FFctl Documentation

Release 0.1 beta

spky

September 27, 2016

Contents

Gateway Server control scripts for Freifunk.

Frei**F**unk**c**on**t**ro**l**

(who would have guessed it?)

Used by Freifunk-Mainz and Freifunk-Wiesbaden.

If you are reading this as `source-code` you could also enjoy the pleasures of a rendered version at readthedocs.

The main Repository is at Github: FFctl

Note: FFctl is currently under development.. Basic things are settled, but still, you have been warned..

Before you install, please read at least about the *Concepts* of FFctl to have an idea of whats going on..

CHAPTER 1

What does FFctl do

On our Gateway-Servers we have the need to run periodic tasks, for example checking if the Gateway-Server is still properly connected to the OpenVPN exit, or syncing keys for fastd (our Node to Gateway VPN) from git repos.

This is perfect for the crontab (or something alike), the best-case scenario would be to initially configure FFctl and then never have the need to login on any of the Gateway-Servers again :)

Let's rather not talk about the real-world scenario :)

Second, FFctl can be used to build a batch of Gluon images on a nightly basis (also by a call via crontab) and automatically release them for the Gluon autoupdater.

In general, the idea of FFctl is to help you in all those common tasks every Freifunk Community is facing.

So, this is an official invitation - No matter which Freifunk Community you are part of (if any), you are invited to improve FFctl.

Please contribute code there or use the issue tracker for questions, ideas, recommendations, etc.

1.1 Why python?

Of course python!

Most parts were already implemented in shell (bash) code, but this brings some problems:

Writing serious shell scripts is best described by using the following words in a specific order:

- pain
- bloody
- ass
- the
- in

serious as in serious business. Not those scripts fixing various problems for the moment, scripts with 150+ lines...

In general you can't clearly see what's going on just by reading the code because you have to wrap every actual shell command with some helper code, to ensure proper functionality.

So the idea was born to write a generic pre-script, which will set up most things beforehand, so we can keep the actual scripts doing the work short and readable. Second common methods are available to safely execute commands or do stuff in a controlled environement.

And while we are on it, why not switch to a proper scripting language?

Python3 is finally easily available for Linux, the standard library is huge and helpful, the syntax just beautiful

Using classes combined with multiple inheritance gives us the power to easily distinguish between helper code and actual code.

Inheriting directly from *util.ffbase.FFbase*, each helper script instantly has a bunch of custom settings and helper methods at hand, allowing short and simple scripts.

CHAPTER 2

Concepts

FFctl is weird from the inside as weird from the outside.

To help you understand what the hell is going on read this: TODO

2.1 FFbase

TODO

2.2 His brother: FFgit

TODO

2.3 The Stage System

TODO

2.4 What is a stage?

TODO

2.5 meta?!?

TODO

CHAPTER 3

Installation

FFctl installation is now simplified!

A simple clone anywhere (in your Homefolder) is the first step. Then you just have to call the setup/fix script:

```
git clone https://github.com/freifunk-mwu/ffctl ~/clones/ffctl
cd ~/clones/ffctl

./ffdoctor.py
```

Now you should have a script called `ffctl` in your `~/bin`-folder, pointing to `~/clones/ffctl/ffctl.py`.

Note: never run any of these scripts as root user.

No matter where the folder of FFctl is, it will always find its locations and paths to the configuration and working directories, as long as you don't switch users.

3.1 Requirements

python You need python3 to run FFctl, for some dependencies see `requirements.txt` file.

git You need git >= 1.8 (because of the heavy used `-C` switch)

pip Please install pip for python3.

For Debian/Ubuntu some would use:

```
sudo aptitude install python3-pip
sudo pip3 install -r requirements.txt
```

3.1.1 Build Server

For using `gluon_builder` you must setup the following:

```
sudo aptitude install build-essential subversion libncurses5-dev zlib1g-dev gawk gcc-multilib flex bi
```

To be able to sign Images, you need the **ecdsautils**. They depend on **libuecc**

libuecc There is a Debian Repo

Install as root:

```
echo "deb http://repo.universe-factory.net/debian/ sid main" > /etc/apt/sources.list.d/universe-facto
apt-key adv --keyserver keyserver.ubuntu.com --recv 16EF3F64CB201D9C
aptitude update
aptitude install libuecc0 libuecc-dev
```

ecdsautils No package in sight, just build it

As a normal user do:

```
git clone https://github.com/tcatm/ecdsautils ~/clones/ecdsautils
cd ~/clones/ecdsautils
mkdir build && cd build

cmake ..
make

sudo make install
```

See the Github-Repo of ecdsautils for more details.

CHAPTER 4

Bobby the Gluonbuilder

The gluon builder builds gluon builds!

