Foto tagger Documentation

Release 1.0

Matthias Baumgartner

March 21, 2016

CONTENTS

CHAPTER

ONE

WHAT'S THIS ABOUT?

In times of digital fotography, taking pictures is very cheap. So people take a lot of pictures. Amateurs and professionals alike. But now the problem is, how to handle large amounts of pictures? Be honest, you'll never ever look at most of them. But at some point, you'll have to go through all of them to sort out the good ones. The ones you want to keep. You need to arrange the pictures somehow, maybe copy some, maybe sort some for later access. That's what we have picture organizers for. Software that aims at organizing your picture gallery. Well, *tagit* is just another one of them. But why write another application when so many already exist? Let's have a look at some I've found and note what's missing in my oppinion. That doesn't mean the software isn't good, on the contrary. Most of them are actually excellent, but simply don't supply what I request.

Fotomanagers	
gThumb	No tagging and Cumbersome navigation through files
digiKam	Not tried
shotwell	I really like this one, because it works with IPTC tags and gives the images much space. It lacks tag-based search and the images could still be a a little bit bigger.
f-spot	They have a neat timeline and it's easy to use and seems well developed. Tagging is cumbersome and you'll need to use the mouse a lot. The app wants to modify the originals
Metadata editors	
geegie	no tag-based search
mapivi	oldschool, hard to use

1.1 Why tagit?

What do I actually expect from a picture organizer? Well, I want to have a tool which is built around the images. Many programs show way too much information around the pictures. So I don't recognize details in the preview anymore. But that's exactly what I want. To see some pictures next to each other, so I can compare them and select the better one. I also want a mechanism to quickly sort and search my images. Grouping similar pictures and showing or hiding them on demand. Or browsing the image collection by keyword.

That's what *tagit* is about. It focusses on these two things only:

1. Image display
2. Tagging

For picture organization, these are the main features. You don't need more than that. For the rest, there's enough tools already. If you want to edit your images, use *Gimp* or *ImageMagick* or one of the above mentioned tools. When viewing my own images, I want to do these two things: **Have a good look at the images and sort them in groups**.

With these two, I can go through my image library and **easily mark images** so that I can quickly browse to an image later on. I can **create and export collections** and run operations on them with third-party tools. I personally prefer the keyboard over the mouse, so tagit comes with full **keyboard shortcuts**. But you can use the mouse just as well.

The IPTC/EXIF support allows to be **compatible** with a huge range of other tools, on many platforms.

Furthermore, I strongly dislike when programs want to modify my originals. There's a good reason to do that, and tagit supports it as well, but **I want to be in control** what happens to the files and when this happens. The user can tune the program to his/her needs.

CHAPTER

TWO

FOR USERS

2.1 Features

2.1.1 Settings

The first thing you want to do is create some settings. A *.tagitrc* file is loaded from your current directory first, then home, then defaults. So you can have a file structure as below, loading the database from wherever you run *tagger*.:

```
~/.tagitrc
~/collections/first/.tagitrc
~/collections/first/first.db
~/collections/first/thumbs
~/collections/second/.tagitrc
~/collections/second/second.db
~/collections/second/thumbs
```

In a settings file, you mostly want to have the following config:

```
"session" : {
      "database"                    : "</pth/to/db>"
    , "window_size"                 : "1024x768"
    , "verbose"                     : false
    , "debug"                       : false
},

"model" : {
      "thumbnail" : {
          "path"                    : "</pth/to/thumbs>"
    }
}
```

Maybe also change the initial row/columns count:

```
"view" : {
      "rows"                        : 6
    , "columns"                     : 6
}
```

And you can specify what tags to generate when importing:

```
"extractor": {
      "constant"                    : ["Tag1", "Tag2", "Tag3"]
    , "path" : {
            "extension"             : false
    }
}
```

A full list of settings is printed by:

```
tagger info settings
```

2.1.2 Importing

Next thing to do is import some images. Check your settings first, especially the *extractor* part. There, you can specify tags to be attached automatically. You can either use the gui or the following command:

```
tagger index -r add /pth/to/images
```

-r makes it go through directories, *add* means that images will ignored if indexed before.

In the gui, hit the add button (+) and select the directory. The gui will freeze until indexing is finished. Which can be a while, so I prefer the command version.

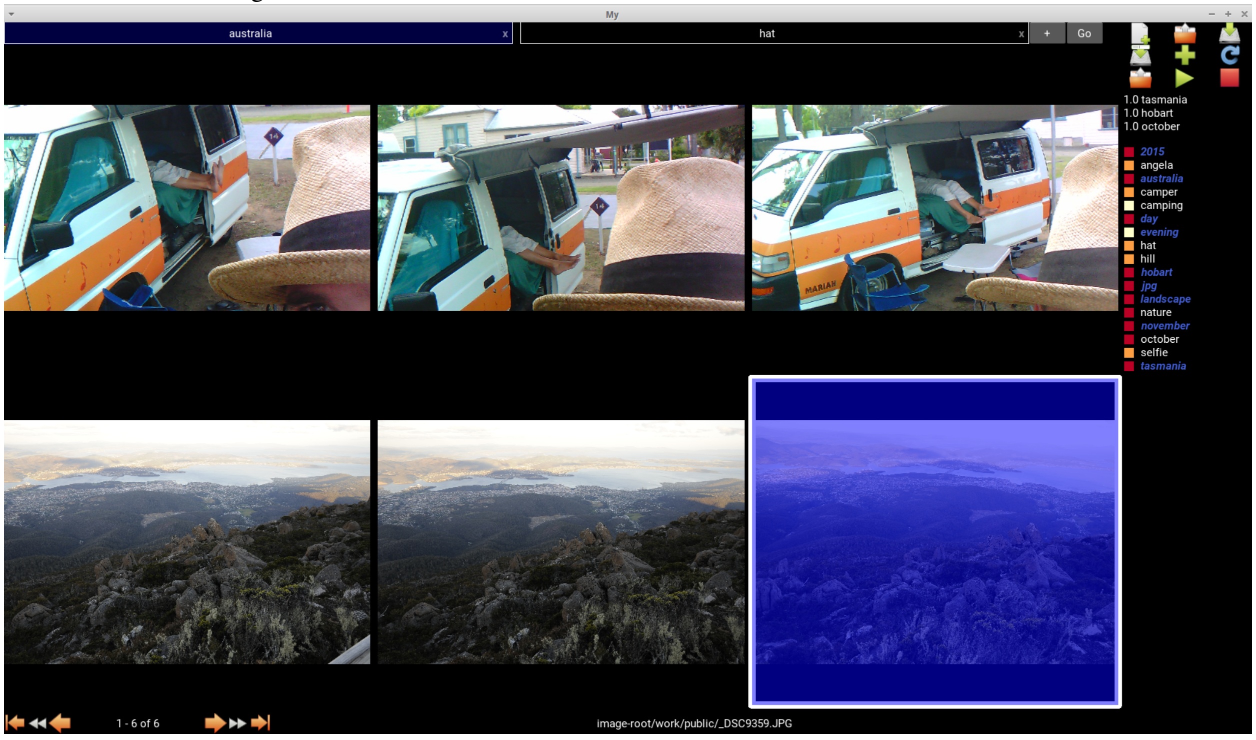
Either way, you end up with your database from the settings file and a couple of thumbnails. You're ready to proceed with the gui.

2.1.3 Viewing Images

Start the gui:

```
cd /pth/to/collection
tagger
```

Hit *F5* to see some images.

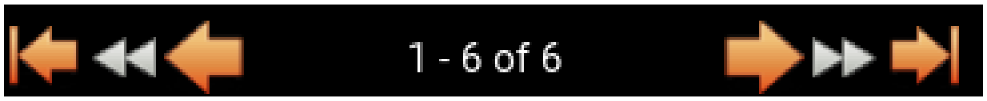


Most of the screen is filled with images. On top, there's the filters (like an address bar). To the side, some buttons and text is displayed.

Hit SHIFT + / (or '?' on an us keyboard) to see the **keybindings**. Go through them, it can really speed the process up. Most of the features are easily reachable via keybindings, and most of them are pretty common anyways. For example, in browsers *CTRL + k* brings you to the search and *CTRL + l* to the address bar. j and k for scrolling is known in *less*, *vim* or *nethack*. *F5* is common for reloads, *SHIFT* and *CTRL* for selection.

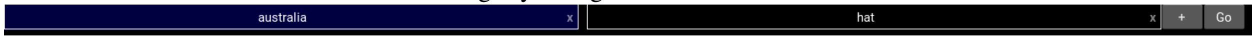
Navigation

There's the browser, showing you all the images. It works like a filebrowser, you can move the cursor and click to select. With the scroll buttons, the keyboard or the scrollwheel, you can move through the images. With CTRL pressed and the scrollwheel, you change the grid size of the browser. Very convenient!



Tags

In addition to that, you move through images with filters. Add a filter by pressing the + button on top or type *CTRL + k*. Enter a tag name and confirm. Only images including this tag will then be shown. Adding more tags stacks the filters and further reduces the number of images you'll get to see.



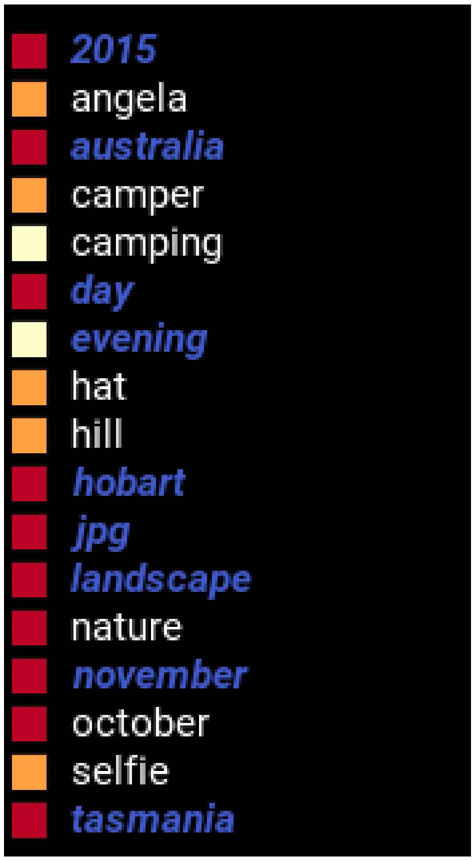
A convenient function is to hit *CTRL + ENTER* or *DEL* on some selected images, to either show only them or remove them.

In order to have this search working, images have to be tagged. For each image imported, you have to add some tags. By selecting several images you can edit tags for many images at once. Add or edit tags via *CTRL + t* or *CTRL + e*.

And you're rewarded with an even cooler feature. The search filters allow you to go one level up or down. Wanting to see the previous screen? Hit *ALT + LEFT*. Apply all filters? Use *ALT + RIGHT*.

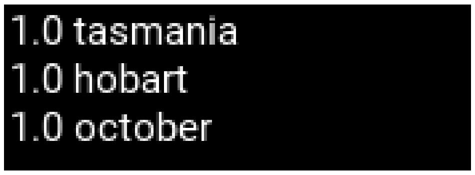
Sidebar

Let's take a bottom up approach.



