Salmon Documentation

Release 3.0.0

Matt Molyneaux

Dec 31, 2017

Contents

Salmon is a pure Python mail server designed to create robust and complex mail applications in the style of modern web frameworks. Salmon is designed to sit behind a traditional mail server in the same way a web application sits behind Apache or Nginx. It has all the features of a web application stack (templates, routing, handlers, state machine) and plays well with other libraries, such as Django and SQLAlchemy.

Salmon has been released uner the GNU GPLv3, as published by the FSF.

CHAPTER 1

Features

Salmon supports running in many contexts for processing mail using the best technology currently available. Since Salmon is aiming to be a modern mail server and Mail processing framework, it has some features you don't find in any other Mail server.

- Written in portable Python that should run on almost any Unix server.
- Handles mail in almost any encoding and format, including attachments, and canonicalizes them for easier processing.
- Sends nearly pristine clean mail that is easier to process by other receiving servers.
- Properly decodes internationalized mail into Python unicode, and translates Python unicode back into nice clean ascii and/or UTF-8 mail.
- Supports working with Maildir queues to defer work and distribute it to multiple machines.
- Can run as an non-root user on privileged ports to reduce the risk of intrusion.
- Salmon can also run in a completely separate virtualenv for easy deployment.
- A flexible and easy to use routing system lets you write stateful or state*less* handlers of your email.
- Helpful tools for unit testing your email applications with nose, including spell checking with PyEnchant.
- Ability to use Jinja2 or Mako templates to craft emails including the headers.
- Easily configurable to use alternative sending and receiving systems, database libraries, or any other systems you need to talk to.
- Yet, you don't *have* to configure everything to get stated. A simple `salmon gen` command lets you get an application up and running quick.
- Finally, many helpful commands for general mail server debugging and cleaning.

CHAPTER 2

Installing

```
pip install salmon-mail
```

CHAPTER 3

Project Information

Project documentation can be found here

3.1 Fork

Salmon is a fork of Lamson. In the summer of 2012 (2012-07-13 to be exact), Lamson was relicensed under a BSD variant that was revokable. The two clauses that were of most concern:

```
4. Contributors agree that any contributions are owned by the copyright holder
and that contributors have absolutely no rights to their contributions.

5. The copyright holder reserves the right to revoke this license on anyone who
uses this copyrighted work at any time for any reason.
```

I read that to mean that I could make a contribution but then have said work denied to me because the orginal author didn't like the colour of my socks. So I went and found the latest version that was available under the GNU GPL version 3.

Salmon is an anagram of Lamson, if you hadn't worked it out already.

3.2 Source

You can find the source on GitHub:

https://github.com/moggers87/salmon

3.3 Status

Salmon has just had some major changes to modernise the code-base. The main APIs should be compatible with releases prior to 3.0.0, but there's no guarantee that older applications won't need changes.

Python versions supported are: 2.7, 3.5, and 3.6.

See the CHANGELOG for more details on what's changed since version 2.

3.4 License

Salmon is released under the GNU GPLv3 license, which can be found here

3.5 Contributing

Pull requests and issues are most welcome.

I will not accept code that has been submitted for inclusion in the original project due to the terms of its new licence.

CHAPTER 4

Testing

The Salmon project needs unit tests, code reviews, coverage information, source analysis, and security reviews to maintain quality. If you find a bug, please take the time to write a test case that fails or provide a piece of mail that causes the failure.

If you contribute new code then your code should have as much coverage as possible, with a minimal amount of mocking.

4.1 Security

Salmon follows the same security reporting model that has worked for other open source projects: If you report a security vulnerability, it will be acted on immediately and a fix with complete full disclosure will go out to everyone at the same time. It's the job of the people using Salmon to keep track of security relate problems.

Additionally, Salmon is written in as secure a manner as possible and assumes that it is operating in a hostile environment. If you find Salmon doesn't behave correctly given that constraint then please voice your concerns.

CHAPTER 5

Development

Salmon is written entirely in Python and runs on Python 2.7 with experimental support for Python 3. It should hopefully run on any platform that supports Python and has Unix semantics.

If you find yourself lost in source code, just yell.

PEP-8 should be followed where possible, but feel free to ignore the 80 character limit it imposes (120 is a good marker IMO).

Contents:

5.1 Getting Started

5.1.1 Setup

Install Salmon from PyPI:

```
$ pip install [--user] salmon-mail
```

Now run the `gen` command to create the basic layout of your first project:

```
$ salmon gen myproject
```

Then change directory to `myproject`

Warning: Users of older versions of Salmon should note that the project template now uses LMPTReceiver as its default

5.1.2 Handlers

Handlers are how your application will process incoming mail. Open `app/handlers/sample.py` and you'll see the following:

```
from salmon.routing import route, route_like


@route("(address)@(host)", address=".+")
def START(message, address=None, host=None):
    return NEW_USER


@route_like(START)
def NEW_USER(message, address=None, host=None):
    return NEW_USER


@route_like(START)
def END(message, address=None, host=None):
    return START
```

Each handler returns the next handler for that sender. `START` is the default handler for senders that Salmon doesn't know about. This *state* is stored in memory by default.

Let's start up a server and see how it all works:

```
$ salmon start
$ salmon status
Salmon running with PID 4557
```

If you look at `logs/salmon.log`, you'll see various start-up messages from Salmon.

Now send an email to our server:

```
$ telnet localhost 8823
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
220 localhost Salmon Mail router LMTPD, version 3
MAIL FROM: sender@example.com
250 Ok
RCPT TO: rcpt@example.com
250 Ok
DATA
354 End data with <CR><LF>.<CR><LF>
Hello
.
250 Ok
QUIT
221 Bye
Connection closed by foreign host.
```

Check `logs/salmon.log` and you'll see the following lines:

```
2016-01-11 00:49:49,947 - root - DEBUG - Message received from Peer: ('127.0.0.1',␣
↪38150), From: 'sender@example.com', to To ['rcpt@example.com'].
2016-01-11 01:00:49,949 - routing - DEBUG - Matched 'rcpt@example.com' against START.
```

```
2016-01-11 01:00:49,949 - sample_app - INFO - START: rcpt@example.com
2016-01-11 01:00:49,950 - routing - DEBUG - Message to rcpt@example.com was handled␣
↪by app.handlers.sample.START
```

If you send the message again you'll see this:

```
2016-01-11 01:01:36,486 - root - DEBUG - Message received from Peer: ('127.0.0.1',␣
↪54628), From: 'sender@example.com', to To ['rcpt@example.com'].
2016-01-11 01:01:36,487 - routing - DEBUG - Matched 'rcpt@example.com' against NEW_␣
↪USER.
2016-01-11 01:01:36,488 - routing - DEBUG - Message to rcpt@example.com was handled␣
↪by app.handlers.sample.NEW_USER
```

As the `NEW_USER` handler returns itself, every message from "sender@example" will now be processed by `NEW_USER`

Once you're done, stop the server:

```
$ salmon stop
Stopping processes with the following PID files: ['./run/stmp.pid']
Attempting to stop salmon at pid 4557
```

5.1.3 Configuration

By deafult, all configuration happens in `config/`

boot.py

This file is used by Salmon during start-up to configure the daemon with various things, such as starting the `LMTPReceiver`. It's a bit like the `wsgi.py` file that Python web apps have. If you want to use a different boot module, you can specify it with the `--boot` argument. E.g. to use `myapp/othermodule.py`, do:

```
salmon start --boot myapp.othermodule
```

testing.py

Just like `boot.py`, except for testing. You can specify `--boot config.testing` when starting Salmon to try it out.

logging.conf and test_logging.conf

Standard Python logging configuration files. See Python's documentation for more details.

settings.py

This file contains generic settings used by the rest of your application, e.g. which port the receiver should listen to. The default settings module is `config.settings`

You can specify a different settings module via the environment variable `SALMON_SETTINGS_MODULE`:

```
SALMON_SETTINGS_MODULE="myapp.othersettings" salmon start
```

5.1.4 Deploying

Salmon is best deployed behind another mailserver such as Postfix or Sendmail - much in the same way as you host a WSGI application behind Apache or Nginx.

