fedmsg Documentation

Release 1.0.1

Ralph Bean, Red Hat. Inc, and others

Sep 29, 2017

Contents

1 User Guide **3**
- 1.1 Installation . . . 3
- 1.2 Configuration . . . 3
- 1.3 Commands . . . 12
- 1.4 Publishing . . . 13
- 1.5 Subscribing . . . 14
- 1.6 Deploying fedmsg . . . 15
- 1.7 Changelog . . . 19

2 API Guide **25**
- 2.1 Developer Interface . . . 25

3 Contributor Guide **39**
- 3.1 Contributing . . . 39

4 Community **43**

Python Module Index **45**

fedmsg (Federated Message Bus) is a library built on ZeroMQ using the PyZMQ Python bindings. fedmsg aims to make it easy to connect services together using ZeroMQ publishers and subscribers.

Receiving messages in Python is as simple as:

```
import fedmsg

# Yield messages as they're available from a generator
for name, endpoint, topic, msg in fedmsg.tail_messages():
    print topic, msg
```

To publish a message:

```
import fedmsg
fedmsg.publish(topic='testing', modname='test', msg={
    'test': "Hello World",
})
```

Note: fedmsg requires some configuration before it can be used. See the *Configuration* documentation for more information.

CHAPTER 1

User Guide

Installation

fedmsg is available on PyPI and may also be available in your distribution's repositories.

Fedora and EPEL

fedmsg is packaged for Fedora, EPEL 6, and EPEL 7. On Fedora:

```
$ sudo dnf install fedmsg
```

On an Enterprise Linux-based distribution with the EPEL repository:

```
$ sudo yum install fedmsg
```

PyPI

To install fedmsg via PyPI, you will need the `openssl` header files, available via `openssl-devel` on Red Hat-based distributions and via `openssl-dev` on Debian-based distributions, and GCC:

```
$ pip install fedmsg[commands,consumers,crypto_ng]
```

Configuration

fedmsg requires some configuration before it will work properly.

General configuration

active

A boolean that, if `True`, will cause the publishing socket to connect to the *relay_inbound* socket, rather than binding its own socket.

topic_prefix

A string prefixed to the topics of all outgoing messages.

The default value is `org.fedoraproject`.

environment

A string that must be one of `['prod', 'stg', 'dev']`. It signifies the environment in which this fedmsg process is running and can be used to weakly separate different logical buses running in the same infrastructure. It is used by *`fedmsg.publish()`* when it is constructing a fully-qualified topic.

status_directory

A string that is the absolute path to a directory where consumers can save the status of their last processed message. In conjunction with *datagrepper_url*, allows for automatic retrieval of backlog on daemon startup.

datagrepper_url

A URL to an instance of the datagrepper web service, such as https://apps.fedoraproject.org/datagrepper/raw. Can be used in conjunction with *status_directory* to allow for automatic retrieval of backlog on daemon startup.

endpoints

`dict` - A mapping of "service keys" to "zeromq endpoints"; the heart of fedmsg.

`endpoints` is "a list of possible addresses from which fedmsg can send messages." Thus, "subscribing to the bus" means subscribing to every address listed in this dictionary.

`endpoints` is also an index where a fedmsg process can look up what port it should bind to to begin emitting messages.

When *`fedmsg.init()`* is invoked, a "name" is determined. It is either passed explicitly, or guessed from the call stack. The name is combined with the hostname of the process and used as a lookup key in the `endpoints` dict.

When sending, fedmsg will attempt to bind to each of the addresses listed under its service key until it can succeed in acquiring the port. There needs to be as many endpoints listed as there will be `processes * threads` trying to publish messages for a given service key.

For example, the following config provides for four WSGI processes on bodhi on the machine app01 to send fedmsg messages.

```
>>> config = dict(
...     endpoints={
...         "bodhi.app01":  [
...               "tcp://app01.phx2.fedoraproject.org:3000",
...               "tcp://app01.phx2.fedoraproject.org:3001",
```

```
...                 "tcp://app01.phx2.fedoraproject.org:3002",
...                 "tcp://app01.phx2.fedoraproject.org:3003",
...             ],
...         },
... )
```

If apache is configured to start up five WSGI processes, the fifth one will produce tracebacks complaining with `IOError("Couldn't find an available endpoint.")`.

If apache is configured to start up four WSGI processes, but with two threads each, four of those threads will raise exceptions with the same complaints.

A process subscribing to the fedmsg bus will connect a zeromq SUB socket to every endpoint listed in the `endpoints` dict. Using the above config, it would connect to the four ports on app01.phx2.fedoraproject.org.

Note: This is possibly the most complicated and hardest to understand part of fedmsg. It is the black sheep of the design. All of the simplicity enjoyed by the python API is achieved at cost of offloading the complexity here.

Some work could be done to clarify the language used for "name" and "service key". It is not always consistent in `fedmsg.core`.

---

srv_endpoints

`list` - A list of domain names for which to query SRV records to get the associated endpoints.

When using fedmsg.config.load_config(), the DNS lookup is done and the resulting endpoints are added to config['endpoint'][$DOMAINNAME]

For example, the following would query the endpoints for foo.example.com.

```
>>> config = dict(
...     srv_endpoints=[foo.example.com]
...)
```

replay_endpoints

`dict` - A mapping of service keys, the same as for *endpoints* to replay endpoints, each key having only one. The replay endpoints are special ZMQ endpoints using a specific protocol to allow the client to request a playback of messages in case some have been dropped, for instance due to network failures.

If the service has a replay endpoint specified, fedmsg will automatically try to detect such failures and properly query the endpoint to get the playback if needed.

relay_inbound

`str` - A list of special zeromq endpoints where the inbound, passive zmq SUB sockets for for instances of `fedmsg-relay` are listening.

Commands like `fedmsg-logger` actively connect here and publish their messages.

See *Commands* for more information.

relay_outbound

`str` - A list of special zeromq endpoints where the outbound sockets for instances of `fedmsg-relay` should bind.

fedmsg.consumers.gateway.port

`int` - A port number for the special outbound zeromq PUB socket posted by `fedmsg.commands.gateway.gateway()`. The `fedmsg-gateway` command is described in more detail in *Commands*.

Authentication and Authorization

The following settings relate to message authentication and authorization.

sign_messages

`bool` - If set to true, then `fedmsg.core` will try to sign every message sent using the machinery from *fedmsg.crypto*.

It is often useful to set this to *False* when developing. You may not have X509 certs or the tools to generate them just laying around. If disabled, you will likely want to also disable *validate_signatures*.

validate_signatures

`bool` - If set to true, then the base class `fedmsg.consumers.FedmsgConsumer` will try to use *fedmsg.crypto.validate()* to validate messages before handing them off to the particular consumer for which the message is bound.

This is also used by *fedmsg.meta* to denote trustworthiness in the natural language representations produced by that module.

crypto_backend

`str` - The name of the *fedmsg.crypto* backend that should be used to sign outgoing messages. It may be either 'x509' or 'gpg'.

crypto_validate_backends

`list` - A list of names of *fedmsg.crypto* backends that may be used to validate incoming messages.

ssldir

`str` - This should be directory on the filesystem where the certificates used by *fedmsg.crypto* can be found. Typically `/etc/pki/fedmsg/`.

crl_location

`str` - This should be a URL where the certificate revocation list can be found. This is checked by *fedmsg.crypto.validate()* and cached on disk.

crl_cache

`str` - This should be the path to a filename on the filesystem where the CRL downloaded from *crl_location* can be saved. The python process should have write access there.

crl_cache_expiry

`int` - Number of seconds to keep the CRL cached before checking *crl_location* for a new one.

ca_cert_location

`str` - This should be a URL where the certificate authority cert can be found. This is checked by *`fedmsg.crypto.validate()`* and cached on disk.

ca_cert_cache

`str` - This should be the path to a filename on the filesystem where the CA cert downloaded from *ca_cert_location* can be saved. The python process should have write access there.

ca_cert_cache_expiry

`int` - Number of seconds to keep the CA cert cached before checking *ca_cert_location* for a new one.

certnames

`dict` - This should be a mapping of certnames to cert prefixes.

The keys should be of the form `<service>.<host>`. For example: `bodhi.app01`.

The values should be the prefixes of cert/key pairs to be found in *ssldir*. For example, if `bodhi-app01.stg.phx2.fedoraproject.org.crt` and `bodhi-app01.stg.phx2.fedoraproject.org.key` are to be found in *ssldir*, then the value `bodhi-app01.stg.phx2.fedoraproject.org` should appear in the *certnames* dict.

