TurboMail Documentation

Release 3.0

Alice Zoe Bevan-McGregor, Felix Schwarz

June 15, 2009

CONTENTS

TurboMail is a Python library to ease sending emails from your application.

By using TurboMail you can:

- Easily construct plain text and HTML emails
- Make your code which sends mails testable
- Use different mail delivery strategies (e.g. your app should not block if your mail server is slow)
- Switch the transport (e.g. SMTP, mailbox) by just changing some values in a config file

TurboMail 3.x was rebuilt from scratch so that you don't have dependencies on TurboGears any longer. Therefore TurboMail is usable with all Python applications including Django, Pylons, Trac, web.py and plain CGI scripts. All versions of Python 2.3-2.6 are supported and there are no other dependencies on external packages for normal operation [1].

Using TurboMail is meant to be simple:

```
from turbomail import Message
message = Message("from@example.com", "to@example.com", "Hello World")
message.plain = "TurboMail is really easy to use."
message.send()
```

This would send a plain text email with the subject "Hello World" and the author (*From:* header) "from@example.com" to "to@example.com". The body just contains "TurboMail is really easy to use.". The code assumes that you use TurboMail inside of a "supported" framework which sets up TurboMail on startup. We'll show you how to start TurboMail manually (e.g. for plain CGI scripts) later (see *TurboMail from Scratch*).

This documentation is an effort of the TurboMail developers to show you the full potential of TurboMail. But we need your help! If you find mistakes in this docs, please notify us. If you think some parts should be clarified or documented differently, please notify us. We really need your input (and your time :-) to make TurboMail a software everyone likes to use.

TurboMail 3 comes with adapters for TurboGears 1 and Pylons/TurboGears 2 (see *TurboMail Adapters*) but supporting a new framework should only be a matter of 4-10 lines of code if the framework uses setuptools (see *Writing Custom Adapters* for more information).

[1] In order to run the full test suite you need to install pymta and its dependencies.

CHAPTER

ONE

HIGH LEVEL OVERVIEW

1.1 Goals of TurboMail

TurboMail is about making sending emails easy. One part of that is constructing email.Message objects with a simple API. This API is designed to cover the easy use cases (which are by far the majority) so your code can be significantly shorter and easier to read than doing it 'the hard way' by using Python's built-in email module. But sometimes you really need the full power of RFC 2822. Then you can always use Python's email module (or another library) and ignore TurboMail's message object. TurboMail can still help you sending the actual message.

TurboMail is *not* a MTA like Exim, Postfix, sendmail or qmail. It is designed to deliver your messages to a real mail server ("smart host") which actually delivers them to the recipient's server. TurboMail does only use an in-memory queue so there is no persistent queue storage which is absolutely crucial for a real MTA. Theoretically you could extend TurboMail so that it behaves like a real MTA but this is not something we will build in the near future [1].

We hope that this got you a better idea what TurboMail is about and what tasks leaves to other software. In the next section we'll show you a high level view over TurboMail's different components.

1.2 Architecture and Components

Message class

The actual mail delivery is done by two components: Managers and transports. *Managers* don't deliver the messages themselves. They have a strategy when to hand over a message to a transport. *Transports* actual deliver messages they get. Each transport is specialized on a specific transport mechanism (e.g. SMTP or mailbox delivery to the filesystem).

[1] Fortunately, there are some free libraries to build an SMTP server on the internet, e.g. Python's smtpd (extremly simple to use, but messy code, not extendable), Twisted Mail (probably the most featureful SMTP server implementation available for Python, uses the twisted framework which can be either a huge advantage or disadvantage, depending on your point of view), and pymta (mostly a proof of concept, but highly customizable). Besides these libraries there is tmda-ofmipd (extended version of Python's smtpd, supports TLS, messy code, only available as a part of a bigger package).

CHAPTER

TWO

USING TURBOMAIL

2.1 Building Mails

One of TurboMail's main goals is to make build emails easy. Therefore we create a Message class which hides all the tedious work like encoding the message header lines and building a MIME compliant body.

class `Message`(*[author, [to, [subject, [**kwargs]]]]*)

All arguments are optional. For the semantics behind these parameters, please read the 'Properties' section below. Through the use of kwargs you can set all properties documented below directly in the constructor (a TypeError will be raised if kwargs contains a key which is not a valid property).

Properties:

- author – Email address or addresses of the author(s)
- bcc – Blind carbon-copy address or addresses.
- cc – Carbon-copy address or addresses.
- date – Date if message creation either a date string as per RFC 2822, a datetime.datetime instance or the number of seconds since 1970 as float (optional, uses the current local time by default)
- disposition – Request disposition notification be sent ("email was read") to this address.
- encoding – Content encoding specific to this message.
- headers – A list of additional messages headers (either as dictionary or as list of tupels).
- nr_retries - How often should TurboMail retry to deliver this message if the delivery failed? (default: 3)
- organization – The descriptive Organization header.
- plain – Plain text content. Will be automatically generated if a rich text part was specified.
- priority – The X-Priority header, in the range of 1-5.
- reply_to – Email address to which replies should be sent (Reply-To header)
- rich – The rich-text ("HTML") part
- sender – email address of the agent which sends the mail (required if you specified more than one author)
- smtp_from – The SMTP envelope address, if different from the sender (this option is only useful for the SMTP-Transport).
- subject – A textual description or summary of the content of the message.

- to – Email address or addresses which should receive the message (To header)

All email addresses can be given either as a string (TurboMail will convert unicode domain names with idna encoding if possible) or as a tuple ('Full Name', 'name@example.com').

The plain- and rich-text parts of the message may be defined as any callable which returns plain- or rich-text. The callable is executed at the time the message is built - usually just prior to the first delivery attempt.

The encoding can be overridden on a per-message basis (you have to pay attention that all characters in your plain or rich contents may be encoded with the chosen encoding). If you use 'UTF-8-QP' as encoding (which is a fake encoding provided by TurboMail), the message body will be encoded with UTF-8 Quoted Printable (Python's email module uses base64 by default for UTF-8 content).

