Federated Message Bus

Release

Aug 09, 2017

Contents

Some time ago, the Fedora Infrastructure team wanted to hook all the services in Fedora Infrastructure up to send messages to one another over a message bus instead of communicating with each other in the heterogenous, "Rube-Goldberg" way they did previously.

`fedmsg` (FEDerated MeSsaGe bus) is a python package and API defining a brokerless messaging architecture to send and receive messages to and from applications. See *Overview* for a thorough introduction.

While originally specific to Fedora, the expansion of the project's name was changed away from the old "Fedora Messaging" to the current "Federated Message Bus" after it was also adopted for use in Debian's infrastructure.

Click here to see a feed of the Fedora bus. There's also a `#fedora-fedmsg` channel on the freenode network with a firehose bot echoing messages to channel to help give you a feel for what's going on.

You can find the list of available topics in Fedora's infrastructure at https://fedora-fedmsg.readthedocs.io

CHAPTER 1

Receiving Messages with Python

```
import fedmsg

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

CHAPTER 2

Receiving Messages from the Shell

```
$ fedmsg-tail --really-pretty
```

CHAPTER 3

Publishing Messages with Python

See *Development* on setting up your environment and workflow.

In a default configuration, sending a message looks like the following:

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

Note: The `endpoints.py` file should have an entry as `"<myprogram>.<myhost>": [ ... ]` where `myprogram` is the name of the program sending the message (can be `__main__` if it is a simple script) and `myhost` is the machine sending the program (corresponds to the output of `hostname -s`).

If you need to publish to a specific endpoint or need a consistent endpoint, you'll need to pass the `name` parameter and adjust the `endpoints.py` accordingly.

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

Note: The `endpoints.py` file should have an entry as `"mybus": [ ... ]`

CHAPTER 4

Publishing Messages from the Shell