As seen above, a new Salmon project will start a LMTP server that listens on `localhost:8823`. You can go into `config/settings.py` and change the host and port Salmon uses. You can also switch out `LMTPReceiver` for `SMTPReceiver` if you require Salmon to use SMTP instead.

Warning: Due to the way Salmon has been implemented it is better suited as a LMTP server than a SMTP server. `SMTPReceiver` is unable to handle multiple recipients in one transaction as it doesn't implement the nessessary features to properly implement this part of the SMTP protocol. This is a compromise `SMTPReceiver` makes in order to allow users more freedom in what they do in their handlers.

`LMTPReceiver` is unaffected by this issue and implements the LMTP protocol fully.

5.2 Routing

Routing in Salmon works via two mechanisms: the `@route` decorator and a finite state machine.

The `@route` decorator uses the recipient to determine which handlers match the message - in a similar way to how web frameworks configure URL handlers.

The finite state machine uses the sender as a key to keep track of which of the matched handlers the message should be given to. For example, a mailing list application might have two handlers with same `@route`, one handler for those who are subscribed and another for those who are not.

Here's a brief overview of how Salmon decides which of your application's handlers should process the message:

1. Match all handlers whose `@route` decorator matches the `to` header
2. Iterate over these and call the following
   (a) any handlers that have been marked as `@stateless`
   (b) the first (and only) stateful handler. If it returns a handler reference, the state for that sender will be updated.
3. If no valid handlers were found, the message is sent to the undeliverable queue

5.2.1 Using Routes

The *route* decorator takes a regex pattern as its first argument and then capture groups as keyword arguments:

```
from salmon.routing import route


@route("(list_name)-(action)@(host)",
    list_name="[a-z]+", action="[a-z]+", host="example\.com")
def START(message, list_name=None, action=None, host=None)
    ....
```

For example, a message to `salmon-subscribe@example.com` would match this handler, but a message to `salmon@example.com` would not - even if `START` was our only handler.

It's quite usual to multiple handlers decorated with the same `route` - we'll cover why in the next section. To save typing, you can have your handler routed exactly like another:

```
from salmon.routing import route_like


@route_like(START):
def CONFIRM(message, list_name=None, action=None, host=None):
    ....
```

Again, a message to `salmon-subscribe@example.com` would match this handler, but a message to `salmon@example.com` would not. How to control which handler out of the two is ultimately used to process a message is discussed in the next section.

5.2.2 The Finite State Machine

The finite state machine is how Salmon knows where to process a message, even when multiple handlers have routes that match the recipient. Before we explain how that is done, let's look at how to control the finite state machine.

First of all, let's flesh out the examples from the previous section. These examples will call some functions defined in `myapp` which we won't define as how they work is not important.:

```
from salmon.routing import route, route_like
from myapp.utils import (
    confirmed_checker,  # returns True or False
    confirm_sender,  # adds sender to subsciber list
    send_confirmation,  # sends a confirmation email
    post_message,  # posts a message to the given mailing list
)


@route("(list_name)-(action)@(host)",
    list_name="[a-z]+", action="[a-z]+", host="example\.com")
def START(message, list_name=None, action=None, host=None)
    if action == "subscribe" and not confirmed_checker(message.from):
        send_confirmation(message.from)
        return CONFIRM
    elif action == "post":
        post_message(message, list_name)
        return
    else:
        # unknown action
        return


@route_like(START):
def CONFIRM(message, list_name=None, action=None, host=None):
    confirm_sender(message.form)
    return START
```

When a message from a previously unknown sender is received, it will be matched against a `START` handler with the correct `route`. In our example, if `action` is `"subscribe"` then the handler returns `CONFIRM` - which is another handler. The next time a message from this sender is received, the `CONFIRM` handler will process the message and the state will return to `START` (as `CONFIRM` always returns `START`).

Note: The `CONFIRM` handler wouldn't reset the state to `START` in a real application, but examples have been kept short to make them easier to understand.

State storage in Salmon is controlled by encoding the current module and sender to a string, then using that string as a key for a `dict`-like object that stores the state as the value for that key. For example, the state storage for our application might look like this:

```
>>> from salmon.routing import Router
>>> print(Router.STATE_STORE.states)
{
    "['myapp', 'user1@example.com']": <function CONFIRM at 0x7f64194fa320>,
    "['myapp', 'user2@example.com']": <function START at 0x7f64194fa398>
}
```

Stateless Processing

If you don't require states for one or more of your handlers, the decorator *`stateless()`* will make sure the state machine is completely bypassed on the way in (but you can still return handles to affect the sender's state):

```
from salmon.routing import stateless, route


@route("admin@example.com")
@stateless
def ADMINS(message):
    # forward the email to admins
    ....
```

Implementing State Storage

The default state storage *`MemoryStorage`* is only intended for testing as it only stores state in memory - states will be lost. For small installations, *`ShelveStorage`* will save state to disk and be performant enough. Add the following lines to your `boot.py` to use it:

```
from myapp.models import ShelveStorage
Router.STATE_STORAGE = ShelveStorage()
```

Larger installations will be required to write their own state storage. Any popular database that can provide some sort of atomic get and set should be capable. For example, Django's ORM could be used:

```
# in your models.py
from django.db import models
from salmon.routing import StateStorage, ROUTE_FIRST_STATE


# this model is incomplete, but should give you a good start
class SalmonState(models.Model):
    key = models.CharField()
    sender = models.CharField()
    state = models.CharField()


class DjangoStateStorage(StateStorage):
```

```
    def get(self, key, sender):
        try:
            state = SalmonState.objects.get(key=key, sender=sender)
            return state.state
        except SalmonState.DoesNotExist:
            return ROUTE_FIRST_STATE

    def set(self, key, sender, state):
        SalmonState.objects.update_or_create(
            key=key, sender=sender, kwargs={"state": state}
        )

    def clear(self):
        SalmonState.objects.all().delete()


# at the end of boot.py
from myapp.models import DjangoStateStorage
Router.STATE_STORAGE = DjangoStateStorage()
```

5.3 Relaying

Mail servers don't just receive mail, they also send mail too. Salmon can do all the required relaying itself, but better performance you might want to use your frontend mailserver to do this for you.

5.3.1 Creating Relay Objects

By default, a *Relay* object expects to find a mailserver on IP `127.0.0.1`, port `25`. The `host` and `port` keyword arguments control this:

```
# probably in your boot.py
from salmon.server import Relay
other_host_relay = Relay(host="example.com", port=123)
```

You can also specify other options such as username, password, and encryption options. See *salmon.server.Relay* for more information.

If you wish to do do all the relaying in Salmon and not delegate to another mailserver, simply set `host` to `None`:

```
resolving_relay = Relay(host=None)
```

This will mean that the MX host for the recipient of the message will be used for delivery.

5.3.2 Creating Responses

Creating responses with HTML and plaintext parts is quite common, so Salmon has the *respond()* function to render via templates:

```
from salmon.view import respond

from salmon.view import respond

variables = {"user": "user1", ...}
```

```
message = respond(variables,
    Body="plaintext_template.txt",
    Html="html_template.html",
    To="me@example.com",
    From="you@example.com",
    Subject="Test")
```

`plaintext_template.txt` and `html_template.html` should be paths that your template engine can find and load. Keyword arguments other than `Body` and `Html` will be passed directly to *`MailResponse`*. Keyword arguments will also be formatted with the contents of variables:

```
>>> message = respond(variables, Subject="Hello %(user)s", ...)
>>> print(message["Subject"])
Hello user1
```

Salmon needs to be configured to use a template engine:

```
# in your boot.py
from salmon import view
from jinja2 import Environment, FileSystemLoader

template_path = "/path/to/templates/"
view.LOADER = Environment(loader=FileSystemLoader(template_path))
```

Note: You don't have to use Jinja 2, but whatever you set `salmon.view.LOADER` to it must have a method *get_template*' which must return an object with the method `render`. Mako and Django template engines have classes that implement these methods. Refer to their documentation for more information.

