicy*, Limited retries policy

3. *FixedDelayUnlimitedRetriesPolicy*, Fixed delay unlimited retries policy
4. *FixedDelayLimitedRetriesPolicy*, Fixed delay limited retries policy

Custom retry policies can be created by implementing the base class *RetryPolicy*

2.4.1 Retry Policy

class rabbitleap.retry_policies.**RetryPolicy**(***kwargs*)
Base class for retry policies.

Subclasses MUST implement *retry()* method.

Unlimited Retries Policy

class rabbitleap.retry_policies.**UnlimitedRetriesPolicy**(*consumer*, *initial_delay*, *max_delay*, *delay_incremented_by*, *retry_queue_suffix='retry'*, ***kwargs*)
Bases: *rabbitleap.retry_policies.BaseRetryPolicy*

Unlimited Retries Policy.

This is an implementation of *RetryPolicy* which does incremental backoff, unlimited retries.

`initial_delay`: is the initial/first backoff delay in seconds

`delay_incremented_by`: is number of seconds the backoff should be incremented by after each death

`max_delay`: is the final/maximum backoff delay in seconds that should net be exceeded

UnlimitedRetriesPolicy.**retry**(*envelope*)
Send message to retry queue to retry handling it later.

Death count is calculated by examining 'x-death' header. Based on the death count, the message is sent to a retry queue where it waits there till it expires and gets sent back to the original queue for handling retry.

Parameters **envelope** (Envelope) – Message envelope

Limited Retries Policy

class rabbitleap.retry_policies.**LimitedRetriesPolicy**(*consumer*, *retry_delays*, *retry_queue_suffix='retry'*, ***kwargs*)
Bases: *rabbitleap.retry_policies.BaseRetryPolicy*

Limited Retries Policy.

This is an implementation of *RetryPolicy* which does incremental backoff, limited number of retries.

`consumer`: message consumer instance

`retry_delays`: immutable list of retry backoff delays in seconds. Message is sent to dlx when this list is exhausted. e.g `(1, 5, 10, 60, 5 * 60)`

`retry_queue_suffix`: suffix str used when naming retry queues.

LimitedRetriesPolicy.**retry**(*envelope*)
Send message to retry queue to retry handling it later.

Death count is calculated by examining 'x-death' header. Based on the death count, the message is sent to a retry queue where it waits there till it expires and gets sent back to the original queue for handling retry.

The death count is used as an index for *retry_delays* list. Where each item in the list represents a retry delay in seconds.

The message will be rejected if the death count exceeded the length of *retry_delays* list.

Parameters **envelope** (Envelope) – Message envelope

Fixed Delay Unlimited Retries Policy

class rabbitleap.retry_policies.**FixedDelayUnlimitedRetriesPolicy**(*consumer*, *delay*, *retry_queue_suffix='retry'*, ***kwargs*)

Bases: *rabbitleap.retry_policies.UnlimitedRetriesPolicy*

Fixed delay unlimited retries policy.

This is an implementation of *RetryPolicy* which does fix backoff delay, unlimited retries.

`consumer`: consumer instance

`delay`: retry delay in seconds

`retry_queue_suffix`: suffix str used when naming retry queues.

FixedDelayUnlimitedRetriesPolicy.**retry**(*envelope*)
Send message to retry queue to retry handling it later.

Death count is calculated by examining 'x-death' header. Based on the death count, the message is sent to a retry queue where it waits there till it expires and gets sent back to the original queue for handling retry.

Parameters **envelope** (Envelope) – Message envelope

Fixed Delay Limited Retries Policy

class rabbitleap.retry_policies.**FixedDelayLimitedRetriesPolicy**(*consumer*, *delay*, *retries_limit*, *retry_queue_suffix='retry'*, ***kwargs*)

Bases: *rabbitleap.retry_policies.LimitedRetriesPolicy*

Fixed delay limited retries policy.

This is an implementation of *RetryPolicy* which does fix backoff delay, limited number of retries.

`consumer`: consumer instance

`delay`: retry delay in seconds.

`retries_limit`: retries limit count.

`retry_queue_suffix`: suffix str used when naming retry queues.

FixedDelayLimitedRetriesPolicy.**retry**(*envelope*)

Send message to retry queue to retry handling it later.

Death count is calculated by examining 'x-death' header. Based on the death count, the message is sent to a retry queue where it waits there till it expires and gets sent back to the original queue for handling retry.

The death count is used as an index for *retry_delays* list. Where each item in the list represents a retry delay in seconds.

The message will be rejected if the death count exceeded the length of *retry_delays* list.

Parameters **envelope** (Envelope) – Message envelope

CHAPTER 3

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

3.1 Types of Contributions

3.1.1 Report Bugs

Report bugs at https://github.com/asahaf/rabbitleap/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

3.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” is open to whoever wants to implement it.