```
$ echo "Hello World." | fedmsg-logger --modname=git --topic=repo.update
$ echo '{"a": 1}' | fedmsg-logger --json-input
$ fedmsg-logger --message="This is a message."
$ fedmsg-logger --message='{"a": 1}' --json-input
```

CHAPTER 5

Testimonials

- Jordan Sissel – “Cool idea, gives new meaning to open infrastructure.”
- David Gay – “It’s like I’m working with software made by people who thought about the future.”

CHAPTER 6

Community

The source for this document can be found on github. The issue tracker can be found there, too.

Almost all discussion happens in `#fedora-apps` on the freenode network. There is also a mailing list.

CHAPTER 7

Table of Contents

Overview

Description of the problem

Fedora Infrastructure is composed of a number of services (koji, fedpkg, pkgdb, etc..) some of which are maintained outside the Fedora Project and some of which were built in-house by the infrastructure team. These are strung together in a pipeline. Think "how an upstream release becomes a package update", "How a new source distribution becomes a package."

At present, many of the steps in this process require the maintainer to wait and watch for a previous step to complete. For instance once a branch of a package is successfully built in koji, the maintainer must submit their update to bodhi (See the new package process for more details).

Other progressions in the pipeline are automated. For instance, AutoQA defines a set of watchers. Most watchers are run as a cron task. Each one looks for certain events and fires off tests when appropriate.

At LinuxFest Northwest (2009), jkeating gave a talk on the problem of complexity in the Fedora infrastructure and how this might be addressed with a message bus architecture. Each service in the infrastructure depends on many of the others. Some pieces directly poke others: git (fedpkg) currently pokes AutoQA from a post-update hook. Other pieces poll others' status: koji scrapes pkgdb for package-owner relationships and email aliases.

This dense coupling of services makes changing, adding, or replacing services more complicated: commits to one project require a spidering of code changes to all the others.

How messaging might address the problem

jkeating's talk on messaging in the Fedora Instructure proposed the adoption of a unified message bus to reduce the complexity of multiple interdependent services. Instead of a service interfacing with its dependencies' implementations, it could subscribe to a *topic*, provide a callback, and respond to events.

For instance, instead of having koji scrape pkgdb on an interval for changed email addresses, pkgdb could emit messages to the `org.fedoraproject.service.pkgdb` topic whenever an account's email address changes. koji could subscribe to the same topic and provide a callback that updates its local email aliases when invoked.

In another case, the git (fedpkg) post-update hook could publish messages to the `org.fedoraproject.service.fedpkg.post-update` topic. AutoQA could subscribe to the same. Now if we wanted to enable another service to act when updates are pushed to fedpkg, that service need only subscribe to the topic and implement its own callback instead of appending its own call to fedpkg's post-update hook (instead of coupling its own implementation with fedpkg's).

A message bus architecture, once complete, would dramatically reduce the work required to update and maintain services in the Fedora infrastructure.

