NAME

MIME−tools − modules for parsing (and creating!) MIME entities

SYNOPSIS

Here's some pretty basic code for **parsing a MIME message,** and outputting its decoded components to a given directory:

```
use MIME::Parser;

### Create parser, and set some parsing options:
my $parser = new MIME::Parser;
$parser−>output_under("$ENV{HOME}/mimemail");

### Parse input:
$entity = $parser−>parse(\*STDIN) or die "parse failed\n";

### Take a look at the top−level entity (and any parts it has):
$entity−>dump_skeleton;
```

Here's some code which **composes and sends a MIME message** containing three parts: a text file, an attached GIF, and some more text:

```
use MIME::Entity;

### Create the top−level, and set up the mail headers:
$top = MIME::Entity−>build(Type    =>"multipart/mixed",
                           From    => "me\ AT myhost DOT com",
                           To      => "you\ AT yourhost DOT com",
                           Subject => "Hello, nurse!");

### Part #1: a simple text document:
$top−>attach(Path=>"./testin/short.txt");

### Part #2: a GIF file:
$top−>attach(Path        => "./docs/mime−sm.gif",
             Type        => "image/gif",
             Encoding    => "base64");

### Part #3: some literal text:
$top−>attach(Data=>$message);

### Send it:
open MAIL, "| /usr/lib/sendmail −t −oi −oem" or die "open: $!";
$top−>print(\*MAIL);
close MAIL;
```

For more examples, look at the scripts in the **examples** directory of the MIME-tools distribution.

DESCRIPTION

MIME-tools is a collection of Perl5 MIME:: modules for parsing, decoding, *and generating* single− or multipart (even nested multipart) MIME messages. (Yes, kids, that means you can send messages with attached GIF files).

REQUIREMENTS

You will need the following installed on your system:

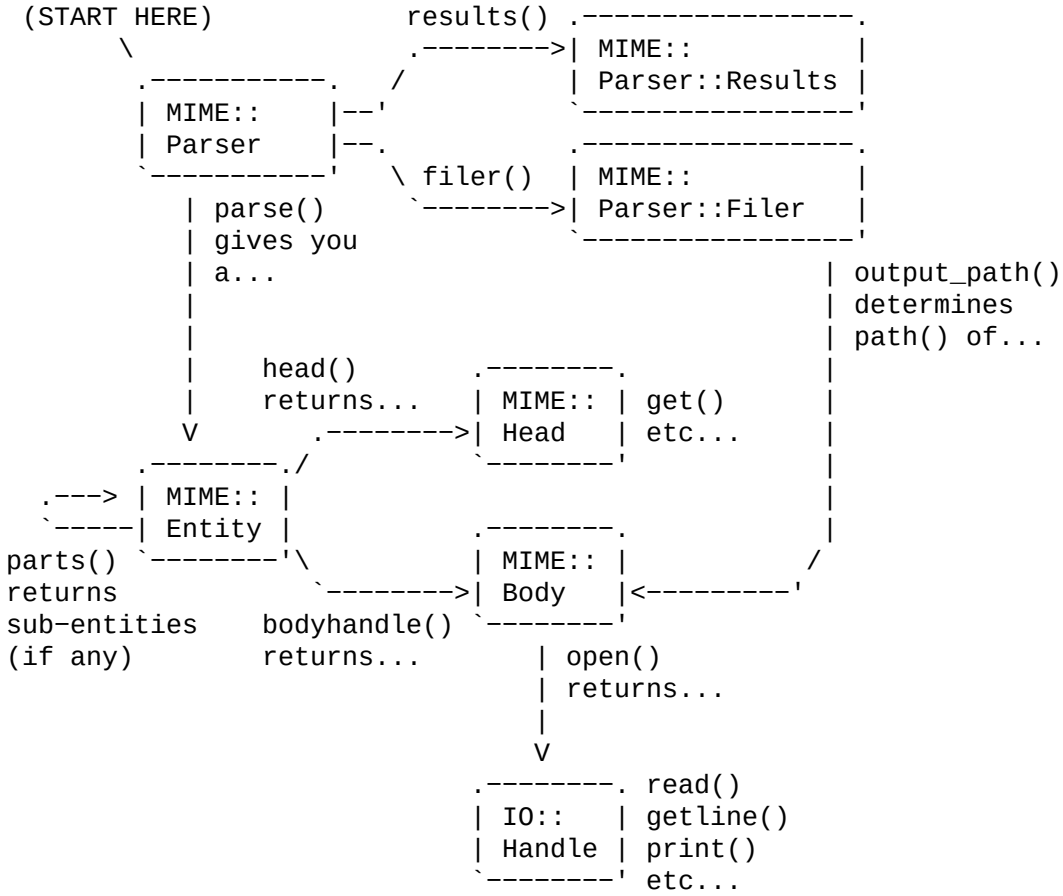
```
File::Path
File::Spec
IPC::Open2              (optional)
MIME::Base64
MIME::QuotedPrint
Net::SMTP
Mail::Internet, ...     from the MailTools distribution.
```

