filuxe

Jan 12, 2022

Contents:

1 A quick spin **1**
- 1.1 LAN Webserver . . . 2
- 1.2 WAN Server . . . 2
- 1.3 LAN Forwarder . . . 2
- 1.4 Buildserver . . . 3

2 Filuxe script **5**

3 Forwarder **7**
- 3.1 Config . . . 8
- 3.2 Rules . . . 8

4 Web server **11**
- 4.1 Write access . . . 11
- 4.2 Authentication . . . 11
- 4.3 Tips and tricks . . . 12

5 Web server direct HTTP **13**
- 5.1 Upload file . . . 13
- 5.2 Get list of files . . . 13
- 5.3 Download file . . . 14

6 Keys and certificates **15**

7 Filebrowser **17**

8 Installing **19**
- 8.1 Dependencies . . . 19
- 8.2 Reading . . . 19
- 8.3 Credits . . . 19

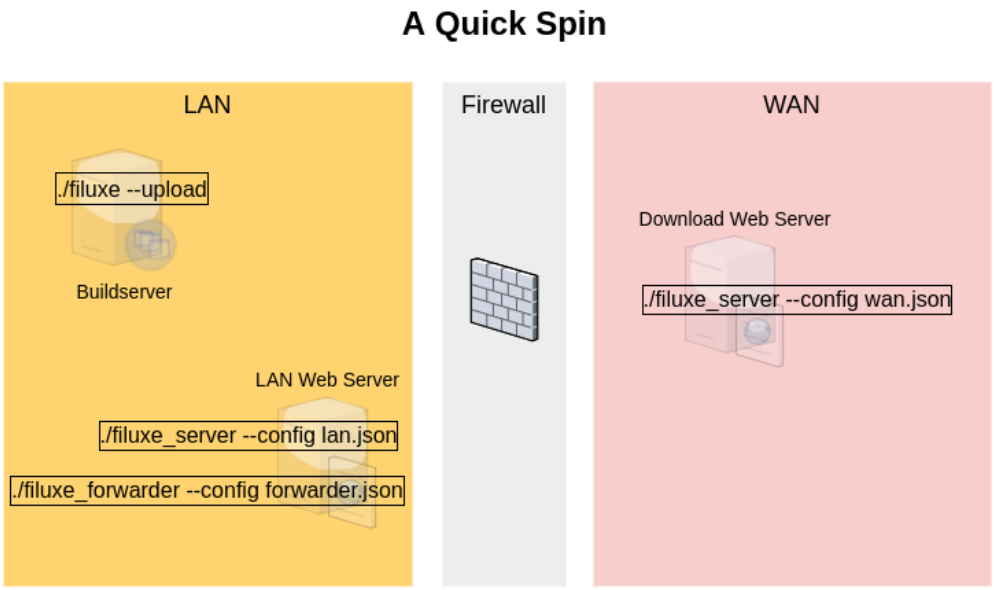
9 Deployment **21**
- 9.1 LAN Webserver . . . 21
- 9.2 WAN Webserver . . . 21
- 9.3 Certificates . . . 21
- 9.4 Others . . . 22

10 Filuxe **23**

CHAPTER 1

A quick spin

The quick spin here consists of starting a webserver and a forwarder application on the 'LAN' and a webserver on a 'WAN' where both servers are in fact just running in parallel on the same local PC.



Finally files are uploaded via HTTP from the fictive buildserver to the LAN webserver with the filuxe script, after which the forwarder will hopefully automatically forward the file from the LAN to the WAN webserver filestorage.

Before starting then make local copies of the three configuration files config_lan, config_wan and config_forwarder.json.template where the .template extension is stripped. Finally make a local copy of the file rules.json.template as well.

Also have a look at *installing* to get the dependencies installed.

Other ways to get filuxe fired up is to run the livetest.py script which starts a filuxe system from scratch in order to run a few testcases and the test_servers.py script that starts a filuxe system (in ./test) and leave it running for testing and

experimenting.

1.1 LAN Webserver

As the very first thing the LAN server needs a filestorage to work on. Since the default in the configuration file is /opt/filestorage_lan then make this directory and make it accessible for the user:

```
# mkdir -p /opt/filestorage_lan
# chown user:group /opt/filestorage_lan
```

Now start the LAN webserver. Its not actually needed unless you would like to work with the LAN filestorage over HTTP but for the sake of this quick walk through that is precisely what you will like to do further below.:

```
./filuxe_server.py --config config_lan.json
```

Verify that it lists the filestorage root as /opt/filestorage_lan.

