Missive

Release 0.8.1

Jan 21, 2021

Contents

Missive is a Python framework for writing message processors.

Please beware that Missive does not (yet) maintain a stable API and is not ready for production use.

Missive's documentation is split into to two parts: narrative documentation to help explain the features (and rationale) of Missive and a reference to consult for the specifics of each class and function.

CHAPTER 1

User guide

This is the narrative part of the documentation and explains Missive piece by piece. If you're short on time (or patience!) simply read the quickstart and get going.

1.1 Quickstart

Get started quickly with this guide, covering all the main features of Missive.

1.1.1 A simple example

```
import missive

from missive.adapters.stdin import StdinAdapter

processor = missive.Processor()

@processor.handle_for(lambda m: m.raw_data == b"Hello")
def greet(message, ctx):
    print("Hello whoever you are")
    ctx.ack()

stdin_processor = StdinAdapter(missive.RawMessage, processor)

if __name__ == "__main__":
    stdin_processor.run()
```

The above code:

1. Creates a new processor called "processor"
2. Creates a new handler for b"Hello" messages
3. Adapts the processor for stdin

4. (If the file is being run directly), runs the processor

Save this as *hello_processor.py* and then run it:

```
python3 hello_processor.py
```

1.1.2 Routing and matchers

Missive routes incoming messages to specific handlers based on the matchers provided. In the example above the matcher is a lambda function but matchers can be any python Callable - for example *def* functions or classes that implement the *__call__* method. Here's a sample class:

```
class HasLabelMatcher:
    def __init__(self, label):
        self.label = label

    def __call__(self, json_message):
          return json_message.get_json().get("label") == label
```

The above matcher class will match any messages with the label matching what it was constructed with. Here's how you might use it:

```
processor = missive.Processor()

@processor.handle_for(HasLabelMatcher("sign-in"))
def record_sign_ins(message, ctx):
    ...

@processor.handle_for(...)
def another_matcher(message, ctx):
    ...
```

The above processor would route messages with the label *"sign-in"* to the *record_sign_ins* handler.

Matchers help ensure that messages of certain types are sent directly to the relevent code for dealing with them.

1.1.3 Message formats

You will notice that the above example had a message with a *get_json* method. That was a *JSONMessage* instead of a *RawMessage*. Processors can be specialised on specific message types. Some popular message types are provided and custom message types can be written easily by subclassing *Message*.

If you are using Python's typechecking facilities you can enforce message types by applying a type to your processor:

```
# All handlers for this message will be typechecked against JSONMessage
json_processor: missive.Processor[missive.JSONMessage] = missive.Processor()
```

1.1.4 Hooks

You can register hooks to run at certain times:

1. *before_processing* - at startup
2. *after_processing* - at shutdown
3. *before_handling* - before each message

4. *after_handling* - after each message

Here's an example that logs the time taken to handle each message

```
from logging import getLogger

proc = missive.Processor()

logger = getLogger(__name__)

@proc.handle_for(...)
def some_handler(message, ctx):
    ...

@proc.before_handling
def record_start_time(processing_ctx, handling_ctx):
    handling_ctx.state.start_time = datetime.utcnow()

@proc.after_handling
def print_end_time(processing_ctx, handling_ctx):
    logger.debug("took %s", datetime.utcnow() - handling_ctx.state.start_time)
```

1.1.5 Pluggable adapters

The initial example used a "stdin" adapter but adapters are pluggable and not (usually) tied up with the message format that you are using.

Instead of running a message processor using unix's stdin and stdout you might want to use Redis's PubSub facility:

```
from missive.adapters.redis import RedisPubSubAdapter
redis_pubsub_processor = RedisPubSubAdapter(
    missive.RawMessage,
    processor)

redis_pubsub_processor.run()
```

As you can see, changing the transport mechanism for messages is just a matter of what adapter is used. Just as with message formats, some adapters are provided but custom adapters can be (somewhat) easily written by subclassing the abstract *Adapter* class.

Note: Using HTTP

One important adapter is the WSGIAdapter, which allows message processors to be run as web applications (via a WSGI server such as gunicorn or uwsgi). This can be a handy way to provide a web API for message senders than for whatever reason can't or don't want to connect to your message bus.