5.3.3 Delivery

Once you have a *`MailResponse`* object ready to send and a *`Relay`* object, delivery is quite simple:

```
new_message = MailResponse()
my_relay.deliver(new_message)
```

Note: If you've `host` to `None`, be sure to have something in place to catch exceptions and retry.

You can also override `To` and `Form` too:

```
my_relay.deliver(new_message, To="someone@example.com", From="another@example.com")
```

5.4 Mail Objects

`MailRequest` and *`MailResponse`* objects are two ways that Salmon represents emails. They provide a simplified interface to Python's own `email` package.

5.4.1 MailRequest

`MailRequest` objects are given to your message handlers when a new email comes in.

To/From properties

`To` and `From` are populated by the `RCPT TO` and `MAIL FROM` commands issued by the sender to Salmon. If you're using *`QueueReceiver`*, these properties will be `None`.

Headers

Headers are accessed a dict-like interface:

```
>>> print(message["Subject"])
My Subject
```

Headers are also case insensitive:

```
>>> print(message["Subject"] == message["sUbJeCt"])
True
```

Methods `keys` and `items` are also supported:

```
message.keys()  # ["To", "From", "Subject"]
message.items()  # [("To", "me@example.com"), ...]
```

Note: Emails can contain multiple headers with the same name. This is quite common with headers such as `Received`, but is completely valid for any header. Be aware of this when iterating over header names from the `keys` method!

Headers can be set too:

```
>>> message["New-Header"] = "My Value"
>>> print(message["New-Header"])
My Value
```

Warning: When headers are added this way, any previous values will be overwritten. This should be no surprise to new users, but might trip up users of Python's `email` package.

Bodies

The `body` property isn't that smart, it just returns the body of the first MIME part of the email. For emails that only have one part or are non-MIME emails this is fine, but there's no guarantee what you'll end up with if your email is a multipart message.

For MIME emails, call the `walk` method to iterate over each part:

```
>>> for part in message.walk():
...     # each part is an instance of MimeBase
...     print("This is a %s part" % part["Content-Type"])
This is a multipart/alternative part
This is a text/html part
This is a text/plain part
```

See *MailBase* for more details.

Detecting Bounce Emails

Detecting bounced emails is quite important - especially if you're sending as well as receiving:

```
>>> if message.is_bounce():
...     print("Message is a bounced email!")
```

`is_bounce` also takes a `threshold` argument that can be used to fine-tune bounce detection:

```
>>> if message.is_bounce(0.5):
...   print("I'm more certain that this is a bounced email than before!")
```

Python Email-like API

If you require an API that is more like Python's `email` package, then the `base` property holds a reference to the corresponding *MailBase* object:

```
mail_base = message.base
```

5.4.2 MailResponse

MailResponse objects can be created to send responses via *salmon.server.Relay*. They can either be created directly:

```
from salmon.mail import MailResponse

msg_html = "<html><body>Hello!</body></html>"
msg_txt = "Hello!"
message = MailResponse(
    Body=msg_txt,
    Html=msg_html,
    To="me@example.com",
    From="you@example.com",
    Subject="Test")
```

Or via *salmon.view.respond()*:

```
from salmon.view import respond

variables = {"user": "user1", ...}
message = respond(variables,
    Body="plaintext_template.txt",
    Html="html_template.html",
    To="me@example.com",
    From="you@example.com",
    Subject="Test")
```

Headers and accessing a Python Email-like API are the same as they are for *MailRequest*.

Attachments

Attachments can be added via the `attach` method:

```
filename = "image.jpg"
file = open(filename, "r")
message.attach(filename=filename, content_type="image/jpeg", data=file.read())
```

5.4.3 MailBase

MailBase contains most of the logic behind *MailRequest* and *MailResponse*, but is less user-friendly as it exposes more of what an email can actually do.

Headers

Headers are accessed by the same dict-like interface as *MailRequest* and *MailResponse*. It also has some additional methods for dealing with multiple headers with the same name.

To fetch all values of a given header name, use the `get_all` method:

```
>>> print(mail_base.get_all("Received"))
["from example.com by localhost...", "from localhost by..."]
>>> print(mail_base.get_all("Not-A-Real-Header"))
[]
```

To add a multiple headers with the same name, use the `append_header` method:

```
>>> print(mail_base.keys())
["To", "From", "Subject"]
>>> mail_base.append_header("Subject", "Another subject header")
>>> print(mail_base.keys())
["To", "From", "Subject", "Subject"]
```

Warning: Be cautious when using this feature, especially with headers that will be displayed to the user such as Subject. There's no telling what email clients will do if presented with multiple headers like this. This feature is better suited to machine read headers such as Received.

Content Encoding

The `content_encoding` property contains the parsed contents of various content encoding headers:

```
>>> print(mail_base["Content-Type"])
text/html; charset="us-ascii"
>>> print(mail_base.content_encoding["Content-Type"])
("text/html", {"charset": "us-ascii"})
```

Content encoding headers can also be set via this property:

```
>>> ct = ("text/html", {"charset": "utf-8"}
>>> mail_base.content_encoding["Content-Type"] = ct
>>> print(mail_base["Content-Type"])
text/html; charset=uft-8
```

Body

The `body` property returns the fully decoded payload of a MIME part. In the case of a "text/*" part this will be decoded fully into a Unicode object, otherwise it will only be decoded into bytes.

Accessing Python `email` API

As Salmon builds upon Python's `email` API, the underlying `email.message.Message` instance is available via the `mime_part` property:

```
email_obj = mail_base.mime_part
```

Thus, if you don't want to bother with all the nice things Salmon does for you in your handlers, you can bypass all that loveliness quite easily:

```
@route_like(START)
def PROCESS(message, **kwargs):
    # grab Message object from incoming message
    email_obj = message.mail_base.mime_part
```

5.5 salmon package

5.5.1 Subpackages

salmon.handlers package

Submodules

salmon.handlers.forward module

Implements a forwarding handler that will take anything it receives and forwards it to the relay host. It is intended to use with the salmon.routing.RoutingBase.UNDELIVERABLE_QUEUE if you want mail that Salmon doesn't understand to be delivered like normal. The Router will dump any mail that doesn't match into that queue if you set it, and then you can load this handler into a special queue receiver to have it forwarded on.

BE VERY CAREFUL WITH THIS. It should only be used in testing scenarios as it can turn your server into an open relay if you're not careful. You are probably better off writing your own version of this that knows a list of allowed hosts your machine answers to and only forwards those.

`salmon.handlers.forward.`**`START`**(*message*, **args*, ***kw*)
Forwards every mail it gets to the relay. BE CAREFUL WITH THIS.

salmon.handlers.log module

Implements a simple logging handler that's actually used by the salmon log command line tool to run a logging server. It simply takes every message it receives and dumps it to the logging.debug stream.

`salmon.handlers.log.`**`START`**(*message*, **args*, ***kw*)
This is stateless and handles every email no matter what, logging what it receives.

salmon.handlers.queue module

Implements a handler that puts every message it receives into the run/queue directory. It is intended as a debug tool so you can inspect messages the server is receiving using mutt or the salmon queue command.

`salmon.handlers.queue.`**`START`**(*message*, **args*, ***kw*)

@stateless and routes however handlers.log.START routes (everything). Has @nolocking, but that's alright since it's just writing to a Maildir.

Module contents

Salmon comes with a few useful handlers that you can add to your salmon.routing configuration. Most of them are implemented as stateless handlers, but some will require you to read the documentation and configure before you can use them.

5.5.2 Submodules

salmon.bounce module

Bounce analysis module for Salmon. It uses an algorithm that tries to simply collect the headers that are most likely found in a bounce message, and then determine a probability based on what it finds.