Putting it all together, this value could be specified as follows:

```
certnames={
    "bodhi.app01": "bodhi-app01.stg.phx2.fedoraproject.org",
    # ... other certname mappings may follow here.
}
```

Note: This is one of the most cumbersome parts of fedmsg. The reason we have to enumerate all these redundant mappings between "service.hostname" and "service-fqdn" has to do with the limitations of reverse dns lookup. Case in point, try running the following on app01.stg inside Fedora Infrastructure's environment.

```
>>> import socket
>>> print socket.getfqdn()
```

You might expect it to print "app01.stg.phx2.fedoraproject.org", but it doesn't. It prints "memcached04.phx2.fedoraproject.org". Since we can't rely on programatically extracting the fully qualified domain names of the host machine during runtime, we need to explicitly list all of the certs in the config.

---

routing_nitpicky

`bool` - When set to True, messages whose topics do not appear in *routing_policy* automatically fail the validation process described in *fedmsg.crypto*. It defaults to `False`.

routing_policy

A Python dictionary mapping fully-qualified topic names to lists of cert names. If a message's topic appears in the *routing_policy* and the name on its certificate does not appear in the associated list, then that message fails the validation process in *fedmsg.crypto*.

For example, a routing policy might look like this:

```
routing_policy={
    "org.fedoraproject.prod.bodhi.buildroot_override.untag": [
        "bodhi-app01.phx2.fedoraproject.org",
        "bodhi-app02.phx2.fedoraproject.org",
        "bodhi-app03.phx2.fedoraproject.org",
        "bodhi-app04.phx2.fedoraproject.org",
    ],
}
```

The above loosely translates to "messages about bodhi buildroot overrides being untagged may only come from the first four app servers." If a message with that topic bears a cert signed by any other name, then that message fails the validation process.

Expect that your *routing_policy* (if you define one) will become quite long.

The default is an empty dictionary.

ZeroMQ

The following settings are ZeroMQ configuration options.

high_water_mark

`int` - An option to zeromq that specifies a hard limit on the maximum number of outstanding messages to be queued in memory before reaching an exceptional state.

For our pub/sub zeromq sockets, the exceptional state means *dropping messages*. See the upstream documentation for ZMQ_HWM and ZMQ_PUB.

A value of `0` means "no limit" and is the recommended value for fedmsg. It is referenced when initializing sockets in *fedmsg.init()*.

io_threads

`int` - An option that specifies the size of a zeromq thread pool to handle I/O operations. See the upstream documentation for zmq_init.

This value is referenced when initializing the zeromq context in *fedmsg.init()*.

post_init_sleep

`float` - A number of seconds to sleep after initializing and before sending any messages. Setting this to a value greater than zero is required so that zeromq doesn't drop messages that we ask it to send before the pub socket is finished initializing.

Experimentation needs to be done to determine and sufficiently small and safe value for this number. `1` is definitely safe, but annoyingly large.

zmq_enabled

`bool` - A value that must be true. It is present solely for compatibility/interoperability with moksha.

zmq_reconnect_ivl

`int` - Number of miliseconds that zeromq will wait to reconnect until it gets a connection if an endpoint is unavailable. This is in miliseconds. See upstream zmq options for more information.

zmq_reconnect_ivl_max

`int` - Max delay that you can reconfigure to reduce reconnect storm spam. This is in miliseconds. See upstream zmq options for more information.

zmq_strict

`bool` - When false, allow splats ('*') in topic names when subscribing. When true, disallow splats and accept only strict matches of topic names.

This is an argument to moksha and arose there to help abstract away differences between the "topics" of zeromq and the "routing_keys" of AMQP.

zmq_tcp_keepalive

`int` - Interpreted as a boolean. If non-zero, then keepalive options will be set. See upstream zmq options and general overview.

zmq_tcp_keepalive_cnt

`int` - Number of keepalive packets to send before considering the connection dead. See upstream zmq options and general overview.

zmq_tcp_keepalive_idle

`int` - Number of seconds to wait after last data packet before sending the first keepalive packet. See upstream zmq options and general overview.

zmq_tcp_keepalive_intvl

`int` - Number of seconds to wait inbetween sending subsequent keepalive packets. See upstream zmq options and general overview.

IRC

irc

`list` - A list of ircbot configuration dicts. This is the primary way of configuring the `fedmsg-irc` bot implemented in `fedmsg.commands.ircbot.ircbot()`.

Each dict contains a number of possible options. Take the following example:

```
>>> config = dict(
...     irc=[
...         dict(
...             network='irc.freenode.net',
...             port=6667,
...             nickname='fedmsg-dev',
...             channel='fedora-fedmsg',
...             timeout=120,
...
...             make_pretty=True,
...             make_terse=True,
...             make_short=True,
...
...             filters=dict(
...                 topic=['koji'],
...                 body=['ralph'],
...             ),
...         ),
...     ],
... )
```

Here, one bot is configured. It is to connect to the freenode network on port 6667. The bot's name will be `fedmsg-dev` and it will join the `#fedora-fedmsg` channel.

`make_pretty` specifies that colors should be used, if possible.

`make_terse` specifies that the "natural language" representations produced by *`fedmsg.meta`* should be echoed into the channel instead of raw or dumb representations.

`make_short` specifies that any url associated with the message should be shortened with a link shortening service. If *True*, the https://da.gd/ service will be used. You can alternatively specify a *callable* to use your own custom url shortener, like this:

```
make_short=lambda url: requests.get('http://api.bitly.com/v3/shorten?login=YOURLOGIN&
↪apiKey=YOURAPIKEY&longUrl=%s&format=txt' % url).text.strip()
```

The `filters` dict is not very smart. In the above case, any message that has 'koji' anywhere in the topic or 'ralph' anywhere in the JSON body will be discarded and not echoed into `#fedora-fedmsg`. This is an area that could use some improvement.

irc_color_lookup

A dictionary mapping module names to MIRC irc color names. For example:

```
>>> irc_color_lookup = {
...     "fas": "light blue",
...     "bodhi": "green",
...     "git": "red",
...     "tagger": "brown",
```

```
...     "wiki": "purple",
...     "logger": "orange",
...     "pkgdb": "teal",
...     "buildsys": "yellow",
...     "planet": "light green",
... }
```

irc_method

the name of the method used to publish the messages on IRC. Valid values are `msg` and `notice`.

The default is `notice`.

STOMP Configuration

When using STOMP, you need to set *zmq_enabled* to `False`. Additionally, if you're using STOMP with TLS (recommended), you do not need fedmsg's cryptographic signatures to validate messages so you can turn those off by setting *validate_signatures* to `False`

stomp_uri

A string of comma-separated brokers. For example:

```
stomp_uri='broker01.example.com:61612,broker02.example.com:61612'
```

There is no default for this setting.

stomp_heartbeat

The STOMP heartbeat interval, in milliseconds.

There is no default for this setting.

stomp_user

The username to use with STOMP when authenticating with the broker.

There is no default for this setting.

stomp_pass

The password to use with STOMP when authenticating with the broker.

There is no default for this setting.

stomp_ssl_crt

The PEM-encoded x509 client certificate to use when authenticating with the broker.

There is no default for this setting.

stomp_ssl_key

The PEM-encoded private key for the *stomp_ssl_crt*.

There is no default for this setting.

stomp_queue

If set, this will cause the Moksha hub to only listen to the specified queue for all fedmsg consumers. If it is not specified, the Moksha hub will listen to all topics declared by all fedmsg consumers.

There is no default for this setting.

Commands

Console Scripts

`fedmsg` provides a number of console scripts for use with random shell scripts.

fedmsg-logger

`fedmsg.commands.logger.`**`logger`**`()`

fedmsg-tail

`fedmsg.commands.tail.`**`tail`**`()`

fedmsg-dg-replay

`fedmsg.commands.replay.`**`replay`**`()`

fedmsg-collectd

fedmsg-check

`fedmsg-check` is used to check the status of consumers and producers. It requires the `moksha.monitoring.socket` key to be set in the configuration.

See usage details with `fedmsg-check --help`.

Service Daemons

fedmsg-hub

`fedmsg.commands.hub.`**`hub`**`()`

fedmsg-relay

fedmsg-signing-relay

fedmsg-irc

fedmsg-gateway

Writing your own fedmsg commands

The *fedmsg.commands* module provides a `@command` decorator to help simplify this.

class fedmsg.commands.**BaseCommand**

Bases: object

daemonizable = False

execute()

extra_args = None

get_config()

Publishing

Before you start publishing messages, it is recommended that you call *fedmsg.init()*. This should be done from every Python thread you intend to publish messages from, and should be done early in your application's initialization. The reason for this is that this call creates a thread-local `zmq.Context` and bind to a socket for publishing.

> **Warning:** Since the network's latency is not 0, it can take some time before a subscriber has set up its connection to the publishing socket. When a publishing socket has no subscribers, it simply drops the published message. Currently, the only way to avoid lost messages is to initialize the socket as early as you can.