In order to simplify the use of the Message class even further, you can set default values for every property in your configuration. These configuration values are used in Message's __init__ (constructor) when you don't set a value explicitely. The configuration key is the name of the property, prefixed with 'mail.message' e.g. 'mail.message.bcc'. Using that mechanism you can send all messages in copy to a certain address (assuming you don't overwrite the bcc property after the message instantiation).

Methods:

attach (*file, [name]*)
: Attach a given file (either on-disk by passing a path, or in-memory by passing a File descendant) to the message. The name argument is required if passing an in-memory File descendant.

embed (*file, name*)
: As per attach, but limited to image files for embedding in the rich-text (HTML) part of the message. Name is the desired CID of the embedded image.

send()
: Send the message via the currently configured manager. In order to send a message, you need to define at least an author, a recipient (either in 'to', 'cc' or 'bcc') and the message content (via 'plain' or 'rich').

2.1.1 Plain Text Messages

Easy things first: You want to send an email with just some information in it (e.g. unhandled exception in your application). This kind of mail doesn't have to look fancy, so we'll just use plain text:

```
from turbomail import Message
message = Message("from@example.com", "to@example.com", "Hello World")
message.plain = "TurboMail is really easy to use."
message.send()
```

You see, building a plain text email is very easy: Just put your message content in the 'plain' attribute and send the mail. Done. In real world scenarios you probably would use some kind of template engine to generate the text body but that's your choice.

2.1.2 Messages with Attachments

Sometimes you need send some files along your message content (e.g. a PDF document or some zip file). There two alternative ways of doing it with TurboMail. This is convenient because sometimes you generate a file on-the-fly (like a PDF report with reportlab/z3c.rml) and don't want to write it to the hard disk (slow, you have to handle disk full conditions etc):

```
from turbomail import Message

message = Message("from@example.com", "to@example.com", "Sales are skyrocketing!")
message.plain = "Have a look at the sales statistics attached."

filename = '/home/fs/latest_sales.pdf'
# The file will be named 'latest_sales.pdf' in the message
message.attach(filename)

# Alternative way to attach a file (please note that you have to provide a
# name for the file to attach):
fp = file(filename)
# The file will be named 'sales_report.pdf' in the message
message.attach(fp, 'sales_report.pdf')

message.send()
```

2.1.3 HTML Messages

Originally emails contained only plain text which does not provide much formatting possibilities especially compared to all those fancy looking websites that came up in the late 90ties. Since the advent of the MIME standard you can use HTML in your messages to produce a so called HTML e-mail.

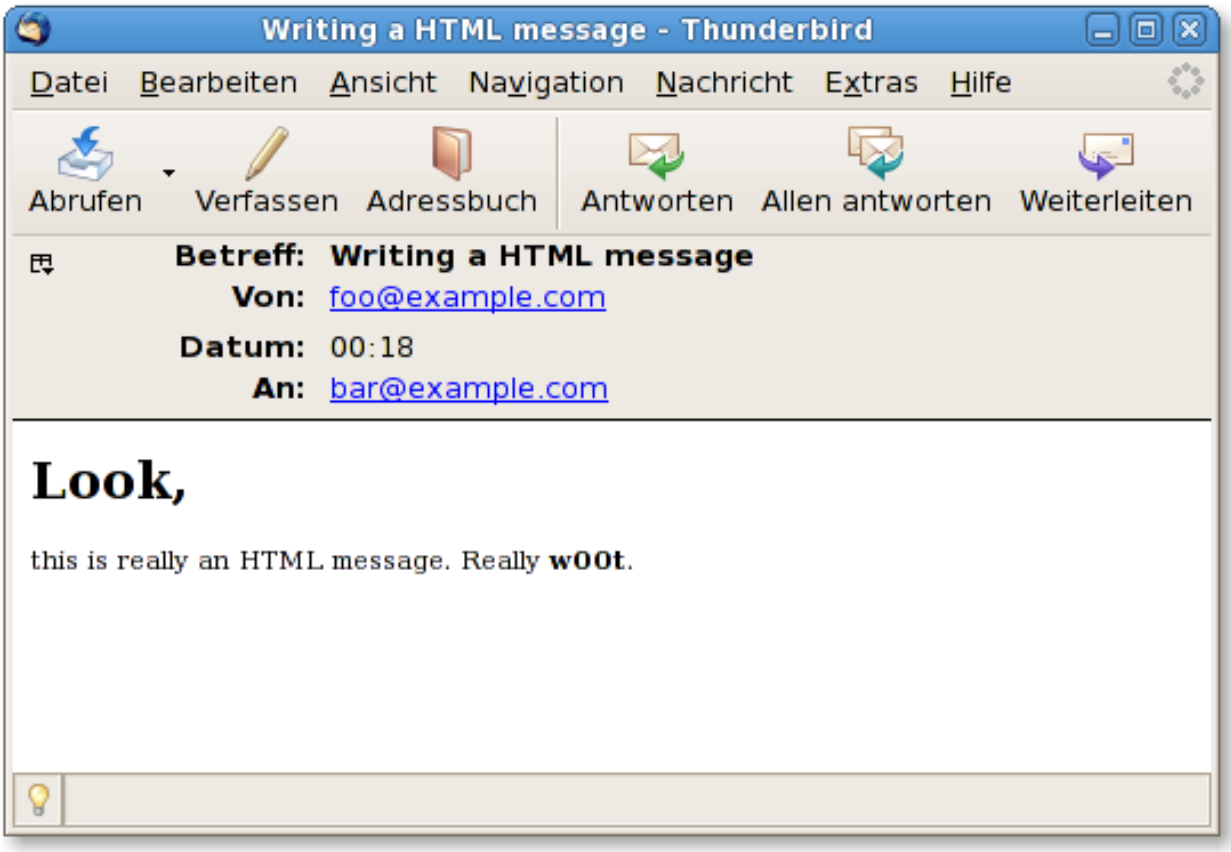
To

```
html_part = """<html><header/>
<body>
  <h1>Look,</h1>

  this is an HTML message. Really <b>w00t</b>.
</body>
</html>
"""

from turbomail import Message
msg = Message("foo@example.com", "bar@example.com", "Writing a HTML message")
msg.rich = html_part
msg.plain = """This is the boring alternative part for people who don't see
the HTML part."""
msg.send()
```

You can build an HTML email just by assigning a string which contains the HTML markup to *msg.rich*. You should always provide *msg.plain* too just in case someone does not want to read your HTML. If you don't provide a plain text TurboMail will derive a plain text part from your HTML by using a very simple algorithm [1].