1.2 WAN Server

Obviously the WAN web server will run on an internet connected server somewhere but here it is just started alongside everything else on the local PC. The web server application itself is the same for both LAN and WAN server.

Since the WAN server will use HTTPS then make a certificates directory and generate the needed SSL keys with a *openssh oneliner*.

Next make a WAN filestorage as /opt/filestorage_wan like it was done for the LAN web server filestorage and then start the server:

```
./filuxe_server.py --config config_wan.json
```

Now verify that this server uses the filestorage root /opt/filestorage_wan.

1.3 LAN Forwarder

The forwarder application will, based on a rules setup, forward files from the LAN filestorage to the WAN webserver via http (in real life through the company firewall). It needs to know how to forward in the form of a json configuration file. It is default assumed to be named rules.json so make a very rudimentary rules.json with the following content:

```
{
    "default": {
        "export": true,
        "delete": true
    },
    "dirs": {
        "test": {
            "include": "zip"
        }
    }
}
```

Next the forwarder is started. It has its own configuration file bridging the two worlds of LAN and WAN, defining the local LAN filestorage root and the address of the remote WAN web server (which is started further down):

```
./filuxe_forwarder.py --config config_forwarder.json --debug
```

Verify that it has started scanning /opt/filestorage_lan and intends to communicate with a "internet" server at localhost:9000.

1.4 Buildserver

With the applications running on the LAN and WAN webservers the backbone of Filuxer is now up and running. What is left is to add and remove files from the LAN webserver filestorage and check if the WAN webserver succeeds as a mirror. Given that everything runs on the same PC there is direct access to the LAN web server filestorage and a quick check is in order to verify that Filuxe is operational::

```
touch /opt/filestorage_lan/direct.zip
```

Check that it got mirrored::

```
ls /opt/filestorage_wan/
direct.zip
```

It were! Now delete the file /opt/filestorage_lan/direct.zip again and watch that the file dissapears from /opt/filestorage_wan/. So far so good.

The problem is that real life buildserver(s) (or whatever the producers are) might have no direct access to the filestorage on the LAN server filesystem. They should instead use the script filuxe.py which is a utility for managing the LAN filestorage via HTTP inside the LAN.

Make a dummy test file:

```
touch test.zap
```

Now anyone on the LAN can add and remove files with the filuxe script. Notice that the source filename and the destination filename are both given as separate entries. Depending on context this can be either handy or rather daft.:

```
./filuxe.py --config config_lan.json --upload --file test.zap --path test/test.zap
```

The servers will automatically construct missing subdirectories found in the –path argument if they don't exist.

Also notice that it didn't actually work. The file appeared in the LAN filestorage but it didn't show up on the WAN server. This is due to the rules.json made earlier, it specifically stated that only zip files should be forwarded from the 'test' directory. So rename the zap to zip and run the filuxe line above once more. Rather than checking in /opt/filestorage_wan then use filuxe to view the WAN filestorage (notice the new config file, now one that points to the WAN server is needed)::

```
./filuxe.py --config config_wan.json --list --pretty --path test
```

Which gives:

```
{
    "filelist": {
        "test": {
            "test.zip": {
                "size": 0,
                "time": 1591819292.2268672
            }
        }
```

(continues on next page)

(continued from previous page)

```
    },
    "info": {
        "dirs": 1,
        "fileroot": "/opt/filestorage_wan/",
        "files": 1
    }
}
```

The net result of the whole exercise to this point is that the file was ultimately saved on the WAN fileserver. Whats left is now that some products or endusers will download files as they see fit. How they do that is not considered part of the filuxe project.

CHAPTER 2

Filuxe script

The filuxe.py script can be used for accessing the LAN and WAN filestorages over HTTP(S). It follows the filuxe philosophy of going all in on HTTP filetransfers via python scripts. Do however keep in mind that what it does is to work on a filestructure on a local server which probably sounds familiar, and that it does so without any builtin security. So if corporate access control is required then SMB or NFS mounts does pretty much the same, but with proper access control.

filuxe.py can be used to upload, download and delete files on either filestorage and it can be used to get a list of files on either as well. filuxe.py is only envisioned to be used on the LAN, the expectation is that accesses to the WAN filestorage by e.g. products will be done with a plain wget rather than with the filuxe python script. That might however be a flawed expectation.