To publish a message, call *fedmsg.publish()*. Your message should be a Python dictionary capable of being JSON-serialized. Additionally, any strings it contains should be a text type. That is, `unicode` in Python 2 and `str` in Python 3.

Publishing Through a Relay

It's possible to avoid having each thread bind to a socket for publishing. To do so, you need to set up a *fedmsg-relay* service. Once it's running, you need to set *active* to `True`, which causes publishing sockets to connect to the socket specified in *relay_inbound*.

Publishing Without ZeroMQ

fedmsg also supports publishing your messages via STOMP. Brokers that support STOMP include RabbitMQ and Apache ActiveMQ. Consult the *STOMP Configuration* for details on how to configure this.

Common Problems

There are a set of problems users commonly encounter when publishing messages, nearly all of them related to configuration. Check the logs to see if there are any helpful messages.

ZeroMQ Not Enabled

fedmsg supports publishing messages using protocols other than ZeroMQ. If you neglect to set *zmq_enabled* to `True`, fedmsg will attempt to publish the message with Moksha's Hub. If the hub has not been initialized, you'll receive an `AttributeError` when you call *`fedmsg.publish()`*.

No Endpoints Available

Currently, unless you are publishing through a relay, you must declare a list of endpoints that fedmsg can bind to. Each Python thread, when *`fedmsg.init()`* is called, iterates through the list and attempts to bind to each address. If it is unable to bind to any address, an `IOError` is raised. The solution is to add more endpoints to the configuration.

Subscribing

There are two approaches to subscribing to messages that have been published.

The Generator Approach

The first approach is to use the *`fedmsg.tail_messages()`* API. This returns a generator that yields messages as they arrive:

```
>>> import fedmsg
>>> for name, endpoint, topic, msg in fedmsg.tail_messages():
...     print topic, msg
```

For this to print anything, you need to be *Publishing* messages.

Note: This approach only works with messages published via ZeroMQ. If you are *Publishing Without ZeroMQ* then you will need to use *The Consumer Approach*.

The Consumer Approach

This approach requires a bit more work, but provides several advantages. Namely, it manages workers for you and supports replaying missed messages via a network service.

The first step is to write a class which extends `fedmsg.consumers.FedmsgConsumer`. The class documentation covers the details of implementing a consumer.

After you have implemented your consumer class and registered it under the `moksha.consumer` Python entry-point, you need to start the *fedmsg-hub* service, which will create your class in one or more worker threads and pass messages to these workers as they arrive.

To see a working example of this pattern, investigate the `fedmsg.consumers.relay` module.

Consuming Non-ZeroMQ Messages

In order to consume messages with STOMP, you will need to set the *STOMP Configuration* options.

Best Practices

When using fedmsg, messages will be lost. Your applications and services should be prepared to receive duplicate messages. Always provide a way for the application or service to recover gracefully for a lost or duplicate message.

Deploying fedmsg

Elsewhere, the emphasis in fedmsg docs is on how to subscribe to an existing fedmsg deployment; how do I listen for koji builds from Fedora Infrastructure? This document, on the other hand, is directed at those who want to deploy fedmsg for their own systems.

This document also only goes as far as setting things up for a single machine. You typically deploy fedmsg across an *infrastructure* but if you just want to try it out for "proof-of-concept", these are the docs for you.

Lastly, the emphasis here is on the practical – there will be lots of examples.

Note: Caveat: fedmsg is deployed at a couple different sites:

- Fedora Infrastructure
- data.gouv.fr
- to some extent, Debian Infrastructure

We wrote this document much later afterwards, so, if you come across errors, or things that don't work right. Please report it.

The basics

First install fedmsg:

```
$ sudo dnf install fedmsg
```

Now you have some `fedmsg-*` cli tools like `fedmsg-tail` and `fedmsg-logger`.

On Fedora systems, fedmsg is configured by default to subscribe to Fedora Infrastructure's bus. Since you are deploying for your own site, you don't want that. So edit `/etc/fedmsg.d/endpoints.py` and *comment out the whole "fedora-infrastructure" section*, like this:

```
#"fedora-infrastructure": [
#    "tcp://hub.fedoraproject.org:9940",
#    #"tcp://stg.fedoraproject.org:9940",
#],
```

Starting fedmsg-relay

Not all fedmsg interactions require the relay, but publishing messages from a terminal does.

Install fedmsg-relay and start it:

```
$ sudo dnf install fedmsg-relay
$ sudo systemctl restart fedmsg-relay
$ sudo systemctl enable fedmsg-relay
```

It has a pid file in `/var/run/fedmsg/fedmsg-relay.pid` and you can view the logs in `journalctl --follow`. On other systems you can find the logs in `/var/log/fedmsg/fedmsg-relay.log`.

Out of the box, it should be listening for incoming messages on `tcp://127.0.0.1:2003` and re-publishing them indiscriminately at `tcp://127.0.0.1:4001`. It is fine to keep these defaults.

Test it out

Try a test! Open two terminals:

- In the first, type `fedmsg-tail --really-pretty`
- In the second, type `echo "Hello world" | fedmsg-logger`

You should see the JSON representation of your message show up in the first terminal. It should look something like this.

```
{
  "username": "root",
  "i": 1,
  "timestamp": 1393878837,
  "msg_id": "2014-f1c49f0b-5caf-49e6-b79a-cc54bcfac602",
  "topic": "org.fedoraproject.dev.logger.log",
  "msg": {
    "log": "Hello world"
  }
}
```

These are two handy tools for debugging the configuration of your bus.

Branching out to two machines

Everything is tied together in fedmsg by the *endpoints* dict. It lets

- A publishing service know what port it should be publishing on.
- A consuming service know where the publisher is so it can connect there.

Let's say you have two machines `hostA` and `hostB`. If you installed that fedmsg-relay on `hostA` as discussed above, then the config file in `/etc/fedmsg.d/relay.py` is going to have values like `tcp://127.0.0.1:4001`. That address will only work for local connectivity. Try changing *all* occurences of `127.0.0.1` in that file to `hostA` so that it looks something like this:

```
config = dict(
    endpoints={
        "relay_outbound": [
            "tcp://hostA:4001",
        ],
    },
    relay_inbound=[
        "tcp://hostA:2003",
```

```
    ],
)
```

To confirm that something's not immediately broken, you can go through the tests of doing `fedmsg-logger` and `fedmsg-tail` on `hostA` again (all "local").

Copy that relay.py file over to `hostB` with `scp /etc/fedmsg.d/relay.py hostB:/etc/fedmsg.d/relay.py`

You should now be able to run `fedmsg-tail` on `hostA` and have it receive a message from `fedmsg-logger` on `hostB` and vice versa have a `fedmsg-tail` session on `hostB` receive a `fedmsg-logger` statement from `hostA`.

The key here is that fedmsg works by having a **shared configuration** that is distributed to all machines. `hostA` only knows where to publish by reading in the config and `hostB` only knows where to consume by reading in the config. If the configs are not the same, then there's going to be a mis-match and your messages won't arrive... anywhere.

It's a far leap ahead, but you're welcome to browse the configuration we're using in production for Fedora Infrastructure.

Store all messages

And now for a different topic.

We use a tool called datanommer to store all the messages that come across the bus in a postgres database. Using whatever relational database you like should be possible just by modifying the config.

Setting up postgres

Here, set up a postgres database:

```
$ sudo dnf install postgresql-server python-psycopg2
$ postgresql-setup initdb
```

Edit the `/var/lib/pgsql/data/pg_hba.conf` as the user postgres. You might find a line like this:

```
host all all 127.0.0.1/32 ident sameuser
host all all ::1/128 ident sameuser
```

Instead of that line, change it to this:

```
host all all 127.0.0.1/32 trust
host all all ::1/128 trust
```

Note: Using `trust` is super unsafe long term. That means that anyone with any password will be able to connect locally. That's fine for our little one-box test here, but you'll want to use md5 or kerberos or something long term.

Start up postgres:

```
$ systemctl start postgresql
$ systemctl enable postgresql
```

Create a database user and the db itself for datanommer and friends:

```
$ sudo -u postgres createuser -SDRPE datanommer
$ sudo -u postgres createdb -E utf8 datanommer -O datanommer
```

Setting up datanommer

Install it:

```
$ sudo dnf install fedmsg-hub python-datanommer-consumer datanommer-commands
```

Edit the configuration to 1) be enabled, 2) point at your newly created postgres db. Edit `/etc/fedmsg.d/datanommer.py` and change the whole thing to look like this:

```
config = {
    'datanommer.enabled': True,
    'datanommer.sqlalchemy.url': 'postgresql://datanommer:password@localhost/
↪datanommer',
}
```

Run the following command from the `datanommer-commands` package to set up the tables. It will read in that connection url from `/etc/fedmsg.d/datanommer.py`:

```
$ datanommer-create-db
```

Start the `fedmsg-hub` daemon, which will pick up the datanommer plugin, which will in turn read in that connection string, start listening for messages, and store them all in the db.

```
$ sudo systemctl start fedmsg-hub
$ sudo systemctl enable fedmsg-hub
```

You can check `journalctl --follow` for logs.