In Mozilla Thunderbird 2 the resulting message looks like this:

[1] If you need more information about TurboMail's built-in HTML to text converter please see *Clever plain text generation for your rich text messages*.



The HTML string above is static. In most applications you probably use a template engine like Genshi to produce the final HTML.

Be warned: *Keep your HTML simple.* E-mail clients generally don't support the full subset of HTML/CSS. If you are frustrated because Internet Explorer does not like your web page wait until you saw your HTML message rendered by Outlook express before you start cursing... Another difficulty is that there are more email clients with significant market share than browsers (you should count different web mail providers as different mail clients because they all cut different parts of your message to protect their webmail users).

2.1.4 HTML Messages with embedded images

If you want to have really good looking emails you can add images. Unfortunately it is somewhat complicated because you can't just attach an image and reference it from your HTML like in a web page. Instead you have to use the embed function which will tell the recipient's mail client that the image is meant to be shown inside the HTML and is not an ordinary attachment.

The other alternative which keeps the mails also very small is to reference your images with absolute urls to your server. So the recipient has to fetch the images every time he wants to read your message. But this method may not be reliable as some mail clients won't load images from remote servers (or require special user interaction to do so like in Mozilla Thunderbird) [2].

[2] Spammers and other evil people used remote images (called web bugs) to track people so they knew who read their emails. Therefore quite few (especially the computer-savvy) people don't like linked images and many popular mail clients refuse to load files from remote URLs, displaying your mail without any images.

2.1.5 Pre-built Messages

Sometimes you don't want to create a Message with TurboMail - maybe the message is already on your hard drive or it will be generated by some other mechanism (be it legacy code or just a highly specialized mail-generation library). However you want to benefit from the facilities TurboMail provides (e.g. testability of mail sending, threaded mail sending, ...). This is when the WrappedMessage comes in:

```
from turbomail import WrappedMessage

# We're using a hard-coded string here - of course you can get this string
# from another library, this is just to make this example as easy as possible.
generated_message = '...'

message = WrappedMessage('from@example.com', 'to@example.com',
                         message=generated_message)
message.send()
```

The WrappedMessage just wraps a message and adds the minimum information which is necessary for delivery. The contents of your message will not be modified in any way.

Please note that you have to specify the 'sender' and the recipient(s) explicitly when building a WrappedMessage - although the message you're about to pass may contain this information already. However, when it comes to delivery mechanisms like SMTP this information can not be extracted from the message content without guessing. You can find an in-depth explanation of the underlying mechanism in your chapter *RFC (2)822 - From, Sender, Disposition, what?*.

2.1.6 How TurboMail handles Internationalization

When RFC 822 was written (1982), the world was still a disc: emails were allowed to contain ASCII characters only. International characters like German umlauts ("ü", "ß") were not allowed. As time went by the original format was extended in a backward compatible way so you could write plain text messages which contain Non-ASCII characters.

The default encoding for TurboMail is still ASCII (the same holds true for almost all mail servers). If you want to send mails which contain Non-ASCII characters, you have to set the configuration option 'mail.message.encoding' appropriately (see *TurboMail Configuration*). The broadest character set available is UTF-8. Unfortunately Python will encode the body part using base64 if you choose "UTF-8". This means that a human can not read the message by looking at the plain text which may help debugging problems. Furthermore it will increase the message's size by approximately 1/3.

To get around that, TurboMail registers a new codec 'UTF-8-QP' which is mostly an alias for UTF-8. Python's email module will use UTF-8 Quoted Printable for messages with 'UTF-8-QP' which is much more human readable than base64 and won't increase the message size as much as base64.

Since several years you can use even internationalized domain names which means that umlauts may appear left from the @ character as in foo@zääz.de. In the future we will probably see internationalized top-level domains. Most email servers will require that you convert internationalized domain names to punycode so that it only contains ASCII characters. With TurboMail you don't have to care about the conversion as TurboMail will do it for you. However if you choose to use WrappedMessage, you must convert IDNs in your email header (not the SMTP envelope) yourself [3].

2.1.7 How to get your Messages through Spam Filters

No, this section is not about how to become a "Spam King". Due to the amount of spam everyone receives on a daily basis, most of your recipients will use spam filters to filter spam (or rely on their providers like Google or Yahoo to do

[3] Technically you *may* not need to do this but often this is necessary so that the readers mail client will display the address correctly.

so). Assuming that your application only sends out legitimate mail, how do you avoid that your mail is accidentally marked as spam?

"Spamminess" is nothing which can defined just by one score value. Every spam filter will use its own rules and thresholds when to classify a message as spam. Most things that influence the spam score are beyond TurboMails' control: message content, having enough innocent text, sending mails from a static IP address with DNS correctly set up etc. It is impossible to assemble a message that will pass all your recipients spam filters!

However, TurboMail users reported that Google Mail classified their messages as spam if they used UTF-8 characters in their message body. The problem was solved by using the UTF-8 Quoted Printable "encoding" [4]. The previous section *How TurboMail handles Internationalization* told you already how to do this.

Furthermore it may help if you add the full name of the recipient as sender and not just the email address (assuming you know the name):

```
# instead of:
msg = Message("foo@example.com", "bar@example.com", ...)
# better write:
msg = Message(("Foo", "foo@example.com"), ("Bar", "bar@example.com"), ...)
```

If you build HTML messages with TurboMail's Message class, you should care about the plain text part, too. Message tries to derive a plain text part from your rich-text ("HTML") part automatically if you don't set any plain text explicitly. But the algorithm for generating the plain text part is not very clever so some parts of your markup may be included in the plain text part. Some users reported that spam filters classified their message as spam due to these left-overs. We try to improve the text filters if we become aware of these issues but you may think about providing a plain text part explicitly or just disabling the plain text generation. Another alternative is to *plug in a better html2text converter*.