See the Makefile.PL in your distribution for the most-comprehensive list of prerequisite modules and their version numbers.

A QUICK TOUR

Overview of the classes

Here are the classes you'll generally be dealing with directly:



To illustrate, parsing works this way:

- **The "parser" parses the MIME stream.** A parser is an instance of `MIME::Parser`. You hand it an input stream (like a filehandle) to parse a message from: if the parse is successful, the result is an ''entity''.
- **A parsed message is represented by an "entity".** An entity is an instance of `MIME::Entity` (a subclass of `Mail::Internet`). If the message had ''parts'' (e.g., attachments), then those parts are ''entities'' as well, contained inside the top-level entity. Each entity has a ''head'' and a ''body''.
- **The entity's "head" contains information about the message.** A ''head'' is an instance of `MIME::Head` (a subclass of `Mail::Header`). It contains information from the message header: content type, sender, subject line, etc.
- **The entity's "body" knows where the message data is.** You can ask to ''open'' this data source for *reading* or *writing*, and you will get back an ''I/O handle''.
- **You can *open()* a "body" and get an "I/O handle" to read/write message data.** This handle is an object that is basically like an IO::Handle... it can be any class, so long as it supports a small,

standard set of methods for reading from or writing to the underlying data source.

A typical multipart message containing two parts — a textual greeting and an ‘‘attached’’ GIF file — would be a tree of MIME::Entity objects, each of which would have its own MIME::Head. Like this:

```
.--------.
| MIME:: | Content-type: multipart/mixed
| Entity | Subject: Happy Samhaine!
`--------'
     |
     `----.
    parts |
          |   .--------.
          |---| MIME:: | Content-type: text/plain; charset=us-ascii
          |   | Entity | Content-transfer-encoding: 7bit
          |   `--------'
          |   .--------.
          |---| MIME:: | Content-type: image/gif
              | Entity | Content-transfer-encoding: base64
              `--------' Content-disposition: inline;
                           filename="hs.gif"
```

Parsing messages

You usually start by creating an instance of **MIME::Parser** and setting up certain parsing parameters: what directory to save extracted files to, how to name the files, etc.

You then give that instance a readable filehandle on which waits a MIME message. If all goes well, you will get back a **MIME::Entity** object (a subclass of **Mail::Internet**), which consists of...

- A **MIME::Head** (a subclass of **Mail::Header**) which holds the MIME header data.
- A **MIME::Body**, which is a object that knows where the body data is. You ask this object to ‘‘open’’ itself for reading, and it will hand you back an ‘‘I/O handle’’ for reading the data: this could be of any class, so long as it conforms to a subset of the **IO::Handle** interface.

If the original message was a multipart document, the MIME::Entity object will have a non-empty list of ‘‘parts’’, each of which is in turn a MIME::Entity (which might also be a multipart entity, etc, etc...).

Internally, the parser (in MIME::Parser) asks for instances of **MIME::Decoder** whenever it needs to decode an encoded file. MIME::Decoder has a mapping from supported encodings (e.g., ’base64’) to classes whose instances can decode them. You can add to this mapping to try out new/experiment encodings. You can also use MIME::Decoder by itself.

Composing messages

All message composition is done via the **MIME::Entity** class. For single-part messages, you can use the **MIME::Entity/build** constructor to create MIME entities very easily.

For multipart messages, you can start by creating a top-level `multipart` entity with ***MIME::Entity::build()***, and then use the similar ***MIME::Entity::attach()*** method to attach parts to that message. *Please note:* what most people think of as ‘‘a text message with an attached GIF file’’ is *really* a multipart message with 2 parts: the first being the text message, and the second being the GIF file.