It will as any other filuxe script need a configuration file. It will have a preference for accessing the LAN filestorage so if it can find a "lan_host" and a "lan_port" entry then this is what it will be using. If LAN entries are not present it will load the wan server address instead. Typically it can therefore be launched with the LAN server config directly or a copy of the forwarder config where the LAN references are deleted forcing it to fall back to WAN access.

The filuxe.py script is just a thin facade on top of the filuxe core script which filuxe.py uses together with the forwarder script for working with HTTP filetransfers.

CHAPTER 3

Forwarder

Note: *The forwarder has a file count limiter where it starts to delete files in a filestorage according to a "max_files" setting. Never enable this in the rules file described below if there are file storages with production files that may not be lost. There are countless ways this could happen, a forwarder gone south, the documentation on this page beeing wrong or outdated or perhaps just regex rules that did't quite work out as expected. Thread with care.*

The forwarder is intended to run on the LAN server where it monitors the LAN filestorage for changes and updates the WAN webserver accordingly by e.g. uploading or deleting files over HTTP(S). It operates based on a rules file defining which kind of files to forward from the LAN to the WAN server and as mentioned above, it can enforce a maximum number of files in a given directory. The use case is a build system which can now upload artifacts from high speed development branches/assets without worrying (too much) about disks running full with obsolete artifacts.

Since the forwarder might be useful as just a file count limiter it can run on a LAN server even though there is no remote WAN server specified (meaning that it will never actually forward anything) and it can just as well run on a WAN server by telling it to use the WAN filestorage as the 'LAN filestorage'.

The forwarder arguments:

```
./filuxe_forwarder.py -h
usage: filuxe_forwarder [-h] [--config CONFIG] [--rules RULES] [--templaterule] [--
↪dryrun] [--verbose] [--info]

optional arguments:
-h, --help       show this help message and exit
--config CONFIG  configuration file, default config_forwarder.json
--rules RULES    rules json file. Default is an empty rule set forwarding everything
--templaterule   make an example rules.json file
--dryrun         don't actually delete files
--verbose        enable verbose messages
--info           enable informational messages
```

3.1 Config

The forwarder requires a configuration file telling it where to find the servers. It can be regarded as containing the parts from the LAN and WAN server configuration files that the forwarder should use.

If the forwarder should run on a server for just using its file count limiter, its configuration could look like

```
{
    "lan_filestorage": "test/filestorage_lan",
    "lan_host": "localhost",
    "lan_port": 8000
}
```

Since there is no wan settings present the actual forwarding is disabled and this forwarder will only be useful as file count limiter given it has a matching rules configuration (next chapter). Since this is most likely the same configuration as the LAN server is using the forwarder can simply be given the LAN server configuration.

The typical forwarder configuration with forwarding from a LAN to a WAN will look something like:

```
{
    "lan_filestorage": "test/filestorage_lan",
    "lan_host": "localhost",
    "lan_port": 8000,
    "wan_host": "localhost",
    "wan_port": 9000,
    "wan_certificate": "test/certificates/cert.pem.devel",
    "certificates": [
        "test/certificates/cert.pem.devel",
        "test/certificates/key.pem.devel"
    ],
    "write_key": "devel"
}
```

Since there are a certificate entry for the WAN server this will be contacted via https where the LAN server by the same logic will use plain http. This configuration is then the merge of the configuration files used for both the LAN and the WAN servers.

There are some forwarder configuration example files used by the live test in config/fwd.

3.2 Rules

The rules file is needed only if forwarding should be changed from the default 'just forward everything as is' and/or the file deleter should be activated with 'max_files' different from the default implicit value of 'unlimited'.

The rules file is in json and consist of a default section and a list of directories needing special settings. Both entries are optional and the rules file itself is optional as well. The default behavior of the forwarder is that all files are forwarded and that there are no limit for the number of files.