It's as simple as that.

Note: Bobby is not related to Little Bobby Tables

The complete build system of FFctl is split into two parts.

You have the **FFbuilder** component, it's job is to integrate Gluon into the concepts of FFctl, and then there is **gluon_builder** (a.k.a. Bobby) to handle the rough compiling part.

4.1 gluon_builder (Bobby)

For reason of performance Bobby is written as a bash script. It depends heavily on FFbuilder because it expects an already prepared Build directory with all files in place and to get all information needed for building a batch of images for a community.

You could strip down Bobby's core function into this weird pseudocode:

```
ffctl --prepare
ffctl --getbuildconfig

for $community in $communities_from_FFctl_config:
    cd $prepared_build_dir_for_community
    make update
    make GLUON_BRANCH=$stablebranch_from_FFctl_config:
    for $branch in $branches_from_FFctl_config:
        make manifest GLUON_BRANCH=$branch
        sign $bobbys_autosign_key images/* $branch.manifest

ffctl --postbuild
```

I hope this helps to get the general idea.

With a manifest file for each branch already in place it becomes easy to add additional signatures to it. To those manifest files will always be symlinked to if the images are published for autoupdate.

4.2 FFbuilder

This module serves two masters.

Bobby Offering `gluon_builder` the possibility to run a prepare and postbuild, as well as passing all necessary information to gluon_builder.

See the Module Documentation of *`ffbuilder.FFbuilder`* if you want to know the details.

You Offer a user interface to publish and sign complete image-batches.

Note: If you are called Bobby, try not to confuse yourself with Bobby the Gluonbuilder or even with Little Bobby Tables!

4.2.1 autobuild

gluon_builder is designed to prepare and clean up everything using FFbuilder.

Using a repeated call to gluon_builder will populate your library folder.

A crontab entry could look like this:

```
42 23 * * * /bin/bash $HOME/clones/ffctl/gluon_builder.sh >> /dev/null 2>&1
```

Note: Do not set too short intervals, once daily should be enough. Parallel building is not supported :)

CHAPTER 5

Modules

If you are looking for information on a specific function, class or method, *this part* of the documentation is for you.

5.1 Modules

FFctl uses inheritance as it's core functionality.

5.1.1 util

Welcome to the core.

Within this folder the most basic stuff is located.

Every FFctl script includes one of the classes in util (either *util.ffbase.FFbase* or *util.ffgit.FFgit*), and operates through them on the methods listed below. Please avoid calling any of the methods from util directly.

ffbase.py

This is the base, from which every class inherit from. Every operation should done through the *util.ffbase.FFbase.m()* method to allow mail-backlog, logging, failsafe command launching and proper system output.

class util.ffbase.**FFbase**(*verbose=True*)

The base class for FFctl.

This is the main starting point for FFctl to work, every script is a child instance of FFbase.

Seealso *__init__()*

Every operation in a child class is done through these methods:

Seealso *m()* and *mail()*.

__init__(*verbose=True*)

Does the most elementary parts to ensure smooth operation:

- Checks if the current user is not the root user, aborts otherwise.
- Checks if the current python version is at least 3, aborts otherwise.
- Loads configuration from disk if any is found.
- Loads a default configuration (generated from DEF and CONF).

•Merges changed/updated values in configuration with local one.

•Writes it back to disk.

•finally sets up logging and logfiles.

Parameters **verbose** – show styled output

Seealso *util.system.check_root()* *util.ffbase.FFbase._FFbase__load_config()* *util.ffbase.FFbase._FFbase__load_logger()*

_FFbase__load_config()
Loads local configuration from file, merges changed values with updates and defaults, then saves the configuration back to file.

Return type A dictionary with all settings

_FFbase__load_logger()
Opens/creates a logfile (according to configuration), and provides logging functionality for FFbase (used in *m()*).

Return type An instance of a python logger

digest(*metaimport=None*)
TODO

get_meta()
TODO

m(*msg*, *cmd=None*, *cmdargs=None*, *cmdkwargs=None*, *sudo=False*, *err=False*, *postmsg=False*)
main method managing multiple massively mixed meshnetworks for FFctl!

Note: every lowercase *m* followed by a *(*-bracket looks like a facepalm! **m(**

This is the main function to interact with the FFctl internals.

Runs either a python function or calls a shell command, returning the untouched output.

m() directs *msg* and command outputs to *stdout* (according to *ffctl_verbose*) and *logger* chains

Parameters

- **msg** – what you are currently doing
- **cmd** – run either a python function or shell command
- **cmdargs** – pass arguments to python function in *cmd*
- **cmdkwargs** – pass keyword arguments to python function in *cmd*
- **sudo** – run shell command from *cmd* as root
- **err** – treat *stdout*/log output as error indicated by "!"
- **postmsg** – output message after everything's done

If *msg* is **False**, all shell output from passed *cmd* functions/calls will be suppressed

mail(*punchline=None*)
Sends mail. Only if *mailalert* is **True**.

