clans Documentation

Release 0.3.0

baldwint

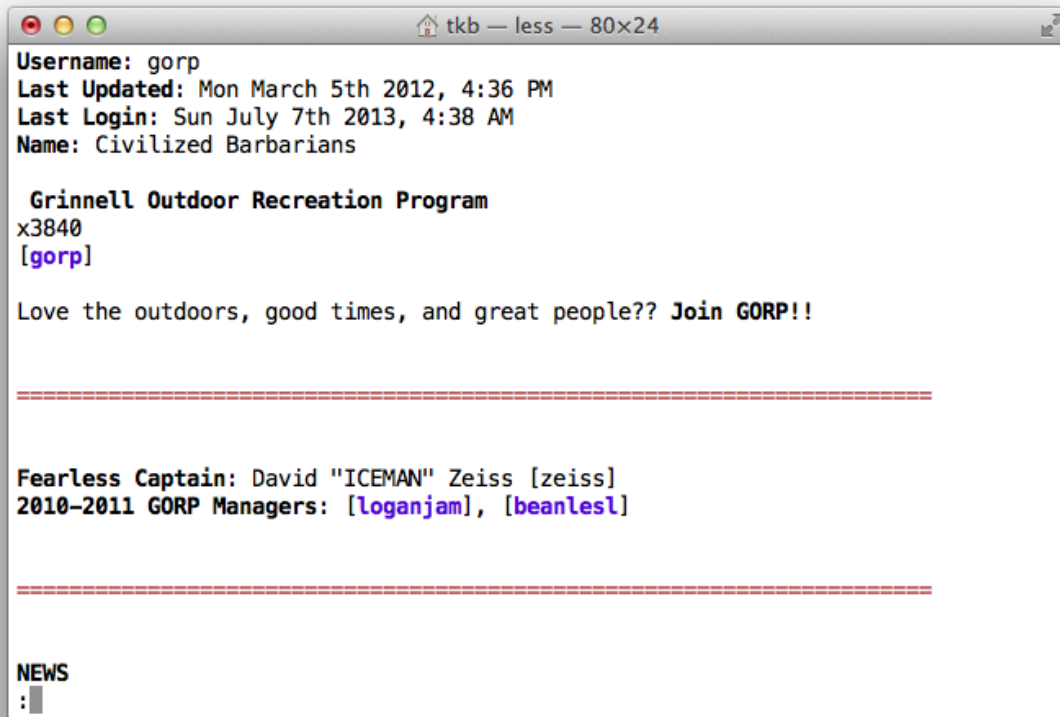June 15, 2016

Clans is a command-line client for the GrinnellPlans social network.

```
●  ●  ●                    ⌂ tkb — less — 80×24                        ⬈
Username: gorp
Last Updated: Mon March 5th 2012, 4:36 PM
Last Login: Sun July 7th 2013, 4:38 AM
Name: Civilized Barbarians

 Grinnell Outdoor Recreation Program
x3840
[gorp]

Love the outdoors, good times, and great people?? Join GORP!!


=====================================================================


Fearless Captain: David "ICEMAN" Zeiss [zeiss]
2010-2011 GORP Managers: [loganjam], [beanlesl]


=====================================================================


NEWS
:
```

Read the [gorp] plan:

```
$ clans read gorp
```

Check quick love:

```
$ clans love
```

Edit your plan in $EDITOR:

```
$ clans edit
```

Not only does clans offer an alternative interface to Plans, but it is also a useful tool for:

- automatically backing up your plan
- scheduling a plan update for a later time
- emailing yourself when new planlove arrives

and much more. No Limits™!

Contents

1.1 Installation

To install clans, you'll need the following:

- A Unix-like operating system (e.g. Linux or Mac OS X)

- Python 2.6+ or 3.3+ (usually preinstalled)

- The pip installer

In addition, clans will only work with Plans accounts that are set to use the **postmodern** interface.

1.1.1 Stable version

Most people will want to use the latest stable release. This installs clans and its dependencies:

```
$ pip install clans
```

If a newer version is available later, update to it with:

```
$ pip install --upgrade clans
```

To uninstall:

```
$ pip uninstall clans
```

1.1.2 Development version

Clans development is versioned using Git. To clone the repository and install it in a single step:

```
$ pip install -e git+https://github.com/baldwint/clans.git#egg=clans
```

This installs clans in *editable* mode - it clones the repository into your `src` directory and configures Python to load it from there.