When building MIME a entity, you’ll have to provide two very important pieces of information: the *content type* and the *content transfer encoding*. The type is usually easy, as it is directly determined by the file format; e.g., an HTML file is `text/html`. The encoding, however, is trickier... for example, some HTML files are `7bit`-compliant, but others might have very long lines and would need to be sent `quoted-printable` for reliability.

See the section on encoding/decoding for more details, as well as ‘‘A MIME PRIMER’’ below.

Sending email

Since MIME::Entity inherits directly from Mail::Internet, you can use the normal Mail::Internet mechanisms to send email. For example,

```
$entity->smtpsend;
```

Encoding/decoding support

The **MIME::Decoder** class can be used to *encode* as well; this is done when printing MIME entities. All the standard encodings are supported (see ''A MIME PRIMER'' below for details):

```
Encoding:         | Normally used when message contents are:
------------------------------------------------------------------
7bit              | 7-bit data with under 1000 chars/line, or multipart.
8bit              | 8-bit data with under 1000 chars/line.
binary            | 8-bit data with some long lines (or no line breaks).
quoted-printable  | Text files with some 8-bit chars (e.g., Latin-1 text).
base64            | Binary files.
```

Which encoding you choose for a given document depends largely on (1) what you know about the document's contents (text vs binary), and (2) whether you need the resulting message to have a reliable encoding for 7-bit Internet email transport.

In general, only `quoted-printable` and `base64` guarantee reliable transport of all data; the other three ''no-encoding'' encodings simply pass the data through, and are only reliable if that data is 7bit ASCII with under 1000 characters per line, and has no conflicts with the multipart boundaries.

I've considered making it so that the content-type and encoding can be automatically inferred from the file's path, but that seems to be asking for trouble... or at least, for Mail::Cap...

Message-logging

MIME-tools is a large and complex toolkit which tries to deal with a wide variety of external input. It's sometimes helpful to see what's really going on behind the scenes. There are several kinds of messages logged by the toolkit itself:

Debug messages

These are printed directly to the STDERR, with a prefix of `"MIME-tools: debug"`.

Debug message are only logged if you have turned ''debugging'' on in the MIME::Tools configuration.

Warning messages

These are logged by the standard Perl *warn()* mechanism to indicate an unusual situation. They all have a prefix of `"MIME-tools: warning"`.

Warning messages are only logged if `$^W` is set true and MIME::Tools is not configured to be ''quiet''.

Error messages

These are logged by the standard Perl *warn()* mechanism to indicate that something actually failed. They all have a prefix of `"MIME-tools: error"`.

Error messages are only logged if `$^W` is set true and MIME::Tools is not configured to be ''quiet''.

Usage messages

Unlike ''typical'' warnings above, which warn about problems processing data, usage-warnings are for alerting developers of deprecated methods and suspicious invocations.

Usage messages are currently only logged if `$^W` is set true and MIME::Tools is not configured to be ''quiet''.

When a MIME::Parser (or one of its internal helper classes) wants to report a message, it generally does so by recording the message to the **MIME::Parser::Results** object immediately before invoking the appropriate function above. That means each parsing run has its own trace-log which can be examined for problems.

Configuring the toolkit

If you want to tweak the way this toolkit works (for example, to turn on debugging), use the routines in the **MIME::Tools** module.

debugging

Turn debugging on or off. Default is false (off).

```
MIME::Tools->debugging(1);
```

quiet

Turn the reporting of warning/error messages on or off. Default is true, meaning that these message are silenced.

```
MIME::Tools->quiet(1);
```

version

Return the toolkit version.

```
print MIME::Tools->version, "\n";
```

THINGS YOU SHOULD DO

Take a look at the examples

The MIME-Tools distribution comes with an ‘‘examples’’ directory. The scripts in there are basically just tossed-together, but they’ll give you some ideas of how to use the parser.

Run with warnings enabled

Always run your Perl script with `-w`. If you see a warning about a deprecated method, change your code ASAP. This will ease upgrades tremendously.

Avoid non-standard encodings

Don’t try to MIME-encode using the non-standard MIME encodings. It’s just not a good practice if you want people to be able to read your messages.