2.2 Sending Mails

When you instantiated a message and added all the information (e.g. message body), you have to tell TurboMail to send the message explicitly. The easiest way of sending a message is calling the send() method of a Message:

```
from turbomail import Message
msg = Message('foo@example.com', 'bar@example.com', 'Subject')
msg.plain = "Foo Bar"
msg.send()    # Message will be sent through the configured manager/transport.
```

When the send method is called, an email.Message will be assembled and handed over to the manager you configured. Whether the message will have to wait in a queue until there are free resources to deliver it to the intended recipient or if the message is given to the configured transport immediately, depends on your configuration.

Please note that every time you call the send method, an email is sent.

In TurboMail 2 you had to call turbomail.enqueue(msg) to send a message. This method is still available for backwards compatibility but its usage is deprecated.

If an error occurs while sending the mail, TurboMail will retry several times to deliver the message. If that is not successful, the message will be dropped. If you use the ImmediateManager, you'll get an exception (smtplib.SMTPError or socket.error). If you use the asynchronous DemandManager, the failure will be logged. We're not satisfied with that behavior but we were not able to fix this in time for TurboMail 3.0. For 3.1 we'll build a more robust and flexible system with hooks so you can do your own error handling routines.

[4] If you use Non-ASCII characters in a message, Python's email module may resort to base64 as encoding for your body. In the early days of mass spam spammers used this trick to get their advertising to the recipients by encoding their message bodies with base64. Most scanners at that time did not preprocess messages and just looked for some keywords but base64 will make an unreadable character stream out of your plain text. Something like 'TurboMail' becomes 'VHVyYm9NYWls\n' when encoded with base64. The users mail client was able to decode base64 without any problems so that may be the reason why most spam filters still give base64 a bad rating.

2.3 TurboMail Configuration

TurboMail's behavior (e.g. what default encoding should be used) is highly configurable. If you use TurboMail in a supported framework which has a TurboMail adapter TurboMail will use the general configuration system of your framework. For TurboGears 1 this means you can put configurations for TurboMail in app.cfg and/or {dev, prod}.cfg and TurboMail will pick them up automatically. The configuration is set when TurboMail starts up and should not be changed afterwards.

Configuration is based on key-value pairs where the key is always a string. What type is required for the value depends on the key. Unless otherwise noted, the values should be specified as strings, too.

Later in this manual we use the ConfigObj syntax when we refer to specific settings. For example, "mail.on = True" means that TurboMail's configuration contains a key "mail.on" with the boolean value True. Which configuration files you have to edit to configure (and if you have to use a special syntax for some values) depends on your framework. We have more information about the supported frameworks in *TurboMail Adapters*.

Throughout this documentation we use the ConfigObj syntax:

```
mail.on              = True        # boolean value True for the option 'mail.on'
mail.manager         = 'immediate' # string value 'immediate'
mail.demand.threads = 5            # int value 5
```

This section deals with general configuration options which are not specific to a single manager or transport. You find specific options in chapter *Detailed TurboMail*.

2.3.1 Configuration options

The single most important option is **mail.on**. If try to use TurboMail without setting this option as True, you will get an exception like this immediately: "MailNotEnabledError: An attempt was made to use a facility of the TurboMail framework but outbound mail hasn't been enabled in the config file [via mail.on]. This was done on purpose because the default transport does not send out mails (so you don't spam real people with mails accidentally generated in your unit tests [5]) but just not sending the mails could be frustrating experience for new users when their mails go to /dev/null. Furthermore this saves some bytes of RAM because neither a manager nor a transport will be loaded when TurboMail is not enabled.

- **mail.on** (boolean, default: False) Enable TurboMail.
- mail.manager (string, default: 'immediate') Specify the name of the manager to use
- **mail.transport** (string, default: 'debug') Specify the name of the transport to use
- mail.message.<property name> (string or tuple) – default values for properties of the Message class (see *Message properties*)

The individual *Managers* and *Transports - Your Work Horses* can have their own configuration options. These are explained in the section where the specific manager/transport is explained in more detail. We just make an exception to explain a mandatory option for the SMTPTransport (which is probably by far the most widely used one):

- **mail.smtp.server** (string, mandatory for SMTP transport) host name or IP address of the SMTP server which will take care of mails sent by TurboMail (only if SMTP transport was enabled).

We deliberately did not set 'localhost' as default value for that because on Un*x machines there is often an SMTP server running on the local machine which may decide to deliver messages to real mail servers. After you mailed a badly written test email to a few thousand customers accidentally you'll understand our paranoia here :-)

[5] You do use foo@example.com as recipient for all your test emails, right? The example.com domain is reserved for documentation so you will never bother someone if you sent a mail to this domain. Usage of existing domains for test purposes is really a pain and kills kittens. And yes, donotreply.com *does* exist.

2.3.2 How transports, managers and extensions are found by TurboMail

TurboMail heavily utilizes setuptools to build a modular, loosely coupled and extensible system. When you specify "mail.transport = 'smtp"' in your configuration, TurboMail tries to load the class associated to the entry point 'turbomail.transports.smtp'. If no class could be loaded, an error will be logged and TurboMail will be disabled.

If you try to use TurboMail after the an error, you will get the same MailNotEnabledError as if you forgot to specify "mail.on = True".

The same holds true for managers (entry point group 'turbomail.managers') and extensions (entry point group 'turbomail.extensions'). The only difference related to extensions is that TurboMail won't be disabled if an extension could not be loaded.

2.3.3 Add new transports without using setuptools

Sometimes you don't want to rely on setuptools for manager/transport discovery. For example if you run the unit test suite for TurboMail itself, we don't want to force you to install the TurboMail egg in your environment. Therefore we added a mechanism so that you can use transports and managers in TurboMail without the requirement that setuptools must be able to find them via its normal entry point mechanism:

```
config = {'mail.on': True, 'mail.manager': 'foo'}
# assuming you built a manager class called MyOwnManagerClass
interface.start(config, extra_classes={'foo': MyOwnManagerClass})
```

If extra classes where provided, TurboMail will look in this dict *before* the lookup with setuptools. If you use this mechanism, you can be sure that the specified manager class is used regardless of the environment configuration which is nice especially for unit tests.