It is a good idea to work inside a virtualenv to keep things separate from stable versions of clans on the same machine. I use the virtualenvwrapper tool to do that. Using this, I would first do

```
$ mkvirtualenv clans
```

and then do the installation step. Then the repository would be cloned into `~/.virtualenvs/clans/src/clans`, but the installation is only active if I first activate the virtualenv using `workon clans`.

As an optional step, install extra dependencies for testing and documentation:

```
$ cd ~/.virtualenvs/clans/src/clans
$ pip install -e .[docs,tests]
```

Getting updates and sharing improvements

To get updates, `cd` to the repository and do:

```
$ git pull
```

You can make your own modifications to clans by editing the Python source code in the repository. If you like, you can commit your changes using Git and contribute them back to the project.

The first step is to publish your modifications. To do this, fork the project on GitHub and add it as a remote in your local copy:

```
$ git remote add myfork https://github.com/your_username/clans.git
```

Now you can publish changes you made locally using `git push myfork master` (although it is often a good idea to work in branches other than `master`). To submit your changes for review, open a pull request on GitHub.

1.2 Usage

To get an overview of available clans commands, run:

```
$ clans --help
```

To get help on a specific subcommand, like `edit`, run:

```
$ clans edit --help
```

This will list all available arguments and option flags.

1.2.1 Logging in

All commands share several option flags related to authentication with the GrinnellPlans server:

> **-u USERNAME, --username USERNAME**   GrinnellPlans username, no brackets.
>
> **-p PASSWORD, --password PASSWORD**   GrinnellPlans password. Omit for secure entry.
>
> **--logout**   Log out before quitting.

By default, you must specify your username with `-u` for every `clans` incantation:

```
$ clans -u <username> read <planname>
```

For example, to log in as user [baldwint], and read the [gorp] plan:

```
$ clans -u baldwint read gorp
```

This can be avoided by *setting a default username* in clans.cfg.

Clans stores active authentications like a browser does a cookie, so it is *not* necessary to specify `--password` each time. In fact, it is a good idea to omit this flag as a rule. If your password is required, you will be prompted for it.

Note: Clans remembers active authentications, but will only use them if `--username` is specified on the command line, or a default username has been set in clans.cfg. This permits having multiple concurrent Plans logins.

Authentications generally expire on the server side after two days of inactivity, unless `--logout` is given, in which case the authentication token will be deleted immediately after the command completes.

In addition, all commands accept `--help` and `--version` options.

1.2.2 Reading Plans and Autoread Lists

To see what's new on your autoread list:

```
$ clans list
```

This returns a list of plans on your autoread lists that have been updated since you last read them.

Note: Unfortunately clans does not currently know how to manage your autoread lists by adding/removing plans to it. This is coming in a future revision.

To read a plan, use the `read` subcommand:

```
$ clans read <planname>
```

This displays the contents of the specified plan in a pager application in HTML format. It's normally easier to read plain text, though:

```
$ clans read <planname> --format text
```

This formats the plan as plain text before displaying it. Run `clans read --help` for a list of available formatters. You can *configure a default formatter* in clans.cfg.

1.2.3 Searching Plans and Quicklove

To search plans, use:

```
$ clans search <term>
```

This returns a lists of plans containing the search term, and a little context. To restrict search to a planlove, use the `--love` flag:

```
$ clans search --love <planname>
```

Searching for love of your own username ("quicklove") gets a shortcut:

```
$ clans love
```

1.2.4 Editing Your Plan

To edit your own plan:

```
$ clans edit
```

This opens your plan for editing in a text editor. Clans decides which editor to use based on the following:

1. The `editor` value configured in the `[clans]` section of `clans.cfg`

2. Failing that, the value of the `$EDITOR` environment variable

3. Failing that, `pico`.

To submit your update, save and close the file. To cancel the update, quit from the editor without saving.

As an alternative to interactively editing your plan, you can use the `--from-file` option to use a text file as input:

```
$ clans edit --from-file <filename>
```

This replaces your *entire* plan with the contents of the specified text file. It will not prompt for confirmation, so use this option with caution!