•The text will be the punchline plus the log output since FFctl startup.

•The recipient is *mailrcpt* (your admin list).

Parameters **punchline** – reason why this mail was sent

Seealso *util.mail.send_mail()*

As punchline you could use something like:

```
self.mail('Dear Sir or Madam, I am writing to inform you about a fire in the building ...')
```

No, that's too formal..

stage(*newstage=None*, *context=False*)
TODO

Parameters **newstage** – Your future stage. If set to `None` it will set the default stage

Do not launch this script directly, or the front will fall off.

ffgit.py

FFgit is a direct successor of *util.ffbase.FFbase* and inherits all it's functions. So either include FFgit or FFbase.

class util.ffgit.**FFgit**
Teaches FFctl how to use git, as a child class of: *util.ffbase.FFbase*.

Every script which messes around with git repositories should be a child instance of *util.ffgit.FFgit* instead of *util.ffbase.FFbase*.

In the constructor the function *ffgit_setup()* should be called to set up things properly beforehand.

__init__()
Calls it's superclass' constructor: *util.ffbase.FFbase.__init__()*

ffgit_publish()
This invokes *ffgit_save()* first, and then pushes the changes into the *remote*

This function will leave your repo checked out at *branch* behind

ffgit_save()
Stages all changes, removes deleted files, commits in your specified *branch* and merges it with the master branch.

This function stops on a merge conflicts sending a mail crying for help!

This function will leave your repo checked out at *branch* behind

ffgit_setup(*target*, *remote*, *branch='master'*, *relativetarget=True*)
Initializes everything you need for git.

Parameters

- **target** – Local folder path for repository
- **remote** (*Some git url (git://... ssh://...)*) – Remote url for repository
- **branch** – Branch to work on in repository
- **relativetarget** – Treat *target* as relative path starting from ffctls data directory

If *realtarget* is set, the repo will be expected to be under `~/.local/share/ffctl/**target**` otherwise `/**target**`

Seealso *util.io.get_datadir()* *util.io.get_backuppath()*

On startup all possible cases are considered, and the proper actions will be launched then:

•No folder under target found: **Fresh clone**

If there is already a folder under target, it checks the folder:

•This folder is not a git repo: **Move to backup folder -> Fresh clone**

•If this folder is a git repo, it compares the remotes with value of remote

- This is another repo: -> **Move to backup folder -> Fresh clone**
- This is the same repo: -> **Stage and commit all changes -> git pull**

The comparison of the remotes with the value of **remote** is just a fuzzy matching, but works properly if the repo was initially created using *FFgit* :)

No matter what the preconditions are, *git_setup* will leave the most recent version of your *remote* behind, all changes commited, checked out at *branch*.

ffgit_target()
Get the current git repo path

Return type Full path as a string

Do not launch this script directly, or the front will fall off.

crypt.py

Deals with crypto stuff..

It can generate both public and private keys for ssh and ecdsa

you'll need the ecdsautils compiled and installed to work with ecdsa keys

Note: Cypherpunks suck!

util.crypt.**ecdsa_genprvkey**(*kfile*)
Generates a private ecdsa key

Note: you'll need the ecdsautils compiled and installed to work with ecdsa keys

Parameters **kfile** – Where to store that key

util.crypt.**ecdsa_genpubkey**(*kfile*, *pfile*)
Generates the public ecdsa key from the given private key and writes it next to the private one.

Note: you'll need the ecdsautils compiled and installed to work with ecdsa keys

Parameters

- **kfile** – Path to the (existing) private key
- **pfile** – Path to the (not yet existing) public key

util.crypt.**ssh_genprvkey**(*kfile*, *ktype*, *ksize*)
Generates a passwordless ssh key pair

Parameters

- **kfile** – Where to store that keypair
- **ktype** – What type of keypair (eg. rsa)
- **ksize** – Key length/size

util.crypt.**ssh_genpubkey**(*kfile*, *pfile*)
Generates the public ssh key from the given private key and writes it next to the private one.