Try testing again with `fedmsg-logger`. After publishing a message, you should see it in the datanommer stats if you run `datanommer-stats`:

```
$ datanommer-stats
[2014-03-03 20:34:43][    fedmsg    INFO] logger has 2 entries
```

Querying datanommer with datagrepper

You can, of course, query datanommer with SQL yourself (and there's a python API for directly querying in the `datanommer.models` module). For the rest here is the HTTP API we have called "datagrepper". Let's set it up:

```
$ sudo dnf install datagrepper mod_wsgi
```

Add a config file for it in `/etc/httpd/conf.d/datagrepper.conf` with these contents:

```
LoadModule wsgi_module modules/mod_wsgi.so

# Static resources for the datagrepper app.
Alias /datagrepper/css /usr/lib/python2.7/site-packages/datagrepper/static/css

WSGIDaemonProcess datagrepper user=fedmsg group=fedmsg maximum-requests=50000 display-
↪name=datagrepper processes=8 threads=4 inactivity-timeout=300
WSGISocketPrefix run/wsgi
```

```
WSGIRestrictStdout Off
WSGIRestrictSignal Off
WSGIPythonOptimize 1

WSGIScriptAlias /datagrepper /usr/share/datagrepper/apache/datagrepper.wsgi

<Directory /usr/share/datagrepper/>
  WSGIProcessGroup datagrepper
  # XXX - The syntax for this is different for different versions of apache
  Require all granted
</Directory>
```

Finally, start up httpd with:

```
$ sudo systemctl restart httpd
$ sudo systemctl enable httpd
```

And it should just work. Open a web browser and try to visit `http://localhost/datagrepper/`.

The whole point of datagrepper is its API, which you might experiment with using the httpie tool:

```
$ sudo dnf install httpie
$ http get http://localhost/datagrepper/raw/ order==desc
```

Outro

This document is a work in progress. Future topics may include selinux and *Cryptography and Message Signing*.

Let us know what you'd like to know if it is missing.

Changelog

1.0.1

Bug fixes

- Fix an issue where messages replayed from datagrepper always failed signature validation despite having valid signatures (#477).
- Fix a Python 3 incompatibility where the downloading the certificate revocation list crashed when attempting to write the file (#478).

Development improvements

- Several loggers now use their full module path as their logger name rather than just "fedmsg" (#479).

Many thanks to all our contributors for this release:

- Jeremy Cline
- Chaitanya Kukde

1.0.0

Backwards incompatible changes

- The `--daemon` option for all fedmsg commands that was deprecated in 0.19.0 has been removed. We recommend using your operating system's init system instead. systemd unit files are available in the git repository (#470).
- Python 2.6 is no longer supported (#469).

Features

- Python 3.4+ is now supported. In order to use x509 certificates to sign and verify messages, you will need cryptography v1.6+ and pyOpenSSL v16.1+. These can be installed with pip via `pip install fedmsg[crypto_ng]` (#449).
- The fedmsg documentation has been re-organized (#453).

Development Improvements

- The m2crypto unit tests were being skipped when the cryptography library was missing. This is no longer the case (#446).
- All usage of the nose library has been removed from the tests and the dependency on nose has been removed (#448).
- `click` has been added as a test dependency (#452).
- Test coverage increased from 54.72% to 58.82%
- Several improvements to the tox.ini file (#458).

Many thanks to all our contributors for this release:

- Lumír 'Frenzy' Balhar
- Ralph Bean
- Jeremy Cline
- Chenxiong Qi

0.19.1

0.19.1 is a bug fix release that addresses several critical regressions introduced in 0.19.0.

Bug fixes

- Fix an issue where messages failed validation because the message certificate and signature were unicode objects (#456).
- Fix an issue where message bodies were not deserialized from JSON before being passed to a consumer because the message bodies were unicode objects (#464).
- Fix an issue where messages never got passed to the consumer because the message pre-processing caused an unhandled exception (#462).

Many thanks to the contributors for this release:

- Kamil Páral
- Jeremy Cline
- Patrick Uiterwijk
- Ralph Bean
- Ricky Elrod

0.19.0

Deprecations

- The `--daemon` option has been deprecated for all fedmsg commands and will be removed in a future release. We recommend using your operating system's init system instead. systemd units and SysV init scripts are available in the git repository (#434).

Features

- A new command, `fedmsg-signing-relay`, has been added that signs messages prior to relaying them (#409).
- A new command, `fedmsg-check`, can be used to check whether or not the expected fedmsg producers and consumers are running (#416).
- If the message contains a `headers` key, these are placed in the message body (#437).
- It is now possible to use cryptography and pyOpenSSL rather than m2crypto (#421).
- The ircbot's URL shortener service is now configurable (#430).

Bug fixes

- Fix an issue where an `AttributeError` wasn't actually raised when calling `fedmsg.publish` before initializing the Moksha hub and using a non-ZeroMQ publishing mechanism (#412).
- The default configuration was missing the `topic_prefix` key (#431).

Development Improvements

- fedmsg is now PEP-8 compliant (#414, #421, #422).
- Tox is used to enforce PEP-8, build the documentation, and run the tests with multiple versions of Python (#417).
- The test suite is now run with pytest rather than nose. (#417).
- Code coverage history is now tracked with codecov.io.

Many thanks to all our contributors for this release:

- Elan Ruusamäe
- Pravin Chaudhary
- Ralph Bean
- Jeremy Cline

0.18.4

Bugs

- Fix an issue introduced in 0.18.3 where monitoring sockets were not being created in the fedmsg relay (#433)

0.18.3

Features

- The `environment` configuration key is no longer restricted to `dev`, `stg`, and `prod`. It now must be an alphanumeric string (#406).

Bug fixes

- fedmsg-logger –json-input can now handle multi-line json (#392).
- Update the documentation on publishing to mention the `endpoints` configuration (#394).
- Start re-branding the library so it's not Fedora-specific (#391).
- Ensure fedmsg-relay doesn't run producers (#395).
- Remove keys added by datagrepper from messages retrieved from the backlog (#402).

Development Improvements

- Fix a mock used by the test suite (#405).

0.18.2

This is a security release which addresses CVE-2017-1000001.

Bug fixes

- Fixes an issue in the validation logic of the base consumer which caused child consumers to not validate the authenticity of messages (5c21cf88a).

0.18.1

Bug fixes

- Only check for STOMP messages after decoding any ZMQMessage (#393).

Development Improvements

- Remove test cases for old versions of the Python six library. fedmsg only supports six-1.9 or greater (#390).

0.18.0

Features

- Cascade IRC connections (#374).
- Get fedmsg-hub working on STOMP (#380).
- Raise the resource limit on open files for fedmsg-hub (#381).
- Add SSL support to irc bot (#386).

Bug fixes

- Return earlier when validate_signatures is turned off (#388).

Documentation Improvements

- Remove the out-dated status page from the documentation (#375).
- Make the introduction less Fedora specific (#377).
- Update the necessary dependencies in the Development section (#385).
- Document turning off validation for other buses (#387).

Development Improvements

- Turn testing Python 2.6 in Travis on (#382).

Older Changes

For older changes, consult the old changelog.

CHAPTER 2

API Guide

Developer Interface

This documentation covers the public interfaces fedmsg provides. Unless otherwise noted, all documented interfaces follow Semantic Versioning 2.0.0. If the interface you depend on is not documented here, it may change without warning in a minor release.

Python

Sending and Receiving Messages

Federated Message Bus Client API

`fedmsg.`**`init`**`(**kw)`
Initialize an instance of `fedmsg.core.FedMsgContext`.

The config is loaded with *`fedmsg.config.load_config()`* and updated by any keyword arguments. This config is used to initialize the context object.

The object is stored in a thread local as `fedmsg.__local.__context`.

`fedmsg.`**`destroy`**`(*args, **kw)`
Destroy a fedmsg context

`fedmsg.`**`publish`**`(*args, **kw)`
Send a message over the publishing zeromq socket.

```
>>> import fedmsg
>>> fedmsg.publish(topic='testing', modname='test', msg={
...     'test': "Hello World",
... })
```

The above snippet will send the message `'{test: "Hello World"}'` over the `<topic_prefix>.dev.test.testing` topic. The fully qualified topic of a message is constructed out of the following pieces:

<*topic_prefix*>.<*environment*>.<modname>.<topic>

This function (and other API functions) do a little bit more heavy lifting than they let on. If the "zeromq context" is not yet initialized, *fedmsg.init()* is called to construct it and store it as `fedmsg.__local.__context` before anything else is done.