1.1.6 Testing

One very important feature is the ability to run tests without sending messages over a real instance of your chosen message bus. Missive includes a test client that allows for this:

```
import json

test_client = json_processor.test_client()
```

(continues on next page)

(continued from previous page)

```
message = missive.JSONMessage(json.dumps({"name": "Cal"}).encode("utf-8"))

test_client.send(message)

assert message in test_client.acked
assert ... # anything else
```

There are a number of advantages to making use of a special test client that cuts out the real message bus:

1. It's easier to assert that messages are acked/nacked/etc
2. It's much faster than using a real message bus (and tests can be run in parallel)
3. It removes the need for test code to navigate the background threading patterns that are common in the real adapters.

1.1.7 Dead letter queues (DLQs)

One of the first questions that comes up in message processing systems is:

> What should I do when an error occurs during message processing?

Unlike when writing request-response model applications (like web APIs), where errors can be reported directly to the client, in publish-subscribe models the emitter of the message often is not able (or interested) in receiving an error from your processor.

What to do then? The answer is to have a special storage location for messages that cause errors in your system so that you can save them for manual inspection or debugging. It might be that some messages are improperly formatted or that your application has bugs.

Note: The "non-ack anti-pattern"

One important anti-pattern to avoid in message processors is failing to ack unprocessable messages. This leaves them on the bus (often causing them to be reprocessed over and over) eventually clogging up the bus and causing further problems.

This special place is called a "dead letter queue". Missive provides a way to register a location in which to put unprocessable messages to get them out of the message bus and somewhere else where they can be kept until they can be debugged.

```
from missive.dlq.sqlite import SQLiteDLQ

# Problem messages will be written to this sqlite database
json_processor.set_dlq(SQLiteDLQ("/var/dlq.db"))
```

Warning: "DLQs" are poorly named

Despite the fact that DLQs are "dead letter *queues*", message queues are usually a bad places for a DLQ. Message queues are designed for fast moving, in-and-out items. Dead letter queues need to be ready to deal with slower moving items that are occasionally very numerous - in the case where someone puts a lot of bad messages onto a shared bus.

A database is usually the right place.

1.1.8 What's not included

Message publication

Missive is focused on message *processing* and not message publication. There are lots of different ways to emit messages and Missive does not try to be an all-encompassing mechanism for being systems that emit and recieve messages.

This would be of limited use anyway - messages are a common means of inter-system communication. The publisher of messages may well be a Java or C++ application.

Message libraries

Likewise Missive does not try to manage message libraries or schemas. There are many many different ways to communicate schemas in-band or out-of-band and Missive aims to be able to handle all of them but does not seek control of the message schema.

Message validation

Missive is not a validation library and if you want to apply validation rules to messages you will need to do that yourself.

1.2 Key features

Missive is not in general an "opinionated" framework but it does have certain ideas as key principles.

> **Warning:** Missive is in an early stage and not all of the promises here are yet implemented.

1.2.1 Easy routing

Routing messages to their appropriate handlers is repetitive, error prone code. Missive provides an easy interface for using message "matchers" to indicate which handlers are to be applied to which messages.

The interface is dead simple: any callable taking a message as an argument and returning a boolean is a valid matcher.

1.2.2 Pluggable adapters

Missive is designed to allow your core message handling code to be agnostic about which message transport system has delivered the message. This is done by providing pluggable adapters into which `missive.Processor` objects are inserted.

Using pluggable adapters has a number of practical advantages.

Pluggable adapters also allows you to easily change your mind about which message transport you will use. If late in the project you learn that a key message publisher will not be allowed to connect to your message bus you will be able to offer them webhook-style access via the `missive.adapters.wsgi.WSGIAdapter`. If your organisation switches from Redis to Kafka (or vice versa) you will be able to switch out one adapter for another and run the same code.

1.2.3 An easy way to write fast tests

Writing automated tests is essential to producing good quality software. Tests that interface with third-party systems such as your message bus are essential.

That said, it is not necessary that every test write to and from your message bus. It's helpful to write the majority of your unit tests assuming that your message bus will work as expected and spare your time suite the time and complexity of putting every test message over the real bus.

Worse yet: some message buses are too proprietary (or licensing too expensive) for developers to be able to run them locally. Having pluggable adapters allows processors to be tested with one adapter (usually `missive.TestAdapter`) and finally deployed onto another. It used to be that this was only the case in the financial sector but with the rise of cloud computing many message transports cannot be run locally at all.