Parameters

- **kfile** – Path to the (existing) private key
- **pfile** – Path to the (not yet existing) public key

git.py

Deals with git repos.

a.k.a. passes more or less the untouched parameters to `git -C CDIR ... ...`

Note: You need git >= 1.8 for most of the functions.

util.git.**git_add**(*cdir*, *target*)
Adds a file to repo

Parameters

- **cdir** – Your repo path
- **target** – The file to add

util.git.**git_checkoutbranch**(*cdir*, *branch='master'*)
Checks out a branch

Parameters

- **cdir** – Your repo path
- **branch** – The branch to checkout

Note: If specified branch already exists in the remote repository, a tracking branch will checked out, otherwise a new one will be created.

util.git.**git_clone**(*cdir*, *remote*)
Clones a git repo from a remote

Parameters

- **cdir** – Your repo path
- **remote** – Url to your remote

util.git.**git_commit**(*cdir*, *message*)
Commits into repo

Parameters

- **cdir** – Your repo path

- **messag** – The commit message

util.git.**git_config**(*field*, *value*)
Configures git

Parameters

- **field** – User field to set (user.name, user.email)
- **value** – Value for field ("Roy Kabel", "roykabel@freifunk.net")

util.git.**git_fetch**(*cdir*)
Fetches from remote

Parameters **cdir** – Your repo path

util.git.**git_log_pretty**(*cdir*, *fstr='%s'*, *n=10*)
Shows formatted entries from git log

Parameters

- **cdir** – Your repo path
- **fstr** – The format string (e.g. use `%h` or `%H` to get the last hashes)
- **n** – How many entries should be returned

Return type String with formatted git output depending on your input

util.git.**git_merge**(*cdir*, *branch*, *message*)
Merges repo

Parameters

- **cdir** – Your repo path
- **branch** – Merge which branch
- **messag** – The merge message

util.git.**git_pull**(*cdir*)
Pulls from remote

Parameters **cdir** – Your repo path

util.git.**git_push**(*cdir*, *branch='master'*)
Pushes to remote

Parameters

- **cdir** – Your repo path
- **branch** – Commit into which remote branch

util.git.**git_remote**(*cdir*)
Shows a list of remotes stored in the repo

Parameters **cdir** – Your repo path

Return type String with output from git remote

util.git.**git_rm**(*cdir*, *target*)
Deletes a file from repo

Parameters

- **cdir** – Your repo path
- **target** – The file to delete

util.git.**git_status**(*cdir*)
Shows current status in repo

Parameters **cdir** – Your repo path

Return type String with git porcelain syntax for current status

util.git.**is_git**(*cdir*)
Checks if given folder is a git repo via running git status

Parameters **cdir** – Your repo path

Return type Boolean

helper.py

Helps doing stuff.

Currently it deals with timestamps and can render jinja2 templates.

util.helper.**render_template**(*tfile*, *tvars*)
Renders a Jinja2 template

Parameters

- **tfile** – Full path to the Jinja2 template file
- **tvars** – Your Jinja2 template variables

util.helper.**timestamp**()
Gets the formatted current timestamp

Return type Current time stamp as string

util.helper.**tstamp**()
Gets the current UNIX time stamp, Redneck style

Return type Current UNIX time stamp as int

io.py

Deals with files.

Can handle files, folders and locations:

- find specific locations on the disk
- split, strip, join, create, copy, delete, ... operations
- read and write files

util.io.**backup_path**(*target*)
Backups files/folders and it's subfolders.

Parameters **target** – What to backup

Seealso *backup_path() unlink_path() del_path()*

util.io.**chmod_path**(*target*, *mask*)
Sets permissions to a path.

Parameters

- **target** – Your path (must exist)

- **mask** – Your mask (e.g. for an octal mask: `0o644`, `0o755`)

`util.io.`**copy_path**(*source*, *target*)
Copies files/folders and it's subfolders to a new location.

Parameters

- **source** – What to copy
- **target** – Where to copy

`util.io.`**del_path**(*target*)
Deletes files/folders and it's subfolders. Be careful!

Parameters **target** – What to delete

Seealso *backup_path() unlink_path() del_path()*

`util.io.`**exists_path**(*target*)
Checks if target exists.

Parameters **target** – Your path

Return type Boolean

`util.io.`**expand_path**(*target*)
Expands paths.

a.k.a. turns `~/my/awesome/folder` into `/home/**username**/my/awesome/folder`

Parameters **target** – Your path

Return type String with path to the config directory

`util.io.`**get_abspath**(*target*)
Strips relative paths into absoulute ones.

a.k.a turns `~/my/not/../awesome/folder` into `/home/USER/my/awesome/folder`

Parameters **target** – Your path

Return type The absoulute path as a string

`util.io.`**get_backuppath**()
Where to store your backups.