An example from Fedora Tagger – SQLAlchemy encoding

Here's an example from fedora-tagger that sends the information about a new tag over `org.fedoraproject.{dev,stg,prod}.fedoratagger.tag.update`:

```
>>> import fedmsg
>>> fedmsg.publish(topic='tag.update', msg={
...     'user': user,
...     'tag': tag,
... })
```

Note that the *tag* and *user* objects are SQLAlchemy objects defined by tagger. They both have `.__json__()` methods which *fedmsg.publish()* uses to encode both objects as stringified JSON for you. Under the hood, specifically, `.publish` uses *fedmsg.encoding* to do this.

`fedmsg` has also guessed the module name (`modname`) of it's caller and inserted it into the topic for you. The code from which we stole the above snippet lives in `fedoratagger.controllers.root`. `fedmsg` figured that out and stripped it down to just `fedoratagger` for the final topic of `org.fedoraproject.{dev,stg,prod}.fedoratagger.tag.update`.

Shell Usage

You could also use the `fedmsg-logger` from a shell script like so:

```
$ echo "Hello, world." | fedmsg-logger --topic testing
$ echo '{"foo": "bar"}' | fedmsg-logger --json-input
```

Parameters

- **topic** (*unicode*) – The message topic suffix. This suffix is joined to the configured topic prefix (e.g. `org.fedoraproject`), environment (e.g. `prod`, `dev`, etc.), and modname.
- **msg** (*dict*) – A message to publish. This message will be JSON-encoded prior to being sent, so the object must be composed of JSON- serializable data types. Please note that if this is already a string JSON serialization will be applied to that string.
- **modname** (*unicode*) – The module name that is publishing the message. If this is omitted, `fedmsg` will try to guess the name of the module that called it and use that to produce an intelligent topic. Specifying `modname` explicitly overrides this behavior.
- **pre_fire_hook** (*function*) – A callable that will be called with a single argument – the dict of the constructed message – just before it is handed off to ZeroMQ for publication.

`fedmsg.`**`tail_messages`**(**args*, ***kw*)

Subscribe to messages published on the sockets listed in *endpoints*.

Parameters

- **topic** (*six.text_type*) – The topic to subscribe to. The default is to subscribe to all topics.
- **passive** (*bool*) – If `True`, bind to the *endpoints* sockets instead of connecting to them. Defaults to `False`.
- ****kw** – Additional keyword arguments. Currently none are used.

Yields *tuple* – A 4-tuple in the form (name, endpoint, topic, message).

Configuration

This module handles loading, processing and validation of all configuration.

The configuration values used at runtime are determined by checking in the following order:

- Built-in defaults
- All Python files in the /etc/fedmsg.d/ directory
- All Python files in the ~/.fedmsg.d/ directory
- All Python files in the current working directory's fedmsg.d/ directory
- Command line arguments

For example, if a config value does not appear in either the config file or on the command line, then the built-in default is used. If a value appears in both the config file and as a command line argument, then the command line value is used.

You can print the runtime configuration to the terminal by using the `fedmsg-config` command implemented by `fedmsg.commands.config.config()`.

`fedmsg.config.`**`load_config`**(*extra_args=None*, *doc=None*, *filenames=None*, *invalidate_cache=False*, *fedmsg_command=False*, *disable_defaults=False*)

Setup a runtime config dict by integrating the following sources (ordered by precedence):

•defaults (unless disable_defaults = True)

•config file

•command line arguments

If the `fedmsg_command` argument is False, no command line arguments are checked.

`fedmsg.config.`**`build_parser`**(*declared_args*, *doc*, *config=None*, *prog=None*)

Return the global `argparse.ArgumentParser` used by all fedmsg commands.

Extra arguments can be supplied with the *declared_args* argument.

`fedmsg.config.`**`execfile`**(*fname*, *variables*)

This is builtin in python2, but we have to roll our own on py3.

Cryptography and Message Signing

`fedmsg.crypto` - Cryptographic component of fedmsg.

Introduction

In general, we assume that 'everything on the bus is public'. Even though all the zmq endpoints are firewalled off from the outside world with iptables, we do have a forwarding service setup that indiscriminantly forwards all messages to anyone who wants them. (See `fedmsg.commands.gateway.gateway` for that service.) So, the issue is not encrypting messages so they can't be read. It is up to sensitive services like FAS to *not send* sensitive information in the first place (like passwords, for instance).

However, since at some point, services will respond to and act on messages that come across the bus, we need facilities for guaranteeing a message comes from where it *ought* to come from. (Tangentially, message consumers need a simple way to declare where they expect their messages to come from and have the filtering and validation handled for them).

There should also be a convenient way to turn crypto off both globally and locally. Justification: a developer may want to work out a bug without any messages being signed or validated. In production, certain senders may send non-critical data from a corner of Fedora Infrastructure in which it's difficult to sign messages. A consumer of those messages should be allowed to ignore validation for those and only those expected unsigned messages

Two backend methods are available to accomplish this:

- `fedmsg.crypto.x509`
- `fedmsg.crypto.gpg`

Which backend is used is configured by the *crypto_backend* configuration value.

Certificates

To accomplish message signing, fedmsg must be able to read certificates and a private key on disk in the case of the `fedmsg.crypto.x509` backend or to read public and private GnuPG keys in the came of the `fedmsg.crypto.gpg` backend. For message validation, it only need be able to read the x509 certificate or gpg public key. Exactly *which* certificates are used are determined by looking up the `certname` in the *certnames* config dict.

We use a large number of certs for the deployment of fedmsg. We have one cert per *service-host*. For example, if we have 3 fedmsg-enabled services and each service runs on 10 hosts, then we have 30 unique certificate/key pairs in all.

The intent is to create difficulty for attackers. If a low-security service on a particular box is compromised, we don't want the attacker automatically have access to the same certificate used for signing high-security service messages.

Furthermore, attempts are made at the sysadmin-level to ensure that fedmsg-enabled services run as users that have exclusive read access to their own keys. See the Fedora Infrastructure SOP for more information (including how to generate new certs/bring up new services).

Routing Policy

Messages are also checked to see if the name of the certificate they bear and the topic they're routed on match up in a *routing_policy* dict. Is the build server allowed to send messages about wiki updates? Not if the routing policy has anything to say about it.

Note: By analogy, "signature validation is to authentication as routing policy checks are to authorization."

If the topic of a message appears in the *routing_policy*, the name borne on the certificate must also appear under the associated list of permitted publishers or the message is marked invalid.

If the topic of a message does *not* appear in the *routing_policy*, two different courses of action are possible:

- If *routing_nitpicky* is set to `False`, then the message is given the green light. Our routing policy doesn't have anything specific to say about messages of this topic and so who are we to deny it passage, right?
- If *routing_nitpicky* is set to `True`, then we deny the message and mark it as invalid.

Typically, you'll deploy fedmsg with nitpicky mode turned off. You can build your policy over time as you determine what services will be sending what messages. Once deployment of fedmsg reaches a certain level of stability, you can turn nitpicky mode on for enhanced security, but by doing so you may break certain message paths that you've forgotten to include in your routing policy.

Configuration

By convention, configuration values for *fedmsg.crypto* are kept in `/etc/fedmsg.d/ssl.py`, although technically they can be kept in any config `dict` in `/etc/fedmsg.d` (or in any of the config locations checked by *fedmsg.config*).

The cryptography routines expect the following values to be defined:

- *crypto_backend*
- *crypto_validate_backends*
- *sign_messages*
- *validate_signatures*
- *ssldir*
- *crl_location*
- *crl_cache*
- *crl_cache_expiry*
- *certnames*
- *routing_policy*
- *routing_nitpicky*

For general information on configuration, see *fedmsg.config*.

Module Contents

fedmsg.crypto encapsulates standalone functions for:

- Message signing.
- Signature validation.
- Stripping crypto information for view.

See `fedmsg.crypto.x509` and `fedmsg.crypto.gpg` for implementation details.

`fedmsg.crypto.`**`init`**(***config*)
Initialize the crypto backend.

The backend can be one of two plugins:

•'x509' - Uses x509 certificates.

•'gpg' - Uses GnuPG keys.

`fedmsg.crypto.`**`sign`**(*message*, ***config*)
Insert two new fields into the message dict and return it.

Those fields are:

•'signature' - the computed message digest of the JSON repr.

•'certificate' - the base64 certificate or gpg key of the signator.

`fedmsg.crypto.`**`strip_credentials`**(*message*)
Strip credentials from a message dict.

A new dict is returned without either *signature* or *certificate* keys. This method can be called safely; the original dict is not modified.

This function is applicable using either using the x509 or gpg backends.

