:mod:`email` Package Architecture

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Lib\email\[cpython-main][Lib][email]architecture.rst`, line 1); *backlink*

Unknown interpreted text role "mod".

Overview

The email package consists of three major components:

Model
: An object structure that represents an email message, and provides an API for creating, querying, and modifying a message.

Parser
: Takes a sequence of characters or bytes and produces a model of the email message represented by those characters or bytes.

Generator
: Takes a model and turns it into a sequence of characters or bytes. The sequence can either be intended for human consumption (a printable unicode string) or bytes suitable for transmission over the wire. In the latter case all data is properly encoded using the content transfer encodings specified by the relevant RFCs.

Conceptually the package is organized around the model. The model provides both "external" APIs intended for use by application programs using the library, and "internal" APIs intended for use by the Parser and Generator components. This division is intentionally a bit fuzzy; the API described by this documentation is all a public, stable API. This allows for an application with special needs to implement its own parser and/or generator.

In addition to the three major functional components, there is a third key component to the architecture:

Policy
: An object that specifies various behavioral settings and carries implementations of various behavior-controlling methods.

The Policy framework provides a simple and convenient way to control the behavior of the library, making it possible for the library to be used in a very flexible fashion while leveraging the common code required to parse, represent, and generate message-like objects. For example, in addition to the default RFC 5322 email message policy, we also have a policy that manages HTTP headers in a fashion compliant with RFC 2616. Individual policy controls, such as the maximum line length produced by the generator, can also be controlled individually to meet specialized application requirements.

The Model

The message model is implemented by the :class:`~email.message.Message` class. The model divides a message into the two fundamental parts discussed by the RFC: the header section and the body. The *Message* object acts as a pseudo-dictionary of named headers. Its dictionary interface provides convenient access to individual headers by name. However, all headers are kept internally in an ordered list, so that the information about the order of the headers in the original message is preserved.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Lib\email\[cpython-main][Lib][email]architecture.rst`, line 51); *backlink*

Unknown interpreted text role "class".

The *Message* object also has a *payload* that holds the body. A *payload* can be one of two things: data, or a list of *Message* objects. The latter is used to represent a multipart MIME message. Lists can be nested arbitrarily deeply in order to represent the message, with all terminal leaves having non-list data payloads.

Message Lifecycle

The general lifecycle of a message is:

Creation
: A *Message* object can be created by a Parser, or it can be instantiated as an empty message by an application.

Manipulation
: The application may examine one or more headers, and/or the payload, and it may modify one or more headers and/or the payload. This may be done on the top level *Message* object, or on any sub-object.

Finalization
    The Model is converted into a unicode or binary stream, or the model is discarded.

Header Policy Control During Lifecycle

One of the major controls exerted by the Policy is the management of headers during the *Message* lifecycle. Most applications don't need to be aware of this.

A header enters the model in one of two ways: via a Parser, or by being set to a specific value by an application program after the Model already exists. Similarly, a header exits the model in one of two ways: by being serialized by a Generator, or by being retrieved from a Model by an application program. The Policy object provides hooks for all four of these pathways.

The model storage for headers is a list of (name, value) tuples.

The Parser identifies headers during parsing, and passes them to the :meth:`~email.policy.Policy.header_source_parse` method of the Policy. The result of that method is the (name, value) tuple to be stored in the model.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Lib\email\[cpython-main][Lib][email]architecture.rst`, line 102);** ***backlink***
>
> Unknown interpreted text role "meth".

When an application program supplies a header value (for example, through the *Message* object *__setitem__* interface), the name and the value are passed to the :meth:`~email.policy.Policy.header_store_parse` method of the Policy, which returns the (name, value) tuple to be stored in the model.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Lib\email\[cpython-main][Lib][email]architecture.rst`, line 106);** ***backlink***
>
> Unknown interpreted text role "meth".

When an application program retrieves a header (through any of the dict or list interfaces of *Message*), the name and value are passed to the :meth:`~email.policy.Policy.header_fetch_parse` method of the Policy to obtain the value returned to the application.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Lib\email\[cpython-main][Lib][email]architecture.rst`, line 111);** ***backlink***
>
> Unknown interpreted text role "meth".

