imag 0.3.0

User Documentation

August 2017

Contents

1 Introduction

This document is the user documentation for imag, the personal information management suite for the commandline. Besides beeing a documentation, it servey also as "roadmap" where this project should go. Parts which are not yet implemented might be documented already, therefore. A list on what is implemented and what is not can be found at the end of this document.

If you have any objections, suggestions for improvements, bugs, etc, please file them. A way to reach out to the imag project maintainer(s) is described in the CONTRIBUTING file of the repository or in this document, on the appropriate section.

1.1 The Problem

The problem imag wants to solve is rather simple. When the project was initiated, there was no PIM-Suite available which

- was for this domain of users ("power-users", "commandline users")
- uses plain text as storage format
- was scriptable
- was modular
- contained functionality to link content

The latter point is the bigger one: "imag" wants to offer the ability for users to link content. This means not only that a contact may be linked to a date, but that *all things* can be linked together. For example that a wiki article can be linked to a date which is linked to a todo which is linked to a note which is linked to a contact.

1.2 The Approach

The approach "imag" takes on solving this problem is to store content in a "store" and persisting content in a unified way. Meta-Information is attached to the content which can be used to store structured data. This can be used to implement a variety of "domain modules" using the store. While content is stored in *one* place, imag does not duplicate content. imag does not copy or move icalendar files, emails, vcard files, music or movies to the store, but indexes them and stores the meta-information in the store.

Detailed explanation on this approach follows in the chapters of this work.

1.3 Implementation

The program is written in the Rust programming language.

The program consists of libraries which can be re-used by other projects to implement and adapt imag functionality. An external program may use a library of the imag distribution to store content in the store of imag and make it visible to imag this way.

This is a technical detail a user does not necessarily need to know, but as imag is intended for power-users anyways, we could say it fits here.

2 Architecture of the imag code

The imag codebase has a rather simple overall architecture. But before introducing the reader to it, a few things have to be introduced.

2.1 Crate types

There are different types of crates in the imag world. A crate is a rust project.

First of all, there are core crates. These crates provide the very core of imag and almost all other crates use them:

- libimagstore - The imag store is the abstraction overbthe filesystem. It provides primitives to get, write and manipulate store entries and their header information.
- libimagrt - The runtime library, which provides functionality to create a store object from libimagstore, helps with configurarion loading and commandline argument handling (through the external "clap" crate).
- libimagerror - Error handling library for handling errors the imag way. Used in all other crates, even the store itself. It also offers functionality to log and trace errors as well as exiting the application, if necessary.
- libimagutil - Utilities.

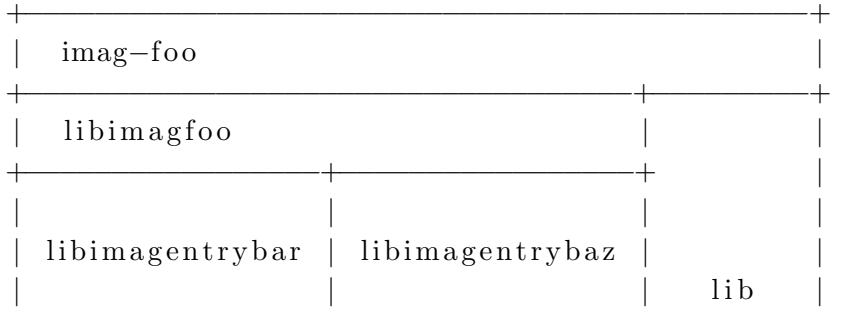
The next type of imag crates are entry extension libraries. Those provide extensional functionality for the types from libimagstore. For example, there is "libimagentrylink" which provides functionality to link two entries in the store.

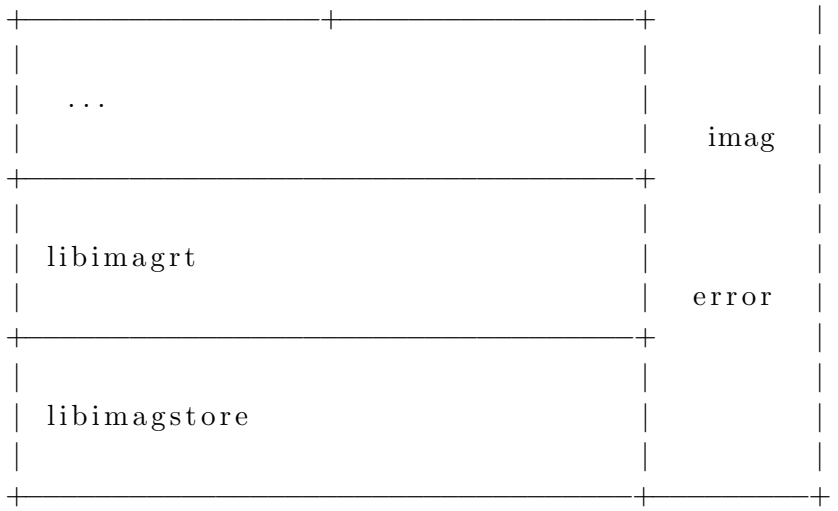
The next kind of crate is the one that offers end-user functionality for a imag aspect, for example "libimagtodo" provides functionality to track todos.

And last, but not least, the commandline frontend crates provide the user interface. These are the kind of crates that are not library crates, but binaries.

2.2 Architecture of an imag module

With the things from above, a module could have the following architecture:





The foundation of all imag modules is the store, as one can see from the image. Above this there is the libimagrt, which provides the basic runtime and access to the Store object. Cross-cutting, there is the error library (and possibly the util library, but we do not care about this one here), which is used through all levels. The highest level of all imag modules is the commandline interface on top of the domain library. In between can be any number of entry extension libraries, or none if not needed.

2.3 Types

The imag core, hence the libimagstore, libimagrt and libimagerror, provide a set of types that a user (as in a library writer) should be aware of.

First of all, there is the Runtime type which is provided by the libimagrt. It provides basic access to whether debugging or verbosity is enabled as well as the most important core object: The Store.