2.3.4 Logging

TurboMail supports logging through Python's standard logging module. All loggers utilized by TurboMail start with 'turbomail.' so you can easily disable all logging (don't forget to stop propagation if you want to silence TurboMail). However we recommend setting the log level for TurboMail to WARNING so you will be notified of potential problems.

Furthermore there is a special delivery log where all delivery attempts are noted. To handle mail delivery log messages differently from other TurboMail log messages (e.g. for later auditing), you can define other handler/log level settings on 'turbomail.delivery' (mail deliveries are logged with log level 'INFO').

2.4 Unit Tests with TurboMail

Testability was advertised as a feature in the beginning. So how to use TurboMail in your (unit) tests?

First of all, you should use the `ImmediateManager` and the `DebugTransport` for testing so you get sent messages as fast as possible (ImmediateManager) and you can collect your messages easily without any interaction with your environment (DebugTransport). Fortunately, this is just the default configuration for TurboMail if you don't configure any managers or transports explicitly.

A simple test case looks like this:

```
import unittest

from turbomail.control import interface
from turbomail.message import Message
```

```
class TestYourAppWithTurboMail(unittest.TestCase):
    def setUp(self):
        config = {'mail.on': True}
        interface.start(config)

    def tearDown(self):
        interface.stop(force=True)
        interface.config = {'mail.on': False}

    def test_fetch_sent_messages(self):
        # call your app - do some interesting things
        # ...
        # no you can retrieve the sent mail:
        send_mails = interface.manager.transport.get_sent_mails()
        self.assertEqual(1, len(send_mails))
        sent_mail = send_mails[0]
        # check that the mail contains the expected information
```

This test assumes that TurboMail is installed in your environment with all the necessary egg information so that it can load the ImmediateManager and the DebugTransport. If you want to be independent of setuptools, please have a look a TurboMail's test_debug_transport.py test case does not rely on setuptools too.

If you want to test a TurboGears 1.0 application and you use a test case which calls turbogears.start() you don't have to issue interface.start and interface.stop because TurboGears will handle that for you. Additionally you should not configure TurboMails' interface but just use turbogears.config to set the appropriate values for TurboMail.

If you want to see some real test cases with TurboMail, you may want to look in TurboMail's own test suite for examples.

2.5 Compatibility with TurboMail 2.x

For TurboMail 3 we reworked all internals. Nevertheless we tried to maintain compatibility for the most important interfaces. The idea is that 90% of all applications using TurboMail 2 will "just work" with TurboMail 3 without the need to change any code (just one line in the configuration). However we deprecated some interfaces and properties. If you use one of these, a DeprecationWarning will be raised. You can disable these warnings in your application easily:

```
from warnings import filterwarnings
filterwarnings('ignore', 'warning text specified with a regular expression',
               category=DeprecationWarning)
# Now you can use the deprecated interface and no warning will be given
```

The most important changes for upgraders are probably that the default manager is now the ImmediateManager (message delivery is synchronous by default, your code needs to deal with exceptions raised during the mail delivery) and the default transport is the DebugTransport which means that **no mail is delivered unless you use the SMTPTransport in the configuration**.

2.5.1 Configuration

One of the main features of TurboMail 3 are different types of transports. In order to send messages with SMTP in your production system you *must* add this configuration option:

```
mail.transport = "smtp"
```

If you forget to do this, no messages will be delivered!

In TurboMail 2 all parameter keys used the naming convention 'mail.<name>'. With TurboMail 3 this often does not make sense because we have different transports now (not only SMTP) and many options are only useful in a SMTP context. Therefore all SMTP related parameters are renamed from 'mail.foo' to 'mail.smtp.foo'. The options mail.server, mail.username, mail.password, mail.debug and mail.tls were renamed. The old parameter names will continue to work but a DeprecationWarning is given when TurboMail encounters one of the old configuration names.

2.5.2 Message class

All properties from the old Message class should be still there. Please note that we renamed the 'recipient' property to 'to' to be more clear which mail header is meant. 'smtpfrom' and 'replyto' were renamed in accordance to PEP 8 (the official style guide for Python code) to 'smtp_from' and 'reply_to'.

The most important change functionality-wise is the rename of 'sender'. We felt that 'sender' is ambiguous because it was used for the From header but there is another header that is called 'sender', too. Therefore the attribute to set the From header is now called 'author' and there the property 'sender' will now set the Sender header. If you don't set the 'author' property but use 'sender', your 'sender' property will be used for the 'author' so we keep the backwards compatibility.

2.5.3 Sending Messages

In TurboMail 2 you submitted messages for sending by doing this:

```
from turbomail import Message, enqueu
msg = Message(...)
enqueue(msg)
```

In TurboMail 3 messages can send themselves so the code above should be written as:

```
from turbomail import Message
msg = Message(...)
msg.send()

# alternatively you can send messages by doing:
from turbomail import send
send(msg)
```

Furthermore we felt that 'enqueue' is not a good name any more because now we have different strategies of sending messages and not all managers will queue your message. Therefore the 'enqueue' method is still there but is marked as deprecated.

2.5.4 Broken Compatibility

We deliberately broke compatibility for some add-on kludges made in TurboMail in order to get a much nicer interface in TurboMail 3. The test mode of TurboMail 2 is completely unsupported although the same functionality is provided by the DebugTransport (see unit testing with TurboMail). Any code that accessed turbomail._queue directly will be broken in TurboMail 3.

As noted above, we changed the default manager behavior to be synchronous and the default transport (DebugTransport) does not deliver any messages to the outside world.

In TurboMail 2.x using the 'UTF-8-QP' encoding had the side effect that Quoted Printable was used to encode all messages with UTF-8. After sending one message with UTF-8-QP all later messages using just UTF-8 would be

encoded with Quoted Printable. In TurboMail 3 using the 'UTF-8-QP' encoding will not affect other messages. If you want to have all UTF-8 encoded messages delivered with Quoted Printable, you need to enable the *UTF8qp extension*.