A rules file could look like this:

```
{
    "default": {
        "include": [".*\\.zip"],
        "max_files": 2
    },
    "dirs": {
```

(continues on next page)

(continued from previous page)

```
        "first": {
            "max_files": 1,
            "exclude": ["unversioned_..zip"],
            "version": ".*?:(\\d+.\\d+.\\d+):.*?",
            "group": ["(.*?)\\:\\d+\\.\\d+\\.\\d+\\:(.*)"],
            "delete_by": "version"
        },
        "second": {
            "include": [".*"],
            "max_files": "unlimited"
        },
        "second/second": {
            "max_files": 2
        }
    }
}
```

Take care. In the rule set above e.g. the 'max_files' equal 2 in the default section will be the default for all directories found recursively which might lead to a lot of unintended file deletions. Use the –dryrun argument on the forwarder in order to spot any unexpected behaviour, with –dryrun no files will actually be deleted.

Forwarder rules files used by the live test can be found in config/rules and the example above is live_test_forwarder_as_deleter.json. A matching testset with files and that exercises these rules can be found in test-data/filestorage_lan.

Note that the default section is the same as the rule for the root directory meaning that settings here will be default inherited for all directories in the entire directory tree. Rules for a given directory is the rules inherited from the parent directory with any explicit rules in a given directory -rewriting- the rules inherited from the parent.

Default rules:

"export": true Setting "export" to false makes it possible to exempt a directory from forwarding from LAN to WAN.

"delete": false Setting "delete" to true will make the forwarder delete files deleted on the LAN filestorage on the WAN filestorage as well. If "export" is true and "delete" is true then the WAN filestorage will be a replica of the LAN filestorage.

"sync_at_startup": false Set to true to syncronize WAN fileserver during start.

Directory rules

These settings can be listed in the default section as well but if they are present in a given directory section these will take precedence.

"max_files": "unlimited" Default is no limit to the number of files. Otherwise the limit as a plain integer.

"include": ".*" Default is to include everything

"exclude": "(?!)" Default is to exclude nothing

"delete_by": "time" Requires a positive value for "max_files". "time" is the default which will delete files from oldest first. "version" will delete files from lowest version first and requires a "version" regex, see below. The third criteria would be "age" but this is not implemented yet.

"version": ".(\d+.\d+.\d+)." Primary regex group for version matching. The regex shown above will look for the pattern ".number.number.number." in the filenames. It currently doesnt handle any 'rcX/ alpha/beta" style extensions which it probably should.

"group": None If there are more than one type of files in a directory then a plain "max_files" putting all files in the same basket makes limited sense. It is possible to specify a list of "group" regex expressions which is used to

divide similar files into specific groups. All files in a given group will then be held up against the "max_files" limit.

Group expressions are tried in the order they are listed in the rules file and files that fails to be parsed by any regex expressions will end up in a common group called "ungrouped" (which is probably not what was wanted). A final group regex "(.*)" will make all otherwise unrecognized files end up in their own individual groups with a matching filecount of 1 and they will then not be able to trigger any file deletions (which is probably not what was wanted either).

An example with the "group" *["(.*?)\:\d+.\d+.\d+\:(.*)"]* containing a single regex:

```
Files                                       Internal group key
a:1.1.1:anytrack:anyarch:unknown.zip        a:anytrack:anyarch:unknown:zip
a:1.1.2:anytrack:anyarch:unknown.zip        a:anytrack:anyarch:unknown:zip
a:1.1.3:anytrack:anyarch:unknown.zip        a:anytrack:anyarch:unknown:zip
b:1.1.3:anytrack:anyarch:unknown.zip        b:anytrack:anyarch:unknown:zip
```

So if "max_files" is 1 and files are deleted by version "\:(\d+.\d+.\d+)\:" then the remaining files will be (*)

```
a:1.1.3:anytrack:anyarch:unknown.zip
b:1.1.3:anytrack:anyarch:unknown.zip
```

(*) at least in theory.

CHAPTER 4

Web server

It is the same webserver application that is used for both LAN and WAN and each domain will provide an individual configuration file to its webserver instance. The webserver can run both HTTPS and plain HTTP, see *here* for generating a selfsigned certificate for HTTPS. For the sake of development and testing Filuxe, the LAN is considered safe and the LAN webserver runs with plain HTTP, while the WAN server is on the roaring internet and uses HTTPS.

Flask debugging is default off, to turn it on while developing launch the webserver as

```
FLASK_DEBUG=1 ./filuxe_server.py ....
```

4.1 Write access

The WAN webserver requires a key for operations that modifies the filestorage. The idea beeing that the clients/products on the WAN that are otherwise able to access the webserver for downloading files wont be at risk for exposing write access to the webserver in case they get compromised. The forwarder service script operating from the LAN will be the only one who need to know the key for the WAN server and the LAN server runs without the key. The key can be found as "write_key" in the WAN server and the forwarder configuration files and should obviously be changed to something more exotic that the default key "devel".