3.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with “feature” is open to whoever wants to implement it.

3.1.4 Write Documentation

RabbitLeap could always use more documentation, whether as part of the official RabbitLeap docs, in docstrings, or even on the web in blog posts, articles, and such.

3.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/asahaf/rabbitleap/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

3.2 Get Started!

Ready to contribute? Here's how to set up *rabbitleap* for local development.

1. Fork the *rabbitleap* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/rabbitleap.git
```

3. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

4. When you're done making changes, check that your changes pass style and unit tests, including testing other Python versions with tox:

```
$ tox
```

To get tox, just pip install it.

5. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

3.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.

3. The pull request should work for Python 2.7, 3.4, 3.5, 3.6, 3.7, pypy, and pypy3. Check https://travis-ci.org/asahaf/rabbitleap under pull requests for active pull requests or run the `tox` command and make sure that the tests pass for all supported Python versions.

3.4 Add a New Test

When fixing a bug or adding features, it's good practice to add a test to demonstrate your fix or new feature behaves as expected. These tests should focus on one tiny bit of functionality and prove changes are correct.

To write and run your new test, follow these steps:

1. Add the new test to *tests/test_rabbitleap.py*. Focus your test on the specific bug or a small part of the new feature.
2. If you have already made changes to the code, stash your changes and confirm all your changes were stashed:

```
$ git stash
$ git stash list
```

3. Run your test and confirm that your test fails. If your test does not fail, rewrite the test until it fails on the original code:

```
$ py.test ./tests
```

4. (Optional) Run the tests with tox to ensure that the code changes work with different Python versions:

```
$ tox
```

5. Proceed work on your bug fix or new feature or restore your changes. To restore your stashed changes and confirm their restoration:

```
$ git stash pop
$ git stash list
```

6. Rerun your test and confirm that your test passes. If it passes, congratulations!

CHAPTER 4

Credits

4.1 Development Lead

- Ahmed AlSahaf <me@asahaf.com> @asahaf

4.2 Contributors

None yet. Why not be the first?

CHAPTER 5

History

5.1 0.1.1 (09/18/2018)

- First release on PyPI.

CHAPTER 6

Reference

6.1 rabbitleap

6.1.1 rabbitleap package

Submodules

rabbitleap.consumer module

The *Consumer* is the main part of RabbitLeap framework. It's responsible for connecting to RabbitMQ, besides, receiving, acknowledging, and rejecting messages. It handles connection failures and automatically reconnects to RabbitMQ. The consumer consumes a share or exclusive RabbitMQ queue.

When the *Consumer* receives a message from RabbitMQ, it packages the message properties, payload, and delivery information in an *Envelope* object, preparing it for handling. The envelope then, is routed to its handler by the *Router*, which the consumer relies on to route envelopes to their handlers.

The consumer can be configured with a retry policy. The *RetryPolicy* defines how to do handling retries incase a timeout or an error happens.

class rabbitleap.consumer.**Consumer**(*amqp_url*, *queue_name*, *durable=True*, *exclusive=False*, *dlx_name=None*, *auto_reconnect=True*, *auto_reconnect_delay=3*)

Bases: *rabbitleap.routing.RuleRouter*

RabbitMQ Consumer.

Public Methods:

start(): start consumer.

stop(): stop consumer.

restart(): restart consumer.

abort(): reject message.

skip(): skip message handling.

error(): raise *HandlingError* exception to report a handling error.

add_exchange_bind(): add exchange bind to the bindings list.

add_rule(): add a routing rule to the routing rules list.

set_default_rule(): set default routing rule to catch all unmatched messages

add_handler(): add handler, it creates a routing rule then add it to the routing rules list.

set_default_handler(): set default handler, it creates a rule then set it as default routing rule

set_retry_policy(): set a retry policy

unset_retry_policy(): un-set retry policy

abort(*reason=None*)

Abort handling the message.

This can be called during *pre_handle()* or handle() to abort handling. It raises *AbortHandling* exception. The exception is handled by the consumer, and causes the consumer to reject the message.

Raise *AbortHandling* when called

NOTE: when called inside the handler, the handler MUST re-raise *AbortHandling* exception to the consumer, in case the exception is handled inside.

Parameters **reason** (*str*) – Reason for aborting handling the message

add_exchange_bind(*exchange_name*, *routing_key*, *declare_exchange=False*, *declare_kwargs=None*)

Add exchange binding to the bindings list

NOTE: Actual exchange binding is happening during the consumer start up, when the connection with RabbitMQ established. Invoking this method after the connection is already established won't take affect until the consumer re-establishes the connection. The method *restart()* can be called to force the consumer to disconnect and reconnect again, doing exchange binding as part of that process.

Raise AssertionError if *declare_exchange* is True and *declare_kwargs* is None or 'type' is not in *declare_kwargs*

Parameters

- **exchange_name** (*str*) – name of the exchange to bind to.
- **routing_key** (*str*) – binding routing key.
- **declare_exchange** (*bool*) – should declare exchange before binding.
- **declare_kwargs** (*dict*) – is *exchange_declare()* arguments.

declare_kwargs dict:

```
'exchange_type': `required`,
'durable': `optional`, default to ``False``.
'auto_delete': `optional`, default to ``False``
'internal': `optional`, default to ``False``
`arguments`: `optional` additional exchange declaration arguments
```

add_handler(*matcher_or_pattern*, *target*, *target_kwargs=None*)

Construct and add a routing rule for the provided handler.

Handlers are added as routing rules. A routing rule (*Rule*) is an object that contains a matcher (*Matcher*) instance, target (Handler subclass or a Router instance), and target kwargs. A routing rule is constructed and added routing rules list.

matcher is an instance of *Matcher* (can be a *str* explained later) which calling its *match()* method, passing *Envelope* instance, returns `True` or `False` indicating match or no match.

target can be a *Handler* subclass or a *Router* instance. When a *Router* instance is provided as a target, it will act as a sub-router which has its own routing logic. This way, a chain of routers can be constructed.

target_kwargs is a dict passed to the handler *initialize()* hook method.

When finding a handler, the *Rule Router* (which the consumer is based on) goes through the list of rules sequentially in the order they were added, invoking the matcher's match method of the each rule.

In case the router finds a match whose target is a router (sub-router) instance, its *find_handler()* is invoked it find handler

In case, the match target is a subclass of *Handler*, the router creates a handler instance, invokes its *initialize()* method passing the handler kwargs, then returns it.

The router stops upon first match, returning the handler to the consumer.

If a *default_handler* or *default_rule* is set, a default rule is added to the end of the routing rules list which will catch all unmatched messages.

The router returns `None` when there is no match and no *default_rule* is set.

Since the rules of the *Rule Router* (which the consumer is based on) are checked sequentially, more specific handlers should be added first, then generic ones later.

When passed matcher is a string, the default matcher *MessageTypeMatches* is constructed and the passed string is its message type regular expression string

Parameters

- **matcher** (*Matcher|str*) – a *Matcher* instance used to determin the match or a pattern string used for the default matcher type *MessageTypeMatches* as its message type regx pattern.
- **target** (*Type[Handler]|Router*) – a subclass of *Handler* or a *Router* instance.
- **target_kwargs** (*dict*) – a dict of kwargs that are passed to handler *initialize()* method. Only used when the target is a *Handler* subclass

error(*error_msg=None*)

Raise *HandlingError* exception.

This method can be called inside the handler or invoked by the consumer in case of an error happened while handling the message. This method raises *HandlingError* which is handled by the consumer. The consumer will retry handling the message if a retry policy is set or reject the message incase no retry policy is set. When calling this method inside the handler, it should be called during *pre_handle()* or *handle()*.

Raise *HandlingError* when called

NOTE: when called inside the handler, the handler should re-raise *HandlingError* exception to the consumer, in case the exception is handled inside.

Parameters **error_msg** (*str*) – Error message

restart()
Restart consumer.

Close the connection and reconnect again.

set_default_handler(*default_target*, *default_target_kwargs=None*)
Construct and add default routing rule for the given target

This method constructs a routing rule for the given target and pass it to *set_default_rule()*.

Parameters

- **default_target** (*Type[Handler]*) – Default target, which will catch all unmatched message. "None" means unset default target, unmached messages will be sent to dlx.
- **default_target_kwargs** (*dict*) – a dict of kwargs that are passed to handler *initialize()* method. Only used when the target is a *Handler* subclass

set_retry_policy(*retry_policy*)
Set retry policy.

Retry policy can be an instance of any *RetryPolicy* subclass.

Parameters **retry_policy** (RetryPolicy) – an instance of a retry policy

skip(*reason=None*)
Skip handling the message.

This can be called during *pre_handle()* or *handle()* to skip handling. It raises *SkipHandling* exception. The exception is handled by the consumer, and causes the consumer to ack and skip the message.

Raise *SkipHandling* when called

NOTE: when called inside the handler, the handler should re-raise *SkipHandling* exception to the consumer, in case the exception is handled inside.

Parameters **reason** (*str*) – Reason for skipping handling the message

start()
Start consumer.

stop()
Stop consumer.

unset_retry_policy()
Unset retry policy.

rabbitleap.envelope module