`fedmsg.crypto.`**`validate`**(*message*, ***config*)
Return true or false if the message is signed appropriately.

`fedmsg.crypto.`**`validate_signed_by`**(*message*, *signer*, ***config*)
Validate that a message was signed by a particular certificate.

This works much like `validate(...)`, but additionally accepts a `signer` argument. It will reject a message for any of the regular circumstances, but will also reject it if its not signed by a cert with the argued name.

Message Encoding

fedmsg messages are encoded as JSON.

Use the functions *fedmsg.encoding.loads()*, *fedmsg.encoding.dumps()*, and *fedmsg.encoding.pretty_dumps()* to encode/decode.

When serializing objects (usually python dicts) with *fedmsg.encoding.dumps()* and *fedmsg.encoding.pretty_dumps()*, the following exceptions to normal JSON serialization are observed.

- `datetime.datetime` objects are correctly converted to seconds since the epoch.
- For objects that are not JSON serializable, if they have a `.__json__()` method, that will be used instead.
- SQLAlchemy models that do not specify a `.__json__()` method will be run through *fedmsg.encoding.sqla.to_json()* which recursively produces a dict of all attributes and relations of the object(!) Be careful using this, as you might expose information to the bus that you do not want to. See *Cryptography and Message Signing* for considerations.

`fedmsg.encoding.`**`loads`**(*s*, *encoding=None*, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)
Deserialize `s` (a `str` or `unicode` instance containing a JSON document) to a Python object.

If `s` is a `str` instance and is encoded with an ASCII based encoding other than utf-8 (e.g. latin-1) then an appropriate `encoding` name must be specified. Encodings that are not ASCII based (such as UCS-2) are not allowed and should be decoded to `unicode` first.

`object_hook` is an optional function that will be called with the result of any object literal decode (a `dict`). The return value of `object_hook` will be used instead of the `dict`. This feature can be used to implement custom decoders (e.g. JSON-RPC class hinting).

`object_pairs_hook` is an optional function that will be called with the result of any object literal decoded with an ordered list of pairs. The return value of `object_pairs_hook` will be used instead of the `dict`. This feature can be used to implement custom decoders that rely on the order that the key and value pairs are decoded (for example, collections.OrderedDict will remember the order of insertion). If `object_hook` is also defined, the `object_pairs_hook` takes priority.

`parse_float`, if specified, will be called with the string of every JSON float to be decoded. By default this is equivalent to float(num_str). This can be used to use another datatype or parser for JSON floats (e.g. decimal.Decimal).

`parse_int`, if specified, will be called with the string of every JSON int to be decoded. By default this is equivalent to int(num_str). This can be used to use another datatype or parser for JSON integers (e.g. float).

`parse_constant`, if specified, will be called with one of the following strings: -Infinity, Infinity, NaN, null, true, false. This can be used to raise an exception if invalid JSON numbers are encountered.

To use a custom `JSONDecoder` subclass, specify it with the `cls` kwarg; otherwise `JSONDecoder` is used.

fedmsg.encoding.**dumps**(*self*, *o*)
Return a JSON string representation of a Python data structure.

```
>>> JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

fedmsg.encoding.**pretty_dumps**(*self*, *o*)
Return a JSON string representation of a Python data structure.

```
>>> JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

SQLAlchemy Encoding Utilities

fedmsg.encoding.sqla houses utility functions for JSONifying sqlalchemy models that do not define their own `.__json__()` methods.

Use at your own risk. *fedmsg.encoding.sqla.to_json()* will expose all attributes and relations of your sqlalchemy object and may expose information you not want it to. See *Cryptography and Message Signing* for considerations.

fedmsg.encoding.sqla.**expand**(*obj*, *relation*, *seen*)
Return the to_json or id of a sqlalchemy relationship.

fedmsg.encoding.sqla.**to_json**(*obj*, *seen=None*)
Returns a dict representation of the object.

Recursively evaluates to_json(...) on its relationships.

"Natural Language" Representation of Messages

fedmsg.meta handles the conversion of fedmsg messages (dict-like json objects) into internationalized human-readable strings: strings like `"nirik voted on a tag in tagger"` and `"lmacken commented on a bodhi update."`

The intent is to use the module 1) in the `fedmsg-irc` bot and 2) in the gnome-shell desktop notification widget. The sky is the limit, though.

The primary entry point is *fedmsg.meta.msg2repr()* which takes a dict and returns the string representation. Portions of that string are in turn produced by *fedmsg.meta.msg2title()*, *fedmsg.meta.msg2subtitle()*, and *fedmsg.meta.msg2link()*.

Message processing is handled by a list of MessageProcessors (instances of *fedmsg.meta.base.BaseProcessor*) which are discovered on a setuptools **entry-point**. Messages for which no MessageProcessor exists are handled gracefully.

The original deployment of fedmsg in *Fedora Infrastructure* uses metadata providers/message processors from a plugin called fedmsg_meta_fedora_infrastructure. If you'd like to add your own processors for your own deployment, you'll need to extend *fedmsg.meta.base.BaseProcessor* and override the appropriate methods. If you package up your processor and expose it on the `fedmsg.meta` entry-point, your new class will need to be added to the *fedmsg.meta.processors* list at runtime.

End users can have multiple plugin sets installed simultaneously.

exception fedmsg.meta.**ProcessorsNotInitialized**
Bases: exceptions.Exception

fedmsg.meta.**conglomerate**(*messages*, *subject=None*, *lexers=False*, ***config*)
Return a list of messages with some of them grouped into conglomerate messages. Conglomerate messages represent several other messages.

For example, you might pass this function a list of 40 messages. 38 of those are git.commit messages, 1 is a bodhi.update message, and 1 is a badge.award message. This function could return a list of three messages, one representing the 38 git commit messages, one representing the bodhi.update message, and one representing the badge.award message.

The `subject` argument is optional and will return "subjective" representations if possible (see msg2subjective(...)).

Functionality is provided by fedmsg.meta plugins on a "best effort" basis.

fedmsg.meta.**legacy_condition**(*cls*)

fedmsg.meta.**make_processors**(***config*)
Initialize all of the text processors.

You'll need to call this once before using any of the other functions in this module.

```
>>> import fedmsg.config
>>> import fedmsg.meta
>>> config = fedmsg.config.load_config([], None)
>>> fedmsg.meta.make_processors(**config)
>>> text = fedmsg.meta.msg2repr(some_message_dict, **config)
```

fedmsg.meta.**msg2agent**(*msg*, *processor=None*, ***config*)
Return the single username who is the "agent" for an event.

An "agent" is the one responsible for the event taking place, for example, if one person gives karma to another, then both usernames are returned by msg2usernames, but only the one who gave the karma is returned by msg2agent.

If the processor registered to handle the message does not provide an agent method, then the *first* user returned by msg2usernames is returned (whether that is correct or not). Here we assume that if a processor implements *agent*, then it knows what it is doing and we should trust that. But if it does not implement it, we'll try our best guess.

If there are no users returned by msg2usernames, then None is returned.

fedmsg.meta.**msg2avatars**(*msg*, *legacy=False*, ***config*)
Return a dict mapping of usernames to avatar URLs.

fedmsg.meta.**msg2emails**(*msg*, *legacy=False*, ***config*)
Return a dict mapping of usernames to email addresses.

fedmsg.meta.**msg2icon**(*msg*, *legacy=False*, ***config*)
Return a primary icon associated with a message.

fedmsg.meta.**msg2lexer**(*msg*, *processor=None*, ***config*)
Return a Pygments lexer able to parse the long_form of this message.

fedmsg.meta.**msg2link**(*msg*, *legacy=False*, ***config*)
Return a URL associated with a message.

fedmsg.meta.**msg2long_form**(*msg*, *legacy=False*, ***config*)
Return a 'long form' text representation of a message.

For most message, this will just default to the terse subtitle, but for some messages a long paragraph-structured block of text may be returned.

fedmsg.meta.**msg2objects**(*msg*, *legacy=False*, ***config*)
Return a set of objects associated with a message.

"objects" here is the "objects" from english grammar.. meaning, the thing in the message upon which action is being done. The "subject" is the user and the "object" is the packages, or the wiki articles, or the blog posts.

Where possible, use slash-delimited names for objects (as in wiki URLs).

fedmsg.meta.**msg2packages**(*msg*, *legacy=False*, ***config*)
Return a set of package names associated with a message.

fedmsg.meta.**msg2processor**(*msg*, ***config*)
For a given message return the text processor that can handle it.