1.2.5 Planwatch

To view a list of recently updated plans, use:

```
$ clans watch
```

By default, this displays a list of every plan updated in the last 12 hours. For a fresher list, you could do

```
$ clans watch 2
```

and only plans updated in the last 2 hours will be displayed.

1.3 Configuration

Clans stores configuration information and other data (login cookies, newlove read state, etc.) in its *profile directory*. The path to this directory is reported by:

```
$ clans config --dir
```

By default, this is a folder inside your operating system's designated place for application data, but it can also be set using the `$CLANS_DIR` environment variable.

Persistent configuration is set in a file called `clans.cfg`, in the clans profile directory. You can go directly to editing the configuration file with:

```
$ clans config
```

`clans.cfg` follows the ConfigParser syntax: essentially, it consists of sections, each led by a `[section]` header and followed by `name:  value` or `name=value` entries.

1.3.1 Getting started

You will probably want to set at least two values in the configuration file:

- your username

- your preferred output format

To set your username, create the `[login]` section and add a `username` entry:

```
[login]
username=baldwint
```

With this value set, I will no longer have to specify `-u baldwint` every time I use clans. I'm also accustomed to passing `--format color` when I read plans. I can avoid passing this every time by setting `format=color` in the `[clans]` section. I add the following:

```
[clans]
format=color
```

Now clans will always make colorized output, unless I specify otherwise.

1.3.2 By section

The `[login]` section sets options to do with authentication. The following configuration options may be set:

> **username** sets a default value for the `--username` flag, if it is not specified.
>
> **url** sets the location of the Plans service to use for login. Defaults to `https://www.grinnellplans.com`.

The `[clans]` section controls how the command-line client behaves.

> **format** sets a default value for the `--format` flag, if it is not specified.
>
> **editor** sets which editor to use when editing your plan, in case you want to use one other than is set by the `EDITOR` environment variable.
>
> **timezone** timezone to use for displaying dates and times, specified as its name in the Olson tz database. Defaults to your local timezone (for text output) or UTC (for JSON output).
>
> **date_format** format string for dates and times, specified in the Unicode style. JSON output ignores this option and will always use the ISO 8601 format.

1.4 Extensions

Clans has a hook-based extension framework for adding features on the client side. Several extensions are already built in, and can be enabled by editing `clans.cfg`.

To enable a built-in extension, such as `newlove`, edit the `[extensions]` section of the configuration file with a line like:

```
[extensions]
newlove=
```

For information on a specific extension, see below.

1.4.1 Newlove

The `newlove` extension tracks the read and unread state of your planlove, much like Ian Young's greasemonkey script of the same name. This allows you to easily see what's new in your quicklove.

To enable this extension, add to `clans.cfg`:

```
[extensions]
newlove=
```

With this line enabled, three new flags are added to `clans love`:

> **-t, --time**   Order results by time first seen.
>
> **-n, --new**   Only show new results.
>
> **--keep-unread**   Preserve read state of any new results.

Now, `clans love -n` behaves roughly like the greasemonkey script: You will only see context snippets that have changed since the last time you checked. Alternately, `clans love -t` will present all past snippets in chronological order.

Keep in mind that this extension doesn't know when planlove was *given*, only when you first *received* the love. By default, `newlove` marks your planlove as read every time you do `clans love`, even if neither of the newlove flags (`-n` and `-t`) is passed. To prevent this, pass `--keep-unread`.

Your planlove read state is stored in a JSON-formatted file called `username.love`, in the clans profile directory. When love is deleted from plans, it is also removed from this file.

Newlove for stalkers

By default, the newlove extension only tracks planlove for the logged-in user, but it can be configured to track the planlove of others, as well as the results of non-planlove searches.

To specify users to track newlove for, set the `log_love` value in the `[newlove]` part of `clans.cfg`. Format it as a comma-separated list:

```
[newlove]
log_love=baldwint,gorp,climb
```

This overrides the default behavior (of tracking your own planlove only), so make sure this list includes yourself.

To track everyone's planlove, leave `log_love` blank:

```
[newlove]
log_love=
```

Non-planlove searches can be tracked by specifying `log_search` in the same way.

1.4.2 Backup

The `backup` extension adds flags to the `clans edit` command to facilitate making local backups whenever you edit your plan. If your edit fails, or the plan truncation troll pays a visit to your plan, you may be able to recover your own lost data.