class rabbitleap.envelope.**Envelope**(*properties*, *payload*, *delivery_info*)
Bases: object

Message envelope

Message properties, payload, and delivery information are all contained in this Envelope class. Envelope is what is passed to the handler instance for handling.

app_id
Retrun message app id.

This is a shortcut for *self.properties.app_id*.

cluster_id
Retrun message cluster id.

This is a shortcut for *self.properties.cluster_id*.

consumer_tag
Retrun message consumer tag.

This is a shortcut for *self.delivery_info.consumer_tag*.

content_encoding
Retrun message content encoding.

This is a shortcut for *self.properties.content_encoding*.

content_type
Retrun message content type.

This is a shortcut for *self.properties.content_type*.

correlation_id
Retrun message correlation id.

This is a shortcut for *self.properties.correlation_id*.

delivery_mode
Retrun message delivery mode.

This is a shortcut for *self.properties.delivery_mode*.

delivery_tag
Retrun message delivery tag.

This is a shortcut for *self.delivery_info.delivery_tag*.

exchange
Retrun message exchange.

This is a shortcut for *self.delivery_info.exchange*.

expiration
Retrun message expiration.

This is a shortcut for *self.properties.expiration*.

get_header(*header*)
Get message header.

headers
Retrun message headers.

This is a shortcut for *self.properties.headers*.

message_id
Retrun message message id.

This is a shortcut for *self.properties.message_id*.

priority
Retrun message priority.

This is a shortcut for *self.properties.priority*.

redelivered
Retrun message redelivered.

This is a shortcut for *self.delivery_info.redelivered*.

reply_to
Retrun message reply_to.

This is a shortcut for *self.properties.reply_to*.

routing_key
Retrun message routing key.

This is a shortcut for *self.delivery_info.routing_key*.

set_header(*header*, *value*)
Set message header

timestamp
Retrun message timestamp.

This is a shortcut for *self.properties.timestamp*.

type
Retrun message type.

This is a shortcut for *self.properties.type*.

user_id
Retrun user id.

This is a shortcut for *self.properties.user_id*.

rabbitleap.exceptions module

exception rabbitleap.exceptions.**AbortHandling**(*reason=None*)
Bases: Exception

This exception is raised when *abort()* method is called.

This exception is handled by the consumer to abort handling the message and reject it.

exception rabbitleap.exceptions.**HandlingError**(*error_msg=None*)
Bases: Exception

This exception is raised when *error()* method is called.

This exception is raise when *error()* method is called upon error in handling message. The exception is handled by the consumer to retry the handling message if a retry policy is set, or reject it otherwise.

exception rabbitleap.exceptions.**SkipHandling**(*reason=None*)
Bases: Exception

This exception is raised when *skip()* method is called.

This exception is handled by the consumer to skip handling the message.

rabbitleap.handling module

Handlers what actually consume message envelopes. When the router routes an envelope and returns a handler instance to the *Consumer*, the *Consumer* executes the handler by invoking its *pre_handle()*, *handle()*, and *post_handle()* methods respectively.

class rabbitleap.handling.**Handler**(*envelope*, ***kwargs*)
Bases: object

Base class for envelope handlers.

`envelope` is a reference to the given message envelope.

Subclasses MUST implement *`handle()`* method.

Methods:

`initialize()`: initialization hook used to initialize the handler with kwargs.

`pre_handle()`: pre handling method invoked before *`handle()`*.

`handle()`: to be implemented by the subclasses for the actual handling logic.

`post_handle()`: post handling method invoked after *`handle()`*.

handle()
Handle message envelope.

This method is invoked by the *`Consumer`* after invoking *`pre_handle()`*. The actual envelope handling should happen in this method.

`Handler` subclasses MUST implement this method.

initialize(***kwargs*)
Initialize handler.

This method is an initialization hook for the handler.

post_handle()
Post handle message envelope.

This method is invoked by the consumer after invoking *`handle()`*. It's meant for clean up and logging after the actual handling

`Handler` subclasses can override this method to add post handling logic.

pre_handle()
Pre handle message envelope.

This method is invoked by the *`Consumer`* before invoking *`handle()`*. It's meant for validation and preparation before the actual handling.

`Handler` subclasses can override this method to add pre handling logic.

class rabbitleap.handling.**MessageHandler**(*consumer*, *envelope*, ***kwargs*)
Bases: *`rabbitleap.handling.Handler`*

Message handler.

This class extens the *`Handler`* class with methods used to reject and skip envelopes, also report handling error. It hold a reference to:class:.*Consumer* instance which does the execution.