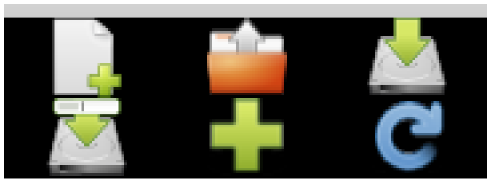
```
🟥 2015
🟧 angela
🟥 australia
🟧 camper
⬜ camping
🟥 day
⬜ evening
🟧 hat
🟧 hill
🟥 hobart
🟥 jpg
🟥 landscape
🟥 nature
🟥 november
🟥 october
🟧 selfie
🟥 tasmania
```

The concept of tags stays influential, also to the right side. You're presented a list of tags. Tags found in the list of currently selected or highlighted (i.e. the cursor) images are marked. To the left there's a color box, indicating how frequent the tag is. The more red, the more often the tag is used. The more white, the rarer it is.



```
1.0 tasmania
1.0 hobart
1.0 october
```

The top entries don't have the colorful box. These are tags recommended to search for (because some secret measure says so).

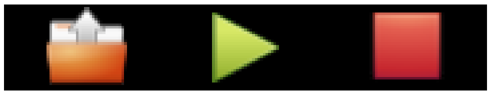


On the very right corner, there's some buttons, usually found in the *File* menu. In order (left-right, top-bottom), they are

- New: Create a new database.
- Load: Load a database file.
- Save: Save the database.
- Save as: Save the database under a different name.

- Index: Add some images. Works recursively on the directory selected.
- Revert: Undo all changes since the last save.

Programmes



Not to mixed up with the database buttons, these three control a programme.

The currently only programme is a breadth-first image tagger. Sounds confusing, but the concept is simple. It's a method to create tag hierarchies.

After starting the program, you select some images and add a tag. They are then removed from the current display. You continue selecting and adding tags. At some point, the display is empty. When this happens, you'll be presented with your previous selections again, one after one. You still continue selecting and adding a tag. Add an empty tag, if you wish to see these images never again (at least not until the program stopped).

Sounds a bit complicated, but the use case is the following. I have some images from a journey. I can easily subdivide these images into episodes, for example the countries. Then departement, then city, Then event.

By presenting fewer images, the oversight is better. And tagging many images at once speeds up the process.

Press the start button for the program, hit the pause button to intercept at any time. The stop button brings you back to the initial screen.

Other constraints

When adding a filter token (*CTRL + k*), you can also go for a image attribute. Like the width or aperture. You always give a range and you always have to maintain the format below. Otherwise, you'll get an error.:

```
!width: [0, 2200]
```

or:

```
!aperture: [2.3, 7.9]
```

More about tags

You can fiddle around with tags statistics on the terminal. Check out:

```
tagger tags -h
```

More features

Though the main features were discussed, *tagit* can do more.

Everything you did took part in the database only. You can write tags back to the image, as IPTC keywords, which is a convenient way to use your work with other programs.

You can also merge two databases, in case you decided to work with seperate collections and regret the decision.

We have some cleanup tools to search the database for dead entries and remove them.

2.2 Usage

2.2.1 Command syntax

Run the command with parameter **–help** to see the syntax. Giving no arguments starts the **GUI mode**.

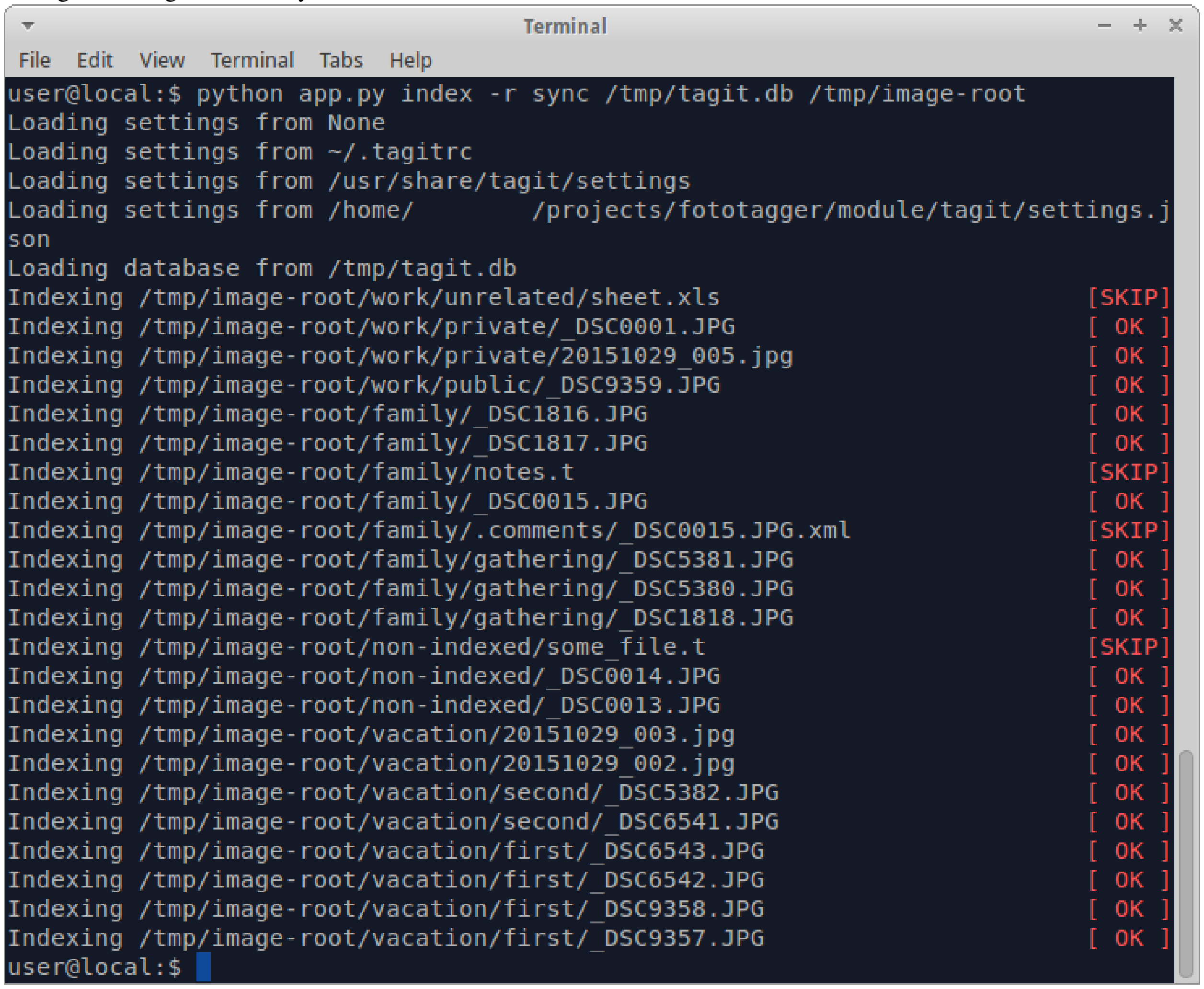
```
usage: tagit [-h] [--settings SETTINGS] [--verbose] [--version]
             {gui,clean,index,sync,merge} ...

Image tagger tool.

positional arguments:
  {gui,clean,index,sync,merge}
    gui                 Normal operation
    clean               Remove nonexistent images
    index               Write the database
    sync                Write the files
    merge               Create the union of two databases

optional arguments:
  -h, --help            show this help message and exit
  --settings SETTINGS, -s SETTINGS
                        Use settings file.
  --verbose, -v
  --version             show program's version number and exit
```

2.2.2 Keybindings

action	keys
Show keybindings	SHIFT + / (‘?’ on the us keyboard)
Select image	whitespace, CTRL + whitespace, SHIFT + whitespace
Deselect image	whitespace, CTRL + whitespace, SHIFT + whitespace
Select all images	CTRL + a
Deselect all images	ESC, CTRL + SHIFT + a
Move the cursor around	left, right, up and down cursor
Scroll one row down	j
Scroll one row up	k
Next page	page down
Previous page	page up
First image	Home
Last image	End
Add tag to selected images	CTRL + t
Edit tags of selected images	CTRL + e
Apply filter / reload	F5
Go back one token	ALT + left cursor, Backspace
Go forth one token	ALT + right cursor
Add filter token	CTRL + k
Filter address line	CTRL + l
Save	CTRL + s
Save as	CTRL + SHIFT + s
Show only selected images	CTRL + ENTER
Remove selected images from view	DEL

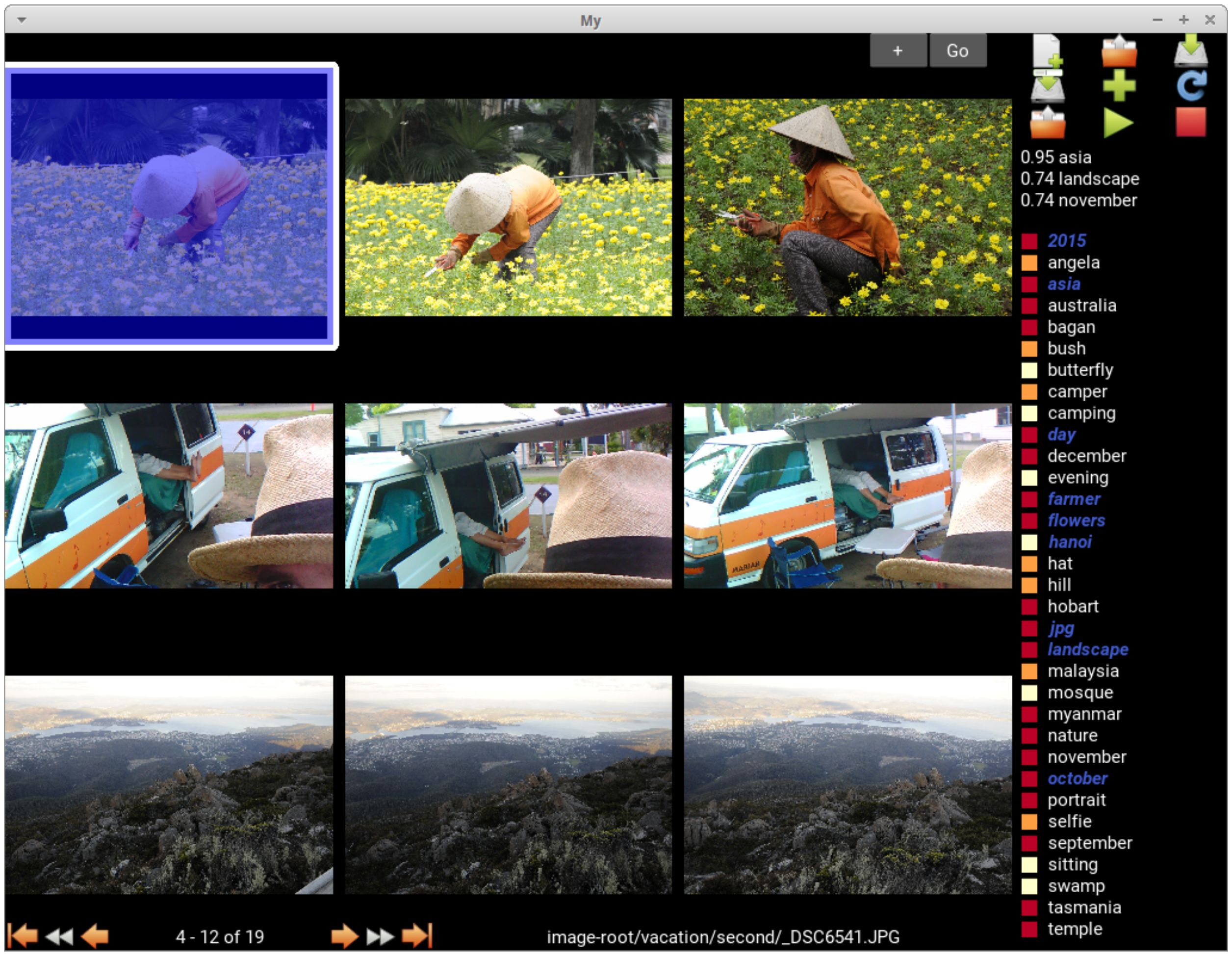
2.3 Screenshots

The images of a directory can be scanned and added to the *tagit* database. It's optional but saves time when scanning through the images. Currently, this action is to be run from the terminal.