`class` `salmon.bounce.`**`BounceAnalyzer`**(*headers*, *score*)

Bases: `object`

BounceAnalyzer collects up the score and the headers and gives more meaningful interaction with them. You can keep it simple and just use is_hard, is_soft, and probable methods to see if there was a bounce. If you need more information then attributes are set for each of the following:

- primary_status – The main status number that determines hard vs soft.
- secondary_status – Advice status.
- combined_status – the 2nd and 3rd number combined gives more detail.
- remote_mta – The MTA that you sent mail to and aborted.
- reporting_mta – The MTA that was sending the mail and has to report to you.
- diagnostic_codes – Human readable codes usually with info from the provider.
- action – Usually 'failed', and turns out to be not too useful.
- content_parts – All the attachments found as a hash keyed by the type.
- original – The original message, if it's found.
- report – All report elements, as salmon.encoding.MailBase raw messages.
- notification – Usually the detailed reason you bounced.

Initializes all the various attributes you can use to analyze the bounce results.

`error_for_humans`()

Constructs an error from the status codes that you can print to a user.

`is_hard`()

Tells you if this was a hard bounce, which is determined by the message being a probably bounce with a primary_status greater than 4.

is_soft()
Basically the inverse of is_hard()

probable(*threshold=0.3*)
Determines if this is probably a bounce based on the score probability. Default threshold is 0.3 which is conservative.

class salmon.bounce.**bounce_to**(*soft=None*, *hard=None*)
Bases: object

Used to route bounce messages to a handler for either soft or hard bounces. Set the soft/hard parameters to the function that represents the handler. The function should take one argument of the message that it needs to handle and should have a route that handles everything.

WARNING: You should only place this on the START of modules that will receive bounces, and every bounce handler should return START. The reason is that the bounce emails come from *mail daemons* not the actual person who bounced. You can find out who that person is using message.bounce.final_recipient. But the bounce handler is *actually* interacting with a message from something like MAILER-DAEMON@somehost.com. If you don't go back to start immediately then you will mess with the state for this address, which can be bad.

salmon.bounce.**detect**(*msg*)
Given a message, this will calculate a probability score based on possible bounce headers it finds and return a salmon.bounce.BounceAnalyzer object for further analysis.

The detection algorithm is very simple but still accurate. For each header it finds it adds a point to the score. It then uses the regex in BOUNCE_MATCHERS to see if the value of that header is parsable, and if it is it adds another point to the score. The final probability is based on how many headers and matchers were found out of the total possible.

Finally, a header will be included in the score if it doesn't match in value, but it WILL NOT be included in the headers used by BounceAnalyzer to give you meanings like remote_mta and such.

Because this algorithm is very dumb, you are free to add to BOUNCE_MATCHERS in your boot files if there's special headers you need to detect in your own code.

salmon.bounce.**match_bounce_headers**(*msg*)
Goes through the headers in a potential bounce message recursively and collects all the answers for the usual bounce headers.

salmon.commands module

salmon.commands.**blast_command**(*parser*)
Given a Maildir, this command will go through each email and blast it at your server. It does nothing to the message, so it will be real messages hitting your server, not cleansed ones.

salmon.commands.**cleanse_command**(*parser*)
Uses Salmon mail cleansing and canonicalization system to take an input Maildir (or mbox) and replicate the email over into another Maildir. It's used mostly for testing and cleaning.

salmon.commands.**function**(*parser*)
Used as a testing sendmail replacement for use in programs like mutt as an MTA. It reads the email to send on the stdin and then delivers it based on the port and host settings.

salmon.commands.**gen_command**(*parser*)
Generates various useful things for you to get you started.

salmon.commands.**log_command**(*parser*)
Runs a logging only server on the given hosts and port. It logs each message it receives and also stores it to the run/queue so that you can make sure it was received in testing.

salmon.commands.**main**()
Salmon script entry point

salmon.commands.**queue_command**(*parser*)
Lets you do most of the operations available to a queue.

salmon.commands.**routes_command**(*parser*)
Prints out valuable information about an application's routing configuration after everything is loaded and ready to go. Helps debug problems with messages not getting to your handlers. Path has the search paths you want separated by a ':' character, and it's added to the sys.path.

salmon.commands.**send_command**(*parser*)
Sends an email to someone as a test message. See the sendmail command for a sendmail replacement.

salmon.commands.**sendmail_command**(*parser*)
Used as a testing sendmail replacement for use in programs like mutt as an MTA. It reads the email to send on the stdin and then delivers it based on the port and host settings.

salmon.commands.**start_command**(*parser*)
Runs a salmon server out of the current directory

salmon.commands.**status_command**(*parser*)
Prints out status information about salmon useful for finding out if it's running and where.

salmon.commands.**stop_command**(*parser*)
Stops a running salmon server

salmon.confirm module

Confirmation handling API that helps you get the whole confirm/pending/verify process correct. It doesn't implement any handlers, but what it does do is provide the logic for doing the following:

- Take an email, put it in a "pending" queue, and then send out a confirm email with a strong random id.
- Store the pending message ID and the random secret someplace for later verification.
- Verify an incoming email against the expected ID, and get back the original.

You then just work this into your project's state flow, write your own templates, and possibly write your own storage.

class salmon.confirm.**ConfirmationEngine**(*pending_queue*, *storage*)
Bases: object

The confirmation engine is what does the work of sending a confirmation, and verifying that it was confirmed properly. In order to use it you have to construct the ConfirmationEngine (usually in settings module) and you write your confirmation message templates for sending.

The primary methods you use are ConfirmationEngine.send and ConfirmationEngine.verify.

The pending_queue should be a string with the path to the salmon.queue.Queue that will store pending messages. These messages are the originals the user sent when they tried to confirm.

Storage should be something that is like ConfirmationStorage so that this can store things for later verification.

cancel(*target*, *from_address*, *expect_secret*)
Used to cancel a pending confirmation.

clear()
Used in testing to make sure there's nothing in the pending queue or storage.

delete_pending(*pending_id*)
Removes the pending message from the pending queue.

get_pending(*pending_id*)
Returns the pending message for the given ID.

make_random_secret()
Generates a random uuid as the secret, in hex form.

push_pending(*message*)
Puts a pending message into the pending queue.

register(*target*, *message*)
Don't call this directly unless you know what you are doing. It does the job of registering the original message and the expected confirmation into the storage.

send(*relay*, *target*, *message*, *template*, *vars*)
This is the method you should use to send out confirmation messages. You give it the relay, a target (i.e. "subscribe"), the message they sent requesting the confirm, your confirmation template, and any vars that template needs.

The result of calling this is that the template message gets sent through the relay, the original message is stored in the pending queue, and data is put into the storage for later calls to verify.

verify(*target*, *from_address*, *expect_secret*)
Given a target (i.e. "subscribe", "post", etc), a from_address of someone trying to confirm, and the secret they should use, this will try to verify their confirmation. If the verify works then you'll get the original message back to do what you want with.

If the verification fails then you are given None.

The message is *not* deleted from the pending queue. You can do that yourself with delete_pending.

class salmon.confirm.**ConfirmationStorage**(*db={}*)
Bases: object

This is the basic confirmation storage. For simple testing purposes you can just use the default hash db parameter. If you do a deployment you can probably get away with a shelf hash instead.

You can write your own version of this and use it. The confirmation engine only cares that it gets something that supports all of these methods.

Change the db parameter to a shelf to get persistent storage.

clear()
Used primarily in testing, this clears out all pending confirmations.

delete(*target*, *from_address*)
Removes a target+from_address from the storage.

get(*target*, *from_address*)
Given a target and a from address, this returns a tuple of (expected_secret, pending_message_id). If it doesn't find that target+from_address, then it should return a (None, None) tuple.

key(*target*, *from_address*)
Used internally to construct a string key, if you write your own you don't need this.

NOTE: To support proper equality and shelve storage, this encodes the key into ASCII. Make a different subclass if you need Unicode and your storage supports it.

store(*target*, *from_address*, *expected_secret*, *pending_message_id*)
Given a target, from_address it will store the expected_secret and pending_message_id of later verification. The target should be a string indicating what is being confirmed. Like "subscribe", "post", etc.