It is provided by the libimagstore library, the heart of everything.

When interacting with the store, two types are visible: FileLockEntry and Entry whereas the former derefs to the latter, which basically means that the former wraps the latter. The FileLockEntry is a necessary wrapper for ensuring that when working concurrently with the store, an entry is only *borrowed* once from the store. It also ensures that the object is alive as long as the store is.

The Entry type provides functionality like reading the actual content, its header and so on. Extensions for its functionality are implemented on this type, not on the FileLockEntry.

The Entry provides access to its header, which is a toml::Value, where toml is the toml-rs crate (external project). Conveniance functionality is provided via the toml−query crate, which is an external project which was initiated and extracted from the imag project.

Error types are also important. All errors in the imag prijects are generated by libimagerror usage and are interoperable. User code hardly handles the actual Error type but its inner one, ErrorKind, which is an enum where each member can be mapped to a representing text. Imag error types do never contain payload (they can via extensions, though it is not really practical and there is no usage for it).

3 The Store

The store is where all the good things happen. The store is basically just a directory on the filesystem imag manages and keeps its state in.

One could say that the store is simply a databases, and it really is. We opted to go for plain text, though, as we believe that plain text is the only sane way to do such a thing. A user should always be able to read her data without great effort and putting everything in a *real* database like sqlite or even postgresql would need a user to install additional software just to read his own data. We don't want that. Text is readable until the worlds end and we think it is therefor better to store the data in plain text.

The following sections describe the store and the file format we use to store data. One may skip the following sections, they are included for users who want to dig into the store with their editors.

3.1 File Format

The contents of the store are encoded in either UTF-8 or ASCII. Either way, a normal text editor (like vim or the other one) will always be sufficient to dog into the store and modify files. For simple viewing even a pager (like `less`) is sufficient.

Each entry in the store consists of two parts:

1. Header
2. Content

The following section describe their purpose.

3.1.1 Header Format

The header format is where imag stores its data. The header is an area at the top of every file which is seperated from the content part by three dashes (`---`). Between these three dashes there is structured data. imag uses TOML as data format for this structured data, because it fits best and the available TOML parser for the rust programming language is really good.

The header can contain any amount of data, but modules (see Section 5) are restricted in their way of altering the data.

So normally there are several sections in the header. One section (`[imag]`) is always present. It contains a version field, which tells imag which version this file was created with (the version information is *also* encoded in the filename, just in case things change in the future). It also contains a `links` field which is an Array of values. This `links` field is for linking (see Section 3.6) to other entries in the store.

Other sections are named like the modules which created them. Every module is allowed to store arbitrary data under its own section and a module may never read other sections than its own. This is not enforced by imag itself, though.

3.1.2 Content Format

The content is the part of the file where the user is free to enter any textual content. The content may be rendered as Markdown or other markup format for the users convenience. The store does never expect and specific markup and actually the markup implementation is not inside the very code of imag.

Technically it would be possible that the content part of a file is used to store binary data. We don't want this, though.

3.1.3 Example

An example for a file in the store follows.

```
---
[imag]
links = ["/home/user/more_kittens.mpeg"]
version = "0.3.0"

[note]
name = "foo"
---

This is an example text, written by the user.
```

3.2 File organization

The "Entries" are stored as files in the "Store", which is a directory the user has access to. The store may exist in the users Home-directory or any other directory the user has read-write-Access to.

Each module stores its data in an own subdirectory in the store. This is because we like to keep things ordered and clean, not because it is technically necessary.

We name the path to a file in the store "Store id" or "Storepath" and we often refer to it by using the store location as root. So if the store exists in /home/user/store/, a file with the storepath /example.file is (on the filesystem) located at /home/user/store/example.file.

A storepath contains predefined parts:

- The module name of the Module the Entry belongs to, as said above. This part is always a directory.
- The version (semantic versioning applies) of the module storing the entry. This part is a postfix to the filename.

The pattern for the storepath is

```
/<module name>/<optional sub-folders>/<file name>~<sem version>
```

So if a module named “example-module” with version “0.1.0” stores a file in the Store, the storepath for a file with the name “example” is “/example-module/example~0.1.0”.

Any number of subdirectories may be used, so creating folder hierarchies is possible and valid. A file “example” for a module “module” in version “0.1.0” would be stored in subfolders like this:

```
/module/some/sub/folder/example~0.1.0
```

For example, it is valid if these files exist at the same time:

- /foo/bar~0.2
- /foo/bar~1.3

It might not be sane, though.

To future-proof the system it is necessary to provide a way for modules to differentiate in their versions on the store level. Thus if a module wants to retrieve a file from the store it must at least accept files from it’s current advertised version. It may accept older files and it may transform them and resubmit them in the newer version.

3.3 Store path links

Linking entries is version independent.

This means if an entry “a” from a module “A” gets written to the store, it may link to an entry “b” from a module “B”, which is in version “0.1.0” at the moment. If the module “B” gets updated, it might update its entries in the store as well. The link from the “a” should never get invalid in this case, though it is not ensured by the core of imag itself.

3.4 Backends

The store itself also has a backend. This backend is the “filesystem abstraction” code.

Note: This is a very core thing. Casual users might want to skip this section.

3.4.1 Problem

First, we had a compiletime backend for the store. This means that the actual filesystem operations were compiled into the stores either as real filesystem operations (in a normal debug or release build) but as a in-memory variant in the ‘test’ case. So tests did not hit the filesystem when running. This gave us us the possibility to run tests concurrently with multiple stores that did not interfere with eachother.

This approach worked perfectly well until we started to test not the store itself but crates that depend on the store implementation. When running tests in a crate that depends on the store, the store itself was compiled with the filesystem-hitting-backend. This was problematic, as tests could not be implemented without hitting the filesystem.

Hence we implemented this.

3.4.2 Implementation

The filesystem is abstracted via a trait FileAbstraction which contains the essential functions for working with the filesystem.

Two implementations are provided in the code:

- FSFileAbstraction
- InMemoryFileAbstraction

whereas the first actually works with the filesystem and the latter works with an in-memory HashMap that is used as filesystem.