To enable this extension, add to `clans.cfg`:

```
[extensions]
backup=
```

With this line enabled, three new flags are added to `clans edit`:

> **-b FILE, --backup FILE**   Backup existing plan to file before editing. To print to stdout, omit filename.
>
> **-s FILE, --save FILE**   Save a local copy of edited plan before submitting.

--skip-update   Don't update the plan or open it for editing.

There are two points at which a backup may be made: before and after you make your edits. To backup your plan as it existed on the server prior to your editing it, use `-b`. To backup your plan as it existed in your text editor before submitting, use `-s`. It doesn't hurt to use both.

Both flags take a filename argument for the backed-up plan. In the case of `-b`, you can omit this and the plan will be piped to standard output - but depending on your operating system, this might not preserve character encodings very well.

To avoid specifying `-b` and `-s` flags all the time, add to `clans.cfg`:

```
[backup]
backup_file=/path/to/plan_backup.txt
save_edit=/path/to/edited_plan.txt
```

and your plan will be backed up to these files every time you edit. Keep in mind that these files will only store the most recent copy of your plan. To keep editions going back several edits, you will need to backup the backup with some other software. My computer regularly backs up my home folder, so I put them in there and they get backed up with everything else.

The `--skip-update` flag forces `clans edit` to quit before opening an interactive editor. When used in combination with `-b`, this is useful for automating your plan backups:

```
$ clans edit --skip-update -b [FILE]
```

is an idiom for grabbing your current edit field text.

1.5 Internals

This section covers technical details of how clans works, in more detail than you probably want to know. It might be useful if you are interested in contributing to the clans source, developing an extension, or writing some other application that will use clans as a library.

1.5.1 Extension Hooks

Clans' extension framework is based on *hooks* - named points in the execution of various commands where it stops to call other code. Extensions are Python modules that define functions with names matching one or more of these hooks.

Hooks all accept clans' controller object, `ClansSession`, as the first argument. Most are also passed a number of other arguments, depending on the context. For example, the `post_search` hook is passed the `ClansSession` as well as the `results` of that search.

To enable a clans extension that you write yourself, make sure the module is on Python's module search path, so that it's importable via `import my_clans_extension`. Then, in `clans.cfg`, add it to the `[extensions]` section:

```
[extensions]
myext=my_clans_extension
```

On the left side of the equal sign is a casual name for your extension, and the right should be its importable name, using Python dot-syntax if necessary.

Some extensions are packaged with clans, like `clans.ext.newlove`, and for these it is unnecessary to specify the full importable path. It is worth looking at the included `clans.ext.example` extension to get an idea of how to write your own.

> **Warning:** I think I've implemented this in a moderately intelligent way, but the hook API should not be considered stable prior to clans 1.0.

List of Hooks

Each hook accepts one or more (usually mutable) arguments, and need not return anything. Typically, arguments can be modified in-place or left untouched.

The first argument passed is always the `ClansSession` object.

post_get_edit_text(*cs*, *plan_text*)

> This hook is called during plan editing, after the edit text has been retrieved from the server.
>
> The edit text is passed as an immutable unicode string as the second argument.
>
> If this hook returns any value other than `None`, clans will skip interactive editing.

post_load_commands(*cs*)

> This hook is called right after the standard commands and arguments are defined.
>
> This hook is a good place to add arguments or subcommands to the command table, which you can do by modifying the `commands` attribute of `cs`.
>
> For example, to add an argument to an existing command:

```
cs.commands['love'].add_argument(
        '-t', '--time', dest='time',
        action='store_true', default=False,
        help="Order results by time first seen.")
```

> or, to add a whole new command:

```
cs.commands.add_command(
        'secrets', secrets, parents=[global_parser],
        description='Glimpse into the souls of others.',
        help='View secrets.')
```

> where `secrets` is a function you define elsewhere in your extension.

post_search(*cs*, *results*)

> This hook is called after a (quicklove or regular) search, and is passed a list containing the results.
>
> Elements of this list are 3-tuples:
>
> > •the name of the plan on which the term was found (str)
> >
> > •the number of instances found (int)
> >
> > •a list of snippets.
>
> Note that the snippet list may not be the same length as the number of instances found.
>
> Lists are mutable, so results may be filtered by modifying this list in-place.

pre_search(*cs*, *term*, *planlove*)

> This hook is called before a (quicklove or regular) search, and is passed the same arguments as is the search function:
>
> > •the search `term`
> >
> > •`planlove`, a boolean of whether to restrict search to planlove.

pre_set_edit_text (*cs*, *edited*)

> This hook is called during plan editing, after the edit text has been modified, but before being submitted to the server.