Other benefits

By adopting a messaging strategy for Fedora Infrastructure we could gain:

- A stream of data which we can watch and from which we can garner statistics about infrastructure activity.
- The de-coupling of services from one another.
- libnotify notifications to developers' desktops.
- jquery.gritter.js notifications to web interfaces.
  - this could be generalized to a `fedmsg.wsgi` middleware layer that injects a fedora messaging dashboard header into every page served by apps *X*, *Y*, and *Z*.
- An irc channel, `#fedora-fedmsg` that echoes every message on the bus.
- An identi.ca account, @fedora-firehose, that echoes every message on the bus.

AMQP, and 0mq

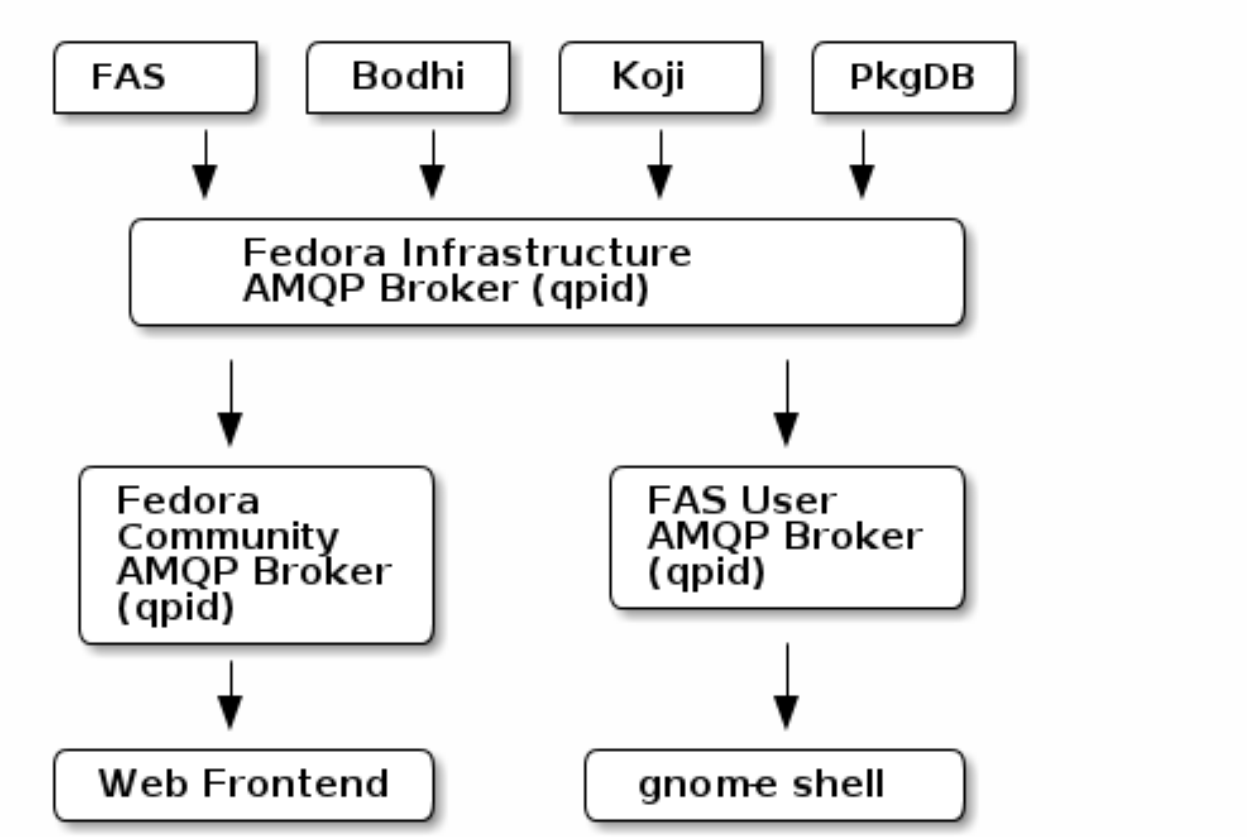
AMQP or "Broker? Damn near killed 'er!"

When discussions on the Fedora Messaging SIG began, AMQP was the choice by default. Since then members of the SIG have become attracted to an alternative messaging interface called 0mq.

Recommended reading:

- What's wrong with AMQP

The following is recreated from J5's Publish/Subscribe Messaging Proposal as an example of how Fedora Infrastructure could be reorganized with AMQP and a set of federated AMQP brokers (qpid).



The gist is that each service in the Fedora Infrastructure would have the address of a central message broker on hand. On startup, each service would connect to that broker, ask the broker to establish its outgoing queues, and begin publishing messages. Similarly, each service would ask the broker to establish incoming queues for them. The broker would handle the routing of messages based on `routing_keys` (otherwise known as *topics*) from each service to the others.

The downshot, in short, is that AMQP requires standing up a single central broker and thus a single-point-of-failure. In the author's work on narcissus I found that for even the most simple of AMQP configurations, my qpid brokers' queues would bloat over time until *pop*, the broker would fall over.

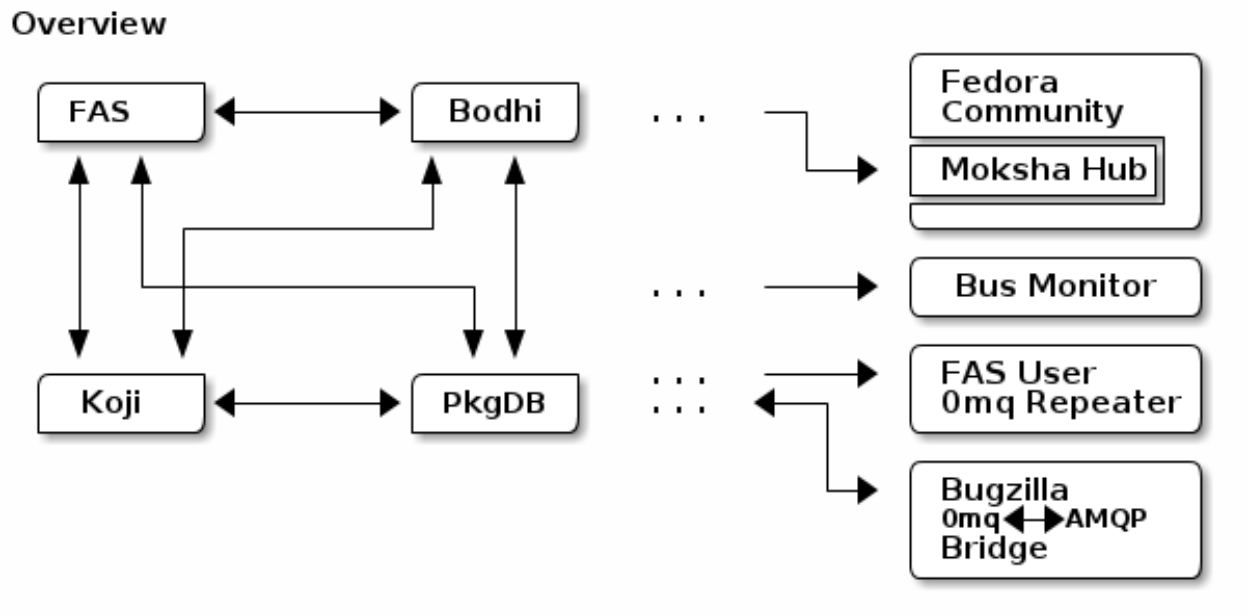
0mq or "Going for Broke(rless)"

0mq is developed by a team that had a hand in the original development of AMQP. It claims to be a number of things: an "intelligent transport layer", a "socket library that acts as a concurrency framework", and the *sine qua non* "Extra Spicy Sockets!"

Recommended reading:

- The Z-guide

The following depicts an overview of a subset of Fedora Infrastructure organized with a decentralized 0mq bus parallel to the spirit of J5's recreated diagram in the AMQP section above.



No broker. The gist is that each service will open a port and begin publishing messages ("bind to" in zmq-language). Each other service will connect to that port to begin consuming messages. Without a central broker doing all the things, 0mq can afford a high throughput. For instance, in initial tests of a 0mq-enabled moksha hub, the Fedora Engineering Team achieved a 100-fold speedup over AMQP.

Service discovery

Shortly after you begin thinking over how to enable Fedora Infrastructure to pass messages over a *fabric* instead of to a *broker*, you arrive at the problem we'll call "service discovery".

In reality, (almost) every service both *produces* and *consumes* messages. For the sake of argument, we'll talk here just about a separate *producing service* and some *consuming services*.

Scenario: the producing service starts up a producing socket (with a hidden queue) and begins producing messages. Consuming services *X*, *Y*, and *Z* are interested in this and they would like to connect.