Further, the trait FileAbstractionInstance was introduced for functions which are executed on actual instances of content from the filesystem, which was previousely tied into the general abstraction mechanism.

So, the FileAbstraction trait is for working with the filesystem, the FileAbstractionInstance trait is for working with instances of content from the filesystem (speak: actual Files).

In case of the FSFileAbstractionInstance, which is the implementation of the FileAbstractionInstance for the actual filesystem-hitting code, the underlying resource is managed like with the old code before. The InMemoryFileAbstractionInstance implementation is corrosponding to the InMemoryFileAbstraction implementation - for the in-memory "filesystem".

3.5 The StdIo backend

Sidenote: The name is "StdIo" because its main purpose is Stdin/Stdio, but it is abstracted over Read/Write actually, so it is also possible to use this backend in other ways, too.

3.5.1 Why?

This is a backend for the imag store which is created from stdin, by piping contents into the store (via JSON or TOML) and piping the store contents (as JSON or TOML) to stdout when the the backend is destructed.

This is one of some components which make command-chaining in imag possible. With this, the application does not have to know whether the store actually lives on the filesystem or just "in memory".

3.5.2 Mappers

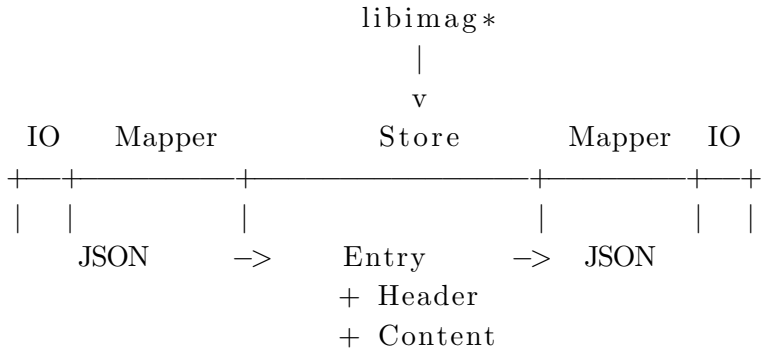
The backend contains a "Mapper" which defines how the contents get mapped into the in-memory store representation: A JSON implementation or a TOML implementation are possible.

The following section assumes a JSON mapper.

The mapper reads the JSON, parses it (thanks serde!) and translates it to a Entry, which is the in-memory representation of the files. The Entry contains a Header part and a Content part.

This is then made available to the store codebase.

To summarize what we do right now, lets have a look at the awesome ascii-art below:



This is what gets translated where for one imag call with a stdio store backend.

3.5.3 The JSON Mapper

The JSON mapper maps JSON which is read from a source into a HashMap which represents the in-memory filesystem.

The strucure is as follows:

```
{
    "version": "0.3.0",
    "store": {
        "example": {
            "header": {
                "imag": {
                    "version": "0.3.0",
                },
            },
            "content": "hi there!",
        },
    },
}
```

3.5.4 TODO

If you look at the version history of this file you will see that this implementation has grown from something complex and probably slow to what we have today.

Still, there's one improvement we could make: abstract all the things away so the libimag* crates handle the header without knowing whether it is JSON or TOML. With this, we would not even have to translate JSON to TOML anymore. We should measure whether this would have actually any performance impact before implementing it.

3.6 Linking from an store entry

As described in Section 1.1 the purpose of imag is to *link* content together. The following section describes, from a technical view, how this is done in imag.

There are two ways of linking in imag. You can either link internally or externally. The following sections describe the differences.

3.6.1 Linking to internal content

Internal links are links between store entries themselfes. This means that one store entry can link to another. Actually, links are not pointers but rather tries between entries, meaning that an link is not directed, but always a two-way pointer.

How linking works from the user interface is described in Section 5.17.

3.6.2 Linking to external content

Linking to external content means linking to files or directories which do not live inside the store itself but outside of it.

Each store entry can store *one link to external content at most.*

External linking should not be used from the user interface but rather the ref feature (Section 5.28) should be used. Section 5.28 describes why that is.

4 Conventions, best practices

This section goes about best practices in the imag codebase. It is mainly focused on developers, but a user may read it for getting to know how imag works.

Lets work our way up from the store and how to extend it to the commandline user interface.

4.1 Store and Entry functionality

A Entry does not offer much functionality by itself. So its the job of libraries to *extend* their functionality. This should never be done by wrapping the Entry type itself but by providing and implementing an extension trait on it.

Same goes for extending the Store type: never wrap it, always provide an extension trait for it.

These two rules ensure that the type does not lose any functionality from a wrapping. Deref could do that, but not over muliple levels, so extension traits it is. It also most likely results in functions inside the extension trait which all return a Result<_, _>.

4.2 Libraries

In the next few sections, conventions and best practices for writing a imag library are written down.

A developer of imag should read this carefully, a user may skip this section or cross-read it for better understanding of the imag project.

4.2.1 Library naming

Libraries which provide functionality for entries or the store (most likely entries or both) should be named "libimagentrything" whereas "thing" stands for what the library provides.

All other libraries should be prefixed with "libimag" at least. Most likely, one will not write such a library but rather a "libimagentrything" library.

4.2.2 Library scope

A library should never introduce utility functionality which could be useful for other libraries as well. If there is no such functionality available, the "libimagutil" might be a place where such a function would be put, or, if not yet available, a "libimagentryutil" could be created.

If a library has to introduce free functions in its public interface, one should think hard whether this is really necessary.

4.2.3 Library error types/kinds

Libraries must use the "libimagerror" tools to create error types and kinds. Most likely, a library needs some kinds for wrapping the errors from underlying libraries, such as the store itself.

A library must *never* introduce multiple error types, but is free to introduce as many error kinds as required. Indeed, more kinds is better than fewer.

4.2.4 Libraries with commandline frontends

Libraries with commandline frontends provide end-user functionality. Normally, they depend on one or more "libimagentrything" libraries. They should be named "libimagthing", though. For example: "libimagdiary", "libimagtimetrack" or "libimagwiki", whereas the commandline frontends would be "imag-diary", "imag-timetrack" and "imag-wiki", respectively.