Return type String with path to the backups directory

Seealso *get_configfilepath() get_datadir() get_logfilepath() get_backuppath() get_stagedir() get_builddir() get_binpath() get_sbinpath() get_sshdir() get_ecdsadir()*

`util.io.`**get_basename**(*target*)
Strips the folder part from a path.

a.k.a turns `/usr/local/bin/my_script.sh` into `my_script.sh`

Parameters **target** – Your path

Return type The leftover part as a string

`util.io.`**get_binpath**()
Where to install the helperscripts

should be in your `$PATH`

Return type String with path to the user's bin directory

Seealso *get_configfilepath()* *get_datadir()* *get_logfilepath()* *get_backuppath()* *get_stagedir()* *get_builddir()* *get_binpath()* *get_sbinpath()* *get_sshdir()* *get_ecdsadir()*

util.io.**get_builddir**()
Where to build the gluon images

Return type String with path to the gluon image compiling directory

Seealso *get_configfilepath()* *get_datadir()* *get_logfilepath()* *get_backuppath()* *get_stagedir()* *get_builddir()* *get_binpath()* *get_sbinpath()* *get_sshdir()* *get_ecdsadir()*

util.io.**get_configfilepath**()
How to find the config file.

Loosely following the XDG folder scheme.

Return type String with path to the config file

Seealso *get_configfilepath()* *get_datadir()* *get_logfilepath()* *get_backuppath()* *get_stagedir()* *get_builddir()* *get_binpath()* *get_sbinpath()* *get_sshdir()* *get_ecdsadir()*

util.io.**get_datadir**()
Where to store your files (e.g. working files, temp files, backups, etc.).

Loosely following the XDG folder scheme.

Return type String with path to the data directory

Seealso *get_configfilepath()* *get_datadir()* *get_logfilepath()* *get_backuppath()* *get_stagedir()* *get_builddir()* *get_binpath()* *get_sbinpath()* *get_sshdir()* *get_ecdsadir()*

util.io.**get_dirname**(*target*)
Strips the file part from a path.

a.k.a turns `/usr/local/bin/my_script.sh` into `/usr/local/bin`

Note: `os.path.dirname` doesn't check if your target is really a file. It just strips the last part after the last **/**

`/usr/local/bin` would become `/usr/local/`, whereas `/usr/local/bin/` would become `/usr/local/bin`

Parameters **target** – Your path

Return type The leftover part as a string

util.io.**get_ecdsadir**()
Where are the ecdsa keys?!

Return type String with path to the user's ecdsa directory (`~/.ecdsa`)

Seealso *get_configfilepath()* *get_datadir()* *get_logfilepath()* *get_backuppath()* *get_stagedir()* *get_builddir()* *get_binpath()* *get_sbinpath()* *get_sshdir()* *get_ecdsadir()*

util.io.**get_logfilepath**()
How to find the logfile.

Return type String with path to the actual log file

Seealso *get_configfilepath()* *get_datadir()* *get_logfilepath()* *get_backuppath()* *get_stagedir()* *get_builddir()* *get_binpath()* *get_sbinpath()* *get_sshdir()* *get_ecdsadir()*

util.io.**get_sbinpath**()
Where to install root's helperscripts (currently for **batctl** only)

Return type String with path to the user's bin directory

Seealso *get_configfilepath()* *get_datadir()* *get_logfilepath()* *get_backuppath()* *get_stagedir()* *get_builddir()* *get_binpath()* *get_sbinpath()* *get_sshdir()* *get_ecdsadir()*

util.io.**get_sshdir**()
Where are the ssh keys?!

Return type String with path to the user's ssh directory

Seealso *get_configfilepath()* *get_datadir()* *get_logfilepath()* *get_backuppath()* *get_stagedir()* *get_builddir()* *get_binpath()* *get_sbinpath()* *get_sshdir()* *get_ecdsadir()*

util.io.**get_stagedir**()
Where is the default stage dir?

Return type String with path to the (specified) stagedir

Seealso *get_configfilepath()* *get_datadir()* *get_logfilepath()* *get_backuppath()* *get_stagedir()* *get_builddir()* *get_binpath()* *get_sbinpath()* *get_sshdir()* *get_ecdsadir()*

util.io.**is_file**(*target*)
Checks if target is a file

Parameters **target** – Your targetpath

Return type Boolean

util.io.**join_path**(*head*, *tail*)
Combine (join) two paths.

Parameters

- **head** – First part of your path
- **tail** – Second part your path

Return type The joined path as a string

util.io.**list_path**(*target*)
Lists files of given folder (flat structure, no depth, no subfolder).

Note: There is no `is_empty` function. Use `len(list_path('/usr/local/bin/'))` instead.

Parameters **target** – Your path

Return type A list of your folder's contents

util.io.**make_path**(*target*)
Creates a folder if it doesn't exists.