1.2.4 Dead letter queues

One of the biggest stumbling blocks in writing message processors is in handling messages that, for whatever reason, cannot be processed.

Counter-intuitively, despite the fact that this messages cannot be processed correctly they must regardless be acked to prevent them from being repeatedly redelivered - lowering throughput and creating congestion.

Unprocessable messages need to be stored somewhere for manual inspection and debugging. Missive provides a simple interface for doing so and comes, "batteries included" with a few well designed dead letter queue options.

1.3 Adapters

Adapters transform *missive.missive.Processor* instances into working message processors for the message transport which they implement.

The adapter system used in Missive allows any processor to be ported easily between different message transports.

Note: Porting between adapters assumes that no transport-specific features are being used! For example, if message leases are being extended via a transport-specific system then that handler is obviously no longer portable between transports.

Missive provides tools to help you avoid transport-specific code but "escape hatches" are always provided.

1.3.1 Built-in adapters

Missive comes with adapters for some message transports. Over time it is hoped that wider support can be added. If support for a transport you want is not present it should not be too hard to add, see *Writing custom adapters*.

Stdin

One useful source of messages (particularly for testing or local reply) is traditional unix pipes and files.

class missive.adapters.stdin.**StdinAdapter**(*message_cls: Type[M], processor: missive.missive.Processor[~M][M], filelike: Optional[BinaryIO] = None*)

ack(*message: M*) → None
Mark a message as acknowledged.

Parameters **message** – The message object to be acknowledged.

nack(*message: M*) → None
Mark a message as negatively acknowledged.

The meaning of this can vary depending on the message transport in question but generally it either returns the message to the message bus queue from which it came or triggers some special processing via some (message bus specific) dead letter queue.

Parameters **message** – The message object to be acknowledged.

WSGI

WSGI is the standard Python way of serving over HTTP. Many different "WSGI servers" exist which will efficiently serve a "WSGI application" for example gunicorn and uwsgi.

The WSGI adapter is useful for implementing "webhooks" (special endpoints that other services will call when events happen).

It also allows you to make your processor available over HTTP to allow access to it for users who for whatever reason aren't able to use a "proper" message transport.

It is also often the easiest thing to get deployed anywhere - running a new web service is typically easy in most organisations but running a new message bus is not.

Note: HTTP is comparatively slow - offering services over HTTP is convenient but there is a much higher associated overhead compared to using a true message transport.

Redis

Missive has built-in support for Redis's Pub/Sub functionality.

1.3.2 Writing custom adapters

Writing a custom adapter is as simple as subclassing *missive.missive.Adapter* and implementing an *ack* and a *nack* method.

class missive.**Adapter**(*processor: missive.missive.Processor[~M][M]*)
Abstract base class representing the API between *missive.Processor* and adapters.

ack(*message: M*) → None
Mark a message as acknowledged.

Parameters **message** – The message object to be acknowledged.

nack(*message: M*) → None
Mark a message as negatively acknowledged.

The meaning of this can vary depending on the message transport in question but generally it either returns the message to the message bus queue from which it came or triggers some special processing via some (message bus specific) dead letter queue.

Parameters **message** – The message object to be acknowledged.

The way that the adapter is to be run is completely undefined. Many adapters define a *run* method that makes the necessary network connections but this can vary widely and is not mandated.

CHAPTER 2

Reference

Documentation for all classes and functions that are part of Missive.

2.1 missive

2.1.1 missive package

Subpackages

missive.adapters package

Submodules

missive.adapters.redis module

missive.adapters.stdin module

class missive.adapters.stdin.**StdinAdapter**(*message_cls: Type[M], processor: missive.missive.Processor[~M][M], filelike: Optional[BinaryIO] = None*)

Bases: *missive.missive.Adapter*

ack(*message: M*) → None
Mark a message as acknowledged.

Parameters **message** – The message object to be acknowledged.

nack(*message: M*) → None
Mark a message as negatively acknowledged.

The meaning of this can vary depending on the message transport in question but generally it either returns the message to the message bus queue from which it came or triggers some special processing via some (message bus specific) dead letter queue.