Methods:

`initialize()`: initialization method, a hook initialize the handler with kwargs.

`pre_handle()`: pre handling method invoked before *`handle()`*.

`handle()`: overridden by the subclass implementing the handling logic.

`post_handle()`: post handling method invoked after *`handle()`*.

`error()`: a shortcut for *`Consumer.error()`* to raise *`HandlingError`* exception.

`abort()`: a shortcut for *`Consumer.abort()`* to reject message.

`skip()`: a shortcut for *`Consumer.skip()`* to skip message handling.

abort(*reason=None*)
Abort handling the message.

This method is a shortcut for *Consumer.abort()*.

NOTE: when called inside the handler, the handler should re-raise *AbortHandling* exception to the consumer if the exception is handled inside it.

Parameters **reason** (*str*) – Reason for aborting handling the message.

channel
Shortcut for self.consumer.channel.

error(*error_msg=None*)
Raise *HandlingError* exception.

This method is a shortcut for *Consumer.error()*.

Raise *HandlingError* when called.

NOTE: when called inside the handler, the handler should re-raise *HandlingError* exception to the consumer, in case the exception is handled inside.

Parameters **error_msg** (*str*) – Error message.

skip(*reason=None*)
Skip handling the message.

This method is a shortcut for *Consumer.skip()*.

NOTE: when called inside the handler, the handler should re-raise *SkipHandling* exception to the consumer if the exception is handled inside it.

Parameters **reason** (*str*) – Reason for skipping handling the message.

rabbitleap.retry_policies module

Noting is perfect, errors and timeouts may happen, and when such failures happen, the consumer has to decide what to do with that. By default, the consumer would reject the envelope (RabbitMQ message) when a failure happens. However, errors and timeouts issues, unless there is a software bug, usually solved with retries. Just like the routing, the consumer doesn't make the retry decision itself, the consumer delegates it to a retry policy. Retry policy defines how the retry is performed. Retries usually happens with back-offs to avoid worsening the situation by hammering other services with more requests, especially if it was a timeout issue. The consumer can be configured to use a retry policy by calling *Consumer.set_retry_policy()*, passing an instance of *RetryPolicy*. When a retry policy is set, the consumer won't reject messages, but rather, it send them to the retry policy to deal with the situation by invoking *RetryPolicy.retry()* method. Based on it's implementation, The retry policy decides how to do retries.

There are 4 different retry policies available:

1. *UnlimitedRetriesPolicy*, Unlimited retries policy
2. *LimitedRetriesPolicy*, Limited retries policy
3. *FixedDelayUnlimitedRetriesPolicy*, Fixed delay unlimited retries policy
4. *FixedDelayLimitedRetriesPolicy*, Fixed delay limited retries policy

Custom retry policies can be created by implementing the base class *RetryPolicy*

class rabbitleap.retry_policies.**BaseRetryPolicy**(*consumer*, *retry_queue_suffix='retry'*, ***kwargs*)
Bases: *rabbitleap.retry_policies.RetryPolicy*

Base retry policy class for *UnlimitedRetriesPolicy* and *LimitedRetriesPolicy*.

It has implementation for geting mesage death count and retry queue creation.

declare_retry_queue(*delay*)
Declare a retry queue for the provided delay.

Each different delay has a different queue where all retry messages with the same delay will be sent to till they expire and get sent back to the original queue for handling retry. The queue is declared with a TTL and automatically gets deleted. The queue TTL is equal to the provided delay. The retry queue's dead letter exchange is (default) direct exchange and the dead letter routing key is the original queue name where the messages originally came from. The messages will be sent back to the original queue when they reach their TTL, for handling retry.

The retry queue is redeclared before every a new message is sent to it. Redeclaration resets the queue's TTL, preventing it from being destroyed.

Parameters **delay** (*int*) – Retry delay in seconds

Returns retry queue name

Return type str

get_death_count(*envelope*)
Return the death count of a message by examining "x-death" header.

Parameters **envelope** (Envelope) – Message envelope

Return int death count

set_original_delivery_info_header(*envelope*)
Save original message delivery infomation in a header.

class rabbitleap.retry_policies.**FixedDelayLimitedRetriesPolicy**(*consumer*, *delay*, *retries_limit*, *retry_queue_suffix='retry'*, ***kwargs*)

Bases: *rabbitleap.retry_policies.LimitedRetriesPolicy*

Fixed delay limited retries policy.

This is an implementation of *RetryPolicy* which does fix backoff delay, limited number of retries.

`consumer`: consumer instance