Parameters **target** – Your path

util.io.**move_path**(*source*, *target*)
Moves files/folders and it's subfolders to a new location.

Parameters

- **source** – What to move
- **target** – Where to move

util.io.**read_file**(*filename*, ***args*)
Reads content of a given file.

Parameters **filename** – Your file to read

Return type Content as a string

util.io.**read_jsonfile**(*filename*, ***args*)
Reads content of a given json file.

Parameters **filename** – Your json file to read

Return type json content as python structure

util.io.**read_yamlfile**(*filename*, ***args*)
Reads content of a given yaml file.

Parameters **filename** – Your yaml file to read

Return type yaml content as python structure

util.io.**symlink_path**(*source*, *target*, *is_directory=False*)
Sets a symlink to source to the location of target

Source can be specified relative to the target

Parameters

- **source** – Your path to set a symlink pointing to (must exist)
- **target** – Your target where to store the symlink
- **is_directory** – Set to `True` if your source is a directory

util.io.**unlink_path**(*target*)
Unlinks files. Be careful!

Use this for symlinks.

Parameters **target** – What file/symlink to unlink (should exist ;))

Seealso *backup_path() unlink_path() del_path()*

util.io.**write_file**(*filename*, *content*, *create_path=False*)
Writes content to a given file.

Parameters

- **filename** – Your file to write
- **content** – The content to write

util.io.**write_jsonfile**(*filename*, *content*, ***kwargs*)
Writes python structures to a json file.

Parameters

- **filename** – Your json file to write
- **content** – Python structure to write

mail.py

Helps sending Emails

util.mail.**send_mail**(*to*, *messagetext*, *sender*, *subject*, ***kwargs*)
Creates a MIMEMultipart message according to RFC23, RFC42, RFC1337, YOLO, SWAG and BBQ to send that mail.

Parameters

- **to** (*Either a list of strings (multiple recipients), or just one string*) – Recipient of the mail
- **messagetext** – Write something nice here, think of something beautiful
- **sender** – The from address (set this correctly not to get caught in the next spam filter)
- **subject** – Subject line of the mail
- **kwargs.punchline** – First line of the mail

Note: You need postfix (or similar) listening on (local) port 25 on your machine for this to work.

Seealso *util.ffbase.FFbase.mail()*, ffctl.mailer()

system.py

Deals with the most basic stuff.

Most methods will be launched only once at startup and the values are written to the config, read from there afterwards.

It also allows running shell commands.

util.system.**check_pythonver**()
Kills FFctl if run with python2

util.system.**check_root**()
Checks if the current user is root or not

Return type Boolean

util.system.**hostname**()
Return the hostname of the current machine (without domain and stuff)

Return type Your freaking hostname as a string

util.system.**kill_me**(*msg*)
This function prints out a message, exiting everything afterwards with an error

Parameters **msg** – Your famous last words

util.system.**sh**(*cmdline*)
This is a simplified wrapper for *shellrun()*

Parameters **cmdline** – Your shell command to run

Return type Mostly stdout, but stderr on error, as a string

util.system.**shellrun**(*cmdline*, *cin=None*)
Executes a shell command, returning the result

Parameters **cmdline** (*As a string or a list of strings*) – Your shell command to run

Return type A dictionary with the keys *stdout* (not always present) *stdin* (if given) *stderr* and *returncode*, or False on fatal errors

All scripts below are successors of either only *util.ffbase.FFbase* or both FFbase and *util.ffgit.FFgit*.

5.1.2 ffctl.py

This file is the starting point for the user.

`ffctl.`**`builder`**`()`

`ffctl.`**`confb`**`(`*par*`)`
invokes *ffconfigbackup.FFconfigbackup.run()*

Parameters **par** – argumentparser namespace object

`ffctl.`**`doctor`**`()`

`ffctl.`**`draw`**`(`*par*`)`
invokes *ffdraw.FFdraw.run()*

Parameters **par** – argumentparser namespace object

`ffctl.`**`ping`**`()`

`ffctl.`**`sync`**`()`

command line interface

You can use the plain `ffctl` command like this to produce an error and exit :)

```
ffctl
```

You *must* specify an action to run. Let's start with help:

```
ffctl --help
ffctl -h
```

Normaly you would specify which action to launch. To run FFconfigbackup use this:

```
ffctl confb
```

You may choose:

- *builder*: launches `ffbuilder.FFbuilder.helper()`
- *confb*: launches *ffconfigbackup.FFconfigbackup.run()*
- *doctor*: launches *ffdoctor.FFdoctor.run()*
- *doctor*: launches *ffdraw.FFdraw.run()*
- *ping*: launches *ffping.FFping.run()*
- *sync*: launches *ffpeersync.FFpeersync.run()*

You can append a `--mail` flag on every action to send the backlog as mail. Syncing peers, sending the output to the admin-list:

```
ffctl sync --mail
```

This allows to put lines like this into the crontab (e.g):

```
* */8 * * * $HOME/bin/ffctl ping --mail
```

5.1.3 ffconfigbackup.py

class `ffconfigbackup.`**FFconfigbackup**
Copies your config files into a git repo, commits and pushes the changes afterwards

The list of config files to copy is in a json file, which lays in the root of the target repo.