With AMQP, this is simplified. You have one central broker and each consuming service need only know it's one address. They connect and the match-making is handled for them. With 0mq, each consuming service needs to somehow *discover* its producer(s) address(es).

There are a number of ways to address this:

- *Write our own broker*; this would not be that difficult. We could (more simply) scale back the project and write our own directory lookup service that would match consumers with their providers. This could be done in surprisingly few lines of python. This issue is that we re-introduce the sticking point of AMQP, a single point of failure.
- *Use DNS*; There is a helpful blog post on how to do this with *djbdns*. DNS is always there anyways: if DNS goes down, we have bigger things to worry about than distributing updates to our messaging topology.
- *Share a raw text file*; This at first appears crude and cumbersome:
  - Maintain a list of all *fedmsg*-enabled producers in a text file
  - Make sure that file is accessible from every consuming service.
  - Have each consuming service read in the file and connect to every (relevant) producer in the list

In my opinion, using DNS is generally speaking the most elegant solution. However, for Fedora Infrastructure in particular, pushing updates to DNS and pushing a raw text file to every server involves much-the-same workflow: *puppet*. Because much of the overhead of updating the text file falls in-line with the rest of Infrastructure work, it makes more sense to go with the third option. Better not to touch DNS when we don't have to.

That configuration is kept in `/etc/fedmsg.d/`, is read by the code in *fedmsg.config*. The config value of interest is *endpoints*.

Namespace considerations

In the above examples, the topic names are derived from the service names. For instance, pkgdb publishes messages to `org.fedoraproject.service.pkgdb*`, AutoQA presumably publishes messages to `org.fedoraproject.service.autoqa*`, and so on.

This convention, while clear-cut, has its limitations. Say we wanted to replace pkgdb whole-sale with our shiney new *threebean-db* (tm). Here, all other services are subscribed to topics that mention pkgdb explicitly. Rolling out threebean-db will require patching every other service; we find ourselves in a new flavor of the same complexity/co-dependency trap described in the first section.

The above *service-oriented* topic namespace is one option. Consider an *object-oriented* topic namespace where the objects are things like users, packages, builds, updates, tests, tickets, and composes. Having bodhi subscribe to `org.fedoraproject.object.tickets` and `org.fedoraproject.object.builds` leaves us less tied down to the current implementation of the rest of the infrastructure. We could replace *bugzilla* with *pivotal* and bodhi would never know the difference - a ticket is a ticket.

That would be nice; but there are too many objects in Fedora Infrastructure that would step on each other. For instance, Koji **tags** packages and Tagger **tags** packages; these two are very different things. Koji and Tagger cannot **both** emit events over `org.fedoraproject.package.tag.*` without widespread misery.

Consequently, our namespace follows a *service-oriented* pattern.

The scheme

Event topics will follow the rule:

```
org.fedoraproject.ENV.CATEGORY.OBJECT[.SUBOBJECT].EVENT
```

Where:

- `ENV` is one of *dev*, *stg*, or *production*.
- `CATEGORY` is the name of the service emitting the message – something like *koji*, *bodhi*, or *fedoratagger*
- `OBJECT` is something like *package*, *user*, or *tag*
- `SUBOBJECT` is something like *owner* or *build* (in the case where `OBJECT` is *package*, for instance)
- `EVENT` is a verb like *update*, *create*, or *complete*.

All 'fields' in a topic **should**:

- Be *singular* (Use *package*, not *packages*)
- Use existing fields as much as possible (since *complete* is already used by other topics, use that instead of using *finished*).

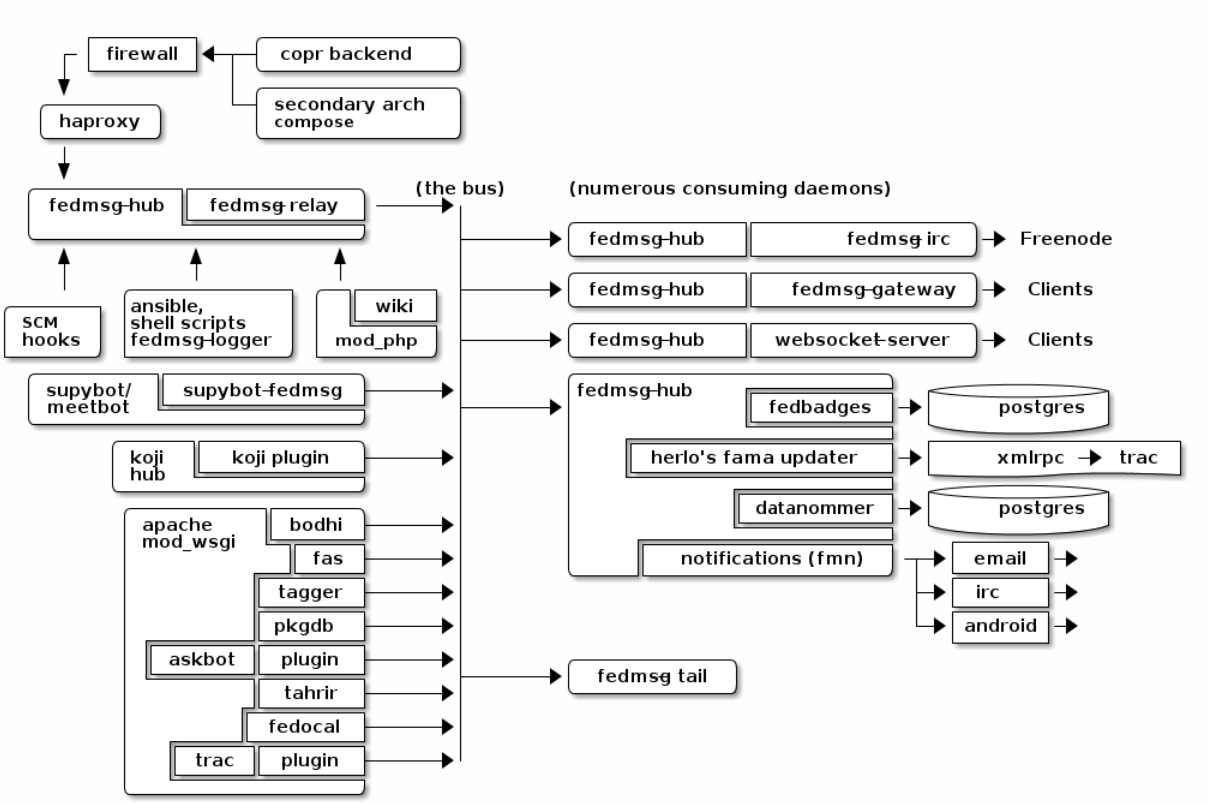
Furthermore, the *body* of messages will contain the following envelope:

- A `topic` field indicating the topic of the message.
- A `timestamp` indicating the seconds since the epoch when the message was published.

- A `msg_id` bearing a unique value distinguishing the message. It is typically of the form <YEAR>-<UUID>. These can be used to uniquely query for messages in the datagrepper web services.
- A `crypto` field indicating if the message is signed with the `X509` method or the `gpg` method.
- A `i` field indicating the sequence of the message if it comes from a permanent service.
- A `username` field indicating the username of the process that published the message (sometimes, `apache` or `fedmsg` or something else).
- Lastly, the application-specific body of the message will be contained in a nested `msg` dictionary.

Bus Topology

This is an overview of the topology of the fedmsg bus as it is deployed in Fedora Infrastructure. This document along with the status will evolve as we are able to integrate more services.



The section on the right labelled `(the bus)` is the NxM connection mesh described in *Overview*.

Frequently Asked Questions

- Is there a list of all messages?
  - Yes! See *List of Message Topics*.
- When will we be getting koji messages?

  - We have them now (we got them in late January, 2013)! Look for `org.fedoraproject.prod.buildsys...`

- When will we be getting bugzilla messages?
  - Not for a while, unfortunately. We're waiting on Red Hat to sort out issues on their end with qpidd.
- Can I deploy fedmsg at my site, even if it has nothing to do with Fedora?
  - Yes. It was designed so that all the Fedora-isms could be separated out into plugins. Get in touch with `#fedora-apps` or create a ticket if you'd like to try this out.
- `fedmsg-tail` isn't showing the same output as the bot in the `#fedora-fedmsg` IRC channel. What's up?
  - Is the formatting just different? Try the following to get those "nice" messages:

```
$ sudo dnf install python-fedmsg-meta-fedora-infrastructure
$ fedmsg-tail --terse
```

  - What you were seeing before was the raw JSON content of the messages. That is interesting to see if you want to develop tools that consume fedmsg messages. You can make those JSON blobs a little more readable with:

```
$ fedmsg-tail --really-pretty
```

  - If you *really are* seeing different messages between fedmsg-bot and fedmsg-tail, please report it in the `#fedora-apps` IRC channel or as a ticket on github.
- I tried installing the `crypto` extra but got an error about something called "swig" during the M2Crypto installation.
  - If you try to install fedmsg from the Python sources (including PyPI), then installation may fail with an error message resembling the following:

```
...
error: command 'swig' failed with exit status 1
...
```

    This happens because the M2Crypto dependency of fedmsg requires the non-Python package "swig" during it's compilation process. Since this can't be readily expressed from within the Python ecosystem, the setup script (unfortunately) just assumes that it's present.

    The easiest way to avoid this problem is to install fedmsg using your system package manager (*e.g.*, `dnf install fedmsg`). This is the recommended way to get fedmsg and has a much higher chance of working. (If it doesn't, then the package is likely broken, and the maintainer(s) will want to know about that.)

    If you have a specific reason to want to install fedmsg from source, then installing M2Crypto with you system package manager (the package is likely called something along the lines of `python-m2crypto`) should satisfy the requirement and allow fedmsg to successfully install. If you're unlucky and your distro doesn't have a packaged version of M2Crypto, then installing swig (which is just called plain `swig` in most package repositories) should allow M2Crypto to build successfully.

Development

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!).

Note: If submitting patches, you should check contributing for style guidelines.

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 for yourself*, 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.

Deploying fedmsg for yourself

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. There is plenty of long-winded explanation over at *Overview*.

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.

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()

Python API: Emitting Messages

`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.

Python API: Consuming Messages

The other side of the *Python API: Emitting Messages* document is consuming messages.

Note: Messages that you consume come with a topic and a body (dict). The content of a message can be useful! For instance, messages from FAS that come when a user edits their profile can tell you who made the change and what fields were changed. fedmsg was designed with security and message validation in mind, but its still so new that you shouldn't trust it. When building consumers, you should *always* verify information with existing webapps before acting on messages.

- **nirik>** trust, but verify. or... don't trust, and verify. ;)

Note: This document is on *how* to consume messages. But if you want to know *what* messages there are, you might check out *List of Message Topics*.

"Naive" Consuming

The most straightforward way, programmatically, to consume messages is to use *fedmsg.tail_messages()*. It is a generator that yields 4-tuples of the form `(name, endpoint, topic, message)`:

```
>>> import fedmsg
>>> import fedmsg.config

>>> # Read in the config from /etc/fedmsg.d/
>>> config = fedmsg.config.load_config()

>>> for name, endpoint, topic, msg in fedmsg.tail_messages(mute=True, **config):
...     print topic, msg  # or use fedmsg.encoding.pretty_dumps(msg)
```

The API is easy to use and should hopefully make your scripts easy to understand and maintain.

For production services, you will want to use the hub-consumer approach described further below.

Note: If you installed fedmsg from PyPI (or source), make sure that you've installed the 'consumers' extra, which may be done like so: `pip install fedmsg[consumers]`.

Note that the *`fedmsg.tail_messages()`* used to be quite inefficient; it spun in a sleep, listen, yield loop that was quite costly in IO and CPU terms. Typically, a script that used *`fedmsg.tail_messages()`* would consume 100% of a CPU. That has since be resolved by introducing the use of a `zmq.Poller`.

Note: The `fedmsg-tail` command described in *Commands* uses *`fedmsg.tail_messages()`* to "tail" the bus.

"Naive" API

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

Tail messages on the bus.

Generator that yields tuples of the form: `(name, endpoint, topic, message)`

The Hub-Consumer Approach

In contrast to the "naive" approach above, a more efficient way of consuming events can be accomplished by way of the fedmsg-hub. The drawback is that programming it is sort of indirect and declarative; it can be confusing at first.

To consume messages and do with them what you'd like, you need to:

- Write a class which extends `fedmsg.consumers.FedmsgConsumer`
- Override certain properties and methods of that class. Namely,
  - `topic` – A string used soley for constraining what messages make their way to the consumer; the consumer can *send* messages on any topic. You may use 'splats' ('*') in the topic and subscribe to `'org.fedoraproject.stg.koji.*'` to get all of the messages from koji in the staging environment.
  - `config_key` – A string used to declare a configuration entry that must be set to `True` for your consumer to be activated by the fedmsg-hub.
  - `consume` – A method that accepts a dict (the message) and contains code that "does what you would like to do".
  - `replay_name` – (optional) The name of the replay endpoint where the system should query playback in case of missing messages. It must match a service key in *replay_endpoints*.
- Register your class on the `moksha.consumer` python entry-point.

A simple example

Luke Macken wrote a simple example of a koji consumer. It's a good place to start if you're writing your own consumer.

An Example From "busmon"

In the busmon app, all messages from the hub are processed to be formatted and displayed on a client's browser. We mark them up with a pretty-print format and use pygments to colorize them.

In the example below, the `MessageColorizer` consumer simply subscribes to '*'; it will receive every message that hits it's local fedmsg-hub.

The `config_key = 'busmon.consumers.enabled'` line means that a `'busmon.consumers.enabled': True` entry must appear in the fedmsg config for the consumer to be enabled.

Here's the full example from busmon, it consumes messages from every topic, formats them in pretty colored HTML and then re-sends them out on a new topic:

```
import pygments.lexers
import pygments.formatters

import fedmsg
import fedmsg.encoding
import fedmsg.consumers

class MessageColorizer(fedmsg.consumers.FedmsgConsumer):
    topic = "*"
    jsonify = False
    config_key = 'busmon.consumers.enabled'

    def consume(self, message):
        destination_topic = "colorized-messages"

        # Just so we don't create an infinite feedback loop.
        if self.destination_topic in message.topic:
            return

        # Format the incoming message
        code = pygments.highlight(
            fedmsg.encoding.pretty_dumps(fedmsg.encoding.loads(message.body)),
            pygments.lexers.JavascriptLexer(),
            pygments.formatters.HtmlFormatter(full=False)
        ).strip()

        # Ship it!
        fedmsg.publish(
            topic=self.destination_topic,
            msg=code,
        )
```

Now, just defining a consumer isn't enough to have it picked up by the `fedmsg-hub` when it runs. You must also declare the consumer as an entry-point in your app's `setup.py`, like this:

```
setup(
    ...
    entry_points={
        'moksha.consumer': (
            'colorizer = busmon.consumers:MessageColorizer',
        ),
    },
)
```

At initialization, `fedmsg-hub` looks for all the objects registered on the `moksha.consumer` entry point and loads them

FedmsgConsumer API

DIY - Listening with Raw zeromq

So you want to receive messages without using any fedmsg libs? (say you're on some ancient system where moksha and twisted won't fly) If you can get python-zmq built, you're in good shape. Use the following example script as a starting point for whatever you want to build:

```
#!/usr/bin/env python

import json
import pprint
import zmq


def listen_and_print():
    # You can listen to stg at "tcp://stg.fedoraproject.org:9940"
    endpoint = "tcp://hub.fedoraproject.org:9940"
    # Set this to something like org.fedoraproject.prod.compose
    topic = 'org.fedoraproject.prod.'

    ctx = zmq.Context()
    s = ctx.socket(zmq.SUB)
    s.connect(endpoint)

    s.setsockopt(zmq.SUBSCRIBE, topic)

    poller = zmq.Poller()
    poller.register(s, zmq.POLLIN)

    while True:
        evts = poller.poll()  # This blocks until a message arrives
        topic, msg = s.recv_multipart()
        print topic, pprint.pformat(json.loads(msg))