`delay`: retry delay in seconds.

`retries_limit`: retries limit count.

`retry_queue_suffix`: suffix str used when naming retry queues.

class rabbitleap.retry_policies.**FixedDelayUnlimitedRetriesPolicy**(*consumer*, *delay*, *retry_queue_suffix='retry'*, ***kwargs*)

Bases: *rabbitleap.retry_policies.UnlimitedRetriesPolicy*

Fixed delay unlimited retries policy.

This is an implementation of *RetryPolicy* which does fix backoff delay, unlimited retries.

`consumer`: consumer instance

`delay`: retry delay in seconds

retry_queue_suffix: suffix str used when naming retry queues.

class rabbitleap.retry_policies.**LimitedRetriesPolicy**(*consumer*, *retry_delays*, *retry_queue_suffix='retry'*, ***kwargs*)

Bases: *rabbitleap.retry_policies.BaseRetryPolicy*

Limited Retries Policy.

This is an implementation of *RetryPolicy* which does incremental backoff, limited number of retries.

consumer: message consumer instance

retry_delays: immutable list of retry backoff delays in seconds. Message is sent to dlx when this list is exhausted. e.g `(1, 5, 10, 60, 5 * 60)`

retry_queue_suffix: suffix str used when naming retry queues.

retry(*envelope*)

Send message to retry queue to retry handling it later.

Death count is calculated by examining 'x-death' header. Based on the death count, the message is sent to a retry queue where it waits there till it expires and gets sent back to the original queue for handling retry.

The death count is used as an index for *retry_delays* list. Where each item in the list represents a retry delay in seconds.

The message will be rejected if the death count exceeded the length of *retry_delays* list.

Parameters **envelope** (Envelope) – Message envelope

class rabbitleap.retry_policies.**RetryPolicy**(***kwargs*)

Bases: object

Base class for retry policies.

Subclasses MUST implement *retry()* method.

retry(*envelope*)

This method is implemented by the subclass.

class rabbitleap.retry_policies.**UnlimitedRetriesPolicy**(*consumer*, *initial_delay*, *max_delay*, *delay_incremented_by*, *retry_queue_suffix='retry'*, ***kwargs*)

Bases: *rabbitleap.retry_policies.BaseRetryPolicy*

Unlimited Retries Policy.

This is an implementation of *RetryPolicy* which does incremental backoff, unlimited retries.

initial_delay: is the initial/first backoff delay in seconds

delay_incremented_by: is number of seconds the backoff should be incremented by after each death

max_delay: is the final/maximum backoff delay in seconds that should net be exceeded

retry(*envelope*)

Send message to retry queue to retry handling it later.

Death count is calculated by examining 'x-death' header. Based on the death count, the message is sent to a retry queue where it waits there till it expires and gets sent back to the original queue for handling retry.

Parameters **envelope** (Envelope) – Message envelope

rabbitleap.routing module

When the *Consumer* receives a message from RabbitMQ, it prepares an *Envelope* object of that message, for the handler. However, the prepared envelope somehow needs to be routed to its handler, where it's actually consumed. The envelope may be routed based on the message type, payload, or any other criteria; It depends on the routing logic. For this reason, the consumer doesn't make the routing decisions itself, it delegates the routing to a router. The router sits between the consumer and handlers. Its responsibility is, to route each incoming envelope to its handler, returning the handler to the consumer for execution.

class rabbitleap.routing.**AnyMatches**
Bases: *rabbitleap.routing.Matcher*

Match all messages macher

This matcher matches nothing. It always returns `False`. It's used in *RuleRouter* when a default rule is set to catch all unroutable envelopes

match(*envelope*)
Always return `True`

class rabbitleap.routing.**Matcher**
Bases: `object`

Base class for matchers

This is the base class for matcher. Matcher is used by *Rule* to check if its target can handle the given *Envelope* or not

Subclasses MUST implement *match()* method.

match(*envelope*)
Return `True` or `False` indicating the target can handle the message.

This method accepts an *Envelope* object as an argument and returns boolean, indicating whether the target can handle the envelope or not.

Subclasses MUST implement this method

Parameters **envelope** (`Envelope`) – Message envelope

Returns can handle or not

Return type bool

class rabbitleap.routing.**MessageTypeMatches**(*message_type_pattern*)
Bases: *rabbitleap.routing.Matcher*

Match messages based on message type macher

This matcher does match based on the message type.

The message type is provided as a regular expression string or a compiled `re.Pattern`

The matcher returns `True` when find a match in the message type, or `False` otherwise.

match(*envelope*)
Return `True` or `False` indicating the target can handle the message.