4.2 Authentication

As a proof of concept the route '/' which is serving a static HTML page can be password protected if a username and a password is specified in the configuration file. Besides that this topic is just left as is for another day. There are (at least) two usecases that are in faviour of password protected access. The first is in case not a file but rather a URL to the WAN server is sent around the world. It will just appear to be not-very-professional if a file can be downloaded without the need for any credentials. The second usecase could be to protect against denial of service attacks where the server is flooded with a gazillion downloads. See also Flask basic HTTP AUTH

4.3 Tips and tricks

When one or both of the LAN and WAN servers are just plainly refusing to talk nicely with e.g. the filuxe.py script then curl can be used to check if the servers are reachable and working as intended. The servers have a default route printing a single HTML text and this will be used to detect a working server. The curl commands below are assumed to be executed from the LAN.

LAN, HTTP:

```
curl -v http://localhost:8000
```

The last part of the output is the HTML output from the server and it should read "filuxe_server_LAN".

WAN, HTTPS:

```
curl --cacert certificates/cert.pem.devel -v https://<server>:9000 --insecure -u␣
↪name:pwd
```

After a lot of SSL goblidigook the last HTML part should now contain "filuxe_server_WAN".

CHAPTER 5

Web server direct HTTP

Rather than using filuxe.py to interact with the LAN webserver, you can use curl and wget for direct HTTP(S) access instead. Since the python script will follow the server script in case of breaking changes, filuxe.py will always be the safest bet for the job of adding and deleting files from the LAN webserver filestorage.

And the mandatory warning: never trust anything you download from the WAN webserver, HTTPS or not. It should be used with end to end encrypted files only. You would never ever work with plain zip files in real life as the examples on this page appears to be doing.

5.1 Upload file

Since the webservers on the LAN and WAN are exactly the same the upload can be used to with the WAN server just as well as the LAN server. That would obviously shortcircuit most of whatever filuxe does and render filuxe itself kind of moot. At time of this writing the commands **upload** and **delete** require a key for the server to accept them. This increases the security by approximately nothing, at least when running plain http.

```
curl <host>/upload/<dest> -H "Content-Type:application/octet-stream" -H "key: secret_
↪write_key" --data-binary @<src>
```

5.2 Get list of files

```
curl -s <LAN host>/filelist/

{
  "filelist": {
    ".": {
      "here_is_a_file.zip": {
        "size": 12,
        "time": 1640218563.1022844
      },
```

(continues on next page)

(continued from previous page)

```
      "here_is_another_file.zip": {
        "size": 56,
        "time": 1640218582.7589521
      }
    }
  },
  "info": {
    "dirs": 1,
    "fileroot": "test/filestorage_lan/",
    "files": 2
  }
}
```

Alternatively “wget -A zip localhost:8000/files -O filelist”

5.3 Download file

This would be one way for products to download files from the WAN web server.

```
wget <host>/download/test.zip
...
2020-05-18 00:41:48 (560 KB/s) - 'test.zip' saved [11/11]
```

For self signed SSL certificates add ‘–no-check-certificate’.

CHAPTER 6

Keys and certificates

Self signed certificates for WAN SSL (HTTPS):

```
openssl req -x509 -newkey rsa:4096 -nodes -out cert.pem.devel -keyout key.pem.devel -
↪days 365
```

Accept everything except setting common name to the WAN hostname (or localhost for testing locally). The above line will give "Certificate for localhost has no *subjectAltName*" warnings from urllib3, which for now are simply silenced. Since the warning is for real the fix is to generate a certificate with a *subjectAltName* which is pending...

For a minimum amount of fuzz then manually make a directory called 'certificates' and run the openssl command from there. All templates and examples expects it to be this way.

The same certificate will have to be generated (or exist) on both the LAN server (to be used by the forwarder) and the WAN server (to be used by the WAN server itself).

CHAPTER 7

Filebrowser

A very nice fullblown webbased filebrowser can be found on github here. Its a standalone executable that can be launched on any server hosting a filestorage. It works like a charm and it would be a petty not to recommend it here.