> The modified edit text is passed as an immutable unicode string as the second argument.

1.5.2 Plans ScrAPI

Plans does not have a complete API, so clans utilizes a Plans-specific scraping library to communicate with the Plans server. This is packaged as a separate sub-module so that it can be used in other Python programs independent of clans.

> **Warning:** Code changes on the server side could break the scrAPI at any time, so it should not be considered in any way stable.

The entire thing is built around one class, `PlansConnection`. Additionally there is the `PlansError` exception. They can be imported like so:

```python
from clans.scraper import PlansConnection, PlansError
```

Then we can instantiate a `PlansConnection`, log into Plans, and begin doing things:

```python
pc = PlansConnection()
pc.plans_login('baldwint', 'not_my_password_lol')
pc.read_plan('gorp')
```

Method summary

class **PlansConnection** (*cookiejar=None*, *base_url='https://www.grinnellplans.com'*, *server_tz='US/Central'*)

> Encapsulates an active login to plans.

> **get_autofinger** ()
>
> > Retrieve all levels of the autofinger (autoread) list.
>
> > Returns a dictionary where the keys are the group names "Level 1", "Level 2", etc. and the values are a list of usernames waiting to be read.

> **get_edit_text** ()
>
> > Retrieve contents of the edit plan field.
>
> > Returns the edit_text of the plan and its md5 hash, as computed on the server side.

> **plans_login** (*username=''*, *password=''*)
>
> > Log into plans.
>
> > Returns True on success, False on failure. Leave username and password blank to check an existing login.

> **planwatch** (*hours=12*)
>
> > Return plans updated in the last `hours` hours.
>
> > The result is a list of (username, timestamp) 2-tuples.

> **read_plan** (*plan*)
>
> > Retrieve the contents of the specified plan.
>
> > **Returns two objects: the plan header (as a python dictionary)** the plan text (in HTML format)

> **search_plans**(*term*, *planlove=False*)
> > Search plans for the provided `term`.
> >
> > If `planlove` is `True`, `term` is a username, and the search will be for incidences of planlove for that user.
> >
> > returns: list of plans upon which the search term was found. each list element is a 3-tuple:
> >
> > > •plan name
> > >
> > > •number of occurrences of search term on the plan
> > >
> > > •list of plan excerpts giving context
> >
> > the length of the excerpt list may be equal to or less than the number of occurrences of the search term, since overlapping excerpts are consolidated.
>
> **set_edit_text**(*newtext*, *md5*)
> > Update plan with new content.
> >
> > To prevent errors, the server does a hash check on the existing plan before replacing it with the new one. We provide an md5 sum to confirm that yes, we really want to update the plan.
> >
> > Returns info message.

exception PlansError
> Exception raised when there is an error talking to plans.

1.6 Cookbook

In addition to providing an alternate user interface for Plans, clans can be used as a utility to achieve functionality not built into Plans itself. Here are some examples.

1.6.1 New planlove notifications

Rather than checking planlove compulsively throughout the day, we can schedule a cron job that runs `clans love` at a reasonable interval, and emails us when there is something new.

For this we should install clans on a machine that is on all the time, and has a mail server installed. We will need to be able to send mail from the command line using `mail`, which usually works like so:

```
$ echo "Hello world!" | mail -s 'subject' username@grinnell.edu
```

After verifying that I can send email to myself using this method, I install clans on the server and configure `clans.cfg` like so:

```
[login]
username=baldwint
[clans]
format=text
[extensions]
newlove=
```

This enables the `newlove` extension, which filters the output of `clans love` to only show unread planlove when I do:

```
$ clans love -tn
```

The `-n` limits output to new planlove. The `-t` flag is for time-ordering, which we only need because it makes the command's output blank when there is no new love.

