Fedora Messaging

Release 1.7.2

Jeremy Cline

Aug 02, 2019

Contents

This package provides tools and APIs to make using Fedora's messaging infrastructure easier. These include a framework for declaring message schemas, a set of synchronous APIs to publish messages to AMQP brokers, a set of asynchronous APIs to consume messages, and services to easily run consumers.

This library is designed to be a replacement for the PyZMQ-backed fedmsg library.

CHAPTER 1

User Guide

1.1 Installation

1.1.1 PyPI

The Python package is available on the Python Package Index (PyPI) as `fedora-messaging`:

```
$ pip install --user fedora-messaging
```

It is, of course, recommended that you install it in a Python virtual environment.

1.1.2 Fedora

The library is available in Fedora 29 and greater as `fedora-messaging`:

```
$ sudo dnf install fedora-messaging
```

1.2 Quick Start

This is a quick-start guide that covers a few common use-cases and contains pointers to more in-depth documentation for the curious.

1.2.1 Local Broker

To publish and consume messages locally can be a useful way to learn about the library, and is also helpful during development of your application or service.

To install the message broker on Fedora:

```
$ sudo dnf install rabbitmq-server
```

RabbitMQ is also available in EPEL7, although it is quite old and the library is not regularly tested against it. You can also install the broker from RabbitMQ directly if you are not using Fedora.

Next, it's recommended that you enable the management interface:

```
$ sudo rabbitmq-plugins enable rabbitmq_management
```

This provides an HTTP interface and API, available at http://localhost:15672/ by default. The "guest" user with the password "guest" is created by default.

Finally, start the broker:

```
$ sudo systemctl start rabbitmq-server
```

You should now be able to consume messages with the following Python script:

```
from fedora_messaging import api, config

config.conf.setup_logging()
api.consume(lambda message: print(message))
```

To learn more about consuming messages, check out the *Consumers* documentation.

You can publish messages with:

```
from fedora_messaging import api, config

config.conf.setup_logging()
api.publish(api.Message(topic="hello", body={"Hello": "world!"}))
```

To learn more about publishing messages, check out the *Publishing* documentation.

1.2.2 Fedora's Public Broker

Fedora's message broker has a publicly accessible virtual host located at `amqps://rabbitmq.fedoraproject.org/%2Fpublic_pubsub`. This virtual host mirrors all messages published to the restricted `/pubsub` virtual host and allows anyone to consume messages being published by the various Fedora services.

These public queues have some restrictions applied to them. Firstly, they are limited to about 50 megabytes in size, so if your application cannot handle the message throughput messages will be automatically discarded once you hit this limit. Secondly, queues that are set to be durable (in other words, not exclusive or auto-deleted) are automatically deleted after approximately an hour.

If you need more robust guarantees about message delivery, or if you need to publish messages into Fedora's message broker, contact the Fedora Infrastructure team about getting access to the private virtual host.

Getting Connected

The public virtual host still requires users to authenticate when connecting, so a public user has been created and its private key and x509 certificate are distributed with fedora-messaging.

If fedora-messaging was installed via RPM, they should be in `/etc/fedora-messaging/` along with a configuration file called `fedora.toml`. If it's been installed via pip, it's easiest to get the key, certificate, and the CA certificate from the upstream git repository and start with the following configuration file:

```
# A basic configuration for Fedora's message broker, using the example callback
# which simply prints messages to standard output.
#
# This file is in the TOML format.
amqp_url = "amqps://fedora:@rabbitmq.fedoraproject.org/%2Fpublic_pubsub"
callback = "fedora_messaging.example:printer"

[tls]
ca_cert = "/etc/fedora-messaging/cacert.pem"
keyfile = "/etc/fedora-messaging/fedora-key.pem"
certfile = "/etc/fedora-messaging/fedora-cert.pem"

[client_properties]
app = "Example Application"
# Some suggested extra fields:
# URL of the project that provides this consumer
app_url = "https://github.com/fedora-infra/fedora-messaging"
# Contact emails for the maintainer(s) of the consumer - in case the
# broker admin needs to contact them, for e.g.
app_contacts_email = ["jcline@fedoraproject.org"]

[exchanges."amq.topic"]
type = "topic"
durable = true
auto_delete = false
arguments = {}

# Queue names *must* be in the normal UUID format: run "uuidgen" and use the
# output as your queue name. If your queue is not exclusive, anyone can connect
# and consume from it, causing you to miss messages, so do not share your queue
# name. Any queues that are not auto-deleted on disconnect are garbage-collected
# after approximately one hour.
#
# If you require a stronger guarantee about delivery, please talk to Fedora's
# Infrastructure team.
[queues.00000000-0000-0000-0000-000000000000]
durable = false
auto_delete = true
exclusive = true
arguments = {}

[[bindings]]
queue = "00000000-0000-0000-0000-000000000000"
exchange = "amq.topic"
routing_keys = ["#"]  # Set this to the specific topics you are interested in.

[consumer_config]
example_key = "for my consumer"

[qos]
prefetch_size = 0
prefetch_count = 25

[log_config]
version = 1
disable_existing_loggers = true
```

(continues on next page)

(continued from previous page)

```
[log_config.formatters.simple]
format = "[%(levelname)s %(name)s] %(message)s"

[log_config.handlers.console]
class = "logging.StreamHandler"
formatter = "simple"
stream = "ext://sys.stdout"

[log_config.loggers.fedora_messaging]
level = "INFO"
propagate = false
handlers = ["console"]

[log_config.loggers.twisted]
level = "INFO"
propagate = false
handlers = ["console"]

[log_config.loggers.pika]
level = "WARNING"
propagate = false
handlers = ["console"]

# If your consumer sets up a logger, you must add a configuration for it
# here in order for the messages to show up. e.g. if it set up a logger
# called 'example_printer', you could do:
#[log_config.loggers.example_printer]
#level = "INFO"
#propagate = false
#handlers = ["console"]

[log_config.root]
level = "ERROR"
handlers = ["console"]
```

Assuming the `/etc/fedora-messaging/fedora.toml`, `/etc/fedora-messaging/cacert.pem`, `/etc/fedora-messaging/fedora-key.pem`, and `/etc/fedora-messaging/fedora-cert.pem` files exist, the following command will create a configuration file called `my_config.toml` with a unique queue name for your consumer:

```
$ sed -e "s/[0-9a-f]\{8\}-[0-9a-f]\{4\}-[0-9a-f]\{4\}-[0-9a-f]\{4\}-[0-9a-f]\{12\}/
↪$(uuidgen)/g" \
    /etc/fedora-messaging/fedora.toml > my_config.toml
```

Warning: Do not skip the step above. This is important because if there are multiple consumers on a queue the broker delivers messages to them in a round-robin fashion. In other words, you'll only get some of the messages being sent.

Run a quick test to make sure you can connect to the broker. The configuration file comes with an example consumer which simply prints the message to standard output:

```
$ fedora-messaging --conf my_config.toml consume
```

Alternatively, you can start a Python shell and use the API:

```
$ FEDORA_MESSAGING_CONF=my_config.toml python
>>> from fedora_messaging import api, config
>>> config.conf.setup_logging()
>>> api.consume(lambda message: print(message))
```

If all goes well, you'll see a log entry similar to:

```
Successfully registered AMQP consumer Consumer(queue=af0f78d2-159e-4279-b404-
↪7b8c1b4649cc, callback=<function printer at 0x7f9a59e077b8>)
```

This will be followed by the messages being sent inside Fedora's Infrastructure. All that's left to do is change the callback in the configuration to use your consumer *callback* and adjusting the routing keys in your *bindings* to receive only the messages your consumer is interested in.

1.2.3 Fedora's Restricted Broker

Connecting the Fedora's private virtual host requires working with the Fedora infrastructure team. The current process and configuration for this is documented in the infrastructure team's development guide.

1.3 Configuration

fedora-messaging can be configured with the `/etc/fedora-messaging/config.toml` file or by setting the `FEDORA_MESSAGING_CONF` environment variable to the path of the configuration file.

Each configuration option has a default value.

Table of Configuration Options

- *qos*

A complete example TOML configuration:

```
# A sample configuration for fedora-messaging. This file is in the TOML format.
amqp_url = "amqp://"
callback = "fedora_messaging.example:printer"
passive_declares = false
publish_exchange = "amq.topic"
topic_prefix = ""

[tls]
ca_cert = "/etc/fedora-messaging/cacert.pem"
keyfile = "/etc/fedora-messaging/fedora-key.pem"
certfile = "/etc/fedora-messaging/fedora-cert.pem"

[client_properties]
app = "Example Application"

# If the exchange or queue name  has a "." in it, use quotes as seen here.
[exchanges."amq.topic"]
type = "topic"
durable = true
auto_delete = false
arguments = {}

[queues.my_queue]
durable = true
auto_delete = false
exclusive = false
arguments = {}

# Note the double brackets below. To add another binding, add another
# [[bindings]] section. To use multiple routing keys, just expand the list here.
[[bindings]]
queue = "my_queue"
exchange = "amq.topic"
routing_keys = ["#"]

[consumer_config]
example_key = "for my consumer"

[qos]
prefetch_size = 0
prefetch_count = 25

[log_config]
version = 1
disable_existing_loggers = true

[log_config.formatters.simple]
format = "[%(levelname)s %(name)s] %(message)s"

[log_config.handlers.console]
class = "logging.StreamHandler"
formatter = "simple"
stream = "ext://sys.stdout"
```

(continues on next page)

(continued from previous page)

```
[log_config.loggers.fedora_messaging]
level = "INFO"
propagate = false
handlers = ["console"]

# Twisted is the asynchronous framework that manages the TCP/TLS connection, as well
# as the consumer event loop. When debugging you may want to lower this log level.
[log_config.loggers.twisted]
level = "INFO"
propagate = false
handlers = ["console"]

# Pika is the underlying AMQP client library. When debugging you may want to
# lower this log level.
[log_config.loggers.pika]
level = "WARNING"
propagate = false
handlers = ["console"]

[log_config.root]
level = "ERROR"
handlers = ["console"]
```

1.3.1 Generic Options

These options apply to both consumers and publishers.

amqp_url

The AMQP broker to connect to. This URL should be in the format described by the `pika.connection.URLParameters` documentation. This defaults to `'amqp://?connection_attempts=3&retry_delay=5`.

Note: When using the Twisted consumer API, which the CLI does by default, any connection-related setting won't apply as Twisted manages the TCP/TLS connection.

passive_declares

A boolean to specify if queues and exchanges should be declared passively (i.e checked, but not actually created on the server). Defaults to `False`.

tls

A dictionary of the TLS settings to use when connecting to the AMQP broker. The default is:

```
{
    'ca_cert': '/etc/pki/tls/certs/ca-bundle.crt',
    'keyfile': None,
```

(continues on next page)

(continued from previous page)

```
    'certfile': None,
}
```

The value of `ca_cert` should be the path to a bundle of CA certificates used to validate the certificate presented by the server. The 'keyfile' and 'certfile' values should be to the client key and client certificate to use when authenticating with the broker.

Note: The broker URL must use the `amqps` scheme. It is also possible to provide these setting via the `amqp_url` setting using a URL-encoded JSON object. This setting is provided as a convenient way to avoid that.

client_properties

A dictionary that describes the client to the AMQP broker. This makes it easy to identify the application using a connection. The dictionary can contain arbitrary string keys and values. The default is:

```
{
    'app': 'Unknown',
    'product': 'Fedora Messaging with Pika',
    'information': 'https://fedora-messaging.readthedocs.io/en/stable/',
    'version': 'fedora_messaging-<version> with pika-<version>',
}
```

Apps should set the `app` along with any additional keys they feel will help administrators when debugging application connections. Do not use the `product`, `information`, and `version` keys as these will be set automatically.

exchanges

A dictionary of exchanges that should be present in the broker. Each key should be an exchange name, and the value should be a dictionary with the exchange's configuration. Options are:

- `type` - the type of exchange to create.
- `durable` - whether or not the exchange should survive a broker restart.
- `auto_delete` - whether or not the exchange should be deleted once no queues are bound to it.
- `arguments` - dictionary of arbitrary keyword arguments for the exchange, which depends on the broker in use and its extensions.

For example:

```
{
    'my_exchange': {
        'type': 'fanout',
        'durable': True,
        'auto_delete': False,
        'arguments': {},
    },
}
```

The default is to ensure the 'amq.topic' topic exchange exists which should be sufficient for most use cases.

log_config

A dictionary describing the logging configuration to use, in a format accepted by `logging.config.dictConfig()`.

Note: Logging is only configured for consumers, not for producers.