If such a library needs to depend on another "libimagthing", for example if "libimagdiary" needs to depend on "libimagnote", one should think about this and whether the functionality could ve outsourced to a more general "libimagentrything". It is not forbidden, though.

A library which implements a functionality for imag may contain helper functions for commandline stuff, but that is discouraged.

4.2.5 Library testing

All libraries should be tested as much as possible. Sometimes it may not be possible without a lot of effort, but still: more tests = better!

4.3 Commandline tools

The next few sections describe how the commandline frontends are implemented. Each imag functionality (or module) has its own library and a commandline frontend for it.

The commandline frontends do contain little to no functionality. They simply translate the commandline parameters and options to calls to the appropriate library functions.

4.4 Commandline tool testing

4.5 Commandline interface

5 Modules

A module is a functionality of the program. There is a huge list of modules available in the imag core distribution.

From a naming perspective, we do not differ between low-level and high-level modules. Some of the modules shipped with imag cover core functionality such as linking, tagging or references to files outside of the store or even the store interface itself (which by the way shouldn't be used by the end-user at all). Others cover things like diary, notes, wiki or bookmarks.

We try really hard to offer a consistent commandline user interface over all of these modules.

The following sections describe each module in detail, including its purpose and its provided backends.

A backend is simply an external tool imag might be able to use. For example, the imag−todo module offers a taskwarrior interface, so imag itself does not cover anything which has to do with todo management, but lets you continue using taskwarrior for that (which does a really good job). So what does the imag−todo module do? Well, it offers you ways to track tasks created in taskwarrior and putting files which can be used as references to tasks then. For example, if you create a task in taskwarrior, you end up with an UUID for this task. imag stores this UUID in a store entry and you are now able to imag−link this file with other files in the store. This way you can link taskwarrior tasks with other data (of course, imag−todo offers some more commands, for searching tasks and so on).

But what if you do not like taskwarrior? That's what backends are for. The goal of imag is to provide backends for not just one tool which implements a PIM aspect, but for many. So you can change the configuration for imag−todo to not use taskwarrior but some other todo tool.

(This is all hypothetical by now because these things are not yet implemented. Anyhow, we aim for exactly what is described above)

5.1 Bibliography

The Bibliography module.

5.2 Bookmarks

The Bookmarks module is for keeping URLs as bookmarks, tagging and categorizing them and finally also open them in the browser.

5.3 Borrow

The Borrow module.

5.4 Calendar

The calendar module implements a commandline calendar like khal. The calendar data itself is retrieved from icalendar files which should be located outside of the imag store. imag does not handle syncing of these files. vdirsyncer may be your tool of choise here.

imag can show events from the calendar(s) like any other commandline calendar tool and of course can also add, delete or edit entries (interactively or via commandline parameters).

5.4.1 Internals

What imag does internally is described in this section.

imag creates one entry in the store for one icalendar file. These entries are basically references to the real data. If an icalendar file is removed from the filesystem, imag does not delete it from the sfore if not told explicitely.

5.5 Category

The Category module is a plumbing command for setting an entrys category. A category must exist before it can be set for an entry. That, and that each entry may have one or no category is the difference from tags.

Also: Categories may have sub-categories.

5.6 Contacts

The Contacts module serves as a vcard viewer which is also able to alter, vcard files (either interactively or via commandline parameters).

The contacts module can also call other programs and pass contact information to them, for example mutt.

5.7 Counter

The Counter module helps you counting things.

In its current state the counter module is capable of simple counting. You can create, list and delete counters which are simply numbers and incremet, decrement, set and reset them.

Future plans include counting functionality which is able to save date and possibly timestamp of your increments/decrements, so you can export this and use (for example) R to visualize this data.

Filters for selecting only certain time ranges when listing/exporting your counters will be added as well.

5.7.1 Examples

Here are some examples how to use the counter module:

```
imag counter create --name example --initval 42 # or: -n example -i 42
imag counter --inc example # or -i example
imag counter --reset example
imag counter --dec example # or -d example
```

5.8 Cuecards

The Cuecards module implements "cuacards-learning" like you probably did it in school.

5.9 Diary

The diary module is for keeping your diary notes.

The diary module giv3s you the possibility to write your diary in imag. It offers daily, hourly and minutely entries (the latter beeing more like a private tumble-blog).

Exporting the diary is possible, so one can write it in markdown and later pass that to pandoc, if desired, to generate a website or book from it.

5.10 Filter

The Filter module is only of use when chaining up imag calls via bash pipes. It can be used to filter out entries based on some parameters, like for example whether a certain header field is set or not.

5.11 Git

The Git module provises a convenient way to call the git executable on the imag store without having to cd to it first. Nothing more.

5.12 GPS

The GPS module is a plumbing command for attaching a GPS coordinate to an entry.

5.13 Habit

The Habit module is a habit tracker. One can add habits, specify how often they should be done and instantiate them.

Example: After creating a new habit "Sunday Run", which should be done on sundays, one can mark (only on sundays of course) that the habit was done. Statistics and number-crunching can be done later on, after there is some habit data there.

Exports to CSV are possible.

5.14 Images

The Images module is for tagging, categorizing and sorting images. GPS coordinates can be attached to image references. Images can be put into collections. Image viewing programs can be called from imag. If an image gets modified (for example via darktable) and a new image file is created, the Image module can be used to group them.

5.15 Item

The Item module is a plumbing command to create entries for items in the imag store.

Items can be anything. For example, one could create a Tomato and a Bread as item to add them later in the shopping list (as in imag-shoppinglist), but also a computer and a printer can be created to use them later in a project (as in imag-project).

5.16 Ledger

The Ledger module implements a ledger like beancount.

5.17 Link

The linking module imag−link is one of the plumbing modules. It offers the possibility to link entries in the store.

It also offers the functionality to link to external sources. This functionality *can* be used to link to external URLs, but the bookmarking module should be used to do this (see Section 5.2).

