messaging Documentation

Release 0.6

Massimo Paladin

April 17, 2012

CONTENTS

Contents:

CHAPTER

ONE

MESSAGE ABSTRACTION

`Message` - abstraction of a *message*

1.1 Synopsis

Example:

```
importing messaging.message as message

# constructor + setters
msg = message.Message()
msg.body = "hello world"
msg.header = {"subject" : "test"}
msg.header["message-id"] = "123"

# fancy constructor
msg = message.Message(
    body = "hello world",
    header = {
        "subject" : "test",
        "message-id" : "123",
    },
)

# getters
if (msg.body) {
    ...
}
id = msg.header["message-id"]

# serialize it
msg.serialize()
# serialize it and compress the body with zlib
msg.serialize({"compression" : "zlib"})
# serialize it and compress the body with snappy
msg.serialize({"compression" : "snappy"})
```

1.1.1 Description

This module provides an abstraction of a *message*, as used in messaging, see for instance: http://en.wikipedia.org/wiki/Enterprise_messaging_system.

A message consists of header fields (collectively called "the header of the message") and a body.

Each header field is a key/value pair where the key and the value are text strings. The key is unique within the header so we can use a dict to represent the header of the message.

The body is either a text string or a binary string. This distinction is needed because text may need to be encoded (for instance using UTF-8) before being stored on disk or sent across the network.

To make things clear:

- a *text string* (aka *character string*) is a sequence of Unicode characters
- a *binary string* (aka *byte string*) is a sequence of bytes

Both the header and the body can be empty.

1.1.2 Json Mapping

In order to ease message manipulation (e.g. exchanging between applications, maybe written in different programming languages), we define here a standard mapping between a Message object and a JSON object.

A message as defined above naturally maps to a JSON object with the following fields:

header the message header as a JSON object (with all values being JSON strings).

body the message body as a JSON string.

text a JSON boolean specifying whether the body is text string (as opposed to binary string) or not.

encoding a JSON string describing how the body has been encoded (see below).

All fields are optional and default to empty/false if not present.

Since JSON strings are text strings (they can contain any Unicode character), the message header directly maps to a JSON object. There is no need to use encoding here.

For the message body, this is more complex. A text body can be put as-is in the JSON object but a binary body must be encoded beforehand because JSON does not handle binary strings. Additionally, we want to allow body compression in order to optionally save space. This is where the encoding field comes into play.

The encoding field describes which transformations have been applied to the message body. It is a + separated list of transformations that can be:

base64 `base64` encoding (for binary body or compressed body).

utf8 `utf8` encoding (only needed for a compressed text body).

zlib `zlib` compression.

snappy Snappy compression (http://code.google.com/p/snappy/).

Here is for instance the JSON object representing an empty message (i.e. the result of `Message()`):

```
{}
```

Here is a more complex example, with a binary body:

```
{
  "header":{"subject":"demo","destination":"/topic/test"},
  "body":"YWJj7g==",
  "encoding":"base64"
}
```

You can use the `Message.jsonify()` method to convert a `Message` object into a dict representing the equivalent JSON object.

Conversely, you can create a new `Message` object from a compatible JSON object (again, a `dict`) with the `dejsonify()` method.

Using this JSON mapping of messages is very convenient because you can easily put messages in larger JSON data structures. You can for instance store several messages together using a JSON array of these messages.

Here is for instance how you could construct a message containing in its body another message along with error information:

```
try:
    import simplejson as json
except ImportError:
    import json
import time
# get a message from somewhere...
msg1 = ...
# jsonify it and put it into a simple structure
body = {
    "message" : msg1.jsonify(),
    "error"   : "an error message",
    "time"    : time.time(),
}
# create a new message with this body
msg2 = message.Message(body = json.dumps(body))
msg2.header["content-type"] = "message/error"
```

A receiver of such a message can easily decode it:

```
try:
    import simplejson as json
except ImportError:
    import json
# get a message from somewhere...
msg2 = ...
# extract the body which is a JSON object
body = json.loads(msg2.body)
# extract the inner message
msg1 = message.dejsonify(body['message'])
```

1.1.3 Stringification and Serialization

In addition to the JSON mapping described above, we also define how to *stringify* and *serialize* a message.