When implementing your own you should *never* allow more than one target+from_address combination.

salmon.encoding module

Salmon takes the policy that email it receives is most likely complete garbage using bizarre pre-Unicode formats that are irrelevant and unnecessary in today's modern world. These are turned into something nice and clean that a regular Python programmer can work with: Unicode.

That's the receiving end, but on the sending end Salmon wants to make the world better by not increasing the suffering. To that end, Salmon will canonicalize all email it sends to be ascii or utf-8 (whichever is simpler and works to encode the data). It is possible to use other encodings (Salmon doesn't live in some fictional world), but this generally frowned upon.

To accomplish these tasks, Salmon goes back to basics and assert a few simple rules on each email it receives:

1. NO ENCODING IS TRUSTED, NO LANGUAGE IS SACRED, ALL ARE SUSPECT.
2. Python wants Unicode, it will get Unicode.
3. Any email that CANNOT become Unicode, CANNOT be processed by Salmon or Python.
4. Email addresses are ESSENTIAL to Salmon's routing and security, and therefore will be canonicalized and properly encoded.
5. Salmon will therefore try to "upgrade" all email it receives to Unicode internally, and cleaning all email addresses.
6. It does this by decoding all codecs, and if the codec LIES, then it will attempt to statistically detect the codec using chardet.
7. If it can't detect the codec, and the codec lies, then the email is bad.
8. All text bodies and attachments are then converted to Python unicode/str (for Python 2.7 and 3.x respectively) in the same way as the headers.
9. All other attachments are converted to raw strings as-is.

Once Salmon has done this, your Python handler can now assume that all MailRequest objects are happily Unicode enabled and ready to go. The rule is:

> IF IT CANNOT BE UNICODE, THEN PYTHON CANNOT WORK WITH IT.

On the outgoing end (when you send a MailResponse), Salmon tries to create the email it wants to receive by canonicalizing it:

1. All email will be encoded in the simplest cleanest way possible without losing information.
2. All headers are converted to 'ascii', and if that doesn't work, then 'utf-8'.
3. All text/* attachments and bodies are converted to ascii, and if that doesn't work, 'utf-8'. It is possible to override this, but you're a bad person if you do
4. All other attachments are left alone.
5. All email addresses are normalized and encoded if they have not been already.

The end result is an email that has the highest probability of not containing any obfuscation techniques, hidden characters, bad characters, improper formatting, invalid non-characterset headers, or any of the other billions of things email clients do to the world. The output rule of Salmon is:

> ALL EMAIL IS ASCII FIRST, THEN ENCODED ASCII-SAFE, AND IF IT CANNOT BE EITHER OF THOSE IT WILL NOT BE SENT.

Following these simple rules, this module does the work of converting email to the canonical format and sending the canonical format. The code is probably the most complex part of Salmon since the job it does is difficult.

Test results show that Salmon can safely canonicalize most email from any culture (not just English) to the canonical form, and that if it can't then the email is not formatted right and/or spam.

If you find an instance where this is not the case, then submit it to the project as a test case.

class salmon.encoding.**ContentEncoding**(*base*)
Bases: object

Wrapper various content encoding headers

The value of each key is returned as a tuple of a string and a dict of params. Note that changes to the params dict won't be reflected in the underlying MailBase unless the tuple is reassigned:

```
>>> value = mail.content_encoding["Content-Type"]
>>> print(value)
('text/html', {'charset': 'us-ascii'})
>>> value[1]['charset'] = 'utf-8'
>>> print(mail["Content-Type"])  # unchanged
('text/html', {'charset': 'us-ascii'})
>>> mail.content_encoding["Content-Type"] = value
>>> print(mail["Content-Type"])
('text/html', {'charset': 'utf-8'})
```

Will raise EncodingError if you try to access a header that isn't in CONTENT_ENCODING_KEYS

get(*key*, *default=None*)

keys()

exception salmon.encoding.**EncodingError**
Bases: exceptions.Exception

Thrown when there is an encoding error.

class salmon.encoding.**MIMEPart**(*type_*, ***params*)
Bases: email.message.Message

A reimplementation of nearly everything in email.mime to be more useful for actually attaching things. Rather than one class for every type of thing you'd encode, there's just this one, and it figures out how to encode what you ask it.

add_text(*content*, *charset=None*)

extract_payload(*mail*)

class salmon.encoding.**MailBase**(*mime_part_or_headers=None*, *parent=None*)
Bases: object

MailBase is used as the basis of salmon.mail and contains the basics of encoding an email. You actually can do all your email processing with this class, but it's more raw.

append_header(*key*, *value*)
Like __set_item__, but won't replace header values

attach_file(*filename*, *data*, *ctype*, *disposition*)
A file attachment is a raw attachment with a disposition that indicates the file name.

attach_text(*data*, *ctype*)
This attaches a simpler text encoded part, which doesn't have a filename.

body

get_all(*key*)

items()

keys()
Returns header keys.

walk()

salmon.encoding.**VALUE_IS_EMAIL_ADDRESS**(*v*)

salmon.encoding.**apply_charset_to_header**(*charset*, *encoding*, *data*)
Given a charset and encoding, decode data into unicode, e.g.

```
>>> print(apply_charset_to_header("utf-8", "Q", "=142ukasz"))
łukasz
```

encoding is case insensitive and must be one of B or Q

salmon.encoding.**attempt_decoding**(*charset*, *dec*)
Attempts to decode bytes into unicode, calls guess_encoding_and_decode if the given charset is wrong.

salmon.encoding.**from_file**(*fileobj*)
Reads an email and cleans it up to make a MailBase.

salmon.encoding.**from_message**(*message*, *parent=None*)
Given a MIMEBase or similar Python email API message object, this will canonicalize it and give you back a pristine MailBase. If it can't then it raises a EncodingError.

salmon.encoding.**from_string**(*data*)
Takes a string, and tries to clean it up into a clean MailBase.

salmon.encoding.**guess_encoding_and_decode**(*original*, *data*, *errors=u'strict'*)

salmon.encoding.**header_from_mime_encoding**(*header*)

salmon.encoding.**header_to_mime_encoding**(*value*, *not_email=False*)

salmon.encoding.**normalize_header**(*header*)

salmon.encoding.**parse_parameter_header**(*message*, *header*)

salmon.encoding.**properly_decode_header**(*header*)
Decodes headers from their ASCII-safe representation

salmon.encoding.**properly_encode_header**(*value*, *encoder*, *not_email*)
The only thing special (weird) about this function is that it tries to do a fast check to see if the header value has an email address in it. Since random headers could have an email address, and email addresses have weird special formatting rules, we have to check for it.

Normally this works fine, but in Librelist, we need to "obfuscate" email addresses by changing the '@' to '-AT-'. This is where VALUE_IS_EMAIL_ADDRESS exists. It's a simple lambda returning True/False to check if a header value has an email address. If you need to make this check different, then change this.

salmon.encoding.**to_file**(*mail*, *fileobj*)
Writes a canonicalized message to the given file.

salmon.encoding.**to_message**(*mail*)
Given a MailBase message, this will construct a MIMEPart that is canonicalized for use with the Python email API.

N.B. this changes the original email.message.Message

salmon.encoding.**to_string**(*mail*, *envelope_header=False*)
Returns a canonicalized email string you can use to send or store somewhere.

salmon.mail module

The salmon.mail module contains nothing more than wrappers around the big work done in salmon.encoding. These are the actual APIs that you'll interact with when doing email, and they mostly replicate the salmon.encoding.MailBase functionality.

The main design criteria is that MailRequest is mostly for reading email that you've received, so it doesn't have functions for attaching files and such. MailResponse is used when you are going to write an email, so it has the APIs for doing attachments and such.

class salmon.mail.**MailRequest**(*Peer*, *From*, *To*, *Data*)

Bases: object

This is what is given to your message handlers. The information you get out of this is *ALWAYS* in Python str (unicode in Python 2.7) and should be usable by any API. Modifying this object will cause other handlers that deal with it to get your modifications, but in general you don't want to do more than maybe tag a few headers.