if __name__ == "__main__":
    listen_and_print()
```

Just bear in mind that you don't reap any of the benefits of *fedmsg.crypto* or *fedmsg.meta*.

Note: In the example above, the topic is just `'org.fedoraproject.prod.'` and *not* `'org.fedoraproject.prod.*'`. The `*` that you see elsewhere is a Moksha convention and it is actually just stripped from the topic.

Why? The `*` has meaning in AMQP, but not zeromq. The Moksha project (which underlies fedmsg) aims to be an abstraction layer over zeromq, AMQP, and STOMP and contains some code that allows use of the `*` for zeromq, in order to make it look more like AMQP or STOMP (superficially). fedmsg (being built on Moksha) inherits this behavior even though it only uses the zeromq backend. See these comments for some discussion.

Message Encoding – JSON

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 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.

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.

Replay

class `fedmsg.replay.`**`ReplayContext`**(***config*)
Bases: `object`

`listen`()

`fedmsg.replay.`**`check_for_replay`**(*name*, *names_to_seq_id*, *msg*, *config*, *context=None*)

`fedmsg.replay.`**`get_replay`**(*name*, *query*, *config*, *context=None*)

Replay queries

Currently, a query is the union of multiple criteria. It means that the more criteria you add, the bigger the result set should become. The following criteria are supported:

seq_ids `list` - A list of integers, 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. Supported by `SqlStore`.

seq_id `int` - A single integer matching the seq_id attribute of the messages. Should return a single message. It is intended as a shorthand for singleton `seq_ids` queries. Supported by `SqlStore`.

seq_id_range `tuple` - A couple of integers defining a range of seq_id to check. Supported by `SqlStore`

msg_ids `list` - a list of UUIDs matching the msg_id attribute of the messages. Supported by `SqlStore`.

msg_id `uuid` - A single UUID for the msg_id attribute. Supported by `SqlStore`.

time `tuple` - Couple of timestamps. It will return all messages emitted inbetween. Supported by `SqlStore`

"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.

Compatibility with Other Messaging Technologies

fedmsg was originally built to specifically support zeromq as its messaging backend.

However, we also originally built it on top of moksha which is a framework that already supports not only zeromq, but also STOMP and AMQP. In 2016, we added the ability for fedmsg to send and receive messages over those protocols (and any others that moksha ends up supporting).

This lets you take advantage of the litany of apps written around the fedmsg ecosystem (FMN, fedmsg-irc, datanommer, fedimg, etc..) and run them in other messaging environments (STOMP, AMQP, ...).

We intentionally support:

- *The hub/consumer approach*. If you have a fedmsg consumer, it should be able to listen to messages over other protocols now.
- *Publishing with fedmsg.publish*. This will only work from a fedmsg consumer. It will figure out the moksha context and ask it to publish on fedmsg's behalf, thereby publishing over STOMP or AMQP, depending on configuration.

We intentionally do not support:

- *The naive listening approach*, i.e. *fedmsg.tail_messages()*. It would be much more work and its not worth it in our opinion. If we find a really good reason to implement support for it, we can always revisit this later.

Configuration

In order to get a fedmsg-hub instance working with STOMP, you should add the following configuration to */etc/fedmsg.d/base.py*:

```
# We almost always want the fedmsg-hub to be sending messages with zmq as
# opposed to amqp or stomp.  You can send with only *one* of the messaging
# backends: zeromq or amqp or stomp.  You cannot send with two or more at
# the same time.  Here, zmq is either enabled, or it is not.  If it is not,
# see the options below for how to configure stomp or amqp.
#zmq_enabled=True,

# On the other hand, if you wanted to use STOMP *instead* of zeromq, you
# could do the following...
zmq_enabled=False,
stomp_uri='broker01.example.com:61612,broker02.example.com:61612',
stomp_heartbeat=1000,
stomp_user='username',
stomp_pass='password',
stomp_ssl_crt='/path/to/ssl.crt',
stomp_ssl_key='/path/to/ssl.key',

# This is optional.
# If present, the hub will listen only to this queue for all fedmsg consumers.
# If absent, the hub will listen to all topics declared by all fedmsg consumers.
#stomp_queue='/queue/Consumer.yourqueue',

# There's usually no point in cryptographically validating messages from
# other busses.  They likely won't bear message certificates and signatures.
# Other busses usually use TLS on the connection itself for authentication
# and authorization.
validate_signatures=False,
```

Configuration

`fedmsg.config` 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
- Config file (/etc/fedmsg.d/*.py)

- Config file (./fedmsg.d/*.py)
- 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.`**`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.

`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.`**`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`.

Glossary of Configuration Values

topic_prefix `str` - A string prefixed to the topics of all outgoing messages. Typically "org.fedoraproject". Used when *`fedmsg.publish()`* constructs the fully-qualified topic for an outgoing message.

environment `str` - 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.

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 *high_water_mark* 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.

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 *endpoints*.

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]
...)
```

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

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 *Bus Topology* and *Commands* for more information.

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 `dict` - A dict 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. It defaults to the empty dict, `{}`.

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*.

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 `dict` - A mapping of `modname` values 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 `str` — the name of the method used to publish the messages on IRC. Valid values are 'msg' and 'notice', the latter being the default.

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.

List of Message Topics

A list of topics for Fedora Infrastructure messages can be found at https://fedora-fedmsg.readthedocs.org/

Note: proposal is a now out-moded document, but is kept here for historical purposes.

Python Module Index

f

Index

L

M

O

P

R

S

T

U

V

W

Z