1.3.2 Publisher Options

The following configuration options are publisher-related.

publish_exchange

A string that identifies the exchange to publish to. The default is `amq.topic`.

topic_prefix

A string that will be prepended to topics on sent messages. This is useful to migrate from fedmsg, but should not be used otherwise. The default is an empty string.

1.3.3 Consumer Options

The following configuration options are consumer-related.

queues

A dictionary of queues that should be present in the broker. Each key should be a queue name, and the value should be a dictionary with the queue's configuration. Options are:

- `durable` - whether or not the queue should survive a broker restart. This is set to `False` for the default queue.
- `auto_delete` - whether or not the queue should be deleted once the consumer disconnects. This is set to `True` for the default queue.
- `exclusive` - whether or not the queue is exclusive to the current connection. This is set to `False` for the default queue.
- `arguments` - dictionary of arbitrary keyword arguments for the queue, which depends on the broker in use and its extensions. This is set to `{}` for the default queue

For example:

```
{
    'my_queue': {
        'durable': True,
        'auto_delete': True,
        'exclusive': False,
        'arguments': {},
    },
}
```

bindings

A list of dictionaries that define queue bindings to exchanges that consumers will subscribe to. The `queue` key is the queue's name. The `exchange` key should be the exchange name and the `routing_keys` key should be a list of routing keys. For example:

```
[
    {
        'queue': 'my_queue',
        'exchange': 'amq.topic',
        'routing_keys': ['topic1', 'topic2.#'],
    },
]
```

This would create two bindings for the `my_queue` queue, both to the `amq.topic` exchange. Consumers will consume from both queues.

callback

The Python path of the callback. This should be in the format `<module>:<object>`. For example, if the callback was called "my_callback" and was located in the "my_module" module of the "my_package" package, the path would be defined as `my_package.my_module:my_callback`. The default is None.

Consult the *Consumers* documentation for details on implementing a callback.

consumer_config

A dictionary for the consumer to use as configuration. The consumer should access this key in its callback for any configuration it needs. Defaults to an empty dictionary. If, for example, this dictionary contains the `print_messages` key, the callback can access this configuration with:

```
from fedora_messaging import config

def callback(message):
    if config.conf["consumer_config"]["print_messages"]:
        print(message)
```

qos

The quality of service settings to use for consumers. This setting is a dictionary with two keys. `prefetch_count` specifies the number of messages to pre-fetch from the server. Pre-fetching messages improves performance by reducing the amount of back-and-forth between client and server. The downside is if the consumer encounters an unexpected problem, messages won't be returned to the queue and sent to a different consumer until the consumer times out. `prefetch_size` limits the size of pre-fetched messages (in bytes), with 0 meaning there is no limit. The default settings are:

```
{
    'prefetch_count': 10,
    'prefetch_size': 0,
}
```

1.4 Publishing

1.4.1 Overview

Publishing messages is simple. Messages are made up of a topic, some optional headers, and a body. Messages are encapsulated in a *fedora_messaging.message.Message* object. For details on defining messages, see the *Messages* documentation. For details on the publishing API, see the *Publishing* API documentation.

Topics

Topics are strings of words separated by the `.` character, up to 255 characters. Topics are used by clients to filter messages, so choosing a good topic helps reduce the number of messages sent to a client. Topics should start broadly and become more specific.

Headers

Headers are key-value pairs attached that are useful for storing information about the message itself. This library adds a header to every message with the `fedora_messaging_schema` key, pointing to the message schema used.

You should not use any key starting with `fedora_messaging` for yourself.

You can write *Header Schema* for your messages to enforce a particular schema.

Body

The only restrictions on the message body is that it must be serializable to a JSON object. You should write a *Body Schema* for your messages to ensure you don't change your message format unintentionally.

1.4.2 Introduction

To publish a message, first create a *fedora_messaging.message.Message* object, then pass it to the *fedora_messaging.api.publish()* function:

```
from fedora_messaging import api, message

msg = message.Message(topic=u'nice.message', headers={u'niceness': u'very'},
                      body={u'encouragement': u"You're doing great!"})
api.publish(msg)
```

The API relies on the *Configuration* you've provided to connect to the message broker and publish the message to an exchange.

1.4.3 Handling Errors

Your message might fail to publish for a number of reasons, so you should be prepared to see (and potentially handle) some errors.

Validation

The message you create may not be successfully validated against its schema. This is not an error you should catch, since it must be fixed by the developer and cannot be recovered from.

Connection Errors

The publish API will attempt to reconnect to the broker several times before an exception is raised. Once this occurs it is up to the application to decide what to do.

Rejected Messages

The broker may reject a message. This could occur because the message is too large, or because the publisher does not have permission to publish messages with a particular topic, or some other reason.

1.5 Consumers

This library is aimed at making implementing a message consumer as simple as possible by implementing common boilerplate code and offering a command line interface to easily start a consumer as a service under init systems like systemd.

1.5.1 Introduction

AMQP consumers configure a queue for their use in the message broker. When a message is published to an exchange and matches the bindings the consumer has declared, the message is placed in the queue and eventually delivered to the consumer. Fedora uses a topic exchange for general-purpose messages.

Fortunately, you don't need to manage the connection to the broker or configure the queue. All you need to do is to implement some code to run when a message is received. The API expects a callable object that accepts a single positional argument:

```
from fedora_messaging import api, config

# The fedora_messaging API does not automatically configure logging so as
# to not destroy application logging setup. This is a convenience method
# to configure the Python logger with the fedora-messaging logging config.
config.conf.setup_logging()

# First, define a function to be used as our callback. This will be called
# whenever a message is received from the server.
def printer_callback(message):
    """
    Print the message to standard output.

    Args:
        message (fedora_messaging.message.Message): The message we received
            from the queue.
    """
    print(str(message))

# Next, we need a queue to consume messages from. We can define
# the queue and binding configurations in these dictionaries:
```

(continues on next page)

(continued from previous page)

```
queues = {
    'demo': {
        'durable': False,  # Delete the queue on broker restart
        'auto_delete': True,  # Delete the queue when the client terminates
        'exclusive': False,  # Allow multiple simultaneous consumers
        'arguments': {},
    },
}
binding = {
    'exchange': 'amq.topic',  # The AMQP exchange to bind our queue to
    'queue': 'demo',  # The unique name of our queue on the AMQP broker
    'routing_keys': ['#'],  # The topics that should be delivered to the queue
}

# Start consuming messages using our callback. This call will block until
# a KeyboardInterrupt is raised, or the process receives a SIGINT or SIGTERM
# signal.
api.consume(printer_callback, bindings=binding, queues=queues)
```

In this example, there's one queue and the queue only has one binding, but it's possible to consume from multiple queues and each queue can have multiple bindings.

1.5.2 Command Line Interface

A command line interface, *fedora-messaging*, is included to make running consumers easier. It's not necessary to write any boilerplate code calling the API, just run `fedora-messaging consume` and provide it the Python path to your callback:

```
$ fedora-messaging consume --callback=fedora_messaging.example:printer
```

Consult the manual page for complete details on this command line interface.

Note: For users of fedmsg, this is roughly equivalent to fedmsg-hub

1.5.3 Consumer API

The introduction contains a very minimal callback. This section covers the complete API for consumers.

The Callback

The callback provided to *fedora_messaging.api.consume()* or the command-line interface can be any callable Python object, so long as it accepts the message object as a single positional argument.

The API will also accept a Python class, which it will instantiate before using as a callable object. This allows you to write a callback with easy one-time initialization or a callback that maintains state between calls:

```
import os

from fedora_messaging import api, config


```

(continues on next page)

(continued from previous page)

```
class SaveMessage(object):
    """
    A fedora-messaging consumer that saves the message to a file.

    A single configuration key is used from fedora-messaging's
    "consumer_config" key, "path", which is where the consumer will save
    the messages::

        [consumer_config]
        path = "/tmp/fedora-messaging/messages.txt"
    """

    def __init__(self):
        """Perform some one-time initialization for the consumer."""
        self.path = config.conf["consumer_config"]["path"]

        # Ensure the path exists before the consumer starts
        if not os.path.exists(os.path.dirname(self.path)):
            os.mkdir(os.path.dirname(self.path))

    def __call__(self, message):
        """
        Invoked when a message is received by the consumer.

        Args:
            message (fedora_messaging.api.Message): The message from AMQP.
        """
        with open(self.path, "a") as fd:
            fd.write(str(message))

api.consume(SaveMessage)
```

When running this type of callback from the command-line interface, specify the Python path to the class object, not the `__call__` method:

```
$ fedora-messaging consume --callback=package_name.module:SaveMessage
```

Exceptions

- Consumers should raise the *fedora_messaging.exceptions.Nack* exception if the consumer cannot handle the message at this time. The message will be re-queued, and the server will attempt to re-deliver it at a later time.
- Consumers should raise the *fedora_messaging.exceptions.Drop* exception when they wish to explicitly indicate they do not want handle the message. This is similar to simply calling `return`, but the server is informed the client dropped the message. What the server does depends on configuration.
- Consumers should raise the *fedora_messaging.exceptions.HaltConsumer* exception if they wish to stop consuming messages.

If a consumer raises any other exception, a traceback will be logged at the error level, the message being processed and any pre-fetched messages will be returned to the queue for later delivery, and the consumer will be canceled.

If the CLI is being used, it will halt with a non-zero exit code. If the API is being used directly, consult the API documentation for exact results, as the synchronous and asynchronous APIs communicate failures differently.

Synchronous and Asynchronous Calls

The AMQP consumer runs in a Twisted event loop. When a message arrives, it calls the callback in a separate Python thread to avoid blocking vital operations like the connection heartbeat. The callback is free to use any blocking (synchronous) calls it likes.

Note: Your callback does not need to be thread-safe. By default, messages are processed serially.

It is safe to start threads to perform IO-blocking work concurrently. If you wish to make use of a Twisted API, you must use the `twisted.internet.threads.blockingCallFromThread()` or `twisted.internet.interfaces.IReactorFromThreads` APIs.

Consumer Configuration

A special section of the fedora-messaging configuration will be available for consumers to use if they need configuration options. Refer to the *consumer_config* in the Configuration documentation for details.

1.5.4 systemd Service

A systemd service file is also included in the Python package for your convenience. It is called `fm-consumer@.service` and simply runs `fedora-messaging consume` with a configuration file from `/etc/fedora-messaging/` that matches the service name:

```
$ systemctl start fm-consumer@sample.service  # uses /etc/fedora-messaging/sample.toml
```

1.6 Messages

Before you release your application, you should create a subclass of *fedora_messaging.message.Message*, define a schema, define a default severity, and implement some methods.

1.6.1 Schema

Defining a message schema is important for several reasons.

First and foremost, if will help you (the developer) ensure you don't accidentally change your message's format. When messages are being generated from, say, a database object, it's easy to make a schema change to the database and unintentionally alter your message, which breaks consumers of your message. Without a schema, you might not catch this until you deploy your application and consumers start crashing. With a schema, you'll get an error as you develop!

Secondly, it allows you to change your message format in a controlled fashion by versioning your schema. You can then choose to implement methods one way or another based on the version of the schema used by a message.

Message schema are defined using JSON Schema. The complete API can be found in the *Message Schemas* API documentation.

Header Schema

The default header schema declares that the header field must be a JSON object with several expected keys. You can leave the schema as-is when you define your own message, or you can refine it. The base schema will always be enforced in addition to your custom schema.

Body Schema

The default body schema simply declares that the header field must be a JSON object.