CHAPTER

THREE

DETAILED TURBOMAIL

3.1 RFC (2)822 - From, Sender, Disposition, what?

This section is dedicated to the plain text format in which email messages are mostly transmitted and focuses on routing information like author and recipients. Only if you know the basics about RFC 822 and SMTP, you can use TurboMail's capabilities to a full extend.

3.1.1 The Anatomy of a message

The original format for email messages was defined in RFC 822 which was superseded by RFC 2822 [1]. The newest standard document about the format is currently RFC 5322. But the basics of RFC 822 still apply, so for the sake of readability we will just use 'RFC 822' to refer to all these RFCs. Please read the official standard documents if this text fails to explain some aspects.

Email messages consist of header lines which contain meta information like the subject and creation date and a body part where the actual message text and attachments are placed. Each header field has a name and a value, separated by a colon (":"):

```
From: foo@example.com
Date: Sun, 26 Oct 2008 12:30:42 -0000
Subject: Test message
```

This should just give you an idea, how email headers look like. This - hopefully - leads to a better understanding of the following paragraphs. We did not talk about the format of the message body because it has no strict format like the header lines and generally you don't need a deeper understanding of that because TurboMail's Message class will take care of generating the message body. If you want to learn more, just examine the emails in your inbox (use the 'view source' command of your email client), have a look at the Wikipedia article about e-mail, or just read the standard documents cited above.

3.1.2 Originator fields

Email messages can have one or more authors which appear in the 'From' line. The author is probably the one who created the contents of the message body. Sometimes, the author is not the same as the one who actually sent the message, e.g. the secretary sends the message for his boss. In this case, the boss is still the author of the message (his name appears in the From: line) but the 'Sender' header tells who did send the mail actually so here you would see the secretary's name. RFC 5322 says: 'The "Sender:" field specifies the mailbox of the agent responsible for the actual transmission of the message.'

[1] There were several other extensions to these RFCs like RFC 2045 through RFC 2049 for MIME message bodies, but most of the message header lines are still unchanged since the publication of RFC 822 - 'Standard for the Format of ARPA Internet Text Messages'.

If there are multiple authors of a message, the explicit specification of a sender is mandated by the RFCs. Otherwise the sender header may be omitted.

3.1.3 Who should receive the message?

There are multiple headers which do encode which addresses should receive this message. The best known header is 'To' where recipients are listed to whom the message text is addressed. Sometimes it can be useful send other people a copy of the message also so they get the information too. This is what the 'CC' header is for. If you don't want send information to someone else too but don't want to reveal this to the other recipients, you can put addresses in 'BCC' which is not a real header but TurboMail will send the message to the addresses listed there too. None of these headers is actually mandated by the RFCs so you can omit any or all of them (e.g. for a newsletter you could decide only to put people in BCC so that subscribers don't see each others email addresses).

3.1.4 Who should receive replies?

Whe you use email as a communication medium, you must be prepared that the recipients will use email for their replies. So you should always provide a working email address which can receive the replies.

Sometimes the author should not receive replies to the message he sent. E.g. the project manager tells everyone in her team about some event but another employee organizes the event so he should get all messages related to that event. You can set a header named 'Reply-To' so that the recipient's email client will use the specified address as a the recipient for a reply. Of course the user may change that but he has to do this explicitly. By default his reply will go to the address specified in the 'Reply-To' header.

When no reply-to address was given, replies will be sent to the from address(es). The sender address is not used for replies.

3.1.5 The SMTP envelope

It is important to differentiate between the message format as defined in RFC822 and SMTP. In the sections above you read something about originator fields, recipients and the like. But when you send email messages with SMTP, all these header fields don't have any effect! During the transport with SMTP your email (headers+body part) is just treated as data [2].

The mechanism works like letter envelopes for snail mail: The post man (SMTP server) will only look at the envelope address to find out to whom the mail should be delivered. In your letter (email header) you can address the letter to whoever you like, the post man will deliver it to the address written on the envelope . So in a SMTP transaction, you have a list of recipients ("RCPT TO") where all recipient email addresses are listed. When you send a message through TurboMail, all addresses from your "To", "CC" and "BCC" header fields are collected and used for "RCPT TO". The "RCPT TO" list is not visible in the delivered message. This mechanism is used for the BCC ("blind carbon copy") property. All addresses in BCC just appear in "RCPT TO" so the recipients don't see if others got the mail too.

In contrary to normal letters, you can send the same message (with the same contents!) to multiple recipients: The SMTP server will take care about sending each recipient a copy of this message. This can potentially same quite a big amount of bandwidth because the message (including attachments) can be easily some megabytes big. But this only works if you send exactly the same message to everyone - if you want to use personalized messages which contain the recipient's name in the message body, you have to generate individual messages for each recipient.

And last but not least there is the SMTP From address. If the post man can not deliver a letter (e.g. the address is wrong), it will be returned to the sender. Exactly the same principle is true for email: There is one address used as sender in the SMTP communication. Normally the address in the 'Sender' header is used. If this is not present, the from address will be taken. All delivery failures ("bounces", e.g. user unknown, mailbox is over quota) are sent to this

[2] Actually, this not quite true because SMTP servers may look into the message to identify spam and other malicious mails. But in general the message routing is totally unaffected by the headers in your email.

address. For example if you sent out a newsletter you may want to use an address for SMTP From which is different from the 'From' header so all bounces will go to a mailbox which is not monitored by human operators. Instead you can use scripts to disable addresses in your database which produce bounces with a permanent error code [3]. In TurboMail you can change the default behavior by setting the 'smtp_from' property of a message explicitly.

Please note that remote SMTP servers are likely to require a valid "SMTP From" address (otherwise they will reject your message immediately). There are many checks to ensure that you provide valid address and of course you can "trick" all checks but you really should provide an email address to which bounces can be delivered - even if this is just a black hole which may be appropriate in some cases.

Gotcha: When you submit an email via authenticated SMTP to a mail server, some mail servers replace your sender header (or create a new one) if the From address is not the same as the authentication name. In the Exim MTA the configuration option is called 'local_from_check'.

That's why you can not trust the "To" header in messages you receive: They don't have any effect on the message routing. So may receive messages addressed to billg@microsoft.com ("To" header) because the sender put a different address in "RCPT TO" than in the message header.