Peer is the remote peer making the connection (sometimes the queue name). From and To are what you think they are. Data is the raw full email as received by the server.

NOTE: It does not handle multiple From headers, if that's even possible. It will parse the From into a list and take the first one.

all_parts()

Returns all multipart mime parts. This could be an empty list.

body()

Always returns a body if there is one. If the message is multipart then it returns the first part's body, if it's not then it just returns the body. If returns None then this message has nothing for a body.

is_bounce(*threshold=0.3*)

Determines whether the message is a bounce message based on salmon.bounce.BounceAnalzyer given threshold. 0.3 is a good conservative base.

items()

keys()

original

to_message()

Converts this to a Python email message you can use to interact with the python mail APIs.

walk()

Recursively walks all attached parts and their children.

class salmon.mail.**MailResponse**(*To=None*, *From=None*, *Subject=None*, *Body=None*, *Html=None*)

Bases: object

You are given MailResponse objects from the salmon.view methods, and whenever you want to generate an email to send to someone. It has the same basic functionality as MailRequest, but it is designed to be written to, rather than read from (although you can do both).

You can easily set a Body or Html during creation or after by passing it as __init__ parameters, or by setting those attributes.

You can initially set the From, To, and Subject, but they are headers so use the dict notation to change them: msg['From'] = 'joe@test.com'.

The message is not fully crafted until right when you convert it with MailResponse.to_message. This lets you change it and work with it, then send it out when it's ready.

all_parts()
Returns all the encoded parts. Only useful for debugging or inspecting after calling to_message().

attach(*filename=None*, *content_type=None*, *data=None*, *disposition=None*)
Simplifies attaching files from disk or data as files. To attach simple text simple give data and a content_type. To attach a file, give the data/content_type/filename/disposition combination.

For convenience, if you don't give data and only a filename, then it will read that file's contents when you call to_message() later. If you give data and filename then it will assume you've filled data with what the file's contents are and filename is just the name to use.

attach_all_parts(*mail_request*)
Used for copying the attachment parts of a mail.MailRequest object for mailing lists that need to maintain attachments.

attach_part(*part*)
Attaches a raw MailBase part from a MailRequest (or anywhere) so that you can copy it over.

clear()
Clears out the attachments so you can redo them. Use this to keep the headers for a series of different messages with different attachments.

items()

keys()

to_message()
Figures out all the required steps to finally craft the message you need and return it. The resulting message is also available as a self.base attribute.

What is returned is a Python email API message you can use with those APIs. The self.base attribute is the raw salmon.encoding.MailBase.

update(*message*)
Used to easily set a bunch of headers from another dict like object.

salmon.queue module

Simpler queue management than the regular mailbox.Maildir stuff. You do get a lot more features from the Python library, so if you need to do some serious surgery go use that. This works as a good API for the 90% case of "put mail in, get mail out" queues.

class salmon.queue.**Queue**(*queue_dir*, *safe=False*, *pop_limit=0*, *oversize_dir=None*)
Bases: object

Provides a simplified API for dealing with 'queues' in Salmon. It currently just supports Maildir queues since those are the most robust, but could implement others later.

This gives the Maildir queue directory to use, and whether you want this Queue to use the SafeMaildir variant which hashes the hostname so you can expose it publicly.

The pop_limit and oversize_queue both set a upper limit on the mail you pop out of the queue. The size is checked before any Salmon processing is done and is based on the size of the file on disk. The purpose is to prevent people from sending 10MB attachments. If a message is over the pop_limit then it is placed into the oversize_dir (which should be a Maildir).

The oversize protection only works on pop messages off, not putting them in, get, or any other call. If you use get you can use self.oversize to also check if it's oversize manually.

clear()
Clears out the contents of the entire queue. Warning: This could be horribly inefficient since it basically pops until the queue is empty.

count()
Returns the number of messages in the queue.

get(*key*)
Get the specific message referenced by the key. The message is NOT removed from the queue.

keys()
Returns the keys in the queue.

oversize(*key*)

pop()
Pops a message off the queue, order is not really maintained like a stack.

It returns a (key, message) tuple for that item.

push(*message*)
Pushes the message onto the queue. Remember the order is probably not maintained. It returns the key that gets created.

remove(*key*)
Removes the queue, but not returned.

exception salmon.queue.**QueueError**(*msg*, *data*)
Bases: exceptions.Exception

class salmon.queue.**SafeMaildir**(*dirname*, *factory=<class rfc822.Message>*, *create=True*)
Bases: mailbox.Maildir

Initialize a Maildir instance.

salmon.routing module

The meat of Salmon, doing all the work that actually takes an email and makes sure that your code gets it.

The three most important parts for a programmer are the Router variable, the StateStorage base class, and the @route, @route_like, and @stateless decorators.

The salmon.routing.Router variable (it's not a class, just named like one) is how the whole system gets to the Router. It is an instance of RoutingBase and there's usually only one.

The salmon.routing.StateStorage is what you need to implement if you want Salmon to store the state in a different way. By default the salmon.routing.Router object just uses a default MemoryStorage to do its job. If you want to use a custom storage, then in your boot modiule you would set salmon.routing.Router.STATE_STORE to what you want to use.

Finally, when you write a state handler, it has functions that act as state functions for dealing with each state. To tell the Router what function should handle what email you use a @route decorator. To tell the Route that one function routes the same as another use @route_like. In the case where a state function should run on every matching email, just use the @stateless decorator after a @route or @route_like.

If at any time you need to debug your routing setup just use the salmon routes command.

Routing Control

To control routing there are a set of decorators that you apply to your functions.

- @route – The main routing function that determines what addresses you are interested in.
- @route_like – Says that this function routes like another one.
- @stateless – Indicates this function always runs on each route encountered, and no state is maintained.
- @nolocking – Use this if you want this handler to run parallel without any locking around Salmon internals. SUPER DANGEROUS, add @stateless as well.
- @state_key_generator – Used on a function that knows how to make your state keys for the module, for example if module_name + message.To is needed to maintain state.

It's best to put @route or @route_like as the first decorator, then the others after that.

The @state_key_generator is different since it's not intended to go on a handler but instead on a simple function, so it shouldn't be combined with the others.

salmon.routing.**DEFAULT_STATE_KEY**(*mod*, *msg*)

class salmon.routing.**MemoryStorage**
Bases: *salmon.routing.StateStorage*

The default simplified storage for the Router to hold the states. This should only be used in testing, as you'll lose all your contacts and their states if your server shuts down. It is also horribly NOT thread safe.

clear()

get(*key*, *sender*)

key(*key*, *sender*)

set(*key*, *sender*, *state*)

class salmon.routing.**RoutingBase**
Bases: object

The self is a globally accessible class that is actually more like a glorified module. It is used mostly internally by the salmon.routing decorators (route, route_like, stateless) to control the routing mechanism.

It keeps track of the registered routes, their attached functions, the order that these routes should be evaluated, any default routing captures, and uses the MemoryStorage by default to keep track of the states.

You can change the storage to another implementation by simple setting:

Router.STATE_STORE = OtherStorage()

in your settings module.

RoutingBase does locking on every write to its internal data (which usually only happens during booting and reloading while debugging), and when each handler's state function is called. ALL threads will go through this lock, but only as each state is run, so you won't have a situation where the chain of state functions will block all the others. This means that while your handler runs nothing will be running, but you have not guarantees about the order of each state function.

However, this can kill the performance of some kinds of state functions, so if you find the need to not have locking, then use the @nolocking decorator and the Router will NOT lock when that function is called. That means while your @nolocking state function is running at least one other thread (more if the next ones happen to be @nolocking) could also be running.

It's your job to keep things straight if you do that.

NOTE: See @state_key_generator for a way to change what the key is to STATE_STORE for different state control options.

call_safely(*func*, *message*, *kwargs*)
Used by self to call a function and log exceptions rather than explode and crash.

clear_routes()
Clears out the routes for unit testing and reloading.

clear_states()
Clears out the states for unit testing.

defaults(**captures*)
Updates the defaults for routing captures with the given settings.