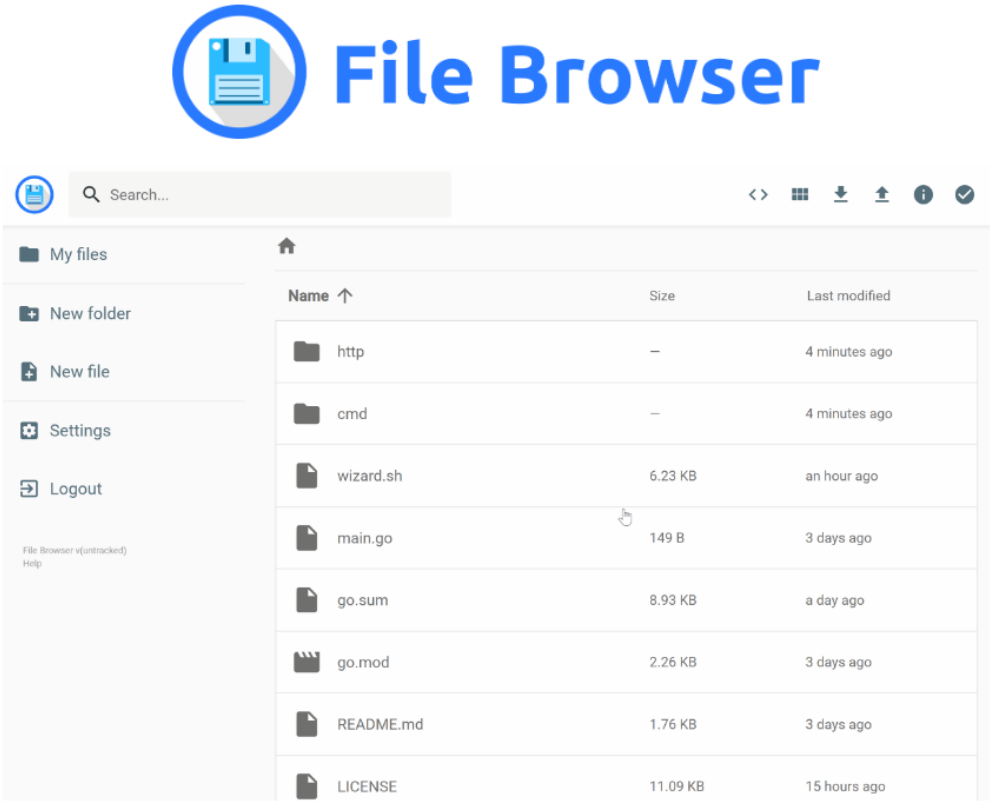


Image captured from the project page at github.

CHAPTER 8

Installing

8.1 Dependencies

A couple of python libraries are required, see ./requirements.txt.

Filuxe has been written on an Arch system with all python dependencies installed as native packages.

Be warned that bringing Filuxe up on e.g. an older Ubuntu with stale python dependencies might be a little challenging.

8.2 Reading

Certificates 101 for localhost by lets encrypt

Certificates 101 for flask

8.3 Credits

Definitely picked up more nice stuff made by others than listed here, but at least there was an attempt:

Chunked file upload with requests see https://gist.github.com/nbari/7335384

Chunked flask file download see https://stackoverflow.com/a/57236538

CHAPTER 9

Deployment

Since the state of the filuxe project is currently play-along-for-fun talking about deployment might be a little pretentious. But still it makes sense to have a page focusing on what needs to be done when filuxe is installed outside a developer pc.

9.1 LAN Webserver

The default LAN webserver is completely open. If that doesn't sound right then make one or more of the following changes in the configuration file:

- make "username" and "password" entries (still just for basic auth).
- make the server run HTTPS by supplying it with a "lan_certificate" entry.
- add a "write_key" needed for operations modifying the LAN filestorage

If about to add everything in then consider to base the LAN configuration file on the WAN configuration file since it uses all of the above.

Change "lan_host" from "localhost" to "0.0.0.0" to make the server listen to all adresses.

9.2 WAN Webserver

Change the "username", "password" and "write_key" entries to something different from the defaults.

9.3 Certificates

If self signing SSL certificates for HTTPS then remember to re-run openssl whenever the host is changed.

9.4 Others

Change any systemd service scripts to be owned and writable by root only.

The rest of filuxe is running as a plain user, including the configuration files, which is rather unambitious. This is just how it is currently.

CHAPTER 10

Filuxe

Filuxe is hosted at github. Please have a look at the readme there for a brief overview.