This will raise a *fedmsg.meta.ProcessorsNotInitialized* exception if *fedmsg.meta.make_processors()* hasn't been called yet.

fedmsg.meta.**msg2repr**(*msg*, *legacy=False*, ***config*)
Return a human-readable or "natural language" representation of a dict-like fedmsg message. Think of this as the 'top-most level' function in this module.

fedmsg.meta.**msg2secondary_icon**(*msg*, *legacy=False*, ***config*)
Return a secondary icon associated with a message.

fedmsg.meta.**msg2subjective**(*msg*, *legacy=False*, ***config*)
Return a human-readable text representation of a dict-like fedmsg message from the subjective perspective of a user.

For example, if the subject viewing the message is "oddshocks" and the message would normally translate into "oddshocks commented on ticket #174", it would instead translate into "you commented on ticket #174".

fedmsg.meta.**msg2subtitle**(*msg*, *legacy=False*, ***config*)
Return a 'subtitle' or secondary text associated with a message.

fedmsg.meta.**msg2title**(*msg*, *legacy=False*, ***config*)
Return a 'title' or primary text associated with a message.

fedmsg.meta.**msg2usernames**(*msg*, *legacy=False*, ***config*)
Return a set of FAS usernames associated with a message.

fedmsg.meta.**with_processor**()

fedmsg.meta.**processors = ProcessorsNotInitialized('You must first call fedmsg.meta.make_processors(**config)',)**

class fedmsg.meta.base.**BaseConglomerator**(*processor*, *internationalization_callable*, ***conf*)
Bases: object

Base Conglomerator. This abstract base class must be extended.

fedmsg.meta "conglomerators" are similar to but different from the fedmsg.meta "processors". Where processors take a single message are return metadata about them (subtitle, a list of usernames, etc..), conglomerators take multiple messages and return a reduced subset of "conglomerate" messages. Think: there are 100 messages where pbrobinson built 100 different packages in koji – we can just represent those in a UI somewhere as a single message "pbrobinson built 100 different packages (click for details)".

This BaseConglomerator is meant to be extended many times over to provide plugins that know how to conglomerate different combinations of messages.

can_handle(*msg*, ***config*)
Return true if we should begin to consider a given message.

conglomerate(*messages*, *subject=None*, *lexers=False*, ***conf*)
Top-level API entry point. Given a list of messages, transform it into a list of conglomerates where possible.

static list_to_series(*items*, *N=3*, *oxford_comma=True*)
Convert a list of things into a comma-separated string.

```
>>> list_to_series(['a', 'b', 'c', 'd'])
'a, b, and 2 others'
>>> list_to_series(['a', 'b', 'c', 'd'], N=4, oxford_comma=False)
'a, b, c and d'
```

matches(*a*, *b*, ***config*)
Return true if message a can be paired with message b.

merge(*constituents*, *subject*, ***config*)
Given N presumably matching messages, return one merged message

classmethod produce_template(*constituents*, *subject*, *lexers=False*, ***config*)
Helper function used by *merge*. Produces the beginnings of a merged conglomerate message that needs to be later filled out by a subclass.

select_constituents(*messages*, ***config*)
From a list of messages, return a subset that can be merged

skip(*message*, ***config*)

class fedmsg.meta.base.**BaseProcessor**(*internationalization_callable*, ***config*)
Bases: object

Base Processor. Without being extended, this doesn't actually handle any messages.

Processors require that an internationalization_callable be passed to them at instantiation. Internationalization is often done at import time, but we handle it at runtime so that a single process may translate fedmsg messages into multiple languages. Think: an IRC bot that runs #fedora-fedmsg, #fedora-fedmsg-es, #fedora-fedmsg-it. Or: a twitter bot that posts to multiple language-specific accounts.

That feature is currently unused, but fedmsg.meta supports future internationalization (there may be bugs to work out).

agent = NotImplemented

avatars(*msg*, ***config*)
Return a dict of avatar URLs associated with a message.

conglomerate(*messages*, ***config*)
Given N messages, return another list that has some of them grouped together into a common 'item'.

A conglomeration of messages should be of the following form:

```
{
  'subtitle': 'relrod pushed commits to ghc and 487 other packages',
  'link': None,  # This could be something.
  'icon': 'https://that-git-logo',
  'secondary_icon': 'https://that-relrod-avatar',
  'start_time': some_timestamp,
  'end_time': some_other_timestamp,
  'human_time': '5 minutes ago',
  'usernames': ['relrod'],
  'packages': ['ghc', 'nethack', ... ],
  'topics': ['org.fedoraproject.prod.git.receive'],
  'categories': ['git'],
  'msg_ids': {
      '2014-abcde': {
          'subtitle': 'relrod pushed some commits to ghc',
          'title': 'git.receive',
```

```
            'link': 'http://...',
            'icon': 'http://...',
        },
        '2014-bcdef': {
            'subtitle': 'relrod pushed some commits to nethack',
            'title': 'git.receive',
            'link': 'http://...',
            'icon': 'http://...',
        },
    },
}
```

The telltale sign that an entry in a list of messages represents a conglomerate message is the presence of the plural `msg_ids` field. In contrast, ungrouped singular messages should bear a singular `msg_id` field.

conglomerators = None

emails(*msg*, ***config*)
Return a dict of emails associated with a message.

handle_msg(*msg*, ***config*)
If we can handle the given message, return the remainder of the topic.

Returns None if we can't handle the message.

icon(*msg*, ***config*)
Return a "icon" for the message.

lexer(*msg*, ***config*)
Return a pygments lexer that can be applied to the long_form.

Returns None if no lexer is associated.

link(*msg*, ***config*)
Return a "link" for the message.

long_form(*msg*, ***config*)
Return some paragraphs of text about a message.

objects(*msg*, ***config*)
Return a set of objects associated with a message.

packages(*msg*, ***config*)
Return a set of package names associated with a message.

secondary_icon(*msg*, ***config*)
Return a "secondary icon" for the message.

subjective(*msg*, *subject*, ***config*)
Return a "subjective" subtitle for the message.

subtitle(*msg*, ***config*)
Return a "subtitle" for the message.

title(*msg*, ***config*)

topic_prefix_re = None

usernames(*msg*, ***config*)
Return a set of FAS usernames associated with a message.

fedmsg.meta.base.**add_metaclass**(*metaclass*)
Compat shim for el7.

Replay

fedmsg.replay.**check_for_replay**(*name*, *names_to_seq_id*, *msg*, *config*, *context=None*)
Check to see if messages need to be replayed.

Parameters

- **name** (*str*) – The consumer's name.
- **names_to_seq_id** (*dict*) – A dictionary that maps names to the last seen sequence ID.
- **msg** (*dict*) – The latest message that has arrived.
- **config** (*dict*) – A configuration dictionary. This dictionary should contain, at a minimum, two keys. The first key, 'replay_endpoints', should be a dictionary that maps `name` to a ZeroMQ socket. The second key, 'io_threads', is an integer used to initialize the ZeroMQ context.
- **context** (*zmq.Context*) – The ZeroMQ context to use. If a context is not provided, one will be created.

Returns A list of message dictionaries.

Return type list

fedmsg.replay.**get_replay**(*name*, *query*, *config*, *context=None*)
Query the replay endpoint for missed messages.

Parameters

- **name** (*str*) – The replay endpoint name.
- **query** (*dict*) – A dictionary used to query the replay endpoint for messages. Queries are dictionaries with the following any of the following keys:
  - 'seq_ids': A `list` of `int`, matching the seq_id attributes of the messages. It should return at most as many messages as the length of the list, assuming no duplicate.
  - 'seq_id': A single `int` matching the seq_id attribute of the message. Should return a single message. It is intended as a shorthand for singleton `seq_ids` queries.
  - 'seq_id_range': A two-tuple of `int` defining a range of seq_id to check.
  - 'msg_ids': A `list` of UUIDs matching the msg_id attribute of the messages.
  - 'msg_id': A single UUID for the msg_id attribute.
  - 'time': A tuple of two timestamps. It will return all messages emitted in between.
- **config** (*dict*) – A configuration dictionary. This dictionary should contain, at a minimum, two keys. The first key, 'replay_endpoints', should be a dictionary that maps `name` to a ZeroMQ socket. The second key, 'io_threads', is an integer used to initialize the ZeroMQ context.
- **context** (*zmq.Context*) – The ZeroMQ context to use. If a context is not provided, one will be created.

Returns A generator that yields message dictionaries.

Return type generator

The fedmsg Protocol

fedmsg uses ZeroMQ Publish-Subscribe (PUBSUB) sockets for the messages sent by `fedmsg.publish()` and the messages received by `fedmsg.tail_messages()` or by way of the Moksha Hub-Consumer approach.

> **Warning:** The message format described below is *not* part of the public API at this time.

The published ZeroMQ message consists of a multi-part message of exactly two frames, formatted on the wire as follows:

- Frame 0: The message topic against which subscribers will perform a binary comparison.
- Frame 1: The JSON-serialized, UTF-8 encoded message.