When a Generator requests a header during serialization, the name and value are passed to the :meth:`~email.policy.Policy.fold` method of the Policy, which returns a string containing line breaks in the appropriate places. The :meth:`~email.policy.Policy.cte_type` Policy control determines whether or not Content Transfer Encoding is performed on the data in the header. There is also a :meth:`~email.policy.Policy.binary_fold` method for use by generators that produce binary output, which returns the folded header as binary data, possibly folded at different places than the corresponding string would be.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Lib\email\[cpython-main][Lib][email]architecture.rst`, line 116);** ***backlink***
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Lib\email\[cpython-main][Lib][email]architecture.rst`, line 116);** ***backlink***
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Lib\email\[cpython-main][Lib][email]architecture.rst`, line 116);** ***backlink***
>
> Unknown interpreted text role "meth".

Handling Binary Data

In an ideal world all message data would conform to the RFCs, meaning that the parser could decode the message into the idealized unicode message that the sender originally wrote. In the real world, the email package must also be able to deal with badly formatted messages, including messages containing non-ASCII characters that either have no indicated character set or are not valid characters in the indicated character set.

Since email messages are *primarily* text data, and operations on message data are primarily text operations (except for binary payloads of course), the model stores all text data as unicode strings. Un-decodable binary inside text data is handled by using the *surrogateescape* error handler of the ASCII codec. As with the binary filenames the error handler was introduced to handle, this allows the email package to "carry" the binary data received during parsing along until the output stage, at which time it is regenerated in its original form.

This carried binary data is almost entirely an implementation detail. The one place where it is visible in the API is in the "internal" API. A Parser must do the *surrogateescape* encoding of binary input data, and pass that data to the appropriate Policy method. The "internal" interface used by the Generator to access header values preserves the *surrogateescaped* bytes. All other interfaces convert the binary data either back into bytes or into a safe form (losing information in some cases).

Backward Compatibility

The :class:`~email.policy.Policy.Compat32` Policy provides backward compatibility with version 5.1 of the email package. It does this via the following implementation of the four+1 Policy methods described above:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Lib\email\[cpython-main][Lib][email]architecture.rst`, line 157); *backlink*

Unknown interpreted text role "class".

header_source_parse

Splits the first line on the colon to obtain the name, discards any spaces after the colon, and joins the remainder of the line with all of the remaining lines, preserving the linesep characters to obtain the value. Trailing carriage return and/or linefeed characters are stripped from the resulting value string.

header_store_parse

Returns the name and value exactly as received from the application.

header_fetch_parse

If the value contains any *surrogateescaped* binary data, return the value as a :class:`~email.header.Header` object, using the character set *unknown-8bit*. Otherwise just returns the value.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Lib\email\[cpython-main][Lib][email]architecture.rst`, line 172); *backlink*

Unknown interpreted text role "class".

fold

Uses :class:`~email.header.Header`'s folding to fold headers in the same way the email5.1 generator did.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Lib\email\[cpython-main][Lib][email]architecture.rst`, line 177); *backlink*

Unknown interpreted text role "class".

binary_fold

Same as fold, but encodes to 'ascii'.

New Algorithm

header_source_parse

Same as legacy behavior.

header_store_parse

Same as legacy behavior.

header_fetch_parse

If the value is already a header object, returns it. Otherwise, parses the value using the new parser, and returns the resulting object as the value. *surrogateescaped* bytes get turned into unicode unknown character code points.

fold

Uses the new header folding algorithm, respecting the policy settings. surrogateescaped bytes are encoded using the `unknown-8bit` charset for `cte_type=7bit` or `8bit`. Returns a string.

At some point there will also be a `cte_type=unicode`, and for that policy fold will serialize the idealized unicode message with RFC-like folding, converting any surrogateescaped bytes into the unicode unknown character glyph.

binary_fold

Uses the new header folding algorithm, respecting the policy settings. surrogateescaped bytes are encoded using the *unknown-8bit* charset for `cte_type=7bit`, and get turned back into bytes for `cte_type=8bit`. Returns bytes.

At some point there will also be a `cte_type=unicode`, and for that policy binary_fold will serialize the message according to :rfc:`5335`.