A *stringified message* is the string representing its equivalent JSON object. A stringified message is a text string and can for instance be used in another message. See the `Message.stringify()` and `destringify()` methods.

A *serialized message* is the UTF-8 encoding of its stringified representation. A serialized message is a binary string and can for instance be stored in a file. See the `Message.serialize()` and `deserialize()` methods.

For instance, here are the steps needed in order to store a message into a file:

1. transform the programming language specific abstraction of the message into a JSON object
2. transform the JSON object into its (text) string representing
3. transform the JSON text string into a binary string using UTF-8 encoding

1 is called `Message.jsonify()`, *1 + 2* is called `Message.stringify()` and *1 + 2 + 3* is called `Message.serialize()`.

To sum up:

```
        Message object
              |  ^
   jsonify() |  | dejsonify()
              v  |
      JSON compatible dict
              |  ^
 JSON encode |  | JSON decode
              v  |
         text string
              |  ^
UTF-8 encode |  | UTF-8 decode
              v  |
        binary string
```


class `messaging.message.`**`Message`**(*body=''*, *header={}*)

A Message abstraction class.

`body`

Returns the body of the message.

`clone`()

Returns a clone of the message.

`equals`(*other*)

Check if the message is equal to the given one.

`get_body`()

Returns the body of the message.

`get_header`()

Return the header of the message.

`get_text`()

Is it a text message?

`header`

Return the header of the message.

`is_text`()

Is it a text message?

`jsonify`(*option={}*)

Transforms the message to JSON.

`md5`()

Return the checksum of the message.

`serialize`(*option={}*)

Serialize message.

`set_body`(*value*)

Set the message body to new value.

`set_header`(*value*)

Set the message header to new value.

set_text(*value*)
Set if the message is text.

size()
Returns an approximation of the message size.

stringify(*option={}*)
Transforms the message to string.

text
Is it a text message?

messaging.message.**dejsonify**(*obj*)
Returns a message from json structure.

messaging.message.**deserialize**(*binary*)
Deserialize a message.

messaging.message.**destringify**(*string*)
Destringify the given message.

messaging.message.**is_ascii**(*string*)
Returns True is the string is ascii.

messaging.message.**is_bytes**(*string*)
Check if given string is a byte string.

CHAPTER

TWO

MESSAGE GENERATOR

2.1 Synopsis

Example:

```
import messaging.generator as generator;

# create the generator
mg = generator.Generator(
    body_content = "binary",
    body_size = 1024,
)

# use it to generate 10 messages
for i in range(10):
    msg = mg.message()
    ... do something with it ...
```

2.2 Description

This module provides a versatile message generator that can be useful for stress testing or benchmarking messaging brokers or libraries.

class messaging.generator.**Generator**(***kwargs*)

A message generator tool.

message()

Returns a newly generated Message object

Options

When creating a message generator, the following options can be given:

body-content

•string: specifying the body content type; depending on this value, the body will be made of:

•base64: only Base64 characters

•binary: anything

- index: the message index number, starting at 1, optionally adjusted to match the C<body-size> (this is the default)
- text: only printable 7-bit ASCII characters

body-size integer specifying the body size

header-count integer specifying the number of header fields

header-value-size integer specifying the size of each header field value (default is -32)

header-name-size integer specifying the size of each header field name (default is -16)

header-name-prefix string to prepend to all header field names (default is C<rnd->)

Note: all integer options can be either positive (meaning exactly this value) or negative (meaning randomly distributed around the value).