Example Schema

```
# Copyright (C) 2018  Red Hat, Inc.
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License along
# with this program; if not, write to the Free Software Foundation, Inc.,
# 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
"""This is an example of a message schema."""

from fedora_messaging import message
from .utils import get_avatar


class BaseMessage(message.Message):
    """
    You should create a super class that each schema version inherits from.
    This lets consumers perform ``isinstance(msg, BaseMessage)`` if they are
    receiving multiple message types and allows the publisher to change the
    schema as long as they preserve the Python API.
    """

    def __str__(self):
        """Return a complete human-readable representation of the message."""
        return "Subject: {subj}\n{body}\n".format(
            subj=self.subject, body=self.email_body
        )

    @property
    def summary(self):
        """Return a summary of the message."""
        return self.subject

    @property
    def subject(self):
        """The email's subject."""
```

(continues on next page)

(continued from previous page)

```
        return 'Message did not implement "subject" property'

    @property
    def email_body(self):
        """The email message body."""
        return 'Message did not implement "email_body" property'

    @property
    def url(self):
        """An URL to the email in HyperKitty

        Returns:
            str or None: A relevant URL.
        """
        base_url = "https://lists.fedoraproject.org/archives"
        archived_at = self._get_archived_at()
        if archived_at and archived_at.startswith("<"):
            archived_at = archived_at[1:]
        if archived_at and archived_at.endswith(">"):
            archived_at = archived_at[:-1]
        if archived_at and archived_at.startswith("http"):
            return archived_at
        elif archived_at:
            return base_url + archived_at
        else:
            return None

    @property
    def app_icon(self):
        """An URL to the icon of the application that generated the message."""
        return "https://apps.fedoraproject.org/img/icons/hyperkitty.png"

    @property
    def usernames(self):
        """List of users affected by the action that generated this message."""
        return []

    @property
    def packages(self):
        """List of packages affected by the action that generated this message."""
        return []


class MessageV1(BaseMessage):
    """
    A sub-class of a Fedora message that defines a message schema for messages
    published by Mailman when it receives mail to send out.
    """

    body_schema = {
        "id": "http://fedoraproject.org/message-schema/mailman#",
        "$schema": "http://json-schema.org/draft-04/schema#",
        "description": "Schema for message sent to mailman",
        "type": "object",
        "properties": {
            "mlist": {
                "type": "object",
```

(continues on next page)

(continued from previous page)

```
                "properties": {
                    "list_name": {
                        "type": "string",
                        "description": "The name of the mailing list",
                    }
                },
            },
            "msg": {
                "description": "An object representing the email",
                "type": "object",
                "properties": {
                    "delivered-to": {"type": "string"},
                    "from": {"type": "string"},
                    "cc": {"type": "string"},
                    "to": {"type": "string"},
                    "x-mailman-rule-hits": {"type": "string"},
                    "x-mailman-rule-misses": {"type": "string"},
                    "x-message-id-hash": {"type": "string"},
                    "references": {"type": "string"},
                    "in-reply-to": {"type": "string"},
                    "message-id": {"type": "string"},
                    "archived-at": {"type": "string"},
                    "subject": {"type": "string"},
                    "body": {"type": "string"},
                },
                "required": ["from", "to", "subject", "body"],
            },
        },
        "required": ["mlist", "msg"],
    }

    @property
    def subject(self):
        """The email's subject."""
        return self.body["msg"]["subject"]

    @property
    def email_body(self):
        """The email message body."""
        return self.body["msg"]["body"]

    @property
    def agent_avatar(self):
        """An URL to the avatar of the user who caused the action."""
        from_header = self.body["msg"]["from"]
        return get_avatar(from_header)

    def _get_archived_at(self):
        return self.body["msg"]["archived-at"]


class MessageV2(BaseMessage):
    """
    This is a revision from the MessageV1 schema which flattens the message
    structure into a single object, but is backwards compatible for any users
    that make use of the properties (``subject`` and ``body``).
    """
```

(continues on next page)

(continued from previous page)

```
    body_schema = {
        "id": "http://fedoraproject.org/message-schema/mailman#",
        "$schema": "http://json-schema.org/draft-04/schema#",
        "description": "Schema for message sent to mailman",
        "type": "object",
        "required": ["mailing_list", "from", "to", "subject", "body"],
        "properties": {
            "mailing_list": {
                "type": "string",
                "description": "The name of the mailing list",
            },
            "delivered-to": {"type": "string"},
            "from": {"type": "string"},
            "cc": {"type": "string"},
            "to": {"type": "string"},
            "x-mailman-rule-hits": {"type": "string"},
            "x-mailman-rule-misses": {"type": "string"},
            "x-message-id-hash": {"type": "string"},
            "references": {"type": "string"},
            "in-reply-to": {"type": "string"},
            "message-id": {"type": "string"},
            "archived-at": {"type": "string"},
            "subject": {"type": "string"},
            "body": {"type": "string"},
        },
    }

    @property
    def subject(self):
        """The email's subject."""
        return self.body["subject"]

    @property
    def email_body(self):
        """The email message body."""
        return self.body["body"]

    @property
    def agent_avatar(self):
        """An URL to the avatar of the user who caused the action."""
        from_header = self.body["from"]
        return get_avatar(from_header)

    def _get_archived_at(self):
        return self.body["archived-at"]
```

Note that message schema can be composed of other message schema, and validation of fields can be much more detailed than just a simple type check. Consult the JSON Schema documentation for complete details.

1.6.2 Message Conventions

Schema are Immutable

Message schema should be treated as immutable. Once defined, they should not be altered. Instead, define a new schema class, mark the old one as deprecated, and remove it after an appropriate transition period.

Provide Accessors

The JSON schema ensures the message sent "on the wire" conforms to a particular format. Messages should provide Python properties to access the deserialized JSON object. This Python API should maintain backwards compatibility between schema. This shields consumers from changes in schema.

1.6.3 Packaging

Finally, you must distribute your schema to clients. It is recommended that you maintain your message schema in your application's git repository in a separate Python package. The package name should be `<your-app-name>_schema`.

A complete sample schema package can be found in the fedora-messaging repository. This includes unit tests, the schema classes, and a setup.py. You can adapt this boilerplate with the following steps:

- Change the package name from `mailman_schema` to `<your-app-name>_schema` in `setup.py`.
- Rename the `mailman_schema` directory to `<your-app-name>_schema`.
- Add your schema classes to `schema.py` and tests to `tests/test_schema.py`.
- Update the `README` file.
- Build the distribution with `python setup.py sdist bdist_wheel`.
- Upload the sdist and wheel to PyPI with `twine`.
- Submit an RPM package for it to Fedora and EPEL.

1.7 Testing

Once you've written code to publish or consume messages, you'll probably want to test it. The *fedora_messaging.testing* module has utilities for common test patterns.

If you find yourself implementing a pattern over and over in your test code, consider contributing it here!

fedora_messaging.testing.**mock_sends**(**expected_messages*)

Assert a block of code results in the provided messages being sent without actually sending them.

This is intended for unit tests. The call to publish is mocked out and messages are captured and checked at the end of the `with`.

For example:

```
>>> from fedora_messaging import api, testing
>>> def publishes():
...     api.publish(api.Message(body={"Hello": "world"}))
...
>>> with testing.mock_sends(api.Message, api.Message(body={"Hello": "world"})):
...     publishes()
...     publishes()
...
```

(continues on next page)

(continued from previous page)

```
>>> with testing.mock_sends(api.Message(body={"Goodbye": "everybody"})):
...     publishes()
...
AssertionError
```

Parameters ***expected_messages** – The messages you expect to be sent. These can be classes instances of classes derived from *fedora_messaging.message.Message*. If the class is provided, the message is checked to make sure it is an instance of that class and that it passes schema validation. If an instance is provided, it is checked for equality with the sent message.

Raises AssertionError – If the messages published don't match the messages asserted.

1.8 Release Notes

1.8.1 1.7.2 (2019-08-02)

Bug Fixes

- Fix variable substitution in log messages. (PR#200)
- Add MANIFEST.in and include tests for sample schema package. (PR#197)

Documentation Improvements

- Document the sent-at header in messages. (PR#199)
- Create a quick-start guide. (PR#196)

Contributors

Many thanks to the contributors of bug reports, pull requests, and pull request reviews for this release:

- Adam Williamson
- Aurélien Bompard
- Jeremy Cline
- Shraddha Agrawal

1.8.2 v1.7.1 (2019-06-24)

Bug Fixes

- Don't declare exchanges when consuming using the synchronous *fedora_messaging.api.consume()* API, which was causing consuming to fail from the Fedora broker (PR#191)

Contributors

Many thanks to the contributors of bug reports, pull requests, and pull request reviews for this release:

- Randy Barlow
- Aurélien Bompard
- Jeremy Cline
- Adam Williamson

Documentation Improvements

- Document some additional app properties and add a note about setting up logging in the fedora.toml and stg.fedora.toml configuration files (PR#188)
- Document how to setup logging in the consuming snippets so any problems are logged to stdout (PR#192)
- Document that logging is only set up for consumers (#181)
- Document the *fedora_messaging.config.conf* and *fedora_messaging.config.DEFAULTS* variables in the API documentation (#182)

1.8.3 v1.7.0 (2019-05-21)

Features

- "fedora-messaging consume" now accepts a "–callback-file" argument which will load a callback function from an arbitrary Python file. Previously, it was required that the callback be in the Python path (#159).

Bug Fixes

- Fix a bug where publishes that failed due to certain connection errors were not retried (#175).
- Fix a bug where AMQP protocol errors did not reset the connection used for publishing messages. This would result in publishes always failing with a ConnectionError (#178).

Documentation Improvements

- Document the `body` attribute on the `Message` class (#164).
- Clearly document what properties message schema classes should override (#166).
- Re-organize the consumer documentation to make the consuming API clearer (#168).

Contributors

Many thanks to the contributors of bug reports, pull requests, and pull request reviews for this release:

- Randy Barlow
- Aurélien Bompard
- Jeremy Cline
- Dusty Mabe

1.8.4 v1.6.1 (2019-04-17)

Bug Fixes

- Fix a bug in publishing where if the broker closed the connection, the client would not properly dispose of the connection object and publishing would fail forever (PR#157).
- Fix a bug in the `fedora_messaging.api.twisted_consume()` function where if the user did not have permissions to read from the specified queue which had already been declared, the Deferred that was returned never fired. It now errors back with a `fedora_messaging.exceptions.PermissionException` (PR#160).

Development Changes

- Stop pinning pytest to 4.0 or less as the incompatibility with pytest-twisted has been resolved (PR#158).

Other Changes

- Include commands to connect to the Fedora broker in the documentation (PR#154).

Contributors

Many thanks to the contributors of bug reports, pull requests, and pull request reviews for this release:

- Aurélien Bompard
- Jeremy Cline

1.8.5 v1.6.0 (2019-04-04)

Dependency Changes

- Twisted is no longer an optional dependency: fedora-messaging requires Twisted 12.2 or greater.

Features

- A new API, `fedora_messaging.api.twisted_consume()`, has been added to support consuming using the popular async framework Twisted. The fedora-messaging command-line interface has been switched to use this API. As a result, Twisted 12.2+ is now a dependency of fedora-messsaging. Users of this new API are not affected by Issue #130 (PR#139).

Bug Fixes

- Only prepend the topic_prefix on outgoing messages. Previously, the topic prefix was incorrectly applied to incoming messages (#143).

Documentation

- Add a note to the tutorial on how to instal the library and RabbitMQ in containers (PR#141).
- Document how to access the Fedora message broker from outside the Fedora infrastructure VPN. Users of fedmsg can now migrate to fedora-messaging for consumers outside Fedora's infrastructure. Consult the new documentation at *Fedora's Public Broker* for details (PR#149).

Contributors

Many thanks to the contributors of bug reports, pull requests, and pull request reviews for this release:

- Aurélien Bompard
- Jeremy Cline
- Shraddha Agrawal

1.8.6 v1.5.0 (2019-02-28)

Dependency Changes

- Replace the dependency on `pytoml` with `toml` (#132).

Features

- Support passive declarations for locked-down brokers (#136).

Bug Fixes

- Fix a bug in the sample schema pachage (#135).

Development Changes

- Switch to Mergify v2 (#129).

Contributors

Many thanks to the contributors of bug reports, pull requests, and pull request reviews for this release:

- Aurélien Bompard
- Jeremy Cline
- Michal Konečný
- Shraddha Agrawal

1.8.7 v1.4.0 (2019-02-07)

Features

- The `topic_prefix` configuration value has been added to automatically add a prefix to the topic of all outgoing messages. (#121)
- Support for Pika 0.13. (#126)
- Add a systemd service file for consumers.

Development Changes

- Use Bandit for security checking.