Plan for thrown exceptions

For example, if your mail-handling code absolutely must not die, then perform mail parsing like this:

```
$entity = eval { $parser->parse(\*INPUT) };
```

Parsing is a complex process, and some components may throw exceptions if seriously-bad things happen. Since ‘‘seriously-bad’’ is in the eye of the beholder, you’re better off *catching* possible exceptions instead of asking me to propagate `undef` up the stack. Use of exceptions in reusable modules is one of those religious issues we’re never all going to agree upon; thankfully, that’s what `eval{}` is good for.

Check the parser results for warnings/errors

As of 5.3xx, the parser tries extremely hard to give you a MIME::Entity. If there were any problems, it logs warnings/errors to the underlying ‘‘results’’ object (see MIME::Parser::Results). Look at that object after each parse. Print out the warnings and errors, *especially* if messages don’t parse the way you thought they would.

Don’t plan on printing exactly what you parsed!

Parsing is a (slightly) lossy operation. Because of things like ambiguities in base64-encoding, the following is *not* going to spit out its input unchanged in all cases:

```
$entity = $parser->parse(\*STDIN);
$entity->print(\*STDOUT);
```

If you’re using MIME::Tools to process email, remember to save the data you parse if you want to send it on unchanged. This is vital for things like PGP-signed email.

Understand how international characters are represented

The MIME standard allows for text strings in headers to contain characters from any character set, by using special sequences which look like this:

```
=?ISO-8859-1?Q?Keld_J=F8rn_Simonsen?=
```

To be consistent with the existing Mail::Field classes, MIME::Tools does *not* automatically unencode these strings, since doing so would lose the character-set information and interfere with the parsing of fields (see ‘‘decode_headers’’ in MIME::Parser for a full explanation). That means you should be prepared to deal with these encoded strings.

The most common question then is, **how do I decode these encoded strings?** The answer depends on what you want to decode them *to*: ASCII, Latin1, UTF-8, etc. Be aware that your ‘‘target’’ representation may not support all possible character sets you might encounter; for example, Latin1 (ISO-8859-1) has no way of representing Big5 (Chinese) characters. A common practice is to represent ‘‘untranslateable’’ characters as ‘‘?’’s, or to ignore them completely.

To unencode the strings into some of the more-popular Western byte representations (e.g., Latin1, Latin2, etc.), you can use the decoders in MIME::WordDecoder (see MIME::WordDecoder). The simplest way is by using `unmime()`, a function wrapped around your ‘‘default’’ decoder, as follows:

```
use MIME::WordDecoder;
...
$subject = unmime $entity->head->get('subject');
```

One place this *is* done automatically is in extracting the recommended filename for a part while parsing. That’s why you should start by setting up the best ‘‘default’’ decoder if the default target of Latin1 isn’t to your liking.

THINGS I DO THAT YOU SHOULD KNOW ABOUT

Fuzzing of CRLF and newline on input

RFC 2045 dictates that MIME streams have lines terminated by CRLF (`"\r\n"`). However, it is extremely likely that folks will want to parse MIME streams where each line ends in the local newline character `"\n"` instead.

An attempt has been made to allow the parser to handle both CRLF and newline-terminated input.

Fuzzing of CRLF and newline when decoding

The `"7bit"` and `"8bit"` decoders will decode both a `"\n"` and a `"\r\n"` end-of-line sequence into a `"\n"`.

The `"binary"` decoder (default if no encoding specified) still outputs stuff verbatim... so a MIME message with CRLFs and no explicit encoding will be output as a text file that, on many systems, will have an annoying ^M at the end of each line... *but this is as it should be*.

Fuzzing of CRLF and newline when encoding/composing

TODO FIXME All encoders currently output the end-of-line sequence as a `"\n"`, with the assumption that the local mail agent will perform the conversion from newline to CRLF when sending the mail. However, there probably should be an option to output CRLF as per RFC 2045

Inability to handle multipart boundaries with embedded newlines

Let’s get something straight: this is an evil, EVIL practice. If your mailer creates multipart boundary strings that contain newlines, give it two weeks notice and find another one. If your mail robot receives MIME mail like this, regard it as syntactically incorrect, which it is.