This method accepts an *Envelope* object as an argument and returns boolean, indicating whether the target can handle the envelope or not.

Subclasses MUST implement this method

Parameters **envelope** (`Envelope`) – Message envelope

Returns can handle or not

Return type bool

class rabbitleap.routing.**NoneMatches**
Bases: *rabbitleap.routing.Matcher*

Match noting macher

This matcher matches nothing. It always returns `False`.

match(*envelope*)
Always return `False`

class rabbitleap.routing.**Router**(***kwargs*)
Bases: `object`

Base class for routers.

Subclasses MUST implement *find_handler()*

find_handler(*envelope*)
Find a handler

This method expects an *Envelope* object as an argument and returns a *Handler* instance to its caller or `None`, indicating the given envelope is unroutable.

Subclasses routers MUST implement method.

class rabbitleap.routing.**Rule**(*matcher*, *target*, *target_kwargs=None*)
Bases: `object`

A matching rule.

A rule (routing rule) is an object that links a matcher (*Matcher*) instance, to a target.

The matcher is used to determines whether the target can handler the envelope. The target can be a *Handler* class or a *Router* instance (sub-router). The target arguments is a dictionary passed to the newly created handler for initialization.

class rabbitleap.routing.**RuleRouter**(*consumer*, *default_rule=None*)
Bases: *rabbitleap.routing.Router*

Rule router.

Rule Router is an implementation of the base class *Router*. It uses routing rules to route envelopes to handlers. The rule router maintains a list routing rules, through which it goes sequentially to find a handler for a given envelope. Rules added to the router are appended to the end of its rules list, and since the router goes through the routing rules sequentially in the order they're added, more specific rules should be added first, then the general ones later.

A routing rule is an instance of the class *Rule*. It has 3 fields, a *Matcher*, target, and target arguments field. The matcher is used to determines whether the target can handle the envelope. The target can be a *Handler* class or a *Router* instance (sub-router). The target arguments is a dictionary passed to the newly created handler for initialization.

When the target is a *Handler* class, the router creates an instance of it, then returns the instance to the caller. However, if the target is a *Router* instance, it would act as a sub-router (child router). The parent router delegates finding the handler to the child router. The sub-router doesn't have to be of the same type, it can be any *Router* implementation. Chained routers let one router delegates the routing to the next one.

The rule router returns `None` when the given envelope is unroutable (no handler can handle it). However, the rule router may be configured with a default routing rule which catches all unroutable envelopes, check *set_default_rule()*.

Rule router always creates a new handler instance when its *find_handler()* is called, even for the same envelope, except when the rule's target is a *Router* instance, which may have different implementation and may not return a new instance.

Actually, the *Consumer* itself is a Rule Router. It implements extra stuff to communicate with RabbitMQ.

add_rule(*rule*)
Add a routing rule to the routing rules list.

A routing rule is added to the end of routing rules list, just before the default rule which is always the last one.

A routing rule is an instance of the class *Rule*. It has 3 fields, a *Matcher*, target, and target arguments field. The matcher is used to determines whether the target can handler the envelope. The target can be a *Handler* class or a *Router* instance (sub-router). The target arguments is a dictionary passed to the newly created handler for initialization.

Since the rules of the *Rule Router* are checked sequentially in the order they added, more specific rules should be added first, then generic ones later.

Parameters **rule** (Rule) – routing rule instance.

find_handler(*envelope*)
Find and return a handler

The router goes through the routing rules list sequentially to find a handler for the given envelope.

When the target, in matched *Rule*, is a *Handler* class, an instance of it is created and returned. However, if the target is a *Router* instance, it would act as a sub-router. The sub-router's *find_handler()* is invoked to get a *Handler* instance.

None is returned when the given envelope is unroutable (no handler can handle it). However, if a default rule is set, its handler instance will be returned

NOTE: The router always creates a new handler instance for each find handler call, even for the same message.

Parameters **envelope** (Envelope) – Message envelope

Return Handler *Handler* instance

set_default_rule(*rule*)
Set default rule.