Contributors

Many thanks to the contributors of bug reports, pull requests, and pull request reviews for this release:

- Aurélien Bompard

1.8.8 v1.3.0 (2019-01-24)

API Changes

- The `Message._body` attribute is renamed to `body`, and is now part of the public API. (PR#119)

Contributors

Many thanks to the contributors of bug reports, pull requests, and pull request reviews for this release:

- Aurélien Bompard
- Jeremy Cline

1.8.9 v1.2.0 (2019-01-21)

Features

- The *`fedora_messaging.api.consume()`* API now accepts a "queues" keyword which specifies the queues to declare and consume from, and the "fedora-messaging" CLI makes use of this (PR#107)
- Utilities were added in the `schema_utils` module to help write the Python API of your message schemas (PR#108)
- No long require "–exchange", "–queue-name", and "–routing-key" to all be specified when using "fedora-messaging consume". If one is not supplied, a default is chosen. These defaults are documented in the command's manual page (PR#117)

Bug Fixes

- Fix the "consumer" setting in config.toml.example to point to a real Python path (PR#104)
- fedora-messaging consume now actually uses the –queue-name and –routing-key parameter provided to it, and –routing-key can now be specified multiple times as was documented (PR#105)
- Fix the equality check on *fedora_messaging.message.Message* objects to exclude the 'sent-at' header (PR#109)
- Documentation for consumers indicated any callable object was acceptable to use as a callback as long as it accepted a single positional argument (the message). However, the implementation required that the callable be a function or a class, which it then instantiated. This has been fixed and you may now use any callable object, such as a method or an instance of a class that implements `__call__` (PR#110)
- Fix an issue where the fedora-messaging CLI would only log if a configuration file was explicitly supplied (PR#113)

Contributors

Many thanks to the contributors of bug reports, pull requests, and pull request reviews for this release:

- Aurélien Bompard
- Jeremy Cline
- Sebastian Wojciechowski
- Tomas Tomecek

1.8.10 v1.1.0 (2018-11-13)

Features

- Initial work on a serialization format for *fedora_messaging.message.Message* and APIs for loading and storing messages. This is intended to make it easy to record and replay messages for testing purposes. (#84)
- Add a module, *fedora_messaging.testing*, to add useful test helpers. Check out the module documentation for details! (#100)

Contributors

Many thanks to the contributors of bug reports, pull requests, and pull request reviews for this release:

- Jeremy Cline
- Sebastian Wojciechowski

1.8.11 v1.0.1 (2018-10-10)

Bug Fixes

- Fix a compatibility issue in Twisted between pika 0.12 and 1.0. (#97)

1.8.12 v1.0.0 (2018-10-10)

API Changes

- The unused `exchange` parameter from the PublisherSession was removed (PR#56)
- The `setupRead` API in the Twisted protocol has been removed and replaced with `consume` and `cancel` APIs which allow for multiple consumers with multiple callbacks (PR#72)
- The name of the entry point is now used to identify the message type (PR#89)

Features

- Ensure proper TLS client cert checking with `service_identity` (PR#51)
- Support Python 3.7 (PR#53)
- Compatibility with Click 7.x (PR#86)
- The complete set of valid severity levels is now available at `fedora_messaging.api.SEVERITIES` (PR#60)
- A `queue` attribute is present on received messages with the name of the queue it arrived on (PR#65)
- The wire format of fedora-messaging is now documented (PR#88)

Development Changes

- Use towncrier to generate the release notes (PR#67)
- Check that our dependencies have Free licenses (PR#68)
- Test coverage is now at 97%.

Other Changes

- The library is available in Fedora as `fedora-messaging`.

Contributors

Many thanks to the contributors of bug reports, pull requests, and pull request reviews for this release:

- Aurélien Bompard
- Jeremy Cline
- Michal Konečný
- Sebastian Wojciechowski

1.8.13 v1.0.0b1

API Changes

- *`fedora_messaging.message.Message.summary`* is now a property rather than a method (#25).
- The non-functional `--amqp-url` parameter has been removed from the CLI (#49).

Features

- Configuration parsing failures now produce point to the line and column of the parsing error (#21).
- *fedora_messaging.message.Message* now come with a set of standard accessors (#32).
- Consumers can now specify whether a message should be re-queued when halting (#44).
- An example consumer that prints to standard output now ships with fedora-messaging. It can be used by running `fedora-messaging consume --callback="fedora_messaging.example:printer"` (#40).
- *fedora_messaging.message.Message* now have a `severity` associated with them (#48).

Bug Fixes

- Fix an issue where invalid or missing configuration files resulted in a traceback rather than a formatted error message from the CLI (#21).
- Client authentication with x509 now works with both the synchronous API and the Twisted API (#29, #35).
- *fedora_messaging.api.publish()* no longer raises a `pika.exceptions.ChannelClosed` exception. Instead, it raises a *fedora_messaging.exceptions.ConnectionException* (#31).
- *fedora_messaging.api.consume()* is now documented to raise a `ValueError` when the callback isn't callable (#47).

Development Features

- The fedora-messaging code base is now compliant with the Black Python formatter and this is enforced with continuous integration.
- Test coverage is moving up and to the right.

Many thanks to the contributors of bug reports, pull requests, and pull request reviews for this release:

- Aurélien Bompard
- Clement Verna
- Ken Dreyer
- Jeremy Cline
- Miroslav Suchý
- Patrick Uiterwijk
- Sebastian Wojciechowski

1.8.14 v1.0.0a1

The initial alpha release for fedora-messaging v1.0.0. The API is not expected to change significantly between this release and the final v1.0.0 release, but it may do so if serious flaws are discovered in it.

1.9 Command Line Interface Manuals

1.9.1 fedora-messaging

Synopsis

`fedora-messaging` COMMAND [OPTIONS] [ARGS]...

Description

`fedora-messaging` can be used to work with AMQP message brokers using the `fedora-messaging` library to start message consumers.

Options

`--help`

Show help text and exit.

`--conf`

Path to a valid configuration file to use in place of the configuration in `/etc/fedora-messaging/config.toml`.

Commands

There is a single sub-command, `consume`, described in detail in its ow section below.

`fedora-messaging consume [OPTIONS]`

Starts a consumer process with a user-provided callback function to execute when a message arrives.

consume

All options below correspond to settings in the configuration file. However, not all available configuration keys can be overridden with options, so it is recommended that for complex setups and production environments you use the configuration file and no options on the command line.

`--app-name`

The name of the application, used by the AMQP client to identify itself to the broker. This is purely for administrator convenience to determine what applications are connected and own particular resources.

This option is equivalent to the `app` setting in the `client_properties` section of the configuration file.

`--callback`

The Python path to the callable object to execute when a message arrives. The Python path should be in the format `module.path:object_in_module` and should point to either a function or a class. Consult the API documentation for the interface required for these objects.

This option is equivalent to the `callback` setting in the configuration file.

`--routing-key`

The AMQP routing key to use with the queue. This controls what messages are delivered to the consumer. Can be specified multiple times; any message that matches at least one will be placed in the message queue.

Setting this option is equivalent to setting the `routing_keys` setting in *all* `bindings` entries in the configuration file.

`--queue-name`

The name of the message queue in AMQP. Can contain ASCII letters, digits, hyphen, underscore, period, or colon. If one is not specified, a unique name will be created for you.

Setting this option is equivalent to setting the `queue` setting in *all* `bindings` entries and creating a `queue.<queue-name>` section in the configuration file.

`--exchange`

The name of the exchange to bind the queue to. Can contain ASCII letters, digits, hyphen, underscore, period, or colon. If one is not specified, the default is the `amq.topic` exchange.

Setting this option is equivalent to setting the `exchange` setting in *all* `bindings` entries in the configuration file.

Exit codes

consume

The `consume` command can exit for a number of reasons:

`0`

The consumer intentionally halted by raising a `HaltConsumer` exception.

`2`

The argument or option provided is invalid.

`10`

The consumer was unable to declare an exchange, queue, or binding in the message broker. This occurs with the user does not have permission on the broker to create the object *or* the object already exists, but does not have the attributes the consumer expects (e.g. the consumer expects it to be a durable queue, but it is transient).

`11`

The consumer encounters an unexpected error while registering the consumer with the broker. This is a bug in fedora-messaging and should be reported.

`12`

The consumer is canceled by the message broker. The consumer is typically canceled when the queue it is subscribed to is deleted on the broker, but other exceptional cases could result in this. The broker administrators should be consulted in this case.

`13`

An unexpected general exception is raised by your consumer callback.

Additionally, consumer callbacks can cause the command to exit with a custom exit code. Consult the consumer's documentation to see what error codes it uses.

Signals

consume

The `consume` command handles the SIGTERM and SIGINT signals by allowing any consumers which are currently processing a message to finish, acknowledging the message to the message broker, and then shutting down. Repeated SIGTERM or SIGINT signals are ignored. To halt immediately, send the SIGKILL signal; messages that are partially processed will be re-delivered when the consumer restarts.

Systemd service

The `consume` subcommand can be started as a system service, and Fedora Messaging provides a dynamic systemd service file.

First, create a valid Fedora Messaging configuration file in `/etc/fedora-messaging/foo.toml`, with the `callback` parameter pointing to your consuming function or class. Remember that you can use the `consumer_config` section for your own configuration.

Enable and start the service in systemd with the following commands:

```
systemctl enable fm-consumer@foo.service
systemctl start fm-consumer@foo.service
```

The service name after the `@` and before the `.service` must match your filename in `/etc/fedora-messaging` (without the `.toml` suffix).

Help

If you find bugs in fedora-messaging or its man page, please file a bug report or a pull request:

```
https://github.com/fedora-infra/fedora-messaging
```

Or, if you prefer, send an email to infrastructure@fedoraproject.org with bug reports or patches.

fedora-messaging's documentation is available online:

```
https://fedora-messaging.readthedocs.io/
```

CHAPTER 2

Tutorial

2.1 Using Fedora Messaging

This tutorial explains how to use the new fedora-messaging library.

2.1.1 Installation

Installing the library

Create a Python virtual environment:

```
mkdir fedora-messaging-tutorial
cd fedora-messaging-tutorial
mkvirtualenv -p python3 -a `pwd` fedora-messaging-tutorial
workon fedora-messaging-tutorial
```

Install the library and its dependencies:

```
pip install fedora-messaging
# Alternatively, install it directly from the git repository
git clone https://github.com/fedora-infra/fedora-messaging.git
cd fedora-messaging
pip install -e .
```

Make sure it is available and working:

```
fedora-messaging --help
```

Setting up RabbitMQ

Install RabbitMQ and start it:

```
dnf install rabbitmq-server
systemctl start rabbitmq-server
```

RabbitMQ has a web admin interface that you can access at: http://localhost:15672/. The username is `guest` and the password is `guest`. This interface lets you change the configuration, send messages and read the messages in the queues. Keep it open in a browser tab, we'll need it later.

If your project uses containers, consult the RabbitMQ documentation about containers.

Configuration

An example of the library configuration file is provided in the `config.toml.example` file. Copy that file to `/etc/fedora-messaging/config.toml` to make it available system-wide. Alternatively, you can copy it to `config.toml` anywhere and set the `FEDORA_MESSAGING_CONF` environement variable to that file's path.

Refer to the documentation for a complete description of the configuration options.

Comment out the `callback` and `bindings` options, and all the `[exchanges.custom_exchange]` and `[queues.my_queue]` sections.

In the `[client_properties]` section, change the `app` value to `Fedora Messaging tutorial`.

2.1.2 Using the API

We will be creating some scripts to publish and subscribe to the bus. First, create a directory to hold the code you will write, than change to this directory.

Publishing

To publish on the Fedora Messaging bus, you just need to use the *`fedora_messaging.api.publish()`* function, passing it an instance of the *`fedora_messaging.message.Message`* class that represents the message you want to publish.

A message has a schema, a topic, a severity, a body, and a set of headers. We'll cover the schema later in this tutorial. The headers and the body are Python dictionaries with JSON-serializable values. The topic is a string containing elements separated by dots that will be used to route messages.

Create a publishing script called `publish.py`:

```
#!/usr/bin/env python3

from fedora_messaging.api import publish, Message
from fedora_messaging.config import conf

conf.setup_logging()
message = Message(
    topic="tutorial.topic",
    body={"reason": "test message"}
)
publish(message)
```

Of course, you can make a smarter script that will use command-line arguments, this is left as an exercice to the reader. Now run it:

```
chmod +x publish.py
./publish.py
```

The script should complete without error. If you go to RabbitMQ's web interface, you'll see that a message has been sent to the `amq.topic` exchange. However, since noone is listening to this topic, the message has been discarded. Now, we'll setup listeners.

Listening

Clients listen on the Fedora Messaging bus by subscribing to a topic or a topic pattern using the hash (`#`) symbol as a wildcard. For exemple you can subscribe to `bodhi.updates.kernel` but also to `bodhi.updates.#`. In the former case you'll get kernel updates, in the latter case you'll get all Bodhi updates.

After subscription, all messages with a topic matching the pattern will be routed to a queue on the server, and clients will consume messages from this queue. In the AMQP language, this is called *binding* a queue to an exchange, and the topic pattern is called the *routing_key*.

In the configuration file, the `bindings` section controls which queues will be subscribed to which topic patterns. Edit the file so the option looks like this:

```
[[bindings]]
queue = "tutorial"
exchange = "amq.topic"
routing_keys = ["tutorial.#"]
```

This means that the queue named `tutorial` will be created and subcribed to the `amq.topic` exchange using the `tutorial.#` pattern. All messages with a topic starting with `tutorial.` will end up in this queue, and no other.

Now configure this new queue's properties in the file using a snippet that looks like this:

```
[queues.tutorial]
durable = true
auto_delete = false
exclusive = false
arguments = {}
```

This means that messages in this queue will survive a client's disconnection and a server restart, and that multiple client can connect to it simultaneously to consume messages in a round-robin fashion.