Ignoring non-header headers

People like to hand the parser raw messages straight from POP3 or from a mailbox. There is often predictable non-header information in front of the real headers; e.g., the initial ‘‘From’’ line in the following message:

```
From - Wed Mar 22 02:13:18 2000
Return-Path: <eryq AT zeegee DOT com>
Subject: Hello
```

The parser simply ignores such stuff quietly. Perhaps it shouldn’t, but most people seem to want that behavior.

Fuzzing of empty multipart preambles

Please note that there is currently an ambiguity in the way preambles are parsed in. The following message fragments *both* are regarded as having an empty preamble (where `\n` indicates a newline character):

```
Content-type: multipart/mixed; boundary="xyz"\n
Subject: This message (#1) has an empty preamble\n
\n
--xyz\n
...

Content-type: multipart/mixed; boundary="xyz"\n
Subject: This message (#2) also has an empty preamble\n
\n
\n
--xyz\n
```

. . .

In both cases, the *first* completely-empty line (after the ‘‘Subject’’) marks the end of the header.

But we should clearly ignore the *second* empty line in message #2, since it fills the role of *“the newline which is only there to make sure that the boundary is at the beginning of a line”*. Such newlines are *never* part of the content preceding the boundary; thus, there is no preamble ‘‘content’’ in message #2.

However, it seems clear that message #1 *also* has no preamble ‘‘content’’, and is in fact merely a compact representation of an empty preamble.

Use of a temp file during parsing

Why not do everything in core? Although the amount of core available on even a modest home system continues to grow, the size of attachments continues to grow with it. I wanted to make sure that even users with small systems could deal with decoding multi-megabyte sounds and movie files. That means not being core-bound.

As of the released 5.3xx, MIME::Parser gets by with only one temp file open per parser. This temp file provides a sort of infinite scratch space for dealing with the current message part. It’s fast and lightweight, but you should know about it anyway.

Why do I assume that MIME objects are email objects?

Achim Bohnet once pointed out that MIME headers do nothing more than store a collection of attributes, and thus could be represented as objects which don’t inherit from Mail::Header.

I agree in principle, but RFC 2045 says otherwise. RFC 2045 [MIME] headers are a syntactic subset of RFC−822 [email] headers. Perhaps a better name for these modules would have been RFC1521:: instead of MIME::, but we’re a little beyond that stage now.

When I originally wrote these modules for the CPAN, I agonized for a long time about whether or not they really should subclass from **Mail::Internet** (then at version 1.17). Thanks to Graham Barr, who graciously evolved MailTools 1.06 to be more MIME-friendly, unification was achieved at MIME-tools release 2.0. The benefits in reuse alone have been substantial.

A MIME PRIMER

So you need to parse (or create) MIME, but you’re not quite up on the specifics? No problem...

Glossary

Here are some definitions adapted from RFC 1521 (predecessor of the current RFC 204[56789] defining MIME) explaining the terminology we use; each is accompanied by the equivalent in MIME:: module terms...

attachment

An ‘‘attachment’’ is common slang for any part of a multipart message — except, perhaps, for the first part, which normally carries a user message describing the attachments that follow (e.g.: ‘‘Hey dude, here’s that GIF file I promised you.’’).

In our system, an attachment is just a **MIME::Entity** under the top-level entity, probably one of its parts.

body

The ‘‘body’’ of an entity is that portion of the entity which follows the header and which contains the real message content. For example, if your MIME message has a GIF file attachment, then the body of that attachment is the base64−encoded GIF file itself.

A body is represented by an instance of **MIME::Body**. You get the body of an entity by sending it a *bodyhandle()* message.

body part

One of the parts of the body of a multipart **/entity**. A body part has a **/header** and a **/body**, so it makes sense to speak about the body of a body part.

Since a body part is just a kind of entity, it’s represented by an instance of **MIME::Entity**.

entity

An ‘‘entity’’ means either a **/message** or a **/body part**. All entities have a **/header** and a **/body**.

An entity is represented by an instance of **MIME::Entity**. There are instance methods for recovering the header (a **MIME::Head**) and the body (a **MIME::Body**).

header
: This is the top portion of the MIME message, which contains the ‘‘Content-type’’, ‘‘Content-transfer-encoding’’, etc. Every MIME entity has a header, represented by an instance of **MIME::Head**. You get the header of an entity by sending it a *head()* message.

message
: A ‘‘message’’ generally means the complete (or ‘‘top-level’’) message being transferred on a network.