For instance:

```
mg = Generator(
    header_count = 10,
    header_value_size = -20,
)
```

It will generate messages with exactly 10 random header fields, each field value having a random size between 0 and 40 and normally distributed around 20.

set(*option*, *value*)
Set Generator option to value provided.

messaging.generator.**maybe_randomize**(*size*)
Maybe randomize int.

messaging.generator.**rndb64**(*size*)
Returns a random text string of the given size (Base64 characters).

messaging.generator.**rndbin**(*size*)
Returns a random binary string of the given size.

messaging.generator.**rndint**(*size*)
Returns a random integer between 0 and 2*size with a normal distribution.

See Irwin-Hall in http://en.wikipedia.org/wiki/Normal_distribution

messaging.generator.**rndstr**(*size*)
Returns a random text string of the given size (all printable characters).

CHAPTER

THREE

MESSAGE QUEUE

`Queue` - abstraction of a message queue

3.1 Synopsis

Example:

```
import messaging.queue as queue

mq = queue.new(type="Foo", ... options ...);
# is identical too
mq = queue.foo.Foo(... options ...);
```

3.2 Description

This module provides an abstraction of a message queue. Its only purpose is to offer a unified method to create a new queue. The functionality is implemented in child modules such as `messaging.queue.dq.DQ`.

`messaging.queue.`**`new`**(*option*)

Create a new message queue object; options must contain the type of queue (which is the name of the child class), see above.

CHAPTER

FOUR

STOMPPY HELPER

This class add support for Message module in stomppy default listener. It can be passed directly to `Connection.set_listener()`.

class `messaging.stomppy.`**`MessageListener`**

This class add support for Message module in stomppy default listener. It can be passed directly to Connection.set_listener().

`connected`(*message*)

Called by the STOMP connection when a CONNECTED frame is received, that is after a connection has been established or re-established.

param message received from the server.

`error`(*message*)

Called by the STOMP connection when an ERROR frame is received.

param message received from the server.

`message`(*message*)

Called by the STOMP connection when a MESSAGE frame is received.

param message received from the server.

`on_connected`(*headers*, *body*)

Translate standard call to custom one.

`on_connecting`(*host_and_port*)

Called by the STOMP connection once a TCP/IP connection to the STOMP server has been established or re-established. Note that at this point, no connection has been established on the STOMP protocol level. For this, you need to invoke the "connect" method on the connection.

param host_and_port a tuple containing the host name and port number to which the connection has been established.

`on_disconnected`()

Called by the STOMP connection when a TCP/IP connection to the STOMP server has been lost. No messages should be sent via the connection until it has been reestablished.

`on_error`(*headers*, *body*)

Translate standard call to custom one.

`on_heartbeat_timeout`()

Called by the STOMP connection when a heartbeat message has not been received beyond the specified period.

on_message(*headers*, *body*)
Translate standard call to custom one.

on_receipt(*headers*, *body*)
Translate standard call to custom one.

on_send(*headers*, *body*)
Translate standard call to custom one.

receipt(*message*)
Called by the STOMP connection when a RECEIPT frame is received, sent by the server if requested by the client using the 'receipt' header.

param message received from the server.

send(*message*)
Called by the STOMP connection when it is in the process of sending a message.

param message being sent.

CHAPTER

FIVE

ERRORS

Errors used in the module.

Copyright (C) 2012 CERN

exception messaging.error.**GeneratorError**
Raised when errors occurs during Generator handling.

exception messaging.error.**MessageError**
Raised when errors occurs during Message handling.

Messaging is a set of Python modules useful to deal with *messages*, as used in messaging, see for instance: http://en.wikipedia.org/wiki/Enterprise_messaging_system.

The modules include a transport independent message abstraction, a versatile message generator and several message queues/spools to locally store messages.

You can download the module at the following link: http://pypi.python.org/pypi/messaging/

An Perl implementation of the same abstractions and queue algorithms is available at the following page: http://search.cpan.org/~lcons/Messaging-Message/

Copyright (C) 2012 CERN

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

m