Python script

Now create the following script, called `consume.py`:

```
#!/usr/bin/env python3

from fedora_messaging.api import consume
from fedora_messaging.config import conf

conf.setup_logging()

def print_message(message):
    print(message)

if __name__ == "__main__":
```

(continues on next page)

(continued from previous page)

```
    conf.setup_logging()
    consume(print_message)
```

The script should run and wait for new messages. Now run the `publish.py` script again in another terminal (remember to activate the virtualenv with `workon fedora-messaging-tutorial`). You should see the message being printed where the `consume.py` script is running.

Python callback

You can also just define the callback function and use the `fedora-messaging` command-line tool to do the listening:

```
fedora-messaging consume --callback="consume:print_message"
```

This should behave identically.

Round robin

When multiple programs are simulaneously consuming from the same queue, they get the messages in a round-robin fashion. Try running another instance of the `consume.py` script, and run the `publish.py` script multiple times. You'll see that `consume.py` instances get a message one after the other.

2.1.3 JSON schemas

Message bodies are JSON objects, that adhere to a schema. Message schemas live in their own Python package, so they can be installed on the producer and on the consumer.

In Fedora Messaging, we follow the JSON Schema standard, and use the jsonschema library.

Creating the schema package

Copy the `docs/sample_schema_package/` directory from the `fedora-messaging` git clone to your app directory.

Edit the `setup.py` file to change the package metadata. Rename the `mailman_schema` directory to something relevant to your app, like `yourapp_message_schemas`. There is no naming convention at the moment. Edit the `README` file too.

Writing the schema

JSON objects are converted to dictionaries in Python. Writing a JSON schema with the jsonschema library means writing a Python dictionary that will describe the message's JSON object body. Read up on the jsonschema library documentation if you have questions about the format.

Open the `schema.py` file, it contains an example schema for Mailman-originating messages on the bus. The schema is a Python class containing an important dictionary attribute: `body_schema`. This is where the JSON schema lives.

For clarity, edit the `setup.py` file and in the entry points list change the `mailman.messageV1` name to something more relevant to your app, like `yourapp.my_messageV1`. The entry point name needs to be unique to your application, so it's best to prefix it with your package or application name.

Schema format

This dictionary describes the possible keys and types in the JSON object being validated, using the following reserved keys:

- `id` (or `$id`): an URI identifing this schema. Change the last part of the example URL to use your app's name.
- `$schema`: an URI describing the validator to use, you can leave that one as it is. It is only present at the root of the dictionary.
- `description`: a fulltext description of the key.
- `type`: the value type for this key. You can choose among: - `null`: equivalent to `None` - `boolean`: equivalent to `True` or `False` - `object`: a Python dictionary - `array`: a Python list - `number`: an `int` or a `float` - `string`: a Python string
- `properties`: a dictionary describing the possible keys contained in the JSON object, where keys are possible key names, and values are JSON schemas. Those schemas can also have `properties` keys to describe all the possible nested keys.
- `required`: a list of keys that must be present in the JSON object.
- `format`: a format validation type. You can choose among: - hostname - ipv4 - ipv6 - email - uri (requires the `rfc3987` package) - date - time - date-time (requires the `strict-rfc3339` package) - regex - color (requires the `webcolors` package)

For information on creating JSON schemas to validate your data, there is a good introduction to JSON Schema fundamentals underway at Understanding JSON Schema.

Example

Now edit the `body_schema` key to use the following schema:

```
{
    'id': 'http://fedoraproject.org/message-schema/fedora-messaging-tutorial#',
    '$schema': 'http://json-schema.org/draft-04/schema#',
    'description': 'Schema for the Fedora Messaging tutorial',
    'type': 'object',
    'properties': {
        'package': {
            'type': 'object',
            'properties': {
                'name': {
                    'type': 'string',
                    'description': 'The name of the package',
                },
                'version': {'type': 'string'},
            }
            'required': ['name'],
        },
        'owner': {
            'description': 'The owner of the package',
            'type': 'string',
        },
    },
    'required': ['package', 'owner'],
}
```

Human readable representation

The schema class also contains a few methods to extract relevant information from the message, or to create a human-readable representation.

Change the `__str__()` method to use the expected items from the message body. For example:

```
return '{owner} did something to the {package} package'.format(
    owner=self.body['owner'], package=self.body['package']['name'])
```

Also edit the `summary` property to return something relevant.

Severity

Messages can also have a severity level. This is used by consumers to determine the importance of a message to an end user. The possibly severity levels are defined in the *Message Severity* API documentation.

You should set a reasonable default for your messages.

Testing it

JSON schemas can also be unit-tested. Check out the `tests/test_schema.py` file and write the unit tests that are appropriate for the message schema and the methods you just wrote. Use the example tests for inspiration.

Using it

To use your new JSON schema, its Python distribution must be available on the system. Run `python setup.py develop` in the schema directory to install it.

Now you can use the `yourapp_message_schemas.schema.Message` class (or however you named the package) to construct your message instances and call `fedora_messaging.api.publish` on them. Edit the `publish.py` script to read:

```
#!/usr/bin/env python3

from fedora_messaging.api import publish
from fedora_messaging.config import conf
from yourapp_message_schema.schema import Message

conf.setup_logging()
message = Message(
    topic="tutorial.topic",
    body={
        "owner": "fedorauser",
        "package": {
            "name": "foobar",
            "version": "1.0",
        }
    }
)
publish(message)
```

Start a consumer, and send the message. Try to comment out the "owner" key and see what happens when you try to send a message that is not valid according to the schema.

Updating it

Message formats can change over time, and the schema must change to reflect that. When that happens, you need to copy the old class to a new class in the schemas package, make the changes you need to do, and import the new one in your publisher. You must also add a new entry in the `entry_points` argument in the schema package's `setup.py` file. The name of the entry point is currently unused, only the class path matters.

However, be warned that messages published with the new class may be dropped by the receivers if they don't have the new schema available locally. Therefore, you should publish the schema package with the new schema, update it on all the receivers, restart them, and then start using the new version in the publishers.

You should keep the old schema versions in the schemas package for a reasonable amount of time, long enough to make sure all receivers are up-to-date. To avoid clutter, we recommend you use a separate module per schema version (`yourapp_message_schemas.v1:Message`, `yourapp_message_schemas.v2:Message`, etc)

Now create a new version and use it in the `publish.py` script. Send a message before restarting the `consume.py` script to see what happens when a message with an unknown schema is received. Now restart the `consume.py` script and re-send the message.

2.1.4 Handling exceptions

All exceptions are located in the *fedora_messaging.exceptions* module.

When publishing

When calling *fedora_messaging.api.publish()*, the following exceptions can be raised:

- `ValidationError`: raised if the message fails validation with its JSON schema. This only depends on the message you are trying to send, the AMQP server is not involved.
- `PublishReturned`: raised if the broker rejects the message.
- `ConnectionException`: raised if a connection error occurred before the publish confirmation arrived.

The `ValidationError` exception means you should fix either the schema (and maybe make a new version) or the message. No need to catch it, this should crash your app during development and testing.

Your app may handle the other two exceptions in whichever way is relevant. It should involve logging, and sending again or discarding may be valid options.

You already noticed the `ValidationError` being raised when you tried sending an invalid message in the previous chapter.

When consuming

Invalid messages according to the JSON schema are automatically rejected by the client.

The callback function can raise the following exceptions:

- `Nack`: raise this to return the message to the queue
- `Drop`: raise this to drop the message
- `HaltConsumer`: raise this to shutdown the consumer and return the message to the queue.

Any other exception will bubble up in the consumer as a `HaltConsumer` exception, shutdown the consumer, and return pending messages to the queue. Your app will have to handle the `HaltConsumer` exception.

Modify the callback function to raise those exceptions and see what happens.

When returning `Nack` systematically, the consumer will just loop on that one message, as it is put back in the queue and delivered again forever.

Notice how raising `HaltConsumer` or another exception stops the consumer, but does not consume the message: it will be re-delivered on the next startup.

2.1.5 Converting a fedmsg application

Converting publishers

Converting a Flask app

Let's use the elections app as an example. Clone the code using the following command:

```
git clone https://pagure.io/elections.git
```

And change to this directory.

In the `elections` app, all calls to publish messages on fedmsg are going through the `fedora_elections.fedmsgshim.publish` wrapper function. We can thus modify this function to make it call Fedora Messaging instead of fedmsg.

JSON schema

First, you will need a Message schema. To write this schema you must know what kind of messages are sent on the bus. A `git grep` command will reveal that all calls are made from the `admin.py` file. Open that file and examine those calls.

In parallel, copy the `docs/sample_schema_package/` directory from the `fedora-messaging` git clone to your app directory. Rename it to `elections-message-schemas`. Edit the `setup.py` file like you did before, to change the package metadata (including the entry point). Use `fedora_elections_message_schemas` for the name. Rename the `mailman_schema` directory to `fedora_elections_message_schemas` and adapt the `setup.py` metadata.

Edit the `schema.py` file and write the basic structure for the elections message schema. According to the different calls in `admin.py`, it could be something like:

```
{
    'id': 'http://fedoraproject.org/message-schema/elections#',
    '$schema': 'http://json-schema.org/draft-04/schema#',
    'description': 'Schema for Fedora Elections',
    'type': 'object',
    'properties': {
        'agent': {'type': 'string'},
        'election': {'type': 'object'},
        'candidate': {'type': 'object'},
    },
    'required': ['agent', 'election'],
}
```

This could be sufficient, but it would be best to list what properties are available in the `election` and `candidate` keys. Unfortunately, those are just JSON dumps of the database model, so you'll have to look further to know the structure.

Examining the `to_json()` methods in `models.py` shows which keys are dumped to JSON. The schema could be written as:

```
{
    'id': 'http://fedoraproject.org/message-schema/elections#',
    '$schema': 'http://json-schema.org/draft-04/schema#',
    'description': 'Schema for Fedora Elections',
    'type': 'object',
    'properties': {
        'agent': {'type': 'string'},
        'election': {
            'type': 'object',
            'properties': {
                'shortdesc': {'type': 'string'},
                'alias': {'type': 'string'},
                'description': {'type': 'string'},
                'url': {'type': 'string', 'format': 'uri'},
                'start_date': {'type': 'string'},
                'end_date': {'type': 'string'},
                'embargoed': {'type': 'number'},
                'voting_type': {'type': 'string'},
            },
            'required': [
                'shortdesc', 'alias', 'description', 'url',
                'start_date', 'end_date', 'embargoed', 'voting_type',
            ],
        },
        'candidate': {
            'type': 'object',
            'properties': {
                'name': {'type': 'string'},
                'url': {'type': 'string', 'format': 'uri'},
            },
            'required': ['name', 'url'],
        },
    },
    'required': ['agent', 'election'],
}
```

Use this schema and adapt the `__str__()` method and the `summary` property.

Since the schema is distributed in a separate python package, it must be added to the `election` app's dependencies in `requirements.txt`.

Wrapper function

Now you can import this class in `fedora_elections/fedmsgshim.py` and use it to encapsulate the messages. The wrapper could look like:

```
import logging

from fedora_elections_message_schemas.schema import Message
from fedora_messaging.api import publish as fm_publish
from fedora_messaging.exceptions import PublishReturned, ConnectionException

LOGGER = logging.getLogger(__name__)

def publish(topic, msg):
    try:
```

(continues on next page)

(continued from previous page)

```
        fm_publish(Message(
            topic="fedora.elections." + topic,
            body=msg,
        ))
    except PublishReturned as e:
        LOGGER.warning(
            "Fedora Messaging broker rejected message %s: %s",
            msg.id, e
        )
    except ConnectionException as e:
        LOGGER.warning("Error sending the message %s: %s", msg.id, e)
```

With this you'll get a couple of nice features over the previous state of things:

- the message format is validated, so it's your responsability to update the schema when you decide to change the format, and not the receiver's responsability to handle any database schema changes you may make that may bleed into the message dictionary. And you'll know during development if you break compatibility.
- you may handle messaging errors in anyway you deem relevant. Here we're just logging them but you could choose to re-send the messages, store them for further analysis, etc.
- when there are no exceptions, you know that the message has reached the broker and has been distributed.

Testing

Let's start the election app and make sure messages are properly sent on the bus. First, we'll create a virtualenv, and install election and fedora-messaging with the following commands:

```
virtualenv venv
source ./venv/bin/activate
pushd elections-message-schemas
python setup.py develop
popd
pip install -r requirements.txt
python setup.py develop
```

Make sure the Fedora Messaging configuration file is correct in `/etc/fedora-messaging/config.toml`. We will add a queue binding to route messages with the `fedora.elections` topic to the `tutorial` queue. Add this entry in the `bindings` list:

```
[[bindings]]
queue = "tutorial"
exchange = "amq.topic"
routing_keys = ["fedora.elections.#"]
```

You could also add `"fedora.elections.#"` to the `"routing_keys"` value in the existing entry.