The linking module offers functionality to add, remove and list both internal (store entry to store entry) and external (store entry to URL) links.

5.17.1 Internal linking

5.17.2 External linking

A store entry can only have *one* external link. Therefor, when you create an external link, the linking module creates a new entry in the store which links to this URL. The linking module then links you entry with this new entry by using an internal link. This way one entry can have multiple external links attached to it and external links are deduplicated automatically.

5.18 Mails

The Mails module implements a commandline email client. Emails can be written (via $EDITOR) and viewed, also in threads. Emails can be crawled for creating new contacts.

A Text User Interface is not planned, but might be there at some point.

The mail module implements a minimal Email client. It does not handle IMAP syncing or SMTP things, it is just a *viewer* for emails (a MUA).

The goal of the initial implementation is only a CLI, not a TUI like mutt offers, for example (but that might be implemented later). As this is an imag module, it also creates references to mails inside the imag store which can be used by other tools then (for example imag−link to link an entry with a mail - or the imag entry representing that mail).

So this module offers functionality to read (Maildir) mailboxes, search for and list mails and mail-threads and reply to mails (by spawning the $EDITOR).

Outgoing mails are pushed to a special directory and can later on be send via imag−mail which calls a MTA (for example msmtp) and also creates store entries for the outgoing mails.

5.18.1 CLI

The CLI of the imag-mail module is planned as follows:

```
imag mail track <path> [opts...]   # track a new mail, mail file passed as p
imag mail scan <path> [opts...]    # scan a maildir and track all untracked
imag mail box <name|path>          # work with the mailbox specified by <nam
imag mail list <args...>           # list mails in a given mailbox for a giv
imag mail show <args...>           # open new mails in the pager
imag mail thread list <args...>    # list mails from a thread
imag mail thread show <args...>    # open new mails from a thread in the pag
imag mail new <args...>            # craft a new mail and safe it in the <ou
imag mail send <args...>           # send emails from the outgoing folder, o
imag mail mv <srcbox> <dstbox>     # move a mail (or thread) from one mailbo
```

5.19 Movies

The Movies module is for categorizing, rating and tagging movies.

5.20 Music

The Music module is for rating, categorizing, tagging and enjoying music. It offers functionality to fetch lyrics, create automatically linkings from genre entries to music files (music files are represented as a entry jn imag), combining songs in albums and albums in artists, etc.

A scrobble server may be implemented at some point to provide more ways to retrieving data over ones music taste. Suggested songs (from the own library of music or via external tools like musicbrainz) may be a feature that comes aith the scrobble server.

5.21 News

The News module is an RSS reader.

5.22 Notes

The Notes module is intended to keep notes. These notes can be inserted as plain text, markdown or other markup languages.

The notes module offers:

- adding, removing and settings of tags
- listing notes, optionally filtered by
  - tags
  - grepping through note content and listing
    - the matches
    - files with matches
- opening a note via xdg−open (rendered as HTML if content is written in a markup language)

5.23 Password

The Password module.

5.24 Project

The Project module can be used to plan and organize projects, though does not offer ways to share these plans with others. If that is desired, a new store should be created, put under version controll (possibly git) and shared via this mechanism.

The project tool integrates the timetracking module as well as the todo module.

A project is represented by a single imag entry. Notes, subprojects, todos, timetrackings and other things are linked to the project. A project does not necessarily have to be a programming project, but could be a project for building a house or losing weight for example.

5.25 Rate

The Rate module is another plumbing command. It simply offers rating functionality, whereas allowed values are 0-10 (0 beeing equal to no rating).

5.26 Read

The Read module is a plumbing command for reading entries from the store and writing them to stdout for further processing.

5.27 Receipt

The Receipt module is for tracking, categorizing, tagging, quering and managing receipts.

5.28 Reference

The Reference module.

5.29 Shoppinglists

The Shoppinglists module provides functionality for creating shopping lists and organizing them. A shoppinglist can be printed and them be used to go shopping, of course.

5.30 Store

The Store module.

5.31 Summary

The Summary module is a wrapper to call a list (specified in the config file) of imag commands and viewing their outputs.

5.32 Tagging

The Tagging module.

A valid tag matches the regex $[a-zA-Z][0-9a-zA-Z]*$.

5.33 Timetrack

The Timetrack module implements a timewarrior-like timetracking functionality for imag.

Each timetracking is a 'tag' which can be started and stopped. These tags are *no* tags as in imag-tag, but timetracking-tags.

Summaries can be printed, also filtered by tags if desired.

5.34 Todo

The Todo module implements taskwarrior functionality by integrating taskwarrior itself into imag.

Each taskwarrior task s referenced from imag and represented as imag entry, thus making it linkable by other imag entries.

5.35 Url

The Url module is a plumbing module to put URLs into the imag store.

5.35.1 Implementation

The implementation of the URL module saves URLs on a per-entry basis. This means that each URL is hashed (with something like SHA512) and the hash is used as filename. The scheme is as follows:

```
/url/<hash of the domain>/<hash of the full URL>
```

This scheme results in grouping URLs of the same domain (for example https://imag-pim.org) but distinction of the actual full URL, while still deduplicating URLs. Entering the same URL twice results in the same entry.

This module does nothing more on its own. Its functionality may be used elsewhere (for example a bookmark module).

5.36 View

The View module.

5.37 Weather

The Weather module.

5.38 Wiki

The Wiki module provides a personal wiki implementation.

5.39 Workout

The Workout module is for tracking workouts. Burned calories, walked kilometers, lifting sets and all the things can be entered.

5.39.1 Submodules

Each type of workout can be handled with one subcommand of the module, whereas training seasions can be used to group types of workouts (for example swimming and running).

A step-counter functionality is implemented as first submodule. Import functionality for the step-counter submodule is available for importing from (for example) an Android step-counter App.

5.40 Write

The Write module is a plumbing command for reading a store feom stdin and writing it to the filesystem.

6 Libraries

This section of the documentation is only relevant for developers and you might skip it if you're only a user of the imag tool.