Note: You can find that json file here: https://github.com/freifunk-mwu/gateway-configs/blob/master/queue.json

---

This is done for comfort and timeline purposes:

- Scrolling through the configs and cross comparing between servers is much more easier in a webbrowser
- Helps answering questions like: 'what did you do at the last meeting?' and 'what is wrong so it doesn't work anymore?'

Beware: In contrast to the name, this is not a real backup!

run(*par=None*)
Actually runs everything - copy, publish, done!

Reads in the json queuefile and copies all mentioned files to a subfolder named like your hostname into your config repo

5.1.4 ffdoctor.py

class `ffdoctor.`**FFdoctor**
Initially sets up your machine.

It ensures the following:

- **you have an ssh-keypair named `YOURHOSTNAME_rsa` and `YOURHOSTNAME_rsa.pub`.**
  - used for write access to git(-hub)
- **you have an script called `ffctl` in your `~/bin` directory.**
  - the ffctl commandline: use `ffctl setup` anywhere to run the setup again
  - this folder should be in your $PATH environement variable.
- **you have some scripts called `batctlMESH` in your `/usr/local/sbin/` directory.**
  - comfort shortcuts for batctl: use `batctlwi o` instead of `batctl -m wiBAT o`

ffbobby_install()
Copies gluon_builder.sh into your ~/bin folder.

Note: This function is intended for Build-Servers only, not for Gateway-Servers!!1!

---

ffecdsakeysetup()
Sets up ecdsa keys on your machine. This is done by the following steps:

•Generates a new ecdsa private-key if it's missing

•Saves the ecdsa public-key next to it

Note: This function is intended for Build-Servers only, no Gateway-Servers!!1!

ffgituser()
Sets the git user and git email accordingly

ffpipreqs()
Installs required python3 dependencies using pip

ffscripts_install()
Generates some helper scripts for ffctl and moves them into your ~/bin folder.

Helper scripts for batctl are written to /usr/local/sbin/.

Both directories should reside in your `$PATH`

ffsshkeysetup()
Sets up ssh keys on your machine. This is done by the following steps:

•Generate a new SSH-keypair (or complement it if the public-key is missing).

•**Appends an entry for GitHub into your SSH configuration file (`~/.ssh/config`).**

– A backup will be made beforehand!

•**Tries to connect with your shiny new key.**

– This ensures that you **will** verify the fingerprint and accept it by typing in `yes`.

GitHub will then be in your `known_hosts` file.

This class is intended to be run once or after changes are made.

Note: You can now use this URL scheme for read/write access to your git repos: `ssh://github_ffctl/...`

Note: Do not forget to add the public-key to your GitHub profile

run(*par=None*)
Runs everything - ssh keys and configuration, ecdsa keys, generating ffctl and batctl helpers, moving them

Parameters

- **par.ecdsa** – setup ecdsa keys also (requires **ecdsakeygen**)
- **par.bob** – setup gluon_builder and ecdsa keys

5.1.5 ffdraw.py

class ffdraw.**FFdraw**
Generates a small website with traffic statistics.

This is done by using vnstat (and *vnstati*). Please configure them correctly beforehand.

run(*par=None*)
Actually runs everything - drawing network statistics and building a status site

5.1.6 ffbuilder.py

class ffbuilder.**FFbuilder**(*verbose=True*)
Helps bobby to build a batch of gluon images for each specified community.

It offers a pre- (prepare all sources, prepare site, setup stage) and postbuild (sorting finished builds into the library) functionality for Bobby. Further, Bobby calles FFbuilder serveral times to receive configuration settings from FFctl.

Second: It offers functionality to declare finished builds in your library as **experimental**, **beta**, and **stable**, setting the according symlinks for release.

_FFbuilder__declare(*decl*, *stagedir*)
Locates specified images in the library, and sets relative symlinks in the serverdir correctly onto it, also seeting manifest symlinks for branches.

We use rsync to distribute the whole directory to the gates from the buildserver thus the nodes can run autoupdate successfuly.

..note:: Do not build on the Gateways! Although it is possible to do so, sometimes you may want to use different versions of a package. Use at least two several virtual machines.