Now make sure that RabbitMQ is still running, and run the `consume.py` script *we used before*. Make sure it is not systematically raising exceptions in the callback function (as we did before).

Now we'll run the election app, but first we need to create a configuration file. Create a file called `config.py` with the following content:

```
FEDORA_ELECTIONS_ADMIN_GROUP = ""
```

This will allow any Fedora account to be an admin on your instance, which is good enough for this tutorial. Now start the app with:

```
python createdb.py
python runserver.py -c config.py
```

Open your browser to http://localhost:5000/admin/new. Login with FAS, then create an election. Check the terminal where the `consume.py` script is running. You should see the message that the `elections` app has sent on election creation. Edit the election, and you should see the corresponding message in the terminal where `consume.py` is running.

Converting a Pyramid app

Let's use the github2fedmsg app as an example. It is a Pyramid webapp that registers a webhook with Github on all subscribed projects, and then broadcasts actions (commits, pull-request, tickets) received on this webhook to the message bus.

Clone the code using the following command:

```
git clone git@github.com:fedora-infra/github2fedmsg.git
```

And change to this directory.

JSON Schema

The only call to fedmsg is in `github2fedmsg/views/webhooks.py`. Since the app transmits the webhook payload almost transparently to the message bus, the structure isn't obvious, so it's harder to define a schema. Fortunately, the Github documentation has a comprehensive list of payload formats.

It would be to long to define precise JSON schemas for each event type, so we'll just use the generic schema.

Sending the messages

Now you can replace the current call to fedmsg with a call to `fedora_messaging.api.publish`. Add these lines in the `github2fedmsg.views.webhook` module:

```
import logging
from fedora_messaging.api import Message, publish
from fedora_messaging.exceptions import PublishReturned, ConnectionException

LOGGER = logging.getLogger(__name__)
```

And replace the call to `fedmsg.publish` with:

```
try:
    msg = Message(
        topic="github." + event_type,
        body=payload,
    )
    publish(msg)
except PublishReturned as e:
    LOGGER.warning(
        "Fedora Messaging broker rejected message %s: %s",
        msg.id, e
    )
```

(continues on next page)

(continued from previous page)

```
except ConnectionException as e:
    LOGGER.warning("Error sending message %s: %s", msg.id, e)
```

Testing it

Make sure the Fedora Messaging configuration file is correct in `/etc/fedora-messaging/config.toml`. We will add a queue binding to route messages with the `github` topic to the `tutorial` queue. Add this entry in the `bindings` list:

```
[[bindings]]
queue = "tutorial"
exchange = "amq.topic"
routing_keys = ["github.#"]
```

You could also add `"github.#"` to the `"routing_keys"` value in the existing entry.

Now make sure that RabbitMQ is still running, and run the `consume.py` script *we used before*. Make sure it is not systematically raising exceptions in the callback function (as we did before).

To setup the `github2fedmsg` application, follow the `README.rst` file:

```
virtualenv venv
source ./venv/bin/activate
python setup.py develop
pip install waitress
```

Go off and register your development application with GitHub. Save the oauth tokens and add the secret one to a new file you create called `secret.ini`. Use the example `secret.ini.example` file.

Create the database and start the application:

```
initialize_github2fedmsg_db development.ini
pserve development.ini --reload
```

Converting consumers

TODO the-new-hotness

CHAPTER 3

API Documentation

3.1 Developer Interface

This documentation covers the public interfaces fedora_messaging provides.

Note: Documented interfaces follow Semantic Versioning 2.0.0. Any interface not documented here may change at any time without warning.

API Table of Contents

3.1.1 Publishing

publish

fedora_messaging.api.**publish**(*message*, *exchange=None*)

Publish a message to an exchange.

This is a synchronous call, meaning that when this function returns, an acknowledgment has been received from the message broker and you can be certain the message was published successfully.

There are some cases where an error occurs despite your message being successfully published. For example, if a network partition occurs after the message is received by the broker. Therefore, you may publish duplicate messages. For complete details, see the *Publishing* documentation.

```
>>> from fedora_messaging import api
>>> message = api.Message(body={'Hello': 'world'}, topic='Hi')
>>> api.publish(message)
```

If an attempt to publish fails because the broker rejects the message, it is not retried. Connection attempts to the broker can be configured using the "connection_attempts" and "retry_delay" options in the broker URL. See `pika.connection.URLParameters` for details.

Parameters

- **message** (`message.Message`) – The message to publish.
- **exchange** (*str*) – The name of the AMQP exchange to publish to; defaults to *publish_exchange*

Raises

- *fedora_messaging.exceptions.PublishReturned* – Raised if the broker rejects the message.

- *fedora_messaging.exceptions.ConnectionException* – Raised if a connection error occurred before the publish confirmation arrived.
- *fedora_messaging.exceptions.ValidationError* – Raised if the message fails validation with its JSON schema. This only depends on the message you are trying to send, the AMQP server is not involved.

3.1.2 Subscribing

twisted_consume

fedora_messaging.api.**twisted_consume**(*callback*, *bindings=None*, *queues=None*)

Start a consumer using the provided callback and run it using the Twisted event loop (reactor).

Note: Callbacks run in a Twisted-managed thread pool using the `twisted.internet.threads.deferToThread()` API to avoid them blocking the event loop. If you wish to use Twisted APIs in your callback you must use the `twisted.internet.threads.blockingCallFromThread()` or `twisted.internet.interfaces.IReactorFromThreads` APIs.

This API expects the caller to start the reactor.

Parameters

- **callback** (*callable*) – A callable object that accepts one positional argument, a *Message* or a class object that implements the `__call__` method. The class will be instantiated before use.
- **bindings** (*dict or list of dict*) – Bindings to declare before consuming. This should be the same format as the *bindings* configuration.
- **queues** (*dict*) – The queue to declare and consume from. Each key in this dictionary should be a queue name to declare, and each value should be a dictionary with the "durable", "auto_delete", "exclusive", and "arguments" keys.

Returns A deferred that fires with the list of one or more *Consumer* objects. Each consumer object has a *Consumer.result* instance variable that is a Deferred that fires or errors when the consumer halts. Note that this API is meant to survive network problems, so consuming will continue until *Consumer.cancel()* is called or a fatal server error occurs. The deferred returned by this function may error back with a *fedora_messaging.exceptions.BadDeclaration* if queues or bindings cannot be declared on the broker, a *fedora_messaging.exceptions.PermissionException* if the user doesn't have access to the queue, or *fedora_messaging.exceptions.ConnectionException* if the TLS or AMQP handshake fails.

Return type twisted.internet.defer.Deferred

Consumer

class fedora_messaging.api.**Consumer**(*queue=None*, *callback=None*)

Represents a Twisted AMQP consumer and is returned from the call to *fedora_messaging.api.twisted_consume()*.

queue

The AMQP queue this consumer is subscribed to.

Type str

callback
The callback to run when a message arrives.

Type callable

result
A deferred that runs the callbacks if the consumer exits gracefully after being canceled by a call to *Consumer.cancel()* and errbacks if the consumer stops for any other reason. The reasons a consumer could stop are: a `fedora_messaging.exceptions.PermissionExecption` if the consumer does not have permissions to read from the queue it is subscribed to, a *HaltConsumer* is raised by the consumer indicating it wishes to halt, an unexpected `Exception` is raised by the consumer, or if the consumer is canceled by the server which happens if the queue is deleted by an administrator or if the node the queue lives on fails.

Type twisted.internet.defer.Deferred

cancel()
Cancel the consumer and clean up resources associated with it. Consumers that are canceled are allowed to finish processing any messages before halting.

Returns A deferred that fires when the consumer has finished processing any message it was in the middle of and has been successfully canceled.

Return type defer.Deferred

consume

fedora_messaging.api.**consume**(*callback*, *bindings=None*, *queues=None*)
Start a message consumer that executes the provided callback when messages are received.

This API is blocking and will not return until the process receives a signal from the operating system.

> **Warning:** This API is runs the callback in the IO loop thread. This means if your callback could run for a length of time near the heartbeat interval, which is likely on the order of 60 seconds, the broker will kill the TCP connection and the message will be re-delivered on start-up.
>
> For now, use the *twisted_consume()* API which runs the callback in a thread and continues to handle AMQP events while the callback runs if you have a long-running callback.

The callback receives a single positional argument, the message:

```
>>> from fedora_messaging import api
>>> def my_callback(message):
...     print(message)
>>> bindings = [{'exchange': 'amq.topic', 'queue': 'demo', 'routing_keys': ['#']}]
>>> queues = {
...     "demo": {"durable": False, "auto_delete": True, "exclusive": True,
↪"arguments": {}}
... }
>>> api.consume(my_callback, bindings=bindings, queues=queues)
```

If the bindings and queue arguments are not provided, they will be loaded from the configuration.

For complete documentation on writing consumers, see the *Consumers* documentation.

Parameters

- **callback** (*callable*) – A callable object that accepts one positional argument, a `Message` or a class object that implements the `__call__` method. The class will be instantiated before use.
- **bindings** (*dict or list of dict*) – Bindings to declare before consuming. This should be the same format as the *bindings* configuration.
- **queues** (*dict*) – The queue or queues to declare and consume from. This should be in the same format as the *queues* configuration dictionary where each key is a queue name and each value is a dictionary of settings for that queue.

Raises

- *fedora_messaging.exceptions.HaltConsumer* – If the consumer requests that it be stopped.
- `ValueError` – If the consumer provide callback that is not a class that implements __call__ and is not a function, if the bindings argument is not a dict or list of dicts with the proper keys, or if the queues argument isn't a dict with the proper keys.

3.1.3 Signals

Signals sent by fedora_messaging APIs using `blinker.base.Signal` signals.

pre_publish_signal

fedora_messaging.api.**pre_publish_signal = <blinker.base.NamedSignal object at 0x7f29b234d9**
A signal triggered before the message is published. The signal handler should accept a single keyword argument, `message`, which is the instance of the *fedora_messaging.message.Message* being sent. It is acceptable to mutate the message, but the `validate` method will be called on it after this signal.

publish_signal

fedora_messaging.api.**publish_signal = <blinker.base.NamedSignal object at 0x7f29b17147f0;**
A signal triggered after a message is published successfully. The signal handler should accept a single keyword argument, `message`, which is the instance of the *fedora_messaging.message.Message* that was sent.

publish_failed_signal

fedora_messaging.api.**publish_failed_signal = <blinker.base.NamedSignal object at 0x7f29b171**
A signal triggered after a message fails to publish for some reason. The signal handler should accept two keyword argument, `message`, which is the instance of the *fedora_messaging.message.Message* that failed to be sent, and `error`, the exception that was raised.

3.1.4 Message Schemas

This module defines the base class of message objects and keeps a registry of known message implementations. This registry is populated from Python entry points in the "fedora.messages" group.

To implement your own message schema, simply create a class that inherits the *Message* class, and add an entry point in your Python package under the "fedora.messages" group. For example, an entry point for the *Message* schema would be:

```
entry_points = {
    'fedora.messages': [
        'base.message=fedora_messaging.message:Message'
    ]
}
```

The entry point name must be unique to your application and is used to map messages to your message class, so it's best to prefix it with your application name (e.g. `bodhi.new_update_messageV1`). When publishing, the Fedora Messaging library will add a header with the entry point name of the class used so the consumer can locate the correct schema.

Since every client needs to have the message schema installed, you should define this class in a small Python package of its own.

Message

class fedora_messaging.message.**Message**(*body=None*, *headers=None*, *topic=None*, *properties=None*, *severity=None*)

Messages are simply JSON-encoded objects. This allows message authors to define a schema and implement Python methods to abstract the raw message from the user. This allows the schema to change and evolve without breaking the user-facing API.

There are a number of properties that are intended to be overridden by users. These fields are used to sort messages for notifications or are used to create human-readable versions of the messages. Properties that are intended for this purpose are noted in their attribute documentation below.

Parameters

- **headers** (*dict*) – A set of message headers. Consult the headers schema for expected keys and values.
- **body** (*dict*) – The message body. Consult the body schema for expected keys and values. This dictionary must be JSON-serializable by the default serializer.
- **topic** (*six.text_type*) – The message topic as a unicode string. If this is not provided, the default topic for the class is used. See the attribute documentation below for details.
- **properties** (*pika.BasicProperties*) – The AMQP properties. If this is not provided, they will be generated. Most users should not need to provide this, but it can be useful in testing scenarios.
- **severity** (*int*) – An integer that indicates the severity of the message. This is used to determine what messages to notify end users about and should be *DEBUG*, *INFO*, *WARNING*, or *ERROR*. The default is *INFO*, and can be set as a class attribute or on an instance-by-instance basis.

id

The message id as a unicode string. This attribute is automatically generated and set by the library and users should only set it themselves in testing scenarios.