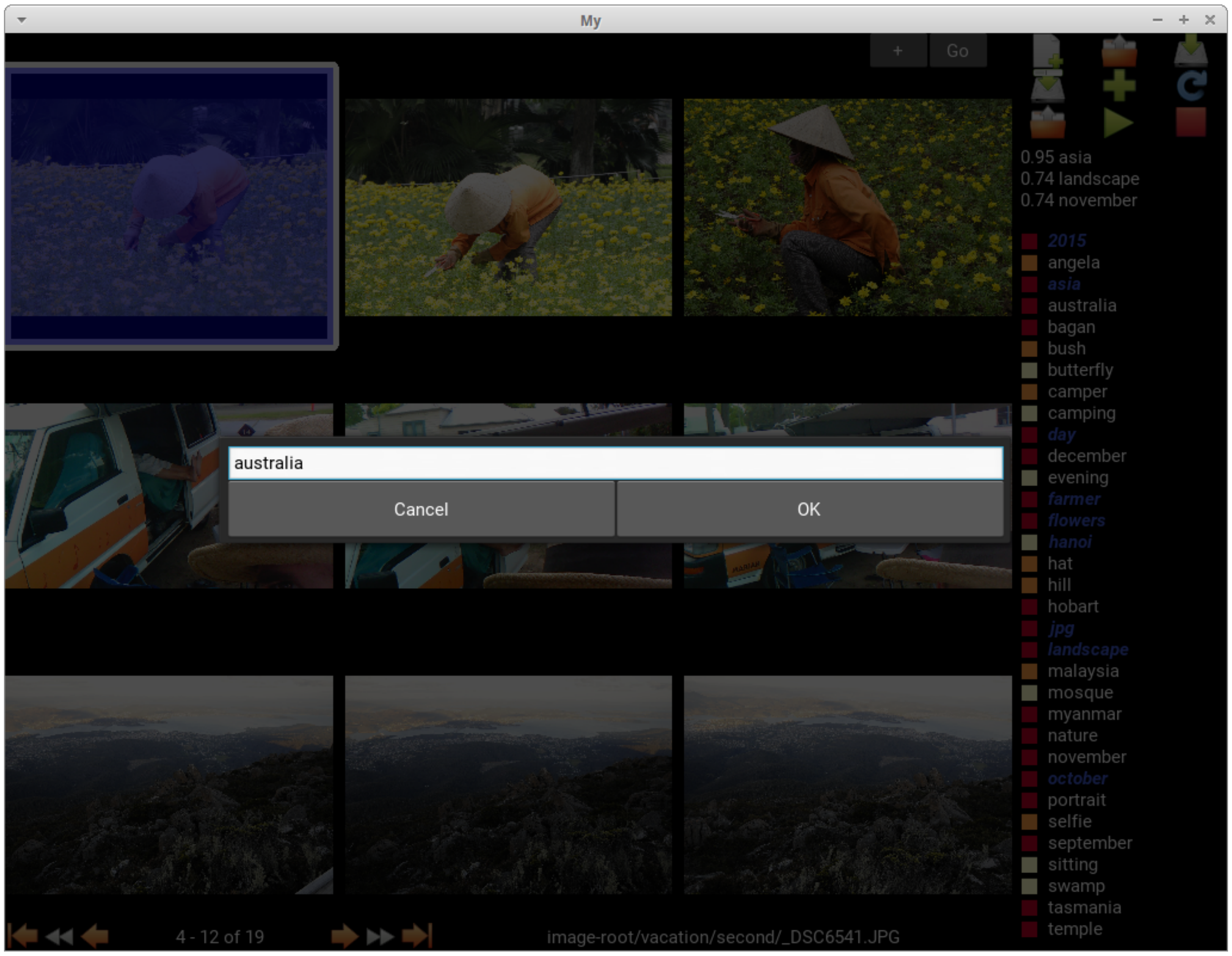


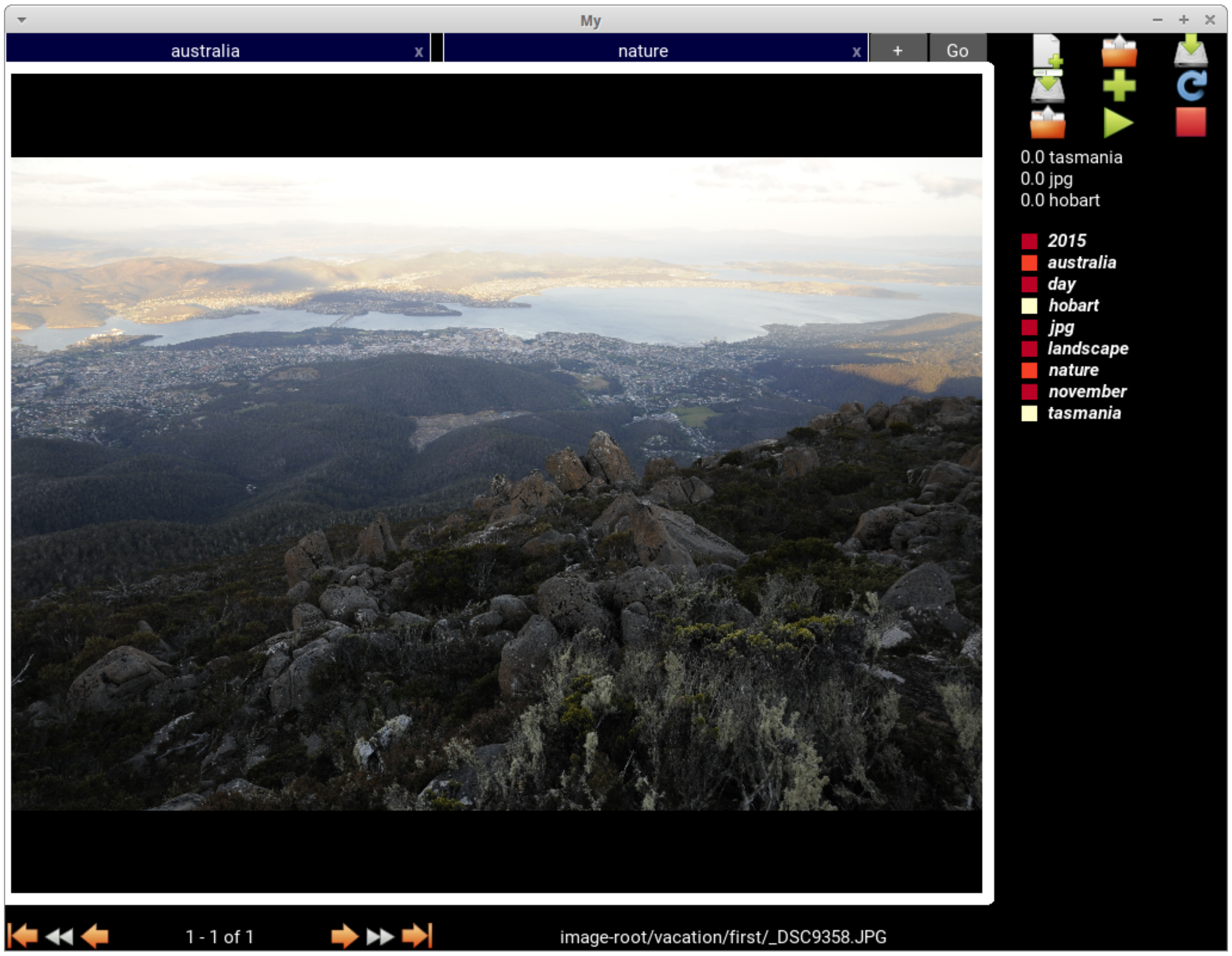
```
user@local:$ python app.py index -r sync /tmp/tagit.db /tmp/image-root
Loading settings from None
Loading settings from ~/.tagitrc
Loading settings from /usr/share/tagit/settings
Loading settings from /home/        /projects/fototagger/module/tagit/settings.j
son
Loading database from /tmp/tagit.db
Indexing /tmp/image-root/work/unrelated/sheet.xls                         [SKIP]
Indexing /tmp/image-root/work/private/_DSC0001.JPG                        [ OK ]
Indexing /tmp/image-root/work/private/20151029_005.jpg                    [ OK ]
Indexing /tmp/image-root/work/public/_DSC9359.JPG                         [ OK ]
Indexing /tmp/image-root/family/_DSC1816.JPG                              [ OK ]
Indexing /tmp/image-root/family/_DSC1817.JPG                              [ OK ]
Indexing /tmp/image-root/family/notes.t                                   [SKIP]
Indexing /tmp/image-root/family/_DSC0015.JPG                              [ OK ]
Indexing /tmp/image-root/family/.comments/_DSC0015.JPG.xml                [SKIP]
Indexing /tmp/image-root/family/gathering/_DSC5381.JPG                    [ OK ]
Indexing /tmp/image-root/family/gathering/_DSC5380.JPG                    [ OK ]
Indexing /tmp/image-root/family/gathering/_DSC1818.JPG                    [ OK ]
Indexing /tmp/image-root/non-indexed/some_file.t                          [SKIP]
Indexing /tmp/image-root/non-indexed/_DSC0014.JPG                         [ OK ]
Indexing /tmp/image-root/non-indexed/_DSC0013.JPG                         [ OK ]
Indexing /tmp/image-root/vacation/20151029_003.jpg                        [ OK ]
Indexing /tmp/image-root/vacation/20151029_002.jpg                        [ OK ]
Indexing /tmp/image-root/vacation/second/_DSC5382.JPG                     [ OK ]
Indexing /tmp/image-root/vacation/second/_DSC6541.JPG                     [ OK ]
Indexing /tmp/image-root/vacation/first/_DSC6543.JPG                      [ OK ]
Indexing /tmp/image-root/vacation/first/_DSC6542.JPG                      [ OK ]
Indexing /tmp/image-root/vacation/first/_DSC9358.JPG                      [ OK ]
Indexing /tmp/image-root/vacation/first/_DSC9357.JPG                      [ OK ]
user@local:$
```

The GUI shows a search bar (top), some extra information (right hand side) and the images. The image grid size (3x3 here) is configurable.