You use this in your handlers or your settings module to set common regular expressions you'll have in your @route decorators. This saves you typing, but also makes it easy to reconfigure later.

For example, many times you'll have a single host="..." regex for all your application's routes. Put this in your settings.py file using route_defaults={'host': '...'} and you're done.

deliver(*message*)
The meat of the whole Salmon operation, this method takes all the arguments given, and then goes through the routing listing to figure out which state handlers should get the gear. The routing operates on a simple set of rules:

> 1) Match on all functions that match the given To in their registered format pattern. 2) Call all @stateless state handlers functions. 3) Call the first method that's in the right state for the From/To.

It will log which handlers are being run, and you can use the 'salmon route' command to inspect and debug routing problems.

If you have an ERROR state function, then when your state blows up, it will transition to ERROR state and call your function right away. It will then stay in the ERROR state unless you return a different one.

get_state(*module_name*, *message*)
Returns the state that this module is in for the given message (using its from).

in_error(*func*, *message*)
Determines if the this function is in the 'ERROR' state, which is a special state that self puts handlers in that throw an exception.

in_state(*func*, *message*)
Determines if this function is in the state for the to/from in the message. Doesn't apply to @stateless state handlers.

load(*handlers*)
Loads the listed handlers making them available for processing. This is safe to call multiple times and to duplicate handlers listed.

match(*address*)
This is a generator that goes through all the routes and yields each match it finds. It expects you to give it a blah@blah.com address, NOT "Joe Blow" <blah@blah.com>.

register_route(*format*, *func*)
Registers this function func into the routes mapping based on the format given. Format should be a regex string ready to be handed to re.compile.

reload()
Performs a reload of all the handlers and clears out all routes, but doesn't touch the internal state.

set_state(*module_name*, *message*, *state*)
Sets the state of the given module (a string) according to the message to the requested state (a string). This is also how you can force another FSM to a required state.

state_key(*module_name*, *message*)
Given a module_name we need to get a state key for, and a message that has information to make the key,

this function calls any registered @state_key_generator and returns that as the key. If none is given then it just returns module_name as the key.

class salmon.routing.**ShelveStorage**(*database_path*)
Bases: *salmon.routing.MemoryStorage*

Uses Python's shelve to store the state of the Routers to disk rather than in memory like with MemoryStorage. This will get you going on a small install if you need to persist your states (most likely), but if you have a database, you'll need to write your own StateStorage that uses your ORM or database to store. Consider this an example.

NOTE: Because of shelve limitations you can only use ASCII encoded keys.

Database path depends on the backing library use by Python's shelve.

clear()
Primarily used in the debugging/unit testing process to make sure the states are clear. In production this could be a bad thing.

get(*key*, *sender*)
This will lock the internal thread lock, and then retrieve from the shelf whatever key you request. If the key is not found then it will set (atomically) to ROUTE_FIRST_STATE.

set(*key*, *sender*, *state*)
Acquires the self.lock and then sets the requested state in the shelf.

class salmon.routing.**StateStorage**
Bases: object

The base storage class you need to implement for a custom storage system.

clear()
This should clear ALL states, it is only used in unit testing, so you can have it raise an exception if you want to make this safer.

get(*key*, *sender*)
You must implement this so that it returns a single string of either the state for this combination of arguments, OR the ROUTE_FIRST_STATE setting.

set(*key*, *sender*, *state*)
Set should take the given parameters and consistently set the state for that combination such that when StateStorage.get is called it gives back the same setting.

salmon.routing.**assert_salmon_settings**(*func*)
Used to make sure that the func has been setup by a routing decorator.

salmon.routing.**attach_salmon_settings**(*func*)
Use this to setup the _salmon_settings if they aren't already there.

salmon.routing.**has_salmon_settings**(*func*)

salmon.routing.**nolocking**(*func*)
Normally salmon.routing.Router has a lock around each call to all handlers to prevent them from stepping on each other. It's assumed that 95% of the time this is what you want, so it's the default. You probably want everything to go in order and not step on other things going off from other threads in the system.

However, sometimes you know better what you are doing and this is where @nolocking comes in. Put this decorator on your state functions that you don't care about threading issues or that you have found a need to manually tune, and it will run it without any locks.

class salmon.routing.**route**(*format*, ***captures*)
Bases: object

The @route decorator is attached to state handlers to configure them in the Router so they handle messages for them. The way this works is, rather than just routing working on only messages being sent to a state handler, it also uses the state of the sender. It's like having routing in a web application use both the URL and an internal state setting to determine which method to run.

However, if you'd rather than this state handler process all messages matching the @route then tag it @stateless. This will run the handler no matter what and not change the user's state.

Sets up the pattern used for the Router configuration. The format parameter is a simple pattern of words, captures, and anything you want to ignore. The captures parameter is a mapping of the words in the format to regex that get put into the format. When the pattern is matched, the captures are handed to your state handler as keyword arguments.

For example, if you have:

```
@route("(list_name)-(action)@(host)",
    list_name='[a-z]+',
    action='[a-z]+', host='test\.com')
def STATE(message, list_name=None, action=None, host=None):
    pass
```

Then this will be translated so that list_name is replaced with [a-z]+, action with [a-z]+, and host with 'test.com' to produce a regex with the right format and named captures to that your state handler is called with the proper keyword parameters.

You should also use the Router.defaults() to set default things like the host so that you are not putting it into your code.

parse_format (*format*, *captures*)
Does the grunt work of conversion format+captures into the regex.

setup_accounting (*func*)
Sets up an accounting map attached to the func for routing decorators.

class salmon.routing.**route_like** (*func*)
Bases: *salmon.routing.route*

Many times you want your state handler to just accept mail like another handler. Use this, passing in the other function. It even works across modules.

salmon.routing.**salmon_setting** (*func*, *key*)
Simple way to get the salmon setting off the function, or None.

salmon.routing.**state_key_generator** (*func*)
Used to indicate that a function in your handlers should be used to determine what they key is for state storage. It should be a function that takes the module_name and message being worked on and returns a string.

salmon.routing.**stateless** (*func*)
This simple decorator is attached to a handler to indicate to the Router.deliver() method that it does NOT maintain state or care about it. This is how you create a handler that processes all messages matching the given format+captures in a @route.

Another way to think about a @stateless handler is that it is a pass-through handler that does its processing and then passes the results on to others.

Stateless handlers are NOT guaranteed to run before the handler with state.

salmon.server module

The majority of the server related things Salmon needs to run, like receivers, relays, and queue processors.

class salmon.server.**LMTPReceiver**(*host=u'127.0.0.1'*, *port=8824*, *socket=None*)
Bases: lmtpd.LMTPServer

Receives emails and hands it to the Router for further processing.

Initializes to bind on the given port and host/IP address. Remember that LMTP isn't for use over a WAN, so bind it to either a LAN address or localhost. If socket is not None, it will be assumed to be a path name and a UNIX socket will be set up instead.

This uses lmtpd.LMTPServer in the __init__, which means that you have to call this far after you use python-daemonize or else daemonize will close the socket.

close()
Doesn't do anything except log who called this, since nobody should. Ever.

process_message(*Peer*, *From*, *To*, *Data*)
Called by lmtpd.LMTPServer when there's a message received.

start()
Kicks everything into gear and starts listening on the port. This fires off threads and waits until they are done.

class salmon.server.**QueueReceiver**(*queue_dir*, *sleep=10*, *size_limit=0*, *oversize_dir=None*)
Bases: object

Rather than listen on a socket this will watch a queue directory and process messages it receives from that. It works in almost the exact same way otherwise.

The router should be fully configured and ready to work, the queue_dir can be a fully qualified path or relative.

process_message(*msg*)
Exactly the same as SMTPReceiver.process_message but just designed for the queue's quirks.

start(*one_shot=False*)
Start simply loops indefinitely sleeping and pulling messages off for processing when they are available.