The following sections contain a short documentation on what the several libraries are supposed to do. It is generated from the README.md files of each library and only gives a general overview what can be done with the library. For a more comprehensive documentation of the library, one might consult the appropriate documentation generated from the source of the library itself.

The documentation of the libraries is sorted *alphabetically*.

6.1 libimagannotation

This library provides annotation functionality for entries.

Annotations are normal Store entries, but their header at annotation.is_annotation is set to true.

Annotations are linked to an entry (as in libimagentrylink).

6.1.1 Library functionality

The library features two traits: One to extend an Entry with annotation functionality and another one for extending the Store with functionality to get annotations of an entry and all annotations in the store.

6.2 libimagbibliography

6.3 libimagbookmark

This library crate implements functionality for bookmarks.

It uses libimagentrylink to create external links and therefor deduplicates equivalent external links (libimagentrylink deduplicates - you cannot store two different store entries for https://imag−pim.org in the store).

It supports bookmark collections and all basic functionality that one might need.

6.4 libimagborrow

6.5 libimagcalendar

6.6 libimagcontacts

6.7 libimagcounter

Library of "imag-counter", usable by other modules as well to implement counter functionality for entries.

6.8 libimagcuecards

6.9 libimagdiary

This library crates implements a full diary.

One can have one or more diaries in the store, each diary can have unlimited entries.

6.9.1 Future plans

The diary should be able to provide *daily*, *hourly* and even *minutely* diary entries, so one can use the diary as normal "Dear diary, today…"-diary, or more fine-grained and more like a journal.

The internal file format as well as the store-path generation for this module is prepared for such functionality.

6.10 libimagentryfilter

Helper library to filter lists of entries by certain predicated. Offers filters for filtering by header values and other predicates, plus this library offers logical operants to combine filters.

A commandline-to-filter DSL is planned for this, so commandline applications can use this to implement a uniform filter interface.

6.11 libimagentrylink

Linking library for linking entries with other entries. Used for "imag-link", the commandline utility, but intended for use in other binaries and libraries as well.

6.12 libimagentrylist

Library for listing entries in different manner.

This includes:

- Plain one-line-one-entry-path listing
- Tree listing by submodule
- Listing with metadata
- One-line-one-entry
- ASCII-Table

6.13 libimagentrymarkdown

Helper crate to add useful functionality in a wrapper around hoedown for imag.

Adds functionality to extract links, parse content into HTML and other things which might be useful for markdown rendering in imag.

6.14 libimagentrytag

Library for tagging entries. Used in "imag-tag" but should be used in all other modules which contain tagging functionality, so the backend and frontend look the same for all modules.

6.15 libimagerror

In imag, we do not panic.

Whatever we do, if we fail as hard as possible, the end-user should *never ever* see a backtrace from a panic!().

Anyways, the user *might* see a error trace generated by imag. That is because imag is software for power-users, for nerds (I use the term "nerd" because for me it is a good thing - I do not want to offend anyone by using it). This target group can read backtraces without getting confused. IO Error and Permission denied Error are things that nerds can understand and they already know what to do in the most obvious cases (such as Permission denied Error).

This library crate is for generating error types and handle them in a nice way. It can be seen as mini-framework inside imag which was written to work with error types in a specified way. All imag crates *must* use this library if they can return errors in any way, except the libimagutil - which is for the most basic utilities.

6.16 libimaghabit

The habit library implements a habit tracker.

A habit can be instantiated with a name and a time-period in which it should be fullfilled (eg. daily, ever 3 days, weekly...).

The module offers ways to generate statistics about habits.

6.17 libimagimages

6.18 libimaginteraction

A crate for more general interaction with the user (interactive commandline interface).

Offers functions for asking the user Y/N questions, for (numeric) values, etc.

6.19 libimagledger

6.20 libimagmails

The mail library implements everything that is needed for beeing used to implement a mail reader (MUA).

It therefor providea reading mailboxes, getting related content or mails, saving attachements to external locations, crafting new mails and responses,...

It also offers, natively, ways to search for mails (which are represented as imag entries) via tags, categories or even other metadata.

For more information on the domain of the imag−mail command, look at the documentation of the Section 5.18 module.

6.21 libimagmovies

6.22 libimagmusic

6.23 libimagnews

The library for the news module implements a RSS reader.

Details, for example where the feeds are stored (inside or outside the imag store) have to be evaluated.

6.24 libimagnotes

6.25 libimagpassword

6.26 libimagproject

6.27 libimagread

This library is for the plumbing command imag−read.

It extends the Runtime object and adds a write_store_to(writer) function (amongst others). After calling this function, the calling program cannot continue to do things, so this consumes the Runtime object and the calling program is expected to exit with the returned error code.

The calling program is expected to *not* print anything to stdout before or after calling this function.

This library is intended for use with the imag−read command only.

6.28 libimagreceipt

6.29 libimagref

This library crate contains functionality to generate *references* within the imag store.

It can be used to create references to other files on the filesystem (reachable via a filesystem path). It differs from libimagentrylink/external linking as it is designed exclusively for filesystem references, not for URLs.

A reference can have several properties, for example can a reference track the content of a filesystem path by hashing the content with a hashsum (SHA1) and one can check whether a file was changed by that. As files can get big (think of debian.iso) *partial hashing* is supported (think of "hash the first 2048 bytes of a file).

The library contains functionality to re-find a moved file automatically by checking the content hash which was stored before.

Permission changes can be tracked as well.

So this library helps to resemble something like a *symlink*.

6.29.1 Limits

Please understand that this is *not* intended to be a version control system or something like that. We also can not use *real symlinks* as we need imag-store-objects to be able to link stuff.

6.29.2 Usecase

This library offers functionality to refer to content outside of the store. It can be used to refer to *nearly static stuff* pretty easily - think of a Maildir - you add new mails by fetching

them, but you mostly do not remove mails and if you do you end up with a "null pointer" in the store, which can then be handled properly.