Default rule, when set, it catches all unroutable envelopes.

Parameters **rule** (Rule) – default routing rule which catches all unmatched messages. None will unset default rule

Module contents

- genindex

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

rabbitleap, 39
rabbitleap.consumer, 27
rabbitleap.envelope, 30
rabbitleap.exceptions, 32
rabbitleap.handling, 32
rabbitleap.retry_policies, 34
rabbitleap.routing, 37

Index

A

abort() (*rabbitleap.consumer.Consumer method*), 28
abort() (*rabbitleap.handling.MessageHandler method*), 33
AbortHandling, 32
add_exchange_bind() (*rabbitleap.consumer.Consumer method*), 28
add_handler() (*rabbitleap.consumer.Consumer method*), 28
add_rule() (*rabbitleap.routing.RuleRouter method*), 39
AnyMatches (*class in rabbitleap.routing*), 37
app_id (*rabbitleap.envelope.Envelope attribute*), 30

B

BaseRetryPolicy (*class in rabbitleap.retry_policies*), 34

C

channel (*rabbitleap.handling.MessageHandler attribute*), 34
cluster_id (*rabbitleap.envelope.Envelope attribute*), 30
Consumer (*class in rabbitleap.consumer*), 27
consumer_tag (*rabbitleap.envelope.Envelope attribute*), 31
content_encoding (*rabbitleap.envelope.Envelope attribute*), 31
content_type (*rabbitleap.envelope.Envelope attribute*), 31
correlation_id (*rabbitleap.envelope.Envelope attribute*), 31

D

declare_retry_queue() (*rabbitleap.retry_policies.BaseRetryPolicy method*), 35
delivery_mode (*rabbitleap.envelope.Envelope attribute*), 31
delivery_tag (*rabbitleap.envelope.Envelope attribute*), 31

E

Envelope (*class in rabbitleap.envelope*), 30
error() (*rabbitleap.consumer.Consumer method*), 29
error() (*rabbitleap.handling.MessageHandler method*), 34
exchange (*rabbitleap.envelope.Envelope attribute*), 31
expiration (*rabbitleap.envelope.Envelope attribute*), 31

F

find_handler() (*rabbitleap.routing.Router method*), 38
find_handler() (*rabbitleap.routing.RuleRouter method*), 39
FixedDelayLimitedRetriesPolicy (*class in rabbitleap.retry_policies*), 35
FixedDelayUnlimitedRetriesPolicy (*class in rabbitleap.retry_policies*), 35

G

get_death_count() (*rabbitleap.retry_policies.BaseRetryPolicy method*), 35
get_header() (*rabbitleap.envelope.Envelope method*), 31

H

handle() (*rabbitleap.handling.Handler method*), 33
Handler (*class in rabbitleap.handling*), 32
HandlingError, 32
headers (*rabbitleap.envelope.Envelope attribute*), 31

I

initialize() (*rabbitleap.handling.Handler method*), 33

L

LimitedRetriesPolicy (*class in rabbitleap.retry_policies*), 36

M

match() (*rabbitleap.routing.AnyMatches method*), 37
match() (*rabbitleap.routing.Matcher method*), 37
match() (*rabbitleap.routing.MessageTypeMatches method*), 37
match() (*rabbitleap.routing.NoneMatches method*), 38
Matcher (*class in rabbitleap.routing*), 37
message_id (*rabbitleap.envelope.Envelope attribute*), 31
MessageHandler (*class in rabbitleap.handling*), 33
MessageTypeMatches (*class in rabbitleap.routing*), 37

N

NoneMatches (*class in rabbitleap.routing*), 38

P

post_handle() (*rabbitleap.handling.Handler method*), 33
pre_handle() (*rabbitleap.handling.Handler method*), 33
priority (*rabbitleap.envelope.Envelope attribute*), 31

R

rabbitleap (*module*), 39
rabbitleap.consumer (*module*), 27
rabbitleap.envelope (*module*), 30
rabbitleap.exceptions (*module*), 32
rabbitleap.handling (*module*), 32
rabbitleap.retry_policies (*module*), 34
rabbitleap.routing (*module*), 37
redelivered (*rabbitleap.envelope.Envelope attribute*), 31
reply_to (*rabbitleap.envelope.Envelope attribute*), 31
restart() (*rabbitleap.consumer.Consumer method*), 29
retry() (*rabbitleap.retry_policies.LimitedRetriesPolicy method*), 36
retry() (*rabbitleap.retry_policies.RetryPolicy method*), 36
retry() (*rabbitleap.retry_policies.UnlimitedRetriesPolicy method*), 36
RetryPolicy (*class in rabbitleap.retry_policies*), 36
Router (*class in rabbitleap.routing*), 38
routing_key (*rabbitleap.envelope.Envelope attribute*), 32
Rule (*class in rabbitleap.routing*), 38
RuleRouter (*class in rabbitleap.routing*), 38

S

set_default_handler() (*rabbitleap.consumer.Consumer method*), 30
set_default_rule() (*rabbitleap.routing.RuleRouter method*), 39
set_header() (*rabbitleap.envelope.Envelope method*), 32
set_original_delivery_info_header() (*rabbitleap.retry_policies.BaseRetryPolicy method*), 35
set_retry_policy() (*rabbitleap.consumer.Consumer method*), 30
skip() (*rabbitleap.consumer.Consumer method*), 30
skip() (*rabbitleap.handling.MessageHandler method*), 34
SkipHandling, 32
start() (*rabbitleap.consumer.Consumer method*), 30
stop() (*rabbitleap.consumer.Consumer method*), 30

T

timestamp (*rabbitleap.envelope.Envelope attribute*), 32
type (*rabbitleap.envelope.Envelope attribute*), 32

U

UnlimitedRetriesPolicy (*class in rabbitleap.retry_policies*), 36
unset_retry_policy() (*rabbitleap.consumer.Consumer method*), 30
user_id (*rabbitleap.envelope.Envelope attribute*), 32