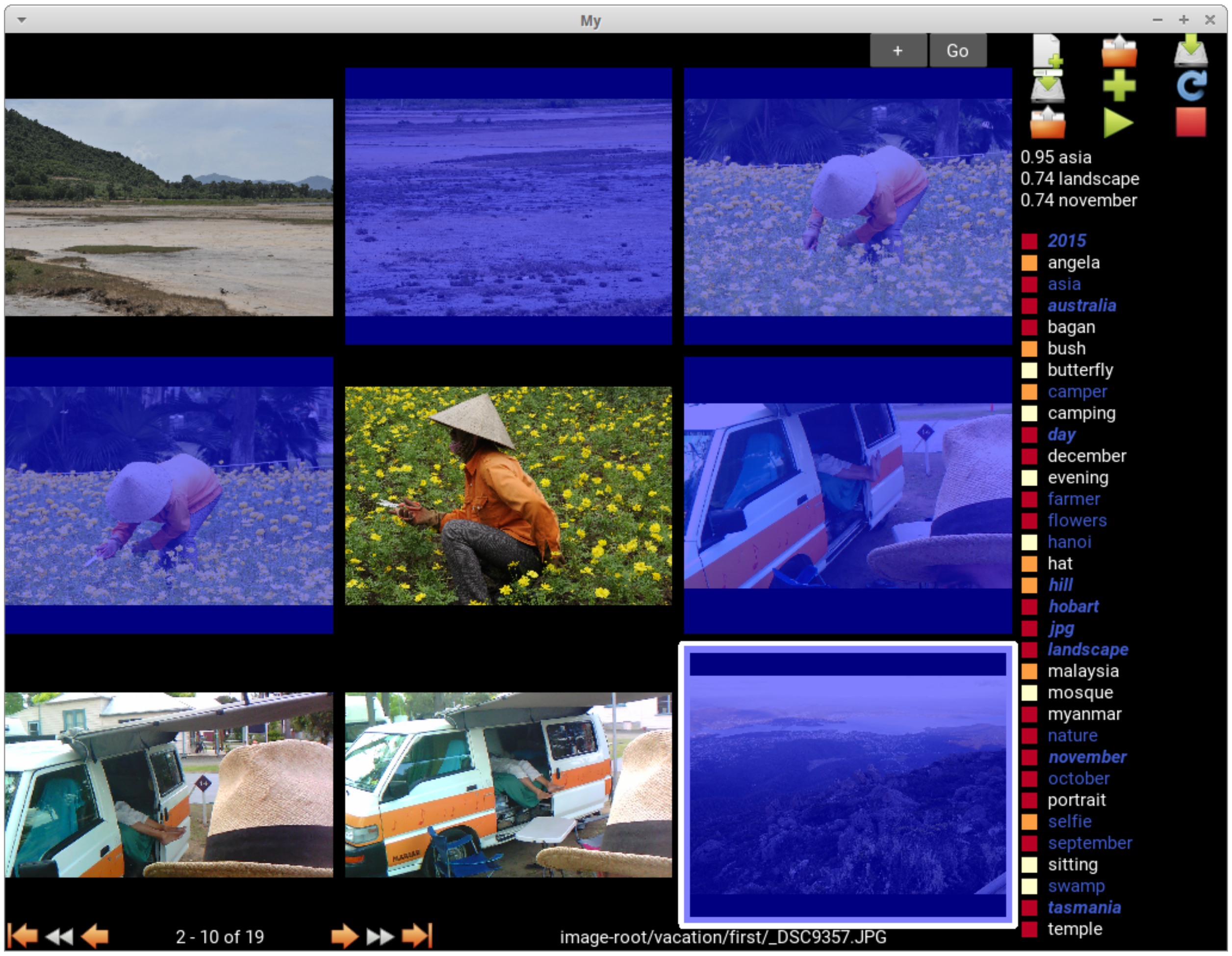


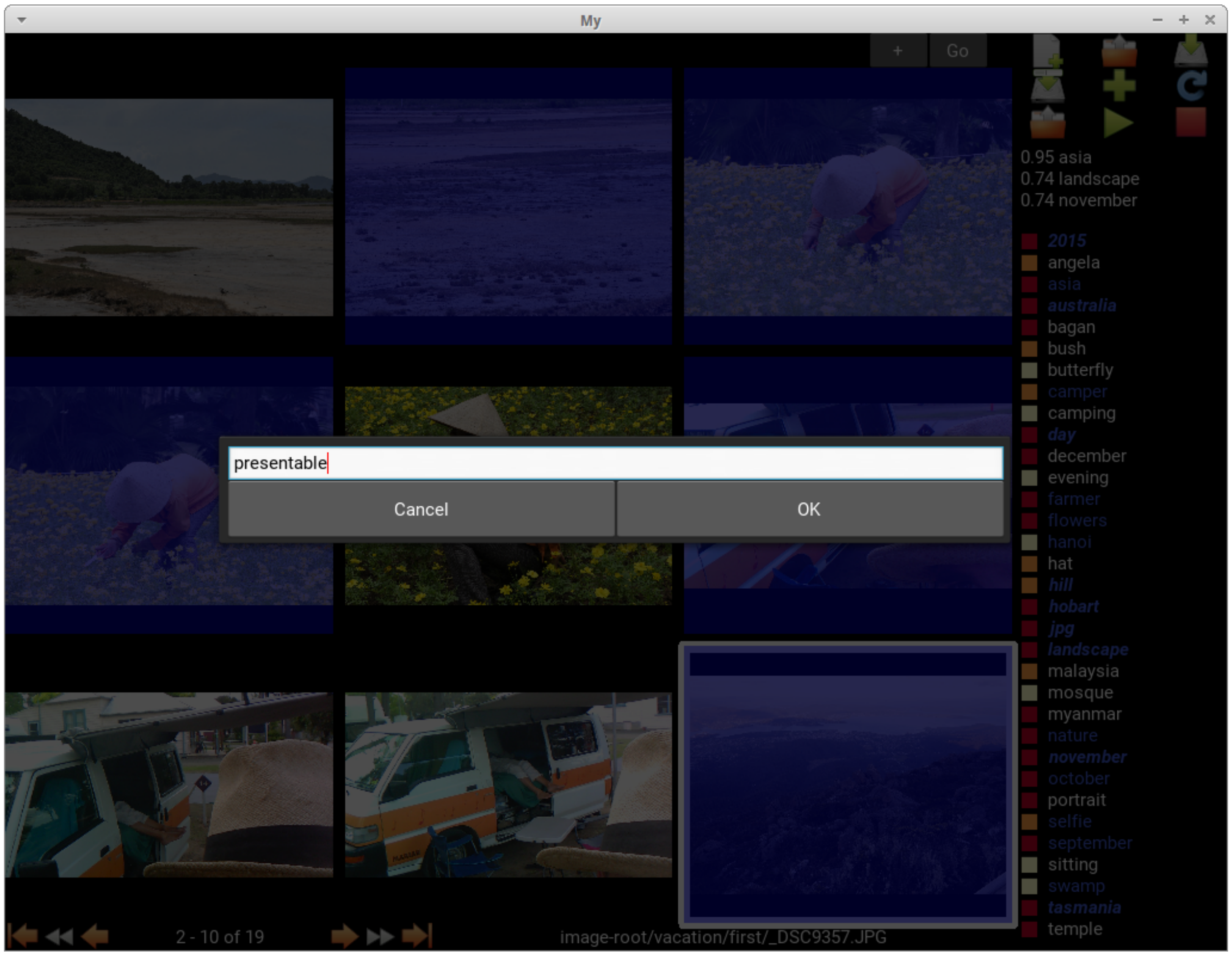
Search filters can be added to restrict the displayed images.


My
+
Go
0.95 asia
0.74 landscape
0.74 november
2015
angela
asia
australia
bagan
bush
butterfly
camper
camping
day
december
evening
farmer
flowers
hanoi
hat
hill
hobart
jpg
landscape
malaysia
mosque
myanmar
nature
november
october
portrait
selfie
september
sitting
swamp
tasmania
temple
australia
Cancel
OK
4 - 12 of 19
image-root/vacation/second/_DSC6541.JPG



Images can be selected and tags can be manipulated.


My
+
Go
0.95 asia
0.74 landscape
0.74 november
2015
angela
asia
australia
bagan
bush
butterfly
camper
camping
day
december
evening
farmer
flowers
hanoi
hat
hill
hobart
jpg
landscape
malaysia
mosque
myanmar
nature
november
october
portrait
selfie
september
sitting
swamp
tasmania
temple
2 - 10 of 19
image-root/vacation/first/_DSC9357.JPG



2.4 License

This project is released under the terms of the 3-clause BSD License.

```
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

2.5 Download

2.5.1 Installation

see *Installation*

2.5.2 License

This project is released under the terms of the 3-clause BSD License. See the section *License* for details. By downloading or using the application you agree to the supplied license's terms and conditions.

2.5.3 Downloads

Warning: This project is still in development and heavy experimental status. Be aware that there's no guarantees whatsoever and support is limited.

- `tagit-1.0` (latest)
- `This documentation (html)` (latest)
- `This documentation (pdf)` (latest)

2.6 Installation

2.6.1 Dependencies

Library	Version	Commit
python	2.7	
pyexiv2	0.3.2	
exiv2	0.25	
magic	0.4.10	d8a89620e7af25e78b21771a810924f2d81a9ed0
kivy	1.9.1	47c60f1cae5cc90b64a3a1fd351514e6036d7ebb
pysqlite	2.6.0	46d999e5302fb58d9636759ff36e0875c0c1eeb2
sqlite	3.8.2	27392118af4c38c5203a04b8013e1afdb1cebd0d

Each of the libraries (except python) come with a lot of their own dependencies... so installation may be a bit tricky...

Linux (Debian-like)

The following instructions are based and tested on the recent **Xubuntu 15.10**. If you use a different system, please do the analoguous and contribute!

Update the system:

```
sudo apt-get update
sudo apt-get upgrade
```

Install *setuptools* (for magic installation), *Python Imaging Library*, *pyexiv2* and *matplotlib* (pylab):

```
sudo apt-get install python-pil python-pyexiv2 python-matplotlib python-setuptools
```

json and *sqlite3* should come with the python installation already. If not, please do install them.

Let's continue with *magic*. Installation is to be done manually but quite straight forward.:

```
wget https://pypi.python.org/packages/source/p/python-magic/python-magic-0.4.10.tar.gz
tar -xzf python-magic-0.4.10.tar.gz
cd python-magic-0.4.10
sudo python setup.py install
```

For *kivy*, version 1.9.1 or above is required. If your system provides this, install *kivy* via standard distribution installation tools (ommit the first two commands). If only a lower version is delivered, a custom repository has to be added like shown below.:

```
sudo add-apt-repository ppa:kivy-team/kivy
sudo apt-get update
sudo apt-get install python-kivy
```

Now comes the *tagger*. Installation is analoguous to magic:

```
wget http://www.igsor.net/projects/tagit/_downloads/tagit-1.0.tar.gz
tar -xzf tagit-1.0.tar.gz
cd tagit-1.0/
sudo python setup.py install
```

So, when all software is installed, let's talk settings. Write the default settings into a file, then change them. You can delete entries if the default values are ok. Make sure to write valid JSON (delete commas, brackets, ...).:

```
tagger info settings > ~/.tagitrc
nano ~/.tagitrc
```

Note that *tagger* looks for a config file (.tagitrc) in the current working directory (*pwd*) first, then the home, then the installation defaults. Repeat the above procedure in a different directory (or copy ~/.tagitrc) and adjust the settings again.

When you're done configuring, run *tagger*.:

```
tagger index -r sync image-root/
tagger
```

CHAPTER

THREE

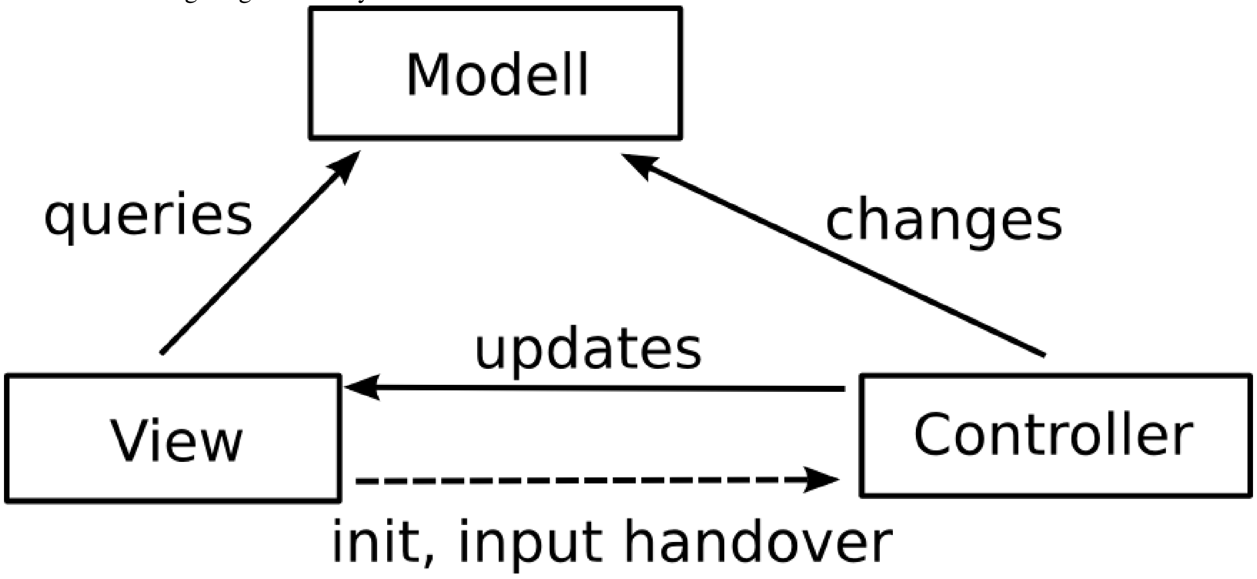
FOR DEVELOPERS

3.1 Concepts

The program is setup in a classic Model-View-Controller (MVC) design. The roles of the three parts are as follows:

- **Model** Data storage and application logic.
- **View** Presentation to the user, based on the model.
- **Controller** Process user input. Update model and view when demanded.

Here's a scheme giving essentially the same information:



3.1.1 Model

Handling pictures gives the following basic data:

- **Image**: The image files.
- **Thumbnails**: Basically an additional image, scaled down and related to the original image file.
- **Tags**: A list of tags for each image.
- **Metadata**: Metadata for internal use, such as timestamps, access statistics, counts, flags, references, etc.

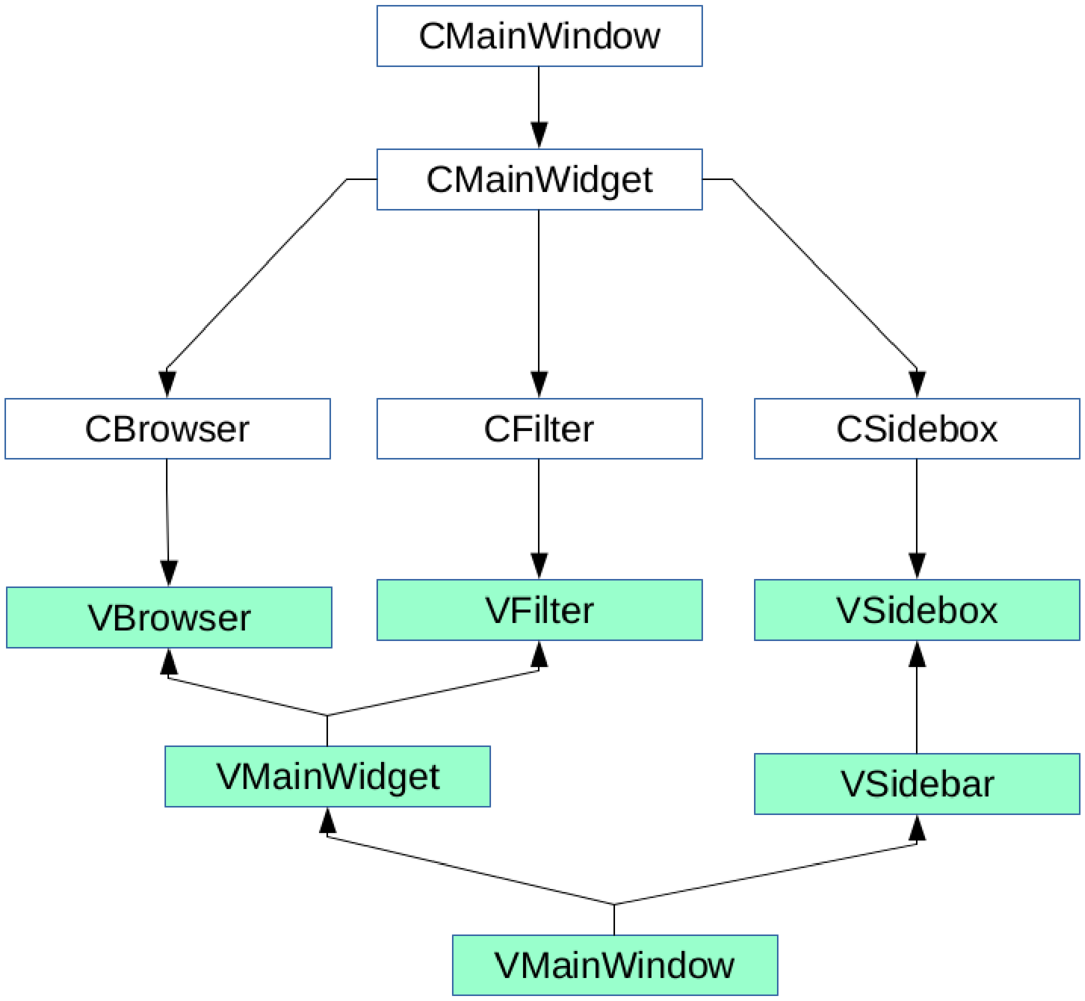
The *Model* defines how and where this data is stored. Due to the non-intrusiveness of *TagIt*, tags and metadata can be stored in an *SQLite* database. Thumbnails can be stored within the database or as extra files. If desired, tags and thumbnails can also be written into the *Image* file directly, via EXIF and IPTC. The specific behaviour of the *Model* is controlled via the configuration.

3.1.2 Controller

The controller processes user input and modifies the model on the user's behalf. If necessary, it can demand the view to update. Most of the application logic is therefore implemented in the *controller*. Another reason for upholding this concept strictly is that the *view* is typically strongly dependent on the UI framework chosen. The controller is not. Logic being static across frameworks is therefore moved to the controller.

The controller comes in a hierarchy, along but not identitcal to the typical widget hierarchy. Since the application is controlled by the UI framework, the view classes have to take care of the controller instantiation. How this is done exactly is defined in the *view*. However, you start with a *root Controller*. Using its *add_child* method, new controllers can be created within the hierarchy. Note that the hierarchy is only towards the top, via the *parent* member, not downwards.

If done right in the view, any widget can easily create a new controller which is attached to a parent. While the controller hierarchy normally follows the widget hierarchy, this is not at all necessary. See the image below to get a grasp of how it's done in *TagIt*.



Having a tree of controllers allows passing events. The event infrastructure is kept rather simple, yet effective. Use *bind* to bind a callback and then *dispatch* the event to run all callbacks. Similar to how *kivy* handles this. Consider the following example.

```
class MyListener(Controller):
    def __init__(self, widget, settings, parent=None):
        super(MyListener, self).__init__(widget, settings, parent)
        self.parent.bind(on_change=self.change_callback)

    def change_callback(self, *args):
        print args # 1, 2, 3, 4
        return True # Stops event processing

class MyDispatcher(Controller):
    def on_action(self):
        self.parent.dispatch('on_change', 1, 2, 3, 4)
```

These events allow distributed keybindings. Each controller can attach itself to the keyboard events of *CMainWindow*. Then it can define its own keybindings in its own space while not interfering with other components.

3.1.3 View

There's two parts here, the UI logic and the design. The design covers how structures are displayed in terms of layout, colors and graphics. The UI logic describes how elements interact with each other. Logic affecting data or application behaviour is implemented in the controller, though.

The *view* is heavily dependent on a UI framework. A framework change will likely result in re-writing the whole *view*. Currently, *kivy* was chosen as the only UI framework and *view* implementation.

The *kivy* sourced view implements its logic in python classes, the layout is described in the *kv language*. These two are somewhat seperate but fitted to each other and the boundary is flexible. This decoupling allows the design to be less independent from the UI logic and behaviour. For example, UI elements (let's say a button) can be referred to by name or throw generic events, so the specific layout is irrelevant for the actions of that element. This way makes it easier to modify the UI in minor ways.

Since *kivy* takes care if input - keyboard as well as clicks and touches - they have to be handed over to the *Controller*. This happens through the *VMainWindow* class, which controlls key events on the *kivy*-side. Key presses are passed to *CMainWindow* and distributed further with the controller events system. Mouse events are directly passed from the widgets to the respective controller.

Controller creation is achieved by extending the *Widget* class during runtime. The widget tree is traversed upwards until a controller is found. From this controller, a child of the specified type is generated and stored within the widget. With this method, a controller is always guaranteed to have a parent (except for the root of course).

3.2 API reference

Contents

3.2.1 Introduction

First, a few remarks to the code documentation. Classes start with an upper case letter and are written in CamelCase. Underscores show inheritance (not always, e.g. controllers). Functions are all lowercase, with underscores seperating words. Variables start with a lowercase letter (except for one-letter variables). They can contain underscores (like functions) or continie with camelCase.

Controller classes start with letter 'C' View classes start with letter 'V'

3.2.2 Core functionality

List operations

`tagit.`**`fst`**`(`*lst*`)`

`tagit.`**`snd`**`(`*lst*`)`

`tagit.`**`head`**`(`*lst*`)`

`tagit.`**`tail`**`(`*lst*`)`

`tagit.`**`unique`**`(`*lst*`)`

`tagit.`**`union`**`(`**args*`)`

`tagit.`**`intersection`**`(`**args*`)`

`tagit.`**`difference`**`(`*lstA*, *lstB*`)`

`tagit.`**`split`**`(`*callback*, *lst*`)`

`tagit.`**`truncate_dir`**`(`*path*, *cutoff=3*`)`
Remove path up to last *cutoff* directories

`tagit.`**`dict_update`**`(`*aggregate*, *candidate*`)`
Update a *dict* recursively.

class `tagit.thread_pool.`**`ThreadPool`**`(`*pool_size=1*`)`

`add_task``(`*task*, **args*, ***kwargs*`)`
Add a callable *task* to the queue. The callback is executed as soon as there's a free worker in the pool. Until then, the call to *add_task* is blocking. Additional arguments are passed to the *task*.

`get_active``()`
Return the number of active workers.

`get_pool_size``()`
Return the pool size.

Settings

class `tagit.`**`Settings`**
Less strict dictionary for missing settings.

Return None instead of raising KeyError if a key is not in the settings dictionary.

The settings dictionary looks like this:

```
{
    "section" : {
        "config" : value,
        ...
    },
    ...
}
```

This class makes the dictionary keys "section" and "config" optional.

`trace``(`**args*`)`
Traverse the *settings* dictionary and subdict's args are (section, [<more dicts>,] key, default value)

`tagit.`**`get_config`**`(`*settings*, **args*`)`
Traverse the *settings* dictionary and subdict's args are (section, [<more dicts>,] key, default value)

class `tagit.bindings.`**`Binding`**
Handle keybindings.

A keybinding is a set of three constraints: * Key code * Inclusive modifiers * Exclusive modifiers

Inclusive modifiers must be present, exclusive ones must not be present. Modifiers occuring in neither of the two lists are ignored.

Modifiers are always lowercase strings. Additionally to SHIFT, CTRL and ALT, the modifiers "all" and "rest" can be used. "all" is a shortcut for all of the modifiers known. "rest" means all modifiers not consumed by the other list yet. "rest" can therefore only occur in at most one of the lists.

Usage example:

```
>>> # From settings, with PGUP w/o modifiers as default
>>> Binding.check(evt, self.settings.trace("bindings", "browser", "page_prev", Binding.simple(Bi

>>> # ESC or CTRL + SHIFT + a
>>> Binding.check(evt, Binding.multi((Binding.ESC, ), (97, (Binding.CTRL, Binding.SHIFT), Bindin
```

static check (*((code, scankey), modifiers), constraint*)
Return True if *evt* matches the *constraint*.

static multi (**args*)
Return binding for multiple constraints.

static simple (*code, inclusive=None, exclusive=None*)
Create a binding constraint.

Console

class `tagit.`**Console** (*verbose*)
Status Line. Create lines like so:

```
fixed_size_part variable_size_part [STAT]
```

The variable_size_part is truncated such that the line fits CONSOLE_WIDTH characters.

If not *verbose*, only failure states are printed

fail ()
Write FAIL result.

ignored ()
Write IGNORED result.

ok ()
Write OK result.

title (*variable, fixed=''*)
Write the text part of the status line.

class `tagit.`**Colors**
Console colors.

Debugging

`tagit.debug.`**debug** (*local, abort=False*)
Enter a debug shell.

In your code, place the following statement

```
>>> debug(locals())
```

to enter the debug shell at that point. You'll have all local variables available, plus some extra modules (see below).

To get the current stack trace, call `tr.print_stack()`.

Extra modules: code, traceback (tr), os, sys, itertools, copy, time.

`tagit.debug.`**doDebug** (*loc*)
Start an interactive debug shell.

```
>>> from debug import doDebug
>>> doDebug(locals())
```

Whenever you do this, you'll be dropped into a shell for debugging. Exit the shell with ctrl+d You'll have your local variables and many debugging helpers readily imported.

3.2.3 Model

class `tagit.model.`**`DataModel`**(*settings*, *path=None*, *meta_adapter=None*)
Open a database from the file *path* or memory (if None, the default, or empty). If a path is given but does not exist, a new database will be created. Version constraints on existing databases may apply.

`get_image`(*image*, *resolution=None*)
Return an openable image from *image*.

`get_metadata`(*image*)
Return all data of *image*.

`has_changes`()
Return True if the database has unsaved changes.

`has_file_connection`()
Return True if the database is stored in a file. As opposed to in-memory database.

`index_dir`(*path*, *recursive=False*, *sync_method='set'*, *insert_only=False*, *update_only=False*)
Index or reindex files in *path*. The database will not be saved!

`index_file`(*path*, *sync_method='set'*, *insert_only=False*, *update_only=False*)
Index or reindex *path*. The database will not be saved!

`index_files`(*files*, *sync_method='set'*, *insert_only=False*, *update_only=False*, *common_root=None*)
Index or reindex may *files*. The database will not be saved.

`load`(*path=None*)
Open a database and re-initialize the instance.

`load_database`(*path=None*)
Load a database from *path*. If *path* is None, the database is created in the memory. If *path* does not exist, a new database is created.

`merge`(*other*, *sync_method='union'*)
Merge this database into *other*. The *other* database will be changed, not this instance. Changes are not commited. Indexed images are merged. If an image is indexed in both, the basic information is taken from the later one. Tags are always merged according to sync_method. SYNC_SET then means to use the information of this database.

`num_images`()
Return the total number of images.

`query`(*tokens*)
Return all images which satisfy all constraining *tokens*. Tokens is a list of tags (exact match).

`remove_dead`(*path*, *recursive=False*)
Remove inexistent files from the database.

`remove_dir`(*path*, *recursive=False*)
Remove images in *path* from the database. Traverses subdirectories if *recursive* is set. *path* has to be a directory.

remove_image (*image*)
Remove *image* from the database.

remove_images (*images*)
Remove *images* from the database.

rollback ()
Undo changes since the last save.

save (*path=None*)
Save changes to the database.

If *path* is set, creates a new database with the current content at that location. The new database is returned, the old one is neither changed (i.e. reverted) nor saved.

If *path* is not set, changes to the database are commited.

If 'write-through' is True the changes are saved to the file as well (-> update_files(sync_method=SYNC_SET))

sync_dir (*path*, *recursive*)
Synchronize all files in *path* with the database. Creates an UNION of file and database and writes to both. Files will be changed, database not saved!

sync_file (*path*)
Synchronize file *path* with the database. Creates an UNION of file and database and writes to both. Files will be changed, database not saved!

sync_files (*files*)
Synchronize all files in *files* with the database. Creates an UNION of file and database and writes to both. Files will be changed, database not saved!

update_dir (*path*, *recursive=False*, *sync_method='set'*)
Update all files in *path* with information from the database. If *path* is a directory, all files in that directory are updated. If *path* is a directory and *recursive* is true, all subdirectories will be changed as well. Files will be changed.

update_file (*path*, *sync_method='set'*)
Update all files in *path* with information from the database. Files will be changed.

update_files (*files*, *sync_method='set'*, *common_root=None*)
Update all files in *path* with information from the database. Files will be changed.

class tagit.model.db_path.**DBpath** (*settings*, *database*)
Path operations on database entries.

exists (*path*)
Return whether *path* exists in the database. *path* should be a file. For directories, the result is always False because directories are not mapped in the database.

getdir (*path*)
List contents of directory *path*. *path* must be a directory. Files and directories are listed, without '.' and '..'

is_dir (*path*)
Check if *path* is a directory.

is_file (*path*)
Check if *path* is a file.

is_prefix (*prefix*, *path*)
Return True if *path* is a subdirectory of *prefix* or a file within *prefix*.

is_temp(*path*)
Return True if *path* is within a temporary location.

join(**parts*)
Join parts of a path.

listdir(*path*)
List contents of directory *path*. *path* must be a directory. Files and directories are listed, without '.' and '..'

listfiles(*path*)
List files of directory *path*. *path* must be a directory.

walk(*top*)
Recursively walk through directory structure from a *top* directory.

Example:

```
>>> for root, files, dirs in walk(top):
>>>     print root, files, dirs
>>>     # root is the base directory
>>>     # files is a list of file names in root
>>>     # dirs is a list of directory names in root
```

Tags

class `tagit.model.tags.`**Tags**
The Tags baseclass and interface.

Tags are keywords which can be attached to images. Tags can be retrieved (get) and modified (add, set, remove). Other actions may be available.

add(*images*, *tags*)
Add all *tags* to each of *images*.

cleanup()
Search and remove tags without images.

get(*image*)
Return tags of *image*.

get_all()
Return all tags in the database.

query(*tags*)
Return all images which satisfy all constraining *tags*.

remove(*images*, *tags*)
Remove all of *tags* from all of *images*.

remove_all(*images*)
Remove all tags from each of *images*.

set(*images*, *tags*)
Set tags of *images* to be exactly *tags*.

class `tagit.model.tags.`**Tags_SQLite**(*conn*)
Bases: `object`

Tag operations based on a SQLite database.

The database link is to be passed as *conn*. Database changes are not saved. It's assumed that the basic scheme is present (i.e. image table).

add (*images*, *tags*)
Add all *tags* to each of *images*. *images* is a list of images (path). If an image is not in the database, it is ignored. *tags* is a list of strings. If a tag is not in the database, it is added.

cleanup ()
Search and remove tags without images. This operation may take some time.

query (*tags*)
Return all images which satisfy all constraining *tags*.

rename (*source*, *target*)
Rename a tag from *source* to *target*. *source* and *target* are the tag names. If *target* exists, the tags will be merged.

class `tagit.model.tags.`**Tags_Exif** (*adapter*)
Bases: `tagit.model.tags.Tags`

Tags-like interface to EXIF/IPTC metadata.

A *meta_adapter* is required to do the IPTC operations. Some operations are not meaningful and return dummy values.

MetaAdapter

class `tagit.model.meta_adapter.`**MetaAdapter**
Handle IPTC/EXIF reads and writes to files.

add (*path*, *tags*)
Add *tags* to IPTC of image *path*. *path* has to be a writeable file (not checked). Image data is written. *tags* is added to the existing tags.

get (*path*)
Read IPTC keywords from *path*. Return a list of tags. *path* has to be a readable file (not checked).

get_author (*path*)
Return the image author.

get_date (*path*)
Return the image date.

get_dimensions (*path*)
Return width, height and orientation of *path*.

get_metrics (*path*)
Return the image metrics.

get_thumbnail (*path*)
Return the thumbnail embedded in image *path*. *path* has to be a readable file.

set (*path*, *tags*)
Write *tags* to IPTC of image *path*. *path* has to be a writeable file (not checked). Image data is written. Existing tags are overwritten.

set_thumbnail (*path*, *image*)
Write thumbnail *image* into *path*. *path* has to be a writeable file (not checked). Image data is written. *image* is written as thumbnail.

class `tagit.model.meta_adapter.`**MetaAdapter_PyExiv2**
Bases: `tagit.model.meta_adapter.MetaAdapter`

Makes use of the pyexiv2 library to manipulate files.

Extractor

class `tagit.model.extractor.`**`Extractor`**

`tags`(*image*, *settings*, *meta_adapter*)
Return a list of automatically extracted tags from *image*.

class `tagit.model.extractor.`**`Extractor_Constant`**
Bases: `tagit.model.extractor.Extractor`

Constant information, user-provided.

class `tagit.model.extractor.`**`Extractor_Date`**
Bases: `tagit.model.extractor.Extractor`

Date information: * Year : Year (4 digit) * Day : Day of month * Month : Month (int 1-12 or name Jan-Dec) * Weekday : Weekday (int 1-7 (iso) or name Mon-Sun) * Hour : Hour (int) * Minute : Minute (int) * Day/Night : day or night

class `tagit.model.extractor.`**`Extractor_Photometrics`**
Bases: `tagit.model.extractor.Extractor`

Image and camera metrics: * Resolution : Image resolution (width, height) * Exposure : Exposure time (float, seconds, inverted) * Focal length : Reported focal length or 35mm equivalent (float, mm) * Aperture : Aperture ('F' + float) * ISO : ISO level (int)

class `tagit.model.extractor.`**`Extractor_Path`**
Bases: `tagit.model.extractor.Extractor`

Path information: * mime : Mime type (image/jpeg, ...) * extension : File extension (jpg, png, ...) * dir_after : Folder after specified path * dir_last : Image's parent directory * dir_after_call: Folder after search root

strip_date: Remove leading or trailing date information

class `tagit.model.extractor.`**`Extractor_Aggregator`**(*children=None*)
Bases: `tagit.model.extractor.Extractor`

Combine multiple *Extractor* instances into one.

`add_child`(*child*)
Add another *Extractor* instance.

class `tagit.model.extractor.`**`Extractor_Aggregator_all`**(*children=None*)
Bases: `tagit.model.extractor.Extractor_Aggregator`

Aggregate all known *Extractors*.

Attributes

class `tagit.model.attributes.`**`Attributes`**

class `tagit.model.attributes.`**`Attributes_SQLite`**(*meta_adapter*, *conn*)
Bases: `tagit.model.attributes.Attributes`

`query`(*attributes=None*, *ranges=None*)

`set`(*db_id*, *path*)
Get attributes from EXIF and write them into the database.

Thumbnailer

class `tagit.model.thumbnail.`**`Thumbnailer`**
Interface for thumbnail operations.

Thumbnailer defines an interface for any thumbnail handling method. It implements creation but remains abstract for *get* and *set*.

`create`(*image*, *resolution=(400, 400)*)
Create a thumbnail of *image* and return the Image object.

`get`(*image*, *resolution=(400, 400)*)
Search for a thumbnail of *image*. Return the path (file-based) or object (embedded) image. Return None if no thumbnail was found.

`has`(*image*)
Search for a thumbnail of *image*. Return success status as boolean.

`set`(*image*, *thumb*)
Write the thumbnail *thumb* to *image*.

class `tagit.model.thumbnail.`**`Thumbnailer_Chain`**(**args*)
Bases: `tagit.model.thumbnail.Thumbnailer`

Link several *Thumbnailers* together.

get: The first match is returned. set: Applied on all childs.

class `tagit.model.thumbnail.`**`Thumbnailer_FS`**(*root*)
Bases: `tagit.model.thumbnail.Thumbnailer`

Read and write thumbnails from/to the file system. Clones the original file structure within a *root* directory.

class `tagit.model.thumbnail.`**`Thumbnailer_FS_Flat`**(*conn*, *root*)
Bases: `tagit.model.thumbnail.Thumbnailer`

Read and write thumbnails from/to the file system. All thumbnails go to the same directory, with a generic file name. The database links the image and its thumb file.

class `tagit.model.thumbnail.`**`Thumbnailer_Exif`**(*meta_adapter*)
Bases: `tagit.model.thumbnail.Thumbnailer`

Read and write the exif thumbnail

class `tagit.model.thumbnail.`**`Thumbnailer_Database`**(*conn*)
Bases: `tagit.model.thumbnail.Thumbnailer`

Read and write thumbnails from/to the database.

`tagit.model.resize.`**`resize`**(*img*, *box*, *fit=False*)
Downsample the image. @param img: Image - an Image-object @param box: tuple(x, y) - the bounding box of the result image (i.e. resolution) @param fit: boolean - crop the image to fill the box (False = keep aspect ratio)

3.2.4 Token

class `tagit.token.`**`Token`**
Bases: `object`

class `tagit.token.`**`Token_Tag`**(*tag*)
Bases: `tagit.token.Token`

class `tagit.token.`**`Token_Image`**(*images*)
Bases: `tagit.token.Token`

class `tagit.token.`**`Token_Include`**(*images*)
Bases: `tagit.token.Token_Image`

class `tagit.token.`**`Token_Exclude`**(*images*)
Bases: `tagit.token.Token_Image`

class `tagit.token.`**`Token_Attribute`**(*attribute*, *range_*)
Bases: `tagit.token.Token`

class `tagit.token.`**`Token_Height`**(*range_*)
Bases: `tagit.token.Token_Attribute`

class `tagit.token.`**`Token_Width`**(*range_*)
Bases: `tagit.token.Token_Attribute`

`tagit.token.`**`token_factory`**(*type_*, **args*, ***kwargs*)

3.2.5 Program

class `tagit.program.program.`**`Program`**
Program State-Machine:

States: * uninitialized * initialized * running * paused * stopped

Transitions: * uninitialized -> initialized (init) * {initialized, stopped} -> uninitialized (unload) * {initialized, stopped} -> running (start) * {running, paused} -> stopped (stop) * running -> paused (pause) * paused -> running (resume)

`load`(*path*)
Load a program state from *path*.

`save`(*path*)
Save the current state to *path*.

class `tagit.program.bfs.`**`Pgm_BFS`**
Bases: `tagit.program.program.Program`

3.2.6 User interface

Controller

class `tagit.controller.controller.`**`Controller`**(*widget*, *settings*, *parent=None*)
Controller base class. Any controller receives a dict with the *settings*, a data *model* and a *parent*.

The *parent* being None implies that the object is the root controller.

The controller supports a simple event meachanism. You can bind callbacks on events on a controller object and later dispatch the event. Note that dispatching an event only affects callback bound to the same controller instance.

```
>>> def callback_fu(arg1, arg2):
    pass
>>>
>>> c = Controller()
>>> c.bind(event_name=callback_fu)
>>> c.dispatch('event_name', arg1, arg2)
>>> Controller().dispatch('event_name', arg1, args2) # Doesn't do anything
```

add_child(*cls*, *wx*, ***kwargs*)
Create and add a child controller.

bind(***kwargs*)
Bind a callback to an event.

```
>>> Controller().bind(event_name=callback_fu)
```

dispatch(*event*, **args*, ***kwargs*)
Dispatch an event.

Extra arguments after the *event* will be passed to the event handler>

get_root()
Return the top element in the controller tree. Return the root of the tree the instance is currently attached to. If the controller is created but not attached to a real controller tree, the result may be unusable.

unbind(***kwargs*)
Release an event binding. The arguments have to be identical to the call to *bind*.

```
>>> c = Controller()
>>> c.bind(event_name=callback_fu)
>>> c.unbind(event_name=callback_fu)
```

class `tagit.controller.controller.`**DataController**(*widget*, *model*, *settings*, *parent=None*)
Bases: `tagit.controller.controller.Controller`

A controller with access to a *model*.

View

`tagit.view.basics.`**get_root**()
Return the main window.

`tagit.view.basics.`**colorize**(*text*, *color*)

`tagit.view.basics.`**fontize**(*text*, *font*)

Kivy widgets will be extended by the following three functions. They allow easy creation of the *Controller* instances.

`tagit.view.wx_adapter.`**on_parent**(*obj*, *self*, *parent*)
Default behaviour if the parent of a widget is changed. Calls *on_parent* of all childs, even though the direct parent has not changed. This behaviour is usefull to pass down widget tree changes to lower widgets.

`tagit.view.wx_adapter.`**get_root**(*self*)
Traverse the widget tree upwards until the root is found. Note that this is not the same as basics.get_root (or app.root from a debug shell) unless the calling widget is attached to the main widget tree. Specifically, if a widget or any of its parents is created but not yet attached.

`tagit.view.wx_adapter.`**get_controller**(*self*, *cls=None*, *wx=None*, ***kwargs*)
Search and return the nearest controller. Create and return an instance of type *cls*, unless *cls* is None. Returns None if no controller was found.

Main

class `tagit.controller.main.`**CMainWindow**(*widget*, *settings*, *parent=None*)
Bases: `tagit.controller.controller.Controller`

MainWindow controller.

Receives the key events and passes them on.

class `tagit.view.main.`**`VMainWindow`**(*settings*, *model*, ***kwargs*)

Bases: `kivy.uix.boxlayout.BoxLayout`

Collection of all widgets makes this the root for the tagit UI.

If you want the tagit UI, create an app that loads this widget.