CHAPTER 3

Contributor Guide

Contributing

If you're submitting patches to fedmsg, please observe the following:

- Check that your code doesn't break the test suite and is PEP8-compliant by running `tox` in the root of the repository.
- If you are adding new code, please write tests for them in `fedmsg/tests`.
- If your change warrants a modification to the docs in `doc/` or any docstrings in `fedmsg/` please make that modification.

I didn't sign up for this!

```
...all I wanted to do was submit a patch!
```

Don't worry. If you have a contribution but the above guidelines are too much to handle, just stop by `#fedora-apps` on freenode and let us know.

Using a virtualenv

Although you don't strictly *have* to, you should use virtualenvwrapper for isolating your development environment. It is to your benefit because you'll be able to install the latest fedmsg from a git checkout without messing with your system fedmsg install (if you have one). The instructions here will assume you are using that.

You can install it with:

```
$ sudo dnf install python-virtualenvwrapper
```

Note: If you decide not to use python-virtualenvwrapper, you can always use latest update of fedmsg in fedora. If you are doing this, simply ignore all `mkvirtualenv` and `workon` commands in these instructions. You can install fedmsg with `sudo dnf install fedmsg`.

Development Dependencies

Get:

```
$ sudo dnf install python-virtualenv libffi-devel openssl-devel \
  zeromq-devel gcc
```

Cloning the Upstream Git Repo

The source code is on github. For read-only access, simply:

```
$ git clone git://github.com/fedora-infra/fedmsg.git
```

Of course, you may want to do the usual fork and then clone pattern if you intend to submit patches/pull-requests (please do!).

Setting up your virtualenv

Create a new, empty virtualenv and install all the dependencies from pypi:

```
$ cd fedmsg
$ mkvirtualenv fedmsg
(fedmsg)$ pip install -e .[all]
```

Note: If the mkvirtualenv command is unavailable try `source /usr/bin/virtualenvwrapper.sh` on Fedora (if you do not run Fedora you might have to adjust the command a little).

Note: As discussed in the FAQ, M2Crypto requires the swig command to be available in order to build successfully. It's recommended that you install M2Crypto using your system package manager, which can be done with `dnf install m2crypto swig` on Fedora.

You should also run the tests, just to make sure everything is sane:

```
(fedmsg)$ python setup.py test
```

Try out the shell commands

Having set up your environment in the *Hacking* section above, open up three terminals. In each of them, activate your virtualenv with:

```
$ workon fedmsg
```

and in one, type:

```
(fedmsg)$ fedmsg-relay
```

In the second, type:

```
(fedmsg)$ fedmsg-tail --really-pretty
```

In the third, type:

```
(fedmsg)$ echo "Hello, world" | fedmsg-logger
```

And you should see the message appear in the `fedmsg-tail` term.

Configuration

There is a folder in the root of the upstream git checkout named `fedmsg.d/`. *`fedmsg.config`* will try to read this whenever the fedmsg API is invoked. If you're starting a new project like a consumer or a webapp that is sending fedmsg messages, you'll need to copy the `fedmsg.d/` directory to the root directory of that project. In *Deploying fedmsg*, that folder is kept in `/etc/fedmsg.d/`.

Note: Watch out: if you have a `/etc/fedmsg.d/` folder and a local `./fedmsg.d/`, fedmsg will read both. Global first, and then local. Local values will overwrite system-wide ones.

Note: The tutorial on consuming FAS messages from stg might be of further help. It was created before these instructions were written.

CHAPTER 4

Community

The source code and issue tracker are on GitHub.

Python Module Index

f

fedmsg, 25
fedmsg.commands, 13
fedmsg.config, 27
fedmsg.crypto, 27
fedmsg.encoding, 30
fedmsg.encoding.sqla, 31
fedmsg.meta, 31
fedmsg.meta.base, 33
fedmsg.replay, 36

Index

A

add_metaclass() (in module fedmsg.meta.base), 35
agent (fedmsg.meta.base.BaseProcessor attribute), 34
avatars() (fedmsg.meta.base.BaseProcessor method), 34

B

BaseCommand (class in fedmsg.commands), 13
BaseConglomerator (class in fedmsg.meta.base), 33
BaseProcessor (class in fedmsg.meta.base), 34
build_parser() (in module fedmsg.config), 27

C

can_handle() (fedmsg.meta.base.BaseConglomerator method), 33
check_for_replay() (in module fedmsg.replay), 36
conglomerate() (fedmsg.meta.base.BaseConglomerator method), 33
conglomerate() (fedmsg.meta.base.BaseProcessor method), 34
conglomerate() (in module fedmsg.meta), 31
conglomerators (fedmsg.meta.base.BaseProcessor attribute), 35

D

daemonizable (fedmsg.commands.BaseCommand attribute), 13
destroy() (in module fedmsg), 25
dumps() (in module fedmsg.encoding), 30

E

emails() (fedmsg.meta.base.BaseProcessor method), 35
execfile() (in module fedmsg.config), 27
execute() (fedmsg.commands.BaseCommand method), 13
expand() (in module fedmsg.encoding.sqla), 31
extra_args (fedmsg.commands.BaseCommand attribute), 13

F

fedmsg (module), 25
fedmsg.commands (module), 13
fedmsg.config (module), 27
fedmsg.crypto (module), 27
fedmsg.encoding (module), 30
fedmsg.encoding.sqla (module), 31
fedmsg.meta (module), 31
fedmsg.meta.base (module), 33
fedmsg.replay (module), 36

G

get_config() (fedmsg.commands.BaseCommand method), 13
get_replay() (in module fedmsg.replay), 36

H

handle_msg() (fedmsg.meta.base.BaseProcessor method), 35
hub() (in module fedmsg.commands.hub), 12

I

icon() (fedmsg.meta.base.BaseProcessor method), 35
init() (in module fedmsg), 25
init() (in module fedmsg.crypto), 29

L

legacy_condition() (in module fedmsg.meta), 32
lexer() (fedmsg.meta.base.BaseProcessor method), 35
link() (fedmsg.meta.base.BaseProcessor method), 35
list_to_series() (fedmsg.meta.base.BaseConglomerator static method), 33
load_config() (in module fedmsg.config), 27
loads() (in module fedmsg.encoding), 30
logger() (in module fedmsg.commands.logger), 12
long_form() (fedmsg.meta.base.BaseProcessor method), 35

M

make_processors() (in module fedmsg.meta), 32

matches() (fedmsg.meta.base.BaseConglomerator method), 34
merge() (fedmsg.meta.base.BaseConglomerator method), 34
msg2agent() (in module fedmsg.meta), 32
msg2avatars() (in module fedmsg.meta), 32
msg2emails() (in module fedmsg.meta), 32
msg2icon() (in module fedmsg.meta), 32
msg2lexer() (in module fedmsg.meta), 32
msg2link() (in module fedmsg.meta), 32
msg2long_form() (in module fedmsg.meta), 32
msg2objects() (in module fedmsg.meta), 32
msg2packages() (in module fedmsg.meta), 33
msg2processor() (in module fedmsg.meta), 33
msg2repr() (in module fedmsg.meta), 33
msg2secondary_icon() (in module fedmsg.meta), 33
msg2subjective() (in module fedmsg.meta), 33
msg2subtitle() (in module fedmsg.meta), 33
msg2title() (in module fedmsg.meta), 33
msg2usernames() (in module fedmsg.meta), 33

O

objects() (fedmsg.meta.base.BaseProcessor method), 35

P

packages() (fedmsg.meta.base.BaseProcessor method), 35
pretty_dumps() (in module fedmsg.encoding), 31
processors (in module fedmsg.meta), 33
ProcessorsNotInitialized, 31
produce_template() (fedmsg.meta.base.BaseConglomerator class method), 34
publish() (in module fedmsg), 25

R

replay() (in module fedmsg.commands.replay), 12

S

secondary_icon() (fedmsg.meta.base.BaseProcessor method), 35
select_constituents() (fedmsg.meta.base.BaseConglomerator method), 34
sign() (in module fedmsg.crypto), 29
skip() (fedmsg.meta.base.BaseConglomerator method), 34
strip_credentials() (in module fedmsg.crypto), 29
subjective() (fedmsg.meta.base.BaseProcessor method), 35
subtitle() (fedmsg.meta.base.BaseProcessor method), 35

T

tail() (in module fedmsg.commands.tail), 12
tail_messages() (in module fedmsg), 26
title() (fedmsg.meta.base.BaseProcessor method), 35
to_json() (in module fedmsg.encoding.sqla), 31
topic_prefix_re (fedmsg.meta.base.BaseProcessor attribute), 35

U

usernames() (fedmsg.meta.base.BaseProcessor method), 35

V

validate() (in module fedmsg.crypto), 30
validate_signed_by() (in module fedmsg.crypto), 30

W

with_processor() (in module fedmsg.meta), 33