Type six.text_type

topic

The message topic as a unicode string. The topic is used by message consumers to filter what messages they receive. Topics should be a string of words separated by '.' characters, with a length limit of 255 bytes. Because of this byte limit, it is best to avoid non-ASCII character. Topics should start general and get more specific each word. For example: "bodhi.update.kernel" is a possible topic. "bodhi" identifies the application, "update" identifies the message, and "kernel" identifies the package in the update. This

can be set at a class level or on a instance level. Dynamic, specific topics that allow for fine-grain filtering are preferred.

Type six.text_type

headers_schema
A JSON schema to be used with `jsonschema.validate()` to validate the message headers. For most users, the default definition should suffice.

Type dict

body_schema
A JSON schema to be used with `jsonschema.validate()` to validate the message body. The body_schema is retrieved on a message instance so it is not required to be a class attribute, although this is a convenient approach. Users are also free to write the JSON schema as a file and load the file from the filesystem or network if they prefer.

Type dict

body
The message body as a Python dictionary. This is validated by the body schema before publishing and before consuming.

Type dict

severity
An integer that indicates the severity of the message. This is used to determine what messages to notify end users about and should be *DEBUG*, *INFO*, *WARNING*, or *ERROR*. The default is *INFO*, and can be set as a class attribute or on an instance-by-instance basis.

Type int

queue
The name of the queue this message arrived through. This attribute is set automatically by the library and users should never set it themselves.

Type str

__str__()
A human-readable representation of this message.

This should provide a detailed, long-form representation of the message. The default implementation is to format the raw message id, topic, headers, and body.

Note: Sub-classes should override this method. It is used to create the body of email notifications and by other tools to display messages to humans.

---

agent_avatar
An URL to the avatar of the user who caused the action.

Note: Sub-classes should override this method if the message was triggered by a particular user.

Returns The URL to the user's avatar.

Return type str or None

app_icon
An URL to the icon of the application that generated the message.

Note: Sub-classes should override this method if their application has an icon and they wish that image to appear in applications that consume messages.

Returns The URL to the app's icon.

Return type str or None

containers
List of containers affected by the action that generated this message.

Note: Sub-classes should override this method if the message pertains to one or more container images. The data returned from this property is used to filter notifications.

Returns A list of affected container names.

Return type list(str)

flatpaks
List of flatpaks affected by the action that generated this message.

Note: Sub-classes should override this method if the message pertains to one or more flatpaks. The data returned from this property is used to filter notifications.

Returns A list of affected flatpaks names.

Return type list(str)

modules
List of modules affected by the action that generated this message.

Note: Sub-classes should override this method if the message pertains to one or more modules. The data returned from this property is used to filter notifications.

Returns A list of affected module names.

Return type list(str)

packages
List of RPM packages affected by the action that generated this message.

Note: Sub-classes should override this method if the message pertains to one or more RPM packages. The data returned from this property is used to filter notifications.

Returns A list of affected package names.

Return type list(str)

summary
A short, human-readable representation of this message.

This should provide a short summary of the message, much like the subject line of an email.

Note: Sub-classes should override this method. It is used to create the subject of email notifications, IRC notification, and by other tools to display messages to humans in short form.

The default implementation is to simply return the message topic.

url
An URL to the action that caused this message to be emitted.

Note: Sub-classes should override this method if there is a URL associated with message.

Returns A relevant URL.

Return type str or None

usernames
List of users affected by the action that generated this message.

Note: Sub-classes should override this method if the message pertains to a user or users. The data returned from this property is used to filter notifications.

Returns A list of affected usernames.

Return type list(str)

validate()
Validate the headers and body with the message schema, if any.

In addition to the user-provided schema, all messages are checked against the base schema which requires certain message headers and the that body be a JSON object.

Warning: This method should not be overridden by sub-classes.

Raises

- `jsonschema.ValidationError` – If either the message headers or the message body are invalid.
- `jsonschema.SchemaError` – If either the message header schema or the message body schema are invalid.

Message Severity

Each message can have a severity associated with it. The severity is used by applications like the notification service to determine what messages to send to users. The severity can be set at the class level, or on a message-by-message basis. The following are valid severity levels:

DEBUG

fedora_messaging.message.**DEBUG = 10**
Indicates the message is for debugging or is otherwise very low priority. Users will not be notified unless they've explicitly requested DEBUG level messages.

INFO

fedora_messaging.message.**INFO = 20**
Indicates the message is informational. End users will not receive notifications for these messages by default. For example, automated tests passed for their package.

WARNING

fedora_messaging.message.**WARNING = 30**
Indicates a problem or an otherwise important problem. Users are notified of these messages when they pertain to packages they are associated with by default. For example, one or more automated tests failed against their package.

ERROR

fedora_messaging.message.**ERROR = 40**
Indicates a critically important message that users should act upon as soon as possible. For example, their package no longer builds.

Utilities

The `schema_utils` module contains utilities that may be useful when writing the Python API of your message schemas.

libravatar_url

fedora_messaging.schema_utils.**libravatar_url**(*email=None*, *openid=None*, *size=64*, *default='retro'*)
Get the URL to an avatar from libravatar.

Either the user's email or openid must be provided.

If you want to use Libravatar federation (through DNS), you should install and use the `libravatar` library instead. Check out the `libravatar.libravatar_url()` function.

Parameters

- **email** (*str*) – The user's email
- **openid** (*str*) – The user's OpenID
- **size** (*int*) – Size of the avatar in pixels (it's a square).
- **default** (*str*) – Default avatar to return if not found.

Returns The URL to the avatar image.

Return type str

Raises `ValueError` – If neither email nor openid are provided.

3.1.5 Exceptions

Exceptions raised by Fedora Messaging.

exception fedora_messaging.exceptions.**BadDeclaration**(*obj_type=None*, *description=None*, *reason=None*)

Raised when declaring an object in AMQP fails.

Parameters

- **obj_type** (*str*) – The type of object being declared. One of "binding", "queue", or "exchange".
- **description** (*dict*) – The description of the object.
- **reason** (*str*) – The reason the server gave for rejecting the declaration.

exception fedora_messaging.exceptions.**BaseException**

The base class for all exceptions raised by fedora_messaging.

exception fedora_messaging.exceptions.**ConfigurationException**(*message*)

Raised when there's an invalid configuration setting

Parameters **message** (*str*) – A detailed description of the configuration problem which is presented to the user.

exception fedora_messaging.exceptions.**ConnectionException**(**args*, ***kwargs*)

Raised if a general connection error occurred.

You may handle this exception by logging it and resending or discarding the message.

exception fedora_messaging.exceptions.**ConsumeException**

Base class for exceptions related to consuming.

exception fedora_messaging.exceptions.**ConsumerCanceled**

Raised when the server has canceled the consumer.

This can happen when the queue the consumer is subscribed to is deleted, or when the node the queue is located on fails.

exception fedora_messaging.exceptions.**Drop**

Consumer callbacks should raise this to indicate they wish the message they are currently processing to be dropped.

exception fedora_messaging.exceptions.**HaltConsumer**(*exit_code=0*, *reason=None*, *requeue=False*, ***kwargs*)

Consumer callbacks should raise this exception if they wish the consumer to be shut down.

Parameters

- **exit_code** (*int*) – The exit code to use when halting.
- **reason** (*str*) – A reason for halting, presented to the user.
- **requeue** (*bool*) – If true, the message is re-queued for later processing.

exception fedora_messaging.exceptions.**Nack**

Consumer callbacks should raise this to indicate they wish the message they are currently processing to be re-queued.

exception fedora_messaging.exceptions.**NoFreeChannels**

Raised when a connection has reached its channel limit

exception fedora_messaging.exceptions.**PermissionException**(*obj_type=None*, *description=None*, *reason=None*)

Generic permissions exception.

Parameters

- **obj_type** (*str*) – The type of object being accessed that caused the permission error. May be None if the cause is unknown.
- **description** (*object*) – The description of the object, if any. May be None.
- **reason** (*str*) – The reason the server gave for the permission error, if any. If no reason is supplied by the server, this should be the best guess for what caused the error.

exception fedora_messaging.exceptions.**PublishException**(*reason=None*, ***kwargs*)

Base class for exceptions related to publishing.

exception fedora_messaging.exceptions.**PublishReturned**(*reason=None*, ***kwargs*)

Raised when the broker rejects and returns the message to the publisher.

You may handle this exception by logging it and resending or discarding the message.

exception fedora_messaging.exceptions.**ValidationError**

This error is raised when a message fails validation with its JSON schema

This exception can be raised on an incoming or outgoing message. No need to catch this exception when publishing, it should warn you during development and testing that you're trying to publish a message with a different format, and that you should either fix it or update the schema.

3.1.6 Configuration

conf

fedora_messaging.config.**conf = {}**

The configuration dictionary used by fedora-messaging and consumers.

DEFAULTS

fedora_messaging.config.**DEFAULTS = {'amqp_url': 'amqp://?connection_attempts=3&retry_delay**

The default configuration settings for fedora-messaging. This should not be modified and should be copied with `copy.deepcopy()`.

3.1.7 Twisted

In addition to the synchronous API, a Twisted API is provided for applications that need an asynchronous API. This API requires Twisted 16.1.0 or greater.

Note: This API is deprecated, please use *fedora_messaging.api.twisted_consume*

Protocol

The core Twisted interface, a protocol represent a specific connection to the AMQP broker.

The `FedoraMessagingProtocolV2` has replaced the deprecated `FedoraMessagingProtocolV2`. This class inherits the `pika.adapters.twisted_connection.TwistedProtocolConnection` class and adds a few additional methods.

When combined with the *`fedora_messaging.twisted.factory.FedoraMessagingFactory`* class, it's easy to create AMQP consumers that last across connections.

For an overview of Twisted clients, see the Twisted client documentation.

class fedora_messaging.twisted.protocol.**FedoraMessagingProtocol**(*parameters*, *confirms=True*)

A Twisted Protocol for the Fedora Messaging system.

This protocol builds on the generic pika AMQP protocol to add calls specific to the Fedora Messaging implementation.

Warning: This class is deprecated, use the `FedoraMessagingProtocolV2`.

Parameters

- **parameters** (*`pika.ConnectionParameters`*) – The connection parameters.
- **confirms** (*`bool`*) – If True, all outgoing messages will require a confirmation from the server, and the Deferred returned from the publish call will wait for that confirmation.

cancel(*queue*)

Cancel the consumer for a queue.

Parameters **queue** (*`str`*) – The name of the queue the consumer is subscribed to.

Returns

A Deferred that fires when the consumer is canceled, or None if the consumer was already canceled. Wrap the call in `defer.maybeDeferred()` to always receive a Deferred.

Return type defer.Deferred

consume(*callback*, *queue*)

Register a message consumer that executes the provided callback when messages are received.

The queue must exist prior to calling this method. If a consumer already exists for the given queue, the callback is simply updated and any new messages for that consumer use the new callback.

If *`resumeProducing()`* has not been called when this method is called, it will be called for you.

Parameters

- **callback** (*`callable`*) – The callback to invoke when a message is received.
- **queue** (*`str`*) – The name of the queue to consume from.

Returns A namedtuple that identifies this consumer.

Return type *fedora_messaging.twisted.protocol.Consumer*

NoFreeChannels: If there are no available channels on this connection. If this occurs, you can either reduce the number of consumers on this connection or create an additional connection.

pauseProducing()
Pause the reception of messages by canceling all existing consumers. This does not disconnect from the server.

Message reception can be resumed with *resumeProducing()*.

Returns fired when the production is paused.

Return type Deferred

resumeProducing()
Starts or resumes the retrieval of messages from the server queue.

This method starts receiving messages from the server, they will be passed to the consumer callback.

Note: This is called automatically when *consume()* is called, so users should not need to call this unless *pauseProducing()* has been called.

Returns fired when the production is ready to start

Return type defer.Deferred

stopProducing()
Stop producing messages and disconnect from the server. :returns: fired when the production is stopped. :rtype: Deferred

class fedora_messaging.twisted.protocol.**Consumer**(*tag*, *queue*, *callback*, *channel*)
A namedtuple that represents a AMQP consumer.

This is deprecated. Use `fedora_messaging.twisted.consumer.Consumer`.

- The `tag` field is the consumer's AMQP tag (`str`).
- The `queue` field is the name of the queue it's consuming from (`str`).
- The `callback` field is the function called for each message (a callable).
- The `channel` is the AMQP channel used for the consumer (`pika.adapters.twisted_connection.TwistedChannel`).

callback
Alias for field number 2

channel
Alias for field number 3

queue
Alias for field number 1

tag
Alias for field number 0

Factory

A Twisted Factory for creating and configuring instances of the *FedoraMessagingProtocol*.