  There currently is no explicit package for ‘‘messages’’; under MIME::, messages are streams of data which may be read in from files or filehandles. You can think of the **MIME::Entity** returned by the **MIME::Parser** as representing the full message.

Content types
: This indicates what kind of data is in the MIME message, usually as *majortype/minortype*. The standard major types are shown below. A more-comprehensive listing may be found in RFC−2046.

  application
  : Data which does not fit in any of the other categories, particularly data to be processed by some type of application program. `application/octet-stream`, `application/gzip`, `application/postscript`...

  audio
  : Audio data. `audio/basic`...

  image
  : Graphics data. `image/gif`, `image/jpeg`...

  message
  : A message, usually another mail or MIME message. `message/rfc822`...

  multipart
  : A message containing other messages. `multipart/mixed`, `multipart/alternative`...

  text Textual data, meant for humans to read. `text/plain`, `text/html`...

  video
  : Video or video+audio data. `video/mpeg`...

Content transfer encodings
: This is how the message body is packaged up for safe transit. There are the 5 major MIME encodings. A more-comprehensive listing may be found in RFC−2045.

  7bit
  : No encoding is done at all. This label simply asserts that no 8−bit characters are present, and that lines do not exceed 1000 characters in length (including the CRLF).

  8bit
  : No encoding is done at all. This label simply asserts that the message might contain 8−bit characters, and that lines do not exceed 1000 characters in length (including the CRLF).

  binary
  : No encoding is done at all. This label simply asserts that the message might contain 8−bit characters, and that lines may exceed 1000 characters in length. Such messages are the *least* likely to get through mail gateways.

  base64
  : A standard encoding, which maps arbitrary binary data to the 7bit domain. Like ‘‘uuencode’’, but very well-defined. This is how you should send essentially binary information (tar files, GIFs, JPEGs, etc.).

  quoted-printable
  : A standard encoding, which maps arbitrary line-oriented data to the 7bit domain. Useful for encoding messages which are textual in nature, yet which contain non-ASCII characters (e.g., Latin−1, Latin−2, or any other 8−bit alphabet).

SEE ALSO

MIME::Parser, MIME::Head, MIME::Body, MIME::Entity, MIME::Decoder, Mail::Header, Mail::Internet

At the time of this writing, the MIME-tools homepage was *http://www.mimedefang.org/static/mime−tools.php*. Check there for updates and support.

The MIME format is documented in RFCs 1521−1522, and more recently in RFCs 2045−2049.

The MIME header format is an outgrowth of the mail header format documented in RFC 822.

SUPPORT

Please file support requests via rt.cpan.org.

CHANGE LOG

Released as MIME-parser (1.0): 28 April 1996. Released as MIME-tools (2.0): Halloween 1996. Released as MIME-tools (4.0): Christmas 1997. Released as MIME-tools (5.0): Mother's Day 2000.

See ChangeLog file for full details.

AUTHOR

Eryq (*eryq AT zeegee DOT com*), ZeeGee Software Inc (*http://www.zeegee.com*). Dianne Skoll (*dfs AT roaringpenguin DOT com*) *http://www.roaringpenguin.com*.

ACKNOWLEDGMENTS

This kit would not have been possible but for the direct contributions of the following:

```
Gisle Aas             The MIME encoding/decoding modules.
Laurent Amon          Bug reports and suggestions.
Graham Barr           The new MailTools.
Achim Bohnet          Numerous good suggestions, including the I/O model.
Kent Boortz           Initial code for RFC−1522−decoding of MIME headers.
Andreas Koenig        Numerous good ideas, tons of beta testing,
                        and help with CPAN−friendly packaging.
Igor Starovoitov      Bug reports and suggestions.
Jason L Tibbitts III  Bug reports, suggestions, patches.
```

Not to mention the Accidental Beta Test Team, whose bug reports (and comments) have been invaluable in improving the whole:

```
Phil Abercrombie
Mike Blazer
Brandon Browning
Kurt Freytag
Steve Kilbane
Jake Morrison
Rolf Nelson
Joel Noble
Michael W. Normandin
Tim Pierce
Andrew Pimlott
Dragomir R. Radev
Nickolay Saukh
Russell Sutherland
Larry Virden
Zyx
```

Please forgive me if I've accidentally left you out. Better yet, email me, and I'll put you in.

LICENSE