3.2 Managers

Every manager component implements a specific strategy when to send messages after they were submitted via message.send(). TurboMail comes with two managers by default: The ImmediateManager and the DemandManager. However, there can be only one active manager in TurboMail. You can configure the manager to use by changing the 'mail.manager' configuration option.

3.2.1 ImmediateManager

The ImmediateManager is extremly simple: Every message is immediately handed over to the transport, all operations are synchronous. This is the manager which is used by default if you don't configure anything.

3.2.2 DemandManager

The DemandManager behaves like TurboMail 2.x: It can spawn multiple threads so that your application is not blocked if you send many messages at once or your mail server is slow.

Configuration options: * mail.manager = 'demand' – enable the demand manager * mail.demand.threads (integer, default: 4) – maximum number of worker threads * mail.demand.divisor (integer, default: 10) – divide the queue size by this number to estimate the number of required threads * mail.demand.timeout (integer, default: 60) – number of seconds a worker thread can be idle before shutting down

Once the DemandManager got a message, it will try to deliver it. You won't get any notification about a failed delivery. Also the DemandManager holds the queue in memory so all unsent messages are lost if you stop the application. Currently fixing these shortcomings is planned for TurboMail 3.1 (patches welcome!).

3.3 Transports - Your Work Horses

Transports actually **deliver** your message so that it can reach the specified recipients like the SMTPTransport. Other possible transports (not yet written) would be some SendmailTransport (calls /usr/bin/sendmail on the command line) or AppEngineMailServiceTransport.

[3] Please note that there is no standardized format for bounce messages and error codes. Therefore you likely need to write a "parser" for different mailer daemons.

3.3.1 SMTPTransport

The SMTP transport sends messages to a SMTP server (smart host).

Configuration options:

- **mail.smtp.server** (string, mandatory) host name or IP address of the SMTP server which will act as a relay for all mails
- mail.smtp.username (string)
- mail.smtp.password (string)
- mail.smtp.tls (boolean, default: None): Use TLS for the server connection (If 'None' is set TurboMail will attempt to auto-detect TLS which might if your server has a bad configuration)
- mail.smtp.debug (boolean, default: False): Print SMTP communication to STDERR
- mail.smtp.max_messages_per_connection (integer, default: 1): The number of messages that are sent over one connection. Increasing the number can help your performance if there is a high latency on connections to your mail server but you might find more (generally harmless) log messages about connections which timed out.

If password and username were given, TurboMail will try to authenticate to the server with these credentials before sending messages.

Notes

Please be aware that your SMTP server can enforce additional restrictions. A very common example are checks for valid recipient and sender addresses (email address is syntactically valid and the domain exists). If you do not ensure that your message fulfills these criteria, the message won't be delivered. If you use the ImmediateManager you may need to catch these exceptions (most notably SMTPSenderRefused and SMTPRecipientsRefused). Please note that SMTPRecipientsRefused exception is only raised if **all** recipients were rejected. If you send a message to two recipients and just one was rejected, you will not see any exception.

If your SMTP server does not listen on the standard SMTP port 25, you need to specifiy the port in the 'mail.smtp.server' option (e.g. 'localhost:4711').

3.3.2 DebugTransport

This is the default transport in case you did not configure a transport explicitly (in order to minimize the risk that you accidentally send out messages). The debug transport just collects all messages which were sent to TurboMail. No delivery takes place. This is very useful for unit tests where you don't want to have interactions with external servers.

There are no configuration options for the DebugTransport, just a method to retrieve the sent messages:

- get_sent_mails() - return a list of collected messages

3.4 TurboMail Extensions - Leverage Super Powers

TurboMail Extensions will be extended in a way that you can add custom behavior to Messages easily (e.g. encryption). However due to time constraints the interface was not finalized for 3.0. Therefore this interface will probably be changed within the 3.x series.

3.4.1 UTF8qp

Unfortunately Python's email module will encode the body part using base64 by default if you have a UTF-8 character set. As written before TurboMail provides a fake encoding called 'UTF-8-QP' which works around that.

In TurboMail 2.0 the behaviour was to reconfigure the 'UTF-8' encoding for messages so that also 'UTF-8' would be encoded with Quoted Printable. This extension provides backwards compatibility by reconfiguring the global UTF-8 setting to use the quoted-printable encoding. Please note that this will affect other Python modules in the same process using the email module.

To enable this extension, you need to enable it in your configuration:

```
mail.utf8qp.on          = True
```

3.5 TurboMail Adapters

TurboMail comes with several so-called 'adapters' which ease the usage of TurboMail in a certain framework. For example an adapter will configure TurboMail in a way so that it can use your framework's configuration mechanism.

3.5.1 TurboGears 1.x

TurboGears 1 uses setuptools to find 'extensions' (entry point 'turbogears.extensions'). If you installed TurboMail correctly, TurboGears 1 will start and stop TurboMail automatically, you don't need to write a single line of custom code. Just put TurboMail's configuration in your app.cfg or dev.cfg/prod.cfg.

3.5.2 Pylons

Pylons tries to be as flexible as possible so there is not fixed 'extension' mechanism in place (as of 0.9.7). Therefore you need to add two lines in your startup configuration. While there are many places where you could add this code, we recommend doing it in the constructor of your Globals object (yourpackage.lib.app_globals):

```
def __init__(self):
    # ...
    from turbomail.adapters import tm_pylons
    tm_pylons.start_extension()
```

To configure TurboMail, just use the [DEFAULT] section in your development.ini/production.ini files.

As there is no shutdown hook either, TurboMail tries to use Python's atexit module to do all the necessary cleanup tasks. Please note that atexit will not call the registered handlers in all cases.

Pylons' configuration module does not convert the configuration values to Python types other than strings (as opposed to ConfigObj used in TurboGears 1). The Pylons adapter calls as_bool() on values for keys that ends with '.on'. Configuration values which only consist of digits will be converted to int. Everything else is returned unmodified.