If you give one_shot=True it will run once rather than do a big while loop with a sleep.

class salmon.server.**Relay**(*host=u'127.0.0.1'*, *port=25*, *username=None*, *password=None*, *ssl=False*, *starttls=False*, *debug=0*, *lmtp=False*)
Bases: object

Used to talk to your "relay server" or smart host, this is probably the most important class in the handlers next to the salmon.routing.Router. It supports a few simple operations for sending mail, replying, and can log the protocol it uses to stderr if you set debug=1 on __init__.

The hostname and port we're connecting to, and the debug level (default to 0). Optional username and password for smtp authentication. If ssl is True smtplib.SMTP_SSL will be used. If starttls is True (and ssl False), smtp connection will be put in TLS mode. If lmtp is true, then smtplib.LMTP will be used. Mutually exclusive with ssl.

configure_relay(*hostname*)

deliver(*message*, *To=None*, *From=None*)
Takes a fully formed email message and delivers it to the configured relay server.

You can pass in an alternate To and From, which will be used in the SMTP/LMTP send lines rather than what's in the message.

reply(*original*, *From*, *Subject*, *Body*)
Calls self.send but with the from and to of the original message reversed.

resolve_relay_host(*To*)

send(*To*, *From*, *Subject*, *Body*)
Does what it says, sends an email. If you need something more complex then look at salmon.mail.MailResponse.

class salmon.server.**SMTPChannel**(*server*, *conn*, *addr*)
Bases: smtpd.SMTPChannel

Replaces the standard SMTPChannel with one that rejects more than one recipient

smtp_RCPT(*arg*)

exception salmon.server.**SMTPError**(*code*, *message=None*)
Bases: exceptions.Exception

You can raise this error when you want to abort with a SMTP error code to the client. This is really only relevant when you're using the SMTPReceiver and the client understands the error.

If you give a message than it'll use that, but it'll also produce a consistent error message based on your code. It uses the errors in salmon.bounce to produce them.

error_for_code(*code*)

class salmon.server.**SMTPReceiver**(*host=u'127.0.0.1'*, *port=8825*)
Bases: smtpd.SMTPServer

Receives emails and hands it to the Router for further processing.

Initializes to bind on the given port and host/IP address. Typically in deployment you'd give 0.0.0.0 for "all internet devices" but consult your operating system.

This uses smtpd.SMTPServer in the __init__, which means that you have to call this far after you use python-daemonize or else daemonize will close the socket.

close()
Doesn't do anything except log who called this, since nobody should. Ever.

handle_accept()

process_message(*Peer*, *From*, *To*, *Data*)
Called by smtpd.SMTPServer when there's a message received.

start()
Kicks everything into gear and starts listening on the port. This fires off threads and waits until they are done.

salmon.server.**undeliverable_message**(*raw_message*, *failure_type*)
Used universally in this file to shove totally screwed messages into the routing.Router.UNDELIVERABLE_QUEUE (if it's set).

salmon.testing module

salmon.utils module

Mostly utility functions Salmon uses internally that don't really belong anywhere else in the modules. This module is kind of a dumping ground, so if you find something that can be improved feel free to work up a patch.

salmon.utils.**check_for_pid**(*pid*, *force*)
Checks if a pid file is there, and if it is sys.exit. If force given then it will remove the file and not exit if it's there.

salmon.utils.**daemonize**(*pid*, *chdir*, *chroot*, *umask*, *files_preserve=None*, *do_open=True*)
Uses python-daemonize to do all the junk needed to make a server a server. It supports all the features daemonize has, except that chroot probably won't work at all without some serious configuration on the system.

salmon.utils.**drop_priv**(*uid*, *gid*)
Changes the uid/gid to the two given, you should give utils.daemonize 0,0 for the uid,gid so that it becomes root, which will allow you to then do this.

salmon.utils.**import_settings**(*boot_also*, *boot_module=u'config.boot'*)
Returns the current settings module, there is no harm in calling it multiple times

The location of the settings module can be control via SALMON_SETTINGS_MODULE

salmon.utils.**make_fake_settings**(*host*, *port*)
When running as a logging server we need a fake settings module to work with since the logging server can be run in any directory, so there may not be a settings module to import.

salmon.utils.**start_server**(*pid*, *force*, *chroot*, *chdir*, *uid*, *gid*, *umask*, *settings_loader*, *debug*, *daemon_proc*)
Starts the server by doing a daemonize and then dropping priv accordingly. It will only drop to the uid/gid given if both are given.

salmon.view module

These are helper functions that make it easier to work with either Jinja2 or Mako templates. You MUST configure it by setting salmon.view.LOADER to one of the template loaders in your boot module.

After that these functions should just work.

salmon.view.**attach**(*msg*, *variables*, *template*, *filename=None*, *content_type=None*, *disposition=None*)
Useful for rendering an attachment and then attaching it to the message given. All the parameters that are in salmon.mail.MailResponse.attach are there as usual.

salmon.view.**load**(*template*)
Uses the registered loader to load the template you ask for. It assumes that your loader works like Jinja2 or Mako in that it has a LOADER.get_template() method that returns the template.

salmon.view.**render**(*variables*, *template*)
Takes the variables given and renders the template for you. Assumes the template returned by load() will have a .render() method that takes the variables as a dict.

Use this if you just want to render a single template and don't want it to be a message. Use render_message if the contents of the template are to be interpreted as a message with headers and a body.

salmon.view.**respond**(*variables*, *Body=None*, *Html=None*, ***kwd*)
Does the grunt work of cooking up a MailResponse that's based on a template. The only difference from the salmon.mail.MailResponse class and this (apart from variables passed to a template) are that instead of giving actual Body or Html parameters with contents, you give the name of a template to render. The kwd variables are the remaining keyword arguments to MailResponse of From/To/Subject.

For example, to render a template for the body and a .html for the Html attachment, and to indicate the From/To/Subject do this:

msg = view.respond(locals(), Body='template.txt', Html='template.html',
From='test@test.com', To='receiver@test.com', Subject='Test body from "%(person)s".')

In this case you're using locals() to gather the variables needed for the 'template.txt' and 'template.html' templates. Each template is setup to be a text/plain or text/html attachment. The From/To/Subject are setup as needed. Finally, the locals() are also available as simple Python keyword templates in the From/To/Subject so you can pass in variables to modify those when needed (as in the %(person)s in Subject).

5.5.3 Module contents

CHAPTER 6

About Salmon

6.1 Changelog

6.1.1 3.0.0

- No changes since *3.0.0rc1*

6.1.2 3.0.0rc1

- Removed lots of cruft (#19)
- Moved from modargs to argparse - command line interface has changed (#28)
  - Improved tests for command line (#47)
- Moved from PyDNS to dnspython
- Tests can now be run without having to start a log-server first (#6)
- MailRequest objects are now wrappers around Python's `email.message.Message` class. (#40)
  - Deserializing incoming messages is now done in a slightly more lazy fashion
  - Also allows access to the "pristine" `Message` object without having to back-convert
  - Header setting now replaces by default (#44)
- End support of Python 2.6 (#42)
- Settings no longer limited to per app "config" module (#38)
- Allow `salmon.server.Relay` to talk to LMTP servers (#41)
- Make `LMTPReceiver` the default in the prototype app (#48)
- Properly work around `SMTPReceiver` bug caused by an assumption about Python's `smtpd` module that should not have been made (#48)

  - This means that Salmon will no longer accept multiple RCPT TOs in the same transaction over SMTP. Consider using `LMTPReceiver` instead as it does not have this restriction.

- Python 3 support (#7)
  - You'll now need `setuptools` to install (this won't be a problem for those upgrading)
  - No more support for Windows - it never worked for production on that platform anyway
- Don't catch `socket.error` when delivering messages via `salmon.server.Relay` (#49)
- Bind to port `0` during tests as this lets the OS choose a free port for us (#51)
- Wrote some documentation (#33)

6.1.3 Earlier Releases

Sorry, we didn't keep a changelog prior to Salmon 3.0!

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W