Parameters **message** – The message object to be acknowledged.

run() → None

missive.adapters.wsgi module

Module contents

missive.dlq package

Submodules

missive.dlq.sqlite module

class missive.dlq.sqlite.**SQLiteDLQ**(*connection_str: str*)

Bases: collections.abc.MutableMapping, typing.Generic

oldest() → Tuple[missive.missive.Message, str, datetime.datetime]

Module contents

Submodules

missive.messages module

class missive.messages.**DictMessage**(*raw_data: bytes, decoder: Callable[[bytes], Dict[Any, Any]]*)

Bases: *missive.missive.Message*

contents() → Dict[Any, Any]

missive.missive module

class missive.missive.**Adapter**(*processor: missive.missive.Processor[~M][M]*)

Bases: typing.Generic

Abstract base class representing the API between missive.Processor and adapters.

ack(*message: M*) → None

Mark a message as acknowledged.

Parameters **message** – The message object to be acknowledged.

nack(*message: M*) → None

Mark a message as negatively acknowledged.

The meaning of this can vary depending on the message transport in question but generally it either returns the message to the message bus queue from which it came or triggers some special processing via some (message bus specific) dead letter queue.

Parameters **message** – The message object to be acknowledged.

class missive.missive.**HandlingContext**(*message: M, processing_ctx: missive.missive.ProcessingContext[~M][M]*)

Bases: typing.Generic

ack() → None

nack() → None

class missive.missive.**JSONMessage**(*raw_data: bytes*)

Bases: *missive.missive.Message*

get_json() → Any

class missive.missive.**Message**(*raw_data: bytes*)

Bases: object

class missive.missive.**ProcessingContext**(*message_cls: Type[M], adapter: missive.missive.Adapter[~M][M], processor: missive.missive.Processor[~M][M]*)

Bases: typing.Generic

ack(*message: M*) → None

handle(*message: M*) → None

handling_context(*message: M*) → Iterator[missive.missive.HandlingContext[~M][M]]

Enter the handling context, including calling hooks.

nack(*message: M*) → None

class missive.missive.**Processor**

Bases: typing.Generic

after_handling(*hook: Callable[[missive.missive.ProcessingContext[~M][M], missive.missive.HandlingContext[~M][M]], None]*) → None

after_processing(*hook: Callable[[missive.missive.ProcessingContext[~M][M]], None]*) → None

before_handling(*hook: Callable[[missive.missive.ProcessingContext[~M][M], missive.missive.HandlingContext[~M][M]], None]*) → None

before_processing(*hook: Callable[[missive.missive.ProcessingContext[~M][M]], None]*) → None

context(*message_cls: Type[M], adapter: missive.missive.Adapter[~M][M]*) → Iterator[missive.missive.ProcessingContext[~M][M]]

Enter the processing context, including calling hooks.

handle_for(*matcher: Callable[[M], bool]*) → Callable[[Callable[[M, missive.missive.HandlingContext[~M][M]], None]], None]

set_dlq(*dlq: MutableMapping[bytes, Tuple[M, str]]*) → None

test_client() → Iterator[missive.missive.TestAdapter[~M][M]]

class missive.missive.**ProcessorHooks**(**args*, ***kwds*)

Bases: typing.Generic

class missive.missive.**RawMessage**(*raw_data: bytes*)

Bases: *missive.missive.Message*

A raw message of just bytes with no interpretation

class missive.missive.**TestAdapter**(*processor: missive.missive.Processor[~M][M]*)

Bases: *missive.missive.Adapter*

ack(*message: M*) → None
Mark a message as acknowledged.

Parameters **message** – The message object to be acknowledged.

close() → None

nack(*message: M*) → None
Mark a message as negatively acknowledged.

The meaning of this can vary depending on the message transport in question but generally it either returns the message to the message bus queue from which it came or triggers some special processing via some (message bus specific) dead letter queue.

Parameters **message** – The message object to be acknowledged.

send(*message: M*) → None

missive.shutdown_handler module

class missive.shutdown_handler.**ShutdownHandler**(*callback: Optional[Callable[[int], None]] = None*)

Bases: object

enable() → None

set_flag() → None

should_exit() → bool

signal_handler(*signal: int*, *frame: Any*) → None

wait_for_flag() → None

Module contents

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

S

T

W