As this library supports custom hashes (you don't have to hash the full file, you can also parse the file and hash only *some* content) this is pretty flexible. For example if you want to implement a imag module which tracks a certain kind of files which constantly change... but the first 5 lines do never change after the file is created - you can write a custom hasher that only uses the first 5 lines for the hash.

6.29.3 Internals

Internally, in the store, the file gets created under /ref/<hash of the path to the file to refer to>. If the content of the file is hashed, we can still re-find the file via the content hash (which is stored in the header of the store entry).

The reference object can, after the path was re-found, be updated.

6.29.4 Long-term TODO

Things which have to be done here or are not yet properly tested:

- [ ] Testing of different Hashers
- [ ] Testing of re-finding of objects, including:
- [ ] Can a moved file automatically be found by content hash?
- [ ] Does a store-reference get updated automatically if it was moved, including links (as in libimaglink)?
- [ ] If the content of a file changes, does the content hash get updated automatically?

("automatically" is a strechable term here, as these things have to be triggered by the user anyways)

6.30 libimagrt

This library provides utility functionality for the modules and the binary frontends, such as reading and parsing the configuration file, a builder helper for the commandline interface and such.

It also contains the store object and creates it from configuration.

the libimagrt :: runtime::Runtime object is the first complex object that comes to live in a imag binary.

6.30.1 Long-term TODO

- [ ] Merge with libimagstore

6.31 libimagshoppinglists

6.32 libimagstore

The store is the heart of everything. Here lives the data, the complexity and the performance bottleneck.

The store offeres read/write access to all entries.

The store itself does not offer functionality, but has a commandline interface "imag-store" which can do basic things with the store.

6.32.1 Long-term TODO

- [ ] Merge with libimagrt

6.33 libimagtimetrack

A library for tracking time events in the imag store.

6.33.1 Store format

Events are stored with a store id like this:

```
/timetrack/<insert-date-year>/<insert-date-month>/<insert-date-day>/<insert
```

Timetrackings contain

- a comment (optional, free text)
- a start date
- an end date
- a tag

by default and might be extended with more header fields as one likes.

The header of a timetrack "work" entry looks like this:

```
[event]
tag = "work"
start = "2017-01-02T03:04:05"
end = "2017-01-02T06:07:08"
```

Normal tags (as in libimagentrytag) are explicitely *not* used for tagging, so the user has the possibility to use normal tags on these entries as well.

The tag field is of type string, as for one tag, one entry is created. This way, one can track overlapping tags, as in:

```
imag timetrack start foo
imag timetrack start bar
imag timetrack stop foo
```

```
imag timetrack start baz
imag timetrack stop bar
imag timetrack stop baz
```

The end field is, of course, only set if the event already ended.

6.33.2 Library functionality

The library uses the libimagentrydatetime::datepath::DatePathBuilder for building StoreId objects.

The library offers two central traits:

- TimeTrackStore, which extends a Store object with functionality to create FileLockEntry objects with a certain setting that is used for time-tracking, and
- TimeTracking, which extends Entry with helper functions to query the entry-metadata that is used for the time tracking functionality

The library does *not* provide functionality to implement imag−timetrack or so, as the core functionality is already given and the commandline application can implement the missing bits in few lines of code.

Aggregating functionality might be provided at a later point in time.

6.34 libimagtodo

The library for the todo module.

Whether this wraps taskwarrior or implements a todo tracking mechanism in imag itself is to be defined. Probably the latter.

6.35 libimagutil

Utility library. Does not depend on other imag crates.

6.36 libimagweather

6.37 libimagwiki

The wiki library implements a complete wiki for personal use.

This basically is a note-taking functionality combined with linking.

6.38 libimagworkout

6.39 libimagwrite

This library is for the plumbing command imag−write.

It extends the Runtime object and adds a read_store_from(reader) function (amongst others). After calling this function, the calling program cannot continue to do things, so this consumes the Runtime object and the calling program is expected to exit with the returned error code.

The calling program is expected to *not* print or read anything to/from stdout/stdin before or after calling this function.

This library is intended for use with the imag−write command only.

7 Todo

This section contains long-term todos. Some kind of roadmap, of one wants to put it that way.

7.1 Modules

Modules imag should offer which are not yet started or in progress, including a short note what each module should do.

First the modules which have been implemented in some way (not necessarily perfect or feature-complete):

- [x] imag-bookmark - A bookmark manager for web browsing.
- [x] imag-diary - A diary, or multiple.
- [x] imag-counter - Counting things.
- [x] imag-link - Linking imag entries to eachother
- [x] imag-notes - Note taking
- [x] imag-ref - Refering to files outside the imag store.
- [x] imag-tag - Tagging imag entries
- [x] imag-view - Viewing imag entries

Now the modules that are not yet started:

- [ ] imag-bibliography - For handling bibliographic references when writing scientific papers. Like Citavi, for example.
- [ ] imag-borrow - If you lend something to someone.
- [ ] imag-calendar - Calendar tooling based on icalendar files. No sync functionality.
- [ ] imag-category - For categorizing imag entries. Categories must exist before an entry can have a category.
- [ ] imag-contact - Contact tooling based on vcard files. No sync functionality.
- [ ] imag-cuecards - Cuecards for learning things, for example vocabulary.
- [ ] imag-filter - command to read the store from stdin, filter out entries based on a predicate specified via the CLI and write the store back to stdout.
- [ ] imag-git - wrapper to call git commands on the imag store no matter whether the current working directory is the store or not
- [ ] imag-gps - Adding GPS coordinates to entries