```
>>> settings = ...
>>> model = ...
>>> from tagit.view import VMainWindow
>>> class MyApp(App):
>>>     def build(self):
>>>         return VMainWindow(settings, model)
>>> app = MyApp()
>>> app.run()
```

Holds the *MainWidget* and *Sidebar*. Handles keyboard events.

`display_keybindings`(*bindings*)

Display an error dialogue.

class `tagit.controller.main.`**`CMainWidget`**(*widget*, *model*, *settings*, *parent=None*)

Bases: `tagit.controller.controller.DataController`

class `tagit.view.main.`**`VMainWidget`**(***kwargs*)

Bases: `kivy.uix.boxlayout.BoxLayout`

Main space for images and image search. Holds the *Browser* and *Filter*. Doesn't do more, actually.

Browser

class `tagit.controller.browser.`**`CBrowser`**(*widget*, *model*, *settings*, *parent=None*)

Bases: `tagit.controller.controller.DataController`

`add_tag`(*text*)

Add tags to images. *text* is a string of tags, split at *TAGS_SEPERATOR*.

`cursor_down`()

Down Cursor: One row down View: If cursor was at bottom row, jump by one row

`cursor_left`()

Left Cursor: One column left View: No change

`cursor_right`()

Right Cursor: One column right View: No change

`cursor_up`()

Down Cursor: One row down View: If cursor was at bottom row, jump by one row

`edit_tag`(*original*, *modified*)

Remove tags from images. Original and modified are strings, split at *TAGS_SEPERATOR*. Tags are added and removed with respect to the difference between those two variables.

`first_image`()

Home Cursor: Jump to first image View: Jump to first page

`last_image`()

End Cursor: Jump to last image View: Jump to last page

`next_page`()

Page Down Cursor: No change View: Jump by N*M images

on_key_down (*wx*, *(code*, *key)*, *modifiers*)
    Watch out for those modifiers.

on_key_up (*wx*, *(code*, *key)*)
    Watch out for those modifiers. Can receive some ups when modifier pressed at app start

on_keyboard (*wx*, *evt*)
    The *CBrowser* portion of key press handling.

on_results_changed (*filter_*, *images*)
    Called when the set of images to be displayed was changed.

previous_page ()
    Page Up Cursor: No change View: Jump by N*M images

redraw ()
    Update the widget.

scroll_down ()
    Scroll Down Cursor: No change View: Jump one row down

scroll_up ()
    Scroll Up Cursor: No change View: Jump one row up

select (*image*, *input_method='mouse'*)
    Set the selection. *image* is the selected (clicked) image.

zoom_in ()
    Decrease grid size.

zoom_out ()
    Increase grid size.

class `tagit.view.browser.`**VBrowser** (***kwargs*)
    Bases: `kivy.uix.boxlayout.BoxLayout`

    Image browser.

    Guaranteed childs (needed for Sideboxes): * cursor Current image (*VImage* instance) * selection Selected images * get_displayed Displayed images

displayed_images ()
    Return the images currently displayed.

go_first ()
    Click on the go first button.

go_last ()
    Click on the go last button.

next_page ()
    Click on the next page button.

on_parent (**args*)
    Set the controller factory. Reinitializes parts of the browser.

on_touch_down (*touch*)
    Scroll in browser.

previous_page ()
    Click on the previous page button.

redraw (*images*, *offset*, *n_results*, *page_size*)
    Update the displayed images.

redraw_cursor(*cursor*)
Redraw cursor decoration.

redraw_selection(*selected_images*)
Redraw selection decoration.

scroll_down()
Click on the scroll down button.

scroll_up()
Click on the scroll up button.

set_grid_size(*num_cols*, *num_rows*)
Set the grid size (cols and rows) according to proposed values.

update_status(*status*)
Update the status line.

Filter

class `tagit.controller.filter.`**CFilter**(*widget*, *model*, *settings*, *parent=None*)
Bases: `tagit.controller.controller.DataController`

add_token(*token*)
Add a token *text*.

add_token_tag(*text*)
Add a token for tag *text*.

add_variable_token(*text*)
Add a token and allow its type to be specified.

apply()
Run the filter.

edit_token(*token*, *data*)
Change *token* to *text*.

go_back(*nSteps=1*)
Remove the last *nSteps* tokens.

go_forth(*nSteps=1*)
Re-add the next *nSteps* tokens.

load(*path*)
Load the filter from *path*.

on_keyboard(*wx*, *evt*)
Handle filter keyboard events.

redraw()
Update the widget.

remove_token(*token*)
Remove a *token* from the token list.

request_add_token()
Add a token.

request_apply()
Apply the filter.

request_edit_token(*token*, *newtext=''*)
Change *token*. If *newtext* is empty, a dialogue will be requested. Otherwise, *token* is changed to *newtext*.

request_remove_token(*token*)
Remove token *token*.

save(*path*)
Save the current filter to *path*.

class `tagit.view.filter.`**VFilter**(***kwargs*)
Bases: `kivy.uix.boxlayout.BoxLayout`

A row of filter tokens (i.e. some filter text) and its management.

A filter is used to search in the image database. The filter consists of several tokens. Each token adds a search constraint.

Tokens can be added, changed and removed. The filter can be applied to an image source. This has effects on other parts of the UI, namely the *Browser*.

error(*text*)
Display an error dialogue.

on_parent(**args*)
Set the controller.

redraw(*tokens*, *cutoff*)
Redraw the tokens.

request_add_token()
Ask the controller to add a token.

request_apply()
Ask the controller to apply the filter.

token_dialogue(*text*, *clbk*)
Show a dialogue for modifying tokens.

Sideboxes

class `tagit.view.sidebar.`**VSidebar**(***kwargs*)
Bases: `kivy.uix.gridlayout.GridLayout`

Load configured sideboxes, present them stacked.

on_widget_change(*instance*, *widget*)
Re-initialize the Sideboxes to a new mainWidget.

class `tagit.controller.sidebox.db_management.`**CSidebox_DB_Management**(*widget*, *model*, *settings*, *parent=None*)
Bases: `tagit.controller.controller.DataController`

A set of functions to manage the model.

on_keyboard(*wx*, *evt*)
Handle keypresses.

request_index()
On index button click.

request_load()
On load button click.

request_new()
On new button click.

request_reload()
On reload button click.

request_save()
On save button click.

request_save_as()
On save as button click.

class `tagit.view.sidebox.db_management.`**`VSidebox_DB_Management`**(***kwargs*)
Bases: `kivy.uix.gridlayout.GridLayout`, `tagit.view.sidebox.sidebox.VSidebox`

A bunch of buttons to control the database.

class `tagit.controller.sidebox.tags.`**`CSidebox_Tags`**(*widget*, *model*, *settings*, *parent=None*)
Bases: `tagit.controller.controller.DataController`

Base class for tag-oriented sideboxes.

class `tagit.controller.sidebox.tags_suggested.`**`CSidebox_Tags_Suggested`**(*widget*, *model*, *settings*, *parent=None*)
Bases: `tagit.controller.sidebox.tags.CSidebox_Tags`

Give a recommendation which tags to include next in the filter.

update(*filter_*, *images*)
Update the sidebox.

class `tagit.view.sidebox.tags_suggested.`**`VSidebox_Tags_Suggested`**(**args*, ***kwargs*)
Bases: `kivy.uix.label.Label`, `tagit.view.sidebox.sidebox.VSidebox`

Show tags to include next in the filter.

class `tagit.controller.sidebox.tags_browser.`**`CSidebox_Tags_Browser`**(*widget*, *model*, *settings*, *parent=None*)
Bases: `tagit.controller.sidebox.tags.CSidebox_Tags`

Show tags if images displayed. Highlight tags of selected images and the cursor.

update()
Update the sidebox widget.

class `tagit.view.sidebox.tags_browser.`**`VSidebox_Tags_Browser`**(**args*, ***kwargs*)
Bases: `kivy.uix.label.Label`, `tagit.view.sidebox.sidebox.VSidebox`

Show tags if images displayed. Highlight tags of selected images and the cursor.

class `tagit.controller.sidebox.tags_selection.`**`CSidebox_Tags_Selection`**(*widget*, *model*, *settings*, *parent=None*)
Bases: `tagit.controller.controller.DataController`

List tags of selected images.

update(*browser*, *selection*)
Update the sidebox widget.

class `tagit.view.sidebox.tags_selection.`**VSidebox_Tags_Selection**(***kwargs*)
Bases: `kivy.uix.label.Label`, `tagit.view.sidebox.sidebox.VSidebox`

List tags of selected images.

class `tagit.controller.sidebox.tags_cursor.`**CSidebox_Tags_Cursor**(*widget*, *model*, *settings*, *parent=None*)
Bases: `tagit.controller.controller.DataController`

List tags of the image under the cursor.

update(*browser*, *cursor*)
Update the sidebox widget.

class `tagit.view.sidebox.tags_cursor.`**VSidebox_Tags_Cursor**(***kwargs*)
Bases: `kivy.uix.label.Label`, `tagit.view.sidebox.sidebox.VSidebox`

List tags of the image under the cursor.

class `tagit.controller.sidebox.pgm.`**CSidebox_Program_Control**(*widget*, *model*, *settings*, *parent=None*, *main_widget=None*)
Bases: `tagit.controller.controller.DataController`

A set of functions to manage the model.

class `tagit.view.sidebox.pgm.`**VSidebox_Program_Control**(***kwargs*)
Bases: `kivy.uix.gridlayout.GridLayout`, `tagit.view.sidebox.sidebox.VSidebox`

A bunch of buttons to control the database.

Dialogues

class `tagit.view.dialogues.dialogue.`**Dialogue**(***kwargs*)
Bases: `kivy.uix.popup.Popup`

Popup dialogue with OK and Cancel buttons.

Use like below:

```
>>> dlg = Dialogue()
>>> dlg.bind(on_ok=....)
>>> dlg.bind(on_cancel=...)
>>> dlg.open()
```

cancel()
User pressed the Cancel button.

ok()
User pressed the OK button.

on_cancel()
Event prototype.

on_ok()
Event prototype.

class `tagit.view.dialogues.`**`TextInputDialogue`**`(`***kwargs*`)`
Bases: `tagit.view.dialogues.dialogue.Dialogue`

Dialogue with a single line text input field.

keyboard_ctrl is an object supporting *request_exclusive_keyboard* and *release_exclusive_keyboard* for exclusive keyboard access. Usually the *root_widget*.

Pass the default text as **text**.

```
>>> TextInputDialogue(text='Hello world', keyboard_ctrl=root_widget).open()
```

In case of touch events, they need to be inhibited to change the focus.

```
>>> FocusBehavior.ignored_touch.append(touch)
```

class `tagit.view.dialogues.`**`LabelDialogue`**`(`***kwargs*`)`
Bases: `tagit.view.dialogues.dialogue.Dialogue`

Dialogue with some text.

Set the default text as **text**.

```
>>> LabelDialogue(text='Hello world').open()
```

class `tagit.view.dialogues.`**`FilePickerDialogue`**`(`***kwargs*`)`
Bases: `tagit.view.dialogues.dialogue.Dialogue`

Dialogue with a file browser to select a file.

Pass the default text as **text**.

```
>>> FilePickerDialogue(text='Hello world').open()
```

class `tagit.view.dialogues.`**`DirPickerDialogue`**`(`***kwargs*`)`
Bases: `tagit.view.dialogues.dialogue.Dialogue`

Dialogue with a file browser to select a directory.

Pass the default text as **text**.

```
>>> DirPickerDialogue(text='Hello world').open()
```

class `tagit.view.dialogues.`**`FileCreatorDialogue`**`(`***kwargs*`)`
Bases: `tagit.view.dialogues.dialogue.Dialogue`

Dialogue with a file browser to select a file.

Pass the default text as **text**.

```
>>> FileCreatorDialogue(text='Hello world').open()
```

class `tagit.view.dialogues.`**`ErrorDialogue`**`(`***kwargs*`)`
Bases: `tagit.view.dialogues.dialogue.Dialogue`

Dialogue with some text.

Set the default text as **text**.

```
>>> try:
>>>     ...
>>> except Exception, e:
>>>     ErrorDialogue(text=e.message).open()
```

class `tagit.view.dialogues.`**`BindingsDialogue`**(*bindings*, **args*, ***kwargs*)

Bases: `tagit.view.dialogues.dialogue.Dialogue`

Dialogue with some text.

Set the default text as **text**.

Externals

class `tagit.external.spellcheck.`**`Spellcheck`**(*lang='en'*)

class `tagit.external.memoize.`**`memoized`**(*func*)

Decorator. Caches a function's return value each time it is called. If called later with the same arguments, the cached value is returned (not reevaluated).

Tools

`tagit.metrics.`**`tags_similarity`**(*tags*)

- Similar case
- Similar word (typo: common errors, change of two letters)
- Synonym
- Prefix (one word is subset of the other)
- Similar results
- One is a subset of the other (w.r.t images)

`tagit.metrics.`**`stat_per_tag`**(*model*)

`tagit.metrics.`**`stat_per_image`**(*model*)

`tagit.tools.`**`tags_histogram`**(*model*)

`tagit.tools.`**`tags_stats`**(*model*)

Tag statistics:

- Tag Histogram
- correlation btw. tags (matrix, top-N list)

`tagit.tools.`**`tags_collect`**(*model*)

Goes through all tags and queries the user if it should be merged with a similar one.

The user can choose the following actions for each pair of tags: * l Merge to the left; Keeps the left tag, removes the right one * r Merge to the right; Keeps the right tag, removes the left one * n Don't merge, keep both tags * s Skip this pair, ask later * q Quit

`tagit.tools.`**`images_export`**(*images*, *target*, *method=None*, *keep_structure=False*, *simulate=False*, *verbose=False*)

Export *images* to a *target* directory.

Method can be either of: * symlink Create symlinks in the *target* directory * hardlink Create hardlinks in the *target* directory * copy Copy the images to the *target* directory

If *keep_structure* is True, the original directory structure is recreated starting after the shared prefix of *images*. Otherwise, the images will be exported to the *target* folder directly. If so, naming conflicts are handled by adding a sequence number to the image name. E.g. /foo/image.jpg, /bar/image.jpg -> export/image.jpg, export/image-1.jpg

If *simulate* is True, path translations are returned (src, trg) but no action is actually taken.

3.2.7 Index

- *genindex*
- *modindex*

3.3 Hic sunt dracones

CHAPTER

FOUR

SEARCH PAGE

search

PYTHON MODULE INDEX