For Pylons you don't have to quote strings in the configuration files. Strings surrounded with quotations marks will be returned unmodified:

```
# Wrong:
# mail.manager = 'demand'

# Good:
mail.manager = demand
```

3.5.3 TurboGears 2

TurboGears 2 is based on Pylons so for now you can just use the Pylons adapter. When TurboGears Ticket 2206 is implemented, we can provide a custom adapter for TurboGears 2 that does not require any code modification.

3.5.4 Others

We like to add TurboGears for other popular frameworks like Trac and Django. However we did not have any time to do that in a sane way so far. Feel free to send patches...

3.6 TurboMail from Scratch

If you don't use a supported framework that means one which has a TurboMail Adapter you have to start the TurboMail machinery by yourself. You can even use TurboMail for Python command line scripts! One goal of TurboMail 3.0 was to get rid of the TurboGears dependency [4]. Therefore starting TurboMail is quite easy:

```
from turbomail.control import interface

turbomail_config = {
    'mail.on': True,
    'mail.transport': 'smtp',
    'mail.smtp.server': 'localhost',
}
# ...
interface.start(turbomail_config)
# now you can send messages
# ...
interface.stop()
```

If your framework has a sophisticated configuration mechanism you probably want to use this instead of hard coding all configuration options by hand. TurboMail just expects that the object used for the configuration provides a dict-like get() method which is how most configuration mechanisms work. If there were no TurboGears 1 adapter, you could write:

```
import turbogears
from turbomail.control import interface

interface.start(turbogears.config)
# ...
interface.stop()
```

Thereby you can leverage all the advantages of having different configurations: TurboGears has a general application and deployment/environment specific configurations which are merged at runtime. So you can change settings easily which change often (e.g. mail server credentials, logging) while leaving most of the other stuff untouched.

3.6.1 Gotcha

Don't forget to call interface.stop() before your application terminates. This is especially important if you use a manager which may send messages asynchronously like the DemandManager. If you don't call interface.stop() and the DemandManager has still some mails in the queue when your app is finished, you will loose these mails!

[4] Don't get us wrong: The authors of TurboMail rely on TurboGears a lot but we felt that TurboMail could be useful for more people than just for TurboGears developers.

CHAPTER

FOUR

EXTENDING TURBOMAIL

4.1 Writing Custom TurboMail Extensions

All implementations of specific behaviors (e.g. managers) can be built using TurboMail extensions. The basic extension interface provided by TurboMail is very simple and takes care of basic life-cycle management:

- The start() method is called before the extension is used the first time. If the extension returns True, TurboMail considers it running. If it returns False, some error happened and the extension instance is considered faulty.
- The stop() method is called when the extension is not needed any longer. The extension should release any resources (e.g. existing connections or locks) but it must not raise any exceptions.

Every extension instance is started and stopped only once. This is being guaranteed by the super class so you must call super() for all overriden methods!

4.2 Writing Custom Managers

Every manager has a method get_new_transport() which can be used to get a new transport instance ready to deliver messages. The manager is responsible to manager the lifecycle of the returned transport instance so it needs to call the stop() method when the transport is not used any more (e.g. the manager's stop() method was called or the manager wants to replace the existing transport instance with a new transport.

managers must be thread safe

4.3 Writing Custom Transports

Interface

turbomail.api.Transport

deliver(message) stop()

helper method config_get(key, default=None, tm2_key=None) “”“Returns the value for the given key from the configuration. If the value was not found, the default value (default None) is returned. If tm2_key was specified, “”“

transports don't have to be thread safe

4.4 Clever plain text generation for your rich text messages

Every HTML email should have a plain text part. If you really need produce nice text emails, you should build the plain text part by hand, the same way you do with the HTML part. But sometimes you just need a text part without much work (this is especially true if your mails' contents do change frequently). So TurboMail has its own naïve implementation of an HTML to text converter called `TagStripper` which is applied to your rich text part if you don't provide any plain text.

There are quite a few, very nice HTML-to-text libraries in the Python space but we did not feel comfortable including one of them in TurboMail due to licensing restrictions (TurboMail should be usable under the MIT license), additional requirements (TurboMail itself should have as few external dependencies as possible and some of those converters requires quite a few packages for themselves) and maintenance (we don't want to ship a private copies of external libraries which we have to keep secure and up-to-date). So we decided to continue shipping a simple converter (like in TurboMail 2) but give you the option to add your own.

4.4.1 TagStripper - Do the simplest thing that could possibly work

`TagStripper` is meant to be the simplest implementation of an html to text converter that is of any use: It just strips the tags from the HTML but does not touch the contents (so it does not care about HTML entities like) at all. The only exception to this rule are script, header and style tags - TagStripper will cut the contents of these tags too because usually they don't contain any information which is useful for plain text messages.

4.4.2 YourMegaCleverHtmlToTextConverter

So you want to roll your own HTML to text converter. First of all you encapsulate your algorithm in a class which inherits from `BaseHTML2TextConverter`:

```
from turbomail.html2text import BaseHTML2TextConverter
# TODO: Probably this should be moved to the api package

class MyHuperDuperConverter(BaseHTML2TextConverter):
    def convert(self, html_content):
        plain_text = ... # Convert the HTML
        return plain_text
```

After that you can configure TurboMail so that it uses your converter. # TODO: What API should we make? # extension point 'turbomail.html2text' # turbomail.html2text = "MyHuperDuper" # interface.start(html2text_converters={ })

4.5 Writing Custom Adapters - use TurboMail at new places

adapters must be thread safe, stop and start can be called multiple times, should not make any difference

4.6 We need you - How to Contribute

4.7 Future Tasks

- Currently TurboMail does not care about queues. A manager may have a queue which holds all mails in-memory but this is up to every manager. I would like to see a general queue concept with two initial implementations:

The InMemoryQueue (same as now) and the DiskStorageQueue which tries to ensure that mails never get lost.

- Making TurboMail less global. Currently there is *one* manager and *one* transport which are stored in global variables. I like to see support for different 'channels' which may have different strategies of sending messages.

CHAPTER

FIVE

TURBOMAIL COMMUNITY

5.1 Getting Help

5.2 Installation of TurboMail

Traditionally, this is among the first chapters but I find this annoying because mostly it's only about './configure && make && make install'. We should make installing TurboMail as easy as possible

INDEX