- [ ] imag-habit - Tracking ones habits (create habits and make sure you do what you've planned)
- [ ] imag-image - Image referencing, categorizing, etc.
- [ ] imag-item - Creating entries for Items in the store
- [ ] imag-ledger - Ledger functionality like beancountcand others
- [ ] imag-list - Managing lists
- [ ] imag-mail - Mail handling tool, a CLI-mutt like tool.
- [ ] imag-movie - Movie database handling, categorization and such things
- [ ] imag-music - Music database handling, categorization and such things. Possibly a scrobble server.
- [ ] imag-news - A RSS Reader
- [ ] imag-project - A project planner, integrated with imag-timetrack and imag-todo
- [ ] imag-rate - Attaching a rating to an entry
- [ ] imag-read - Command to load the store and pipe it to stdout (usefull for piping/ chaining commands)
- [ ] imag-receipt - Creating, categorizing, managing receipts
- [ ] imag-shoppinglists - Managing shopping lists
- [ ] imag-store - Low Level CLI Store interface
- [ ] imag-summary - Meta-command to call a set of imag commands (configurable which) and displaying their outputs
- [ ] imag-timetrack - Tracking time like with timewarrior
- [ ] imag-todo - Tracking tasks like with taskwarrior
- [ ] imag-url - Extracting URLs from enties, saving URLs to the imag store
- [ ] imag-weather - Weather tooling for getting forecast and recording a history of weather
- [ ] imag-weight - Weight tracker
- [ ] imag-wiki - A wiki for personal use
- [ ] imag-workout - Tools for tracking workouts. One sub-module will be step-counter tracking
- [ ] imag-write - Command to read the store from stdin and write it to the filesystem store (usefull for piping/chaining commands)

7.2 Libraries

- [ ] Ensure all libraries are implemented as extension traits rather than wrapping store types
- [ ] Rewrite logger to allow config/env-var based module white/blacklisting and writing log to file

7.3 User Interface

- [ ] Ensure store ids are always passed as positional arguments

7.4 Project structure and development

- [ ] Move away from github
- [ ] Have own issue tracking (possibly git-dit)
- [ ] Find a solution to having no travis-ci via github anymore
- [ ] Setup a viewer for the mailinglist archive
- [ ] Add maintainer scripts to repository
- [ ] To check patches for Signed-off-by lines
  - [ ] To automatically craft a reply to a contributor about a missing signed-off-by line
  - [ ] To automatically craft a reply to a contributor about a patchset that cannot be applied
- [ ] To check pull requests for GPG signatures
- [ ] To apply a patchset in a new branch

8 Contributing to imag

So you want to contribute to imag! Thank you, that's awesome!

All contributors agree to the developer certificate of origin by contributing to imag.

If you already have something in mind, go ahead with the prerequisites section. If you don't know what you could do, start here.

8.1 Without Github

If you do not want to use github for your contribution, this is completely okay. Feel free to contact us via our mailinglist via mail, feel also free to submit patches via mail (use git format−patch and git send−email, always add a cover letter to describe your submission).

Also ensure that each commit has a "Signed-off-by:" line. By adding that line, you agree to our developer certificate of origin. If you do not add the "Signed-off-by:" line, I cannot take your patch, sorry.

Once *I am* okay with your patchset, I will submit it as PR in the github repository, so more people can review it and CI can test it (the mailinglist is not yet used as much as github). I might come back to you if something broke in CI or someone has a suggestion how to improve your PR. I will keep you as author of the commits.

The following sections describe the way how to contribute with github.

8.2 Finding an issue

Finding an issue is simple: We have a special label in our issues section for easy-to-solve issues. You can start there, don't hesitate to ask questions if you do not understand the issue comment!

Also, if you've found bugs or outdated stuff in our documentation, feel free to file issues about them or even better: Write a pull request to fix them!

8.3 Prerequisites

- cargo and rust compiler in current version (stable)

Dependencies are listed in the default.nix file, though you do not have to have nix installed to build imag.

make can be helpful to automate the build process.

Note that this software is targeted towards commandline linux users and we do not aim to be portable to Windows or Mac OSX (though I wouldn't mind merging patches for OS X compatibility).

If you want to build the documentation (you don't have to) you'll need:

- pandoc
- pandoc-citeproc
- texlive
- lmodern (font package)
- make

All dependencies are installable with the nix package manager by using a nix−shell, if you have the nix package manager installed on your system.

8.4 Commit guidelines

Please don't refer to issues or PRs from inside a commit message, if possible. Make sure your PR does not contain "Fixup" commits when publishing it, but feel free to push "Fixup" commits in the review process. We will ask you to clean your history before merging! If you're submitting via patch-mail, I will do the fixup squashing myself.

Make sure to prefix your commits with "doc: " if you change the document. Do not change document and code in one commit, always separate them.

We do not follow some official Rust styleguide for our codebase, but we try to write minimal and readable code. 100 characters per line, as few lines as possible, avoid noise in the codebase, ... you get it.

Not all of your commits have to be buildable. But your PR has to be.

8.5 PR guidelines

We'd like to have one PR per module change. This means you *should* only change one imag module in one commit or PR (library plus belonging binary is okay). As this is not always possible, we do not enforce this, though we might ask you to split your commits/PR into two smaller ones.

Use feature branches. If you could name them "/", for example "libimagstore/add-debugging-calls", that would be awesome.

You are welcome to publish your PR as soon as there is one commit in your branch. This gives us the possibility to review whether your ideas go into a nice direction or whether there are issues with your approach and we can report them to you rather quickly. Rewriting a whole PR is not satisfactory and we'd like to make your contribution process enjoyable.

9 Merging tools which use the imag core functionality into this repo

If you're writing an application or module for imag, feel free to propose integrating it into the imag core distribution, if it fulfills the following requirements:

1. It is written in Rust
2. It has a commandline interface which is the main interface to the module OR it is a utility library for creating new kinds of functionality within the imag core.
3. It is licensed under the terms of GNU LGPLv2.1 OR all of your contributors approve a commit which changes the license of your codebase to GNU LGPLv2.1 (The word "approve" in this sentence is to be defined).

(If your tool does not fulfill these requirements, I won't merge it into the imag core distribution.)

9.1 Code of Conduct

We use the same code of conduct as the rust community does.

Basically: Be kind, encourage others to ask questions - you are encouraged to ask questions as well!

9.2 Contact

Feel free to reach out via mail/mailinglist or IRC.

9.3 Developer Certificate of Origin

```
Developer Certificate of Origin
Version 1.1
```

Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or

(b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or

(c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.

(d) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.