Chapter 1

Overview

The etar library provides support for archiving data using the pax/tar format.

1.1 Concepts

1.1.1 Archive

An archive is a series of blocks of size 512 bytes. An entry consists of exactly one header block, followed by zero or more payload blocks. At the end of the archive there are two consecutive blocks with all zero bytes.

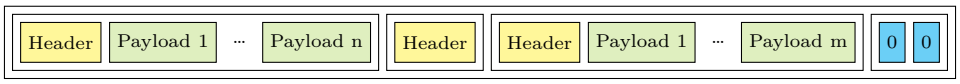

Figure 1.1: Structure of an archive

1.2 Error Handling

API still unstable

Chapter 2

ARCHIVE

The `ARCHIVE` class is the central piece of this library. It allows clients to manipulate archives.

2.1 Initialization

To create a new instance of the `ARCHIVE` class, the client has to use the make feature and provide a `STORAGE_BACKEND` which will then be used to do I/O. A client then can either open the archive for archiving or for unarchiving. It is not possible to archive and unarchive simultaneously.

2.2 Archiving

To use the archiving mode, one has to use the `open_archive` feature of `ARCHIVE`. Then, one can add entries using `add_entry`. Once all entries are written, the user has to call finalize, which will write the end-of-archive indicator and then close the archive.

2.3 Unarchiving

To use the unarchiving mode, one has to call `open_unarchive` after creation. Next, the client may register arbitrarily many `UNARCHIVER`s using `add_unarchiver`. Once all of them were added, the client may either use `unarchive` which will then unarchive all entries and close the archive or `unarchive_next_entry` until `unarchiving_finished` becomes `True`. In the latter case, the client has to call `close` himself.

Chapter 3

STORAGE_BACKEND

STORAGE_BACKEND provides a unified interface for different storage methods an archive could use. Currently the only implementation is FILE_STORAGE_BACKEND, providing support for archives that are stored in a file.

3.1 FILE_STORAGE_BACKEND

A FILE_STORAGE_BACKEND is either created from a file with make_from_file or from a filename with make_from_filename.

3.2 Implementing a Custom STORAGE_BACKEND

To implement a custom STORAGE_BACKEND, one has to implement the following features:

3.2.1 open_read

open_read
Open backend for read access. Reading should start from the beginning.

3.2.2 open_write

open_write
Open backend for write access. Writing should start from the beginning.

3.2.3 close

close
Close backend.

3.2.4 archive_finished

archive_finished: BOOLEAN
Indicate whether the next two blocks contain the end-of-archive indicator (only zero bytes). The next read_block calls should not skip these two blocks but read them again (not

necessarily from the backend again, the implementation is free to chache these blocks). `archive_finished` should return `True` too, if an error occured (or occurs while checking for the end-of-archive indicator), does not have enough blocks available or if the backend is closed.

3.2.5 block_ready

`block_ready: BOOLEAN`
Indicate whether there is a block that can be read with `last_block` `False` if an error occured.

3.2.6 is_readable

`is_readable: BOOLEAN`
Indicates whether this backend can be read from. If an error occured, this has to return `False`

3.2.7 is_writable

`is_writable: BOOLEAN`
Indicates whether this backend can be written to. If an error occured, this has to return `False`

3.2.8 is_closed

`is_closed: BOOLEAN`
Indicates whether this backend is closed.

3.2.9 read_block

`read_block`
Read next block from backend. If there are not enough bytes for a full block, an error should be reported.

3.2.10 last_block

`last_block: MANAGED_POINTER`
Last block that was read.

3.2.11 write_block

`write_block (block: MANAGED_POINTER)`
Write `block` to the backend (starting from the beginning).

3.2.12 finalize

`finalize`
Write the end-of-archive indicator and close backend.

Chapter 4

ARCHIVABLE

Chapter 5

UNARCHIVER