Parameters

- **decl** – Status to declare (**experimental**, **beta**, **stable**)
- **stagedir** – Path to your library entry with images to declare

_FFbuilder__get_gluon_ident()
Get the Version-String for Gluon. Will be used in the filename of gluon images, library folders and other meta files generated while building.

Note: calling this function before running __prepare() results in an incomplete Version String

---

_FFbuilder__load_buildconfig()
Populates the buildconfig by copying the values from FFctl's configuration.

The buildconfig is vital, the values there will be used inside FFbuilder and passed over to bobby.

_FFbuilder__postbuild()
Sorts fresh builds into the library

Will be called from within Bobby after the build for each community was successful.

It counts the Number of Images (sysupgrade needs three files more than the factory folder).

Moving the buildstage into a subfolder of your library named after the Gluon release string, copying all images from the built communities creates a so called *endlager* per community.

_FFbuilder__prepare()
Ensure everything is ready, so Bobby can start his job:

•Creates a stage (subfolder of *serverdir*, so it is publicly viewable if a build is currently running) and a new build directory.

•Deletes old files if any

Per community it does:

•Check out fresh Gluon sources

•**Check out the siteconf**

– for `builder_siteswitchcommunity` the siteconf is switched using `builder_siteswitch` command.

This function gets called within `gluon_builder.sh` as one of the first.

_FFbuilder__sign_release(*stagedir*, *signkey*)
TODO

_FFbuilder__trimanifest(*mfile*)

run(*par=None*)
Actually runs everything - Helps Bobby (`gluon_builder.sh`) getting values from the buildconfig, as well as helping FFctl commandline to declare and sign Gluon releases in your library.

Parameters

- **par.prepare** – Pre-build
- **par.postbuild** – Post-build
- **par.metalog** – Send log message to meta
- **par.metaimport** – Import json file into meta
- **par.declare_experimental** – Path into library for Gluon images to declare and release as experimental
- **par.declare_beta** – Path into library for Gluon images to declare and release as beta
- **par.declare_stable** – Path into library for Gluon images to declare and release as stable
- **par.sign_release** – Path into library for Gluon images to sign
- **par.key** – Use this key to sign Gluon images

5.1.7 ffping.py

class `ffping.`**FFping**
Checks if the gateway is still a gateway through it's OpenVPN connection.

First a list of hosts will be pinged, and if a minimum of *ping_percreq %* is reachable it adds the gateway flag to B.A.T.M.A.N., otherwise removes it.

Our gateways each are serving two mesh networks at once (one for Mainz, one for Wiesbaden), so everything except the pings will be done twice.

checkbatman(*interface*)
Retrieves the current gateway flag state of B.A.T.M.A.N.

Parameters **interface** – B.A.T.M.A.N. interface to use

Return type string with output from B.A.T.M.A.N. (either "off" or "server 54MBit/54MBit" (or similar) or error)

ping(*interface*)
Pings a list of host (defined in *ping_hosts*) and watches the results.

It counts the number of reachable hosts and will give you a proposal if you should set or remove the gateway flag.

Return type boolean (**True** if server flag should be set, **False** if server flag should be removed)

run(*par=None*)
Actually runs everything - sends pings and adds or removes the gateway flag for B.A.T.M.A.N. depending on the ping result

Parameters

- **par.batserverbw** – use custom batserver bandwith setting (e.g `ffctl ping --batserverbw "54Mbit/64Mbit"`)
- **par.batif** – use custom batman interface (e.g "wiBAT")
- **par.exitif** – use custom exit interface (e.g "exitVPN")

5.1.8 ffpeersync.py

class `ffpeersync.`**FFpeersync**
Synchronizes the **peers folder** of your **fastd** instances, sending a `SIGHUP` on changes to tell **fastd** to reload it's configuration and keys.

Our gateways each are serving two mesh networks at once (one for Mainz, one for Wiesbaden), so everything will be done twice.

Functionality purely relies on methods from *util.ffgit.FFgit*, the main part is a *util.ffgit.FFgit.ffgit_setup()* followed by a *util.ffgit.FFgit.ffgit_publish()*

run(*par=None*)
Actually runs everything - [syncpeers(x) for x in [mainz, wiesbaden]]

Parameters **par.meshif** – use custom mesh interface (e.g "wiVPN")

5.1.9 ffrsync.py

class `ffrsync.`**FFrsync**
Calls the correct rsync command for Gluon Image mirroring on the Gateways

run(*par=None*)
Actually runs everything - checking for output directory and rsync

- genindex
- modindex
- search

f

u

Symbols

B

C

D

E

F

G

H

I

J

K

L

M

P

R

S

T

U

W