A factory is used to implement automatic re-connections by producing protocol instances (connections) on demand. Twisted uses factories for its services APIs.

See the Twisted client documentation for more information.

class fedora_messaging.twisted.factory.**FedoraMessagingFactory**(*parameters*, *confirms=True*, *exchanges=None*, *queues=None*, *bindings=None*)

Reconnecting factory for the Fedora Messaging protocol.

buildProtocol(*addr*)
Create the Protocol instance.

See the documentation of *twisted.internet.protocol.ReconnectingClientFactory* for details.

cancel(*queue*)
Cancel the consumer for a queue.

This removes the consumer from the list of consumers to be configured for every connection.

Parameters **queue** (*str*) – The name of the queue the consumer is subscribed to.

Returns

Either a Deferred that fires when the consumer is canceled, or None if the consumer was already canceled. Wrap the call in `defer.maybeDeferred()` to always receive a Deferred.

Return type defer.Deferred or None

clientConnectionFailed(*connector*, *reason*)
Called when the client has failed to connect to the broker.

See the documentation of *twisted.internet.protocol.ReconnectingClientFactory* for details.

clientConnectionLost(*connector*, *reason*)
Called when the connection to the broker has been lost.

See the documentation of *twisted.internet.protocol.ReconnectingClientFactory* for details.

consume(*callback*, *queue*)
Register a new consumer.

This consumer will be configured for every protocol this factory produces so it will be reconfigured on network failures. If a connection is already active, the consumer will be added to it.

Parameters

- **callback** (*callable*) – The callback to invoke when a message arrives.
- **queue** (*str*) – The name of the queue to consume from.

protocol
alias of *fedora_messaging.twisted.protocol.FedoraMessagingProtocol*

publish(*message*, *exchange=None*)
Publish a *fedora_messaging.message.Message* to an exchange on the message broker. This call will survive connection failures and try until it succeeds or is canceled.

Parameters

- **message** (message.Message) – The message to publish.

- **exchange** (*str*) – The name of the AMQP exchange to publish to; defaults to *publish_exchange*

Returns A deferred that fires when the message is published.

Return type defer.Deferred

Raises

- `PublishReturned` – If the published message is rejected by the broker.
- `ConnectionException` – If a connection error occurs while publishing. Calling this method again will wait for the next connection and publish when it is available.

startedConnecting(*connector*)
Called when the connection to the broker has started.

See the documentation of *twisted.internet.protocol.ReconnectingClientFactory* for details.

stopFactory()
Stop the factory.

See the documentation of *twisted.internet.protocol.ReconnectingClientFactory* for details.

stopTrying()
Stop trying to reconnect to the broker.

See the documentation of *twisted.internet.protocol.ReconnectingClientFactory* for details.

whenConnected()
Get the next connected protocol instance.

Returns

A deferred that results in a connected `FedoraMessagingProtocol`.

Return type defer.Deferred

class fedora_messaging.twisted.factory.**FedoraMessagingFactoryV2**(*parameters*, *confirms=True*)
Reconnecting factory for the Fedora Messaging protocol.

buildProtocol(*addr*)
Create the Protocol instance.

See the documentation of *twisted.internet.protocol.ReconnectingClientFactory* for details.

cancel(*consumers*)
Cancel a consumer that was previously started with consume.

Parameters **consumer** (*list of fedora_messaging.api.Consumer*) – The consumers to cancel.

consume(*callback*, *bindings*, *queues*)
Start a consumer that lasts across individual connections.

Parameters

- **callback** (*callable*) – A callable object that accepts one positional argument, a `Message` or a class object that implements the `__call__` method. The class will be instantiated before use.
- **bindings** (*dict or list of dict*) – Bindings to declare before consuming. This should be the same format as the *bindings* configuration.

- **queues** (*dict*) – The queues to declare and consume from. Each key in this dictionary is a queue, and each value is its settings as a dictionary. These settings dictionaries should have the "durable", "auto_delete", "exclusive", and "arguments" keys. Refer to *queues* for details on their meanings.

Returns A deferred that fires with the list of one or more `fedora_messaging.twisted.consumer.Consumer` objects. These can be passed to the *FedoraMessagingFactoryV2.cancel()* API to halt them. Each consumer object has a `result` instance variable that is a Deferred that fires or errors when the consumer halts. The Deferred may error back with a BadDeclaration if the user does not have permissions to consume from the queue.

Return type defer.Deferred

stopFactory()
Stop the factory.

See the documentation of *twisted.internet.protocol.ReconnectingClientFactory* for details.

when_connected()
Retrieve the currently-connected Protocol, or the next one to connect.

Returns

A Deferred that fires with a connected `FedoraMessagingProtocolV2` instance. This is similar to the whenConnected method from the Twisted endpoints APIs, which is sadly isn't available before 16.1.0, which isn't available in EL7.

Return type defer.Deferred

Service

Twisted Service to start and stop the Fedora Messaging Twisted Factory.

This Service makes it easier to build a Twisted application that embeds a Fedora Messaging component. See the `verify_missing` service in fedmsg-migration-tools for a use case.

See https://twistedmatrix.com/documents/current/core/howto/application.html

class fedora_messaging.twisted.service.**FedoraMessagingService**(*amqp_url=None*, *exchanges=None*, *queues=None*, *bindings=None*, *consumers=None*)

A Twisted service to connect to the Fedora Messaging broker.

Parameters

- **on_message** (*callable|None*) – Callback that will be passed each incoming messages. If None, no message consuming is setup.
- **amqp_url** (*str*) – URL to use for the AMQP server.
- **exchanges** (*list of dicts*) – List of exchanges to declare at the start of every connection. Each dictionary is passed to `pika.channel.Channel.exchange_declare()` as keyword arguments, so any parameter to that method is a valid key.

- **queues** (*list of dicts*) – List of queues to declare at the start of every connection. Each dictionary is passed to `pika.channel.Channel.queue_declare()` as keyword arguments, so any parameter to that method is a valid key.
- **bindings** (*list of dicts*) – A list of bindings to be created between queues and exchanges. Each dictionary is passed to `pika.channel.Channel.queue_bind()`. The "queue" and "exchange" keys are required.
- **consumers** (*dict*) – A dictionary where each key is a queue name and the value is a callable object to handle messages on that queue. Consumers will be set up after each connection is established so they will survive networking issues.

factoryClass
alias of *fedora_messaging.twisted.factory.FedoraMessagingFactory*

class fedora_messaging.twisted.service.**FedoraMessagingServiceV2**(*amqp_url=None*, *publish_confirms=True*)

A Twisted service to connect to the Fedora Messaging broker.

Parameters

- **amqp_url** (*str*) – URL to use for the AMQP server.
- **publish_confirms** (*bool*) – If true, use the RabbitMQ publisher confirms AMQP extension.

stopService()
Gracefully stop the service.

Returns

a Deferred which is triggered when the service has finished shutting down.

Return type defer.Deferred

3.2 Message Format

This documentation covers the format of AMQP messages sent by this library. If you are interested in using a language other than Python to send or receive messages sent by Fedora applications, this document is for you.

3.2.1 Overview

Messages are AMQP Basic content. Basic messages have the content type, content encoding, a table of headers, delivery mode, priority, correlation ID, reply-to, expiration, message ID, timestamp, type, user ID, and app ID fields.

Your messages *MUST* have a content-type of `application/json` and a content-encoding of `utf-8`. The message ID should be a version 4 UUID.

3.2.2 Headers

Required

Messages must have, at a minimum, the `fedora_messaging_severity`, `fedora_messaging_schema`, and `sent-at` keys.

The `fedora_messaging_severity` key should be set to an integer that indicates the importance of the message to an end user, with 10 being debug-level information, 20 being informational, 30 being warning-level, and 40 being critically important.

The `fedora_messaging_schema` key should be set to a string that uniquely identifies the type of message. In the Python library this is the entry point name, which is mapped to a class containing the schema and a Python API to interact with the message object.

The `sent-at` key should be a ISO8601 date time that should include the UTC offset and should *not* include microseconds. For example: `2019-07-30T19:12:22+00:00`.

The header's json-schema is:

```
{
   "$schema": "http://json-schema.org/draft-04/schema#",
   "description": "Schema for message headers",
   "type": "object",
   "properties": {
      "fedora_messaging_severity": {
         "type": "number",
         "enum": [10, 20, 30, 40],
      },
      "fedora_messaging_schema": {"type": "string"},
      "sent-at": {"type": "string"},
   },
}
```

Optional

In addition to the required headers, there are a number of optional headers you can set that have special meaning. The general format of these headers is `fedora_messaging_<object>_<id>` where the `<object>` is one of `user`, `rpm`, `container`, `module`, or `flatpak` and `<id>` uniquely identifies the object. Set these headers when the message pertains to the referenced object.

For example, if the user `jcline` submitted a build for the `python-requests` RPM, the message about that event would have `fedora_messaging_user_jcline` and `fedora_messaging_rpm_python-requests` set.

At this time the value of the header key is not used and should always be set to a Boolean value of true.

3.2.3 Body

The message body must match the content-type and content-encoding. That is, it must be UTF-8 encoded JSON. Additionally, it must be a JSON Object. Beyond that, there are no restrictions. Messages should be validated using their JSON schema. If you are publishing a new message type, please write a json-schema for it and provide it to the Fedora infrastructure team. It will be distributed to applications that wish to consume the message.

CHAPTER 4

Contributor Guide

4.1 Contributing

Thanks for considering contributing to fedora-messaging, we really appreciate it!

Quickstart:

1. Look for an existing issue about the bug or feature you're interested in. If you can't find an existing issue, create a new one.
2. Fork the repository on GitHub.
3. Fix the bug or add the feature, and then write one or more tests which show the bug is fixed or the feature works.
4. Submit a pull request and wait for a maintainer to review it.

More detailed guidelines to help ensure your submission goes smoothly are below.

Note: If you do not wish to use GitHub, please send patches to infrastructure@lists.fedoraproject.org.

4.1.1 Guidelines

Python Support

fedora-messaging supports Python 2.7 and Python 3.4 or greater. This is automatically enforced by the continuous integration (CI) suite.

Code Style

We follow the PEP8 style guide for Python. This is automatically enforced by the CI suite.

We are using *Black <https://github.com/ambv/black>* to automatically format the source code. It is also checked in CI. The Black webpage contains instructions to configure your editor to run it on the files you edit.

Tests

The test suites can be run using tox by simply running `tox` from the repository root. All code must have test coverage or be explicitly marked as not covered using the `# no-qa` comment. This should only be done if there is a good reason to not write tests.

Your pull request should contain tests for your new feature or bug fix. If you're not certain how to write tests, we will be happy to help you.

Release notes

To add entries to the release notes, create a file in the `news` directory with the `source.type` name format, where `type` is one of:

- `feature`: for new features
- `bug`: for bug fixes
- `api`: for API changes
- `dev`: for development-related changes
- `author`: for contributor names
- `other`: for other changes

And where the `source` part of the filename is:

- `42` when the change is described in issue `42`
- `PR42` when the change has been implemented in pull request `42`, and there is no associated issue
- `Cabcdef` when the change has been implemented in changeset `abcdef`, and there is no associated issue or pull request.
- `username` for contributors (`author` extention). It should be the username part of their commits' email address.

A preview of the release notes can be generated with `towncrier --draft`.

Licensing

Your commit messages must include a Signed-off-by tag with your name and e-mail address, indicating that you agree to the Developer Certificate of Origin version 1.1:

```
Developer Certificate of Origin
Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors.
1 Letterman Drive
Suite D4700
San Francisco, CA, 94129

Everyone is permitted to copy and distribute verbatim copies of this
license document, but changing it is not allowed.


Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:
```

(continues on next page)

(continued from previous page)

```
(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the best
    of my knowledge, is covered under an appropriate open source
    license and I have the right under that license to submit that
    work with modifications, whether created in whole or in part
    by me, under the same open source license (unless I am
    permitted to submit under a different license), as indicated
    in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```

Use `git commit -s` to add the Signed-off-by tag.

Releasing

When cutting a new release, follow these steps:

- update the version in `fedora_messaging/__init__.py`
- generate the changelog by running `towncrier`
- change the `Development Status` classifier in `setup.py` if necessary
- commit the changes
- tag the commit
- push to GitHub
- generate a tarball and push to PyPI with the commands:

```
python setup.py sdist bdist_wheel
twine upload -s dist/*
```

Python Module Index

f

Index

Symbols

A

B

C

D

E

F

H

I

L

M

N

P

Q

R

S

T

U

V

W