I write a script, `lovenotify.sh`, which pipes the output into an email if it's not blank:

```bash
#!/bin/bash

CLANS='/full/path/to/clans'
LOVE=`$CLANS love -tn`;

if [ -n "$LOVE" ]; then
    echo "$LOVE" | mail -s 'new planlove' username@grinnell.edu
fi
```

I make it executable (`chmod +x lovenotify.sh`) and make an entry in my crontab:

```
00 * * * * path/to/lovenotify.sh
```

This will run the script every hour on the hour. If you're less obsessive than me, you might prefer to run it less frequently:

```
00 */3 * * * path/to/lovenotify.sh # every 3 hours
00 */6 * * * path/to/lovenotify.sh # every 6 hours
48 07 * * * path/to/lovenotify.sh # every morning at 7:48
```

Please try not to cook the plans server by hitting your planlove every minute. On the other hand, don't schedule it less often than once per day, since plans will log you out after 2 days of inactivity.

1.6.2 Automated plan backups

With the *backup extension*, clans can be configured to save a local copy of the plan every time we invoke `clans edit`. But it would be nice for this to also back up edits done on the web site, and it would be extra helpful to keep a versioned history of every edit we have ever made. We can achieve this by scheduling another job on the same server we used to run the newlove notifications.

First, I add `backup=` to the `[extensions]` section of `clans.cfg` to enable the extension. Next, I create a folder `plans_backups` in my home directory, which will contain my first plans backup:

```
$ mkdir plans_backups
$ cd plans_backups
$ clans edit --skip-update --b baldwint.txt
```

Now I put the directory under version control. I use git, which is total overkill, but is familiar to me:

```
$ git init
$ git add baldwint.txt
$ git commit -m "initial commit"
```

Finally I schedule a cron job to periodically run the following script:

```bash
#!/bin/bash

CLANS='/full/path/to/clans'

REPO="full/path/to/plans_backups"
BAKFILE="$REPO/baldwint.txt"

$CLANS edit --skip-update -b $BAKFILE
```

```
(cd $REPO && git commit -am "Automated commit `date`" >> /dev/null)
```

This backs up and commits a version of my plan every time it is run. Usually, the plan will not have changed since the last time the script was run, in which case the call to `git commit` will fail. That's expected, so I silence its output by piping to `/dev/null`.

1.6.3 Scheduling a plan update

If you have in mind a hilarious April Fool's day joke to post on your plan, but will be away from the computer on that day, you can prepare it ahead of time and schedule clans to submit it at the proper time.

First copy the contents of your existing plan into a text file. This is straightforward to do with the *backup extension* enabled:

```
$ clans edit --skip-update --backup myplan.txt
```

Now edit and re-save this file so that it includes the desired update. The command we should give to our task scheduler to run on the morning of April 1 is:

```
$ clans edit --from-file myplan.txt
```

We could use `cron` to schedule this, as we did in the previous examples, or some equivalent thereof. I did this on a Mac, using `launchd`, and the following LaunchAgent:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>clans.edit</string>
    <key>ProgramArguments</key>
    <array>
        <string>/Users/tkb/bin/clans</string>
        <string>edit</string>
        <string>--from-file</string>
        <string>/Users/tkb/myplan.txt</string>
    </array>
    <key>StartCalendarInterval</key>
    <dict>
        <key>Day</key>
        <integer>1</integer>
        <key>Hour</key>
        <integer>7</integer>
        <key>Minute</key>
        <integer>48</integer>
    </dict>
</dict>
</plist>
```

This schedules the job to run at 7:48 AM on the 1st of the month. Note that:

- I used the `/full/path/to/clans` and `/full/path/to/myplan.txt`, so the agent can run outside the environment defined by my shell.

- Any change I make to the plan before the job runs will be overwritten when it eventually does.

- This job will actually run on the 1st of *every* month, so I'll need to remember to disable it before the 1st of May.

Loading LaunchAgents by hand is super-cumbersome, so I usually use the Lingon app to schedule them.

1.6.4 Using clans on multitple computers

If you use multiple computers, you can sync clans data between them using a service such as Dropbox.

By default, clans stores its data in its *profile directory*. This contains the `clans.cfg` file as well as other data (login cookies, newlove read state, etc.). By symlinking this directory into your Dropbox, the configuration file and all other data can be shared by your clans installations.

The profile directory location is reported by `clans config --dir`. Move it, and leave a symlink in its place:

```
$ mv -r "`clans config --dir`" ~/Dropbox/clansdata
$ ln -s ~/Dropbox/clansdata "`clans config --dir`"
```

Then repeat the second step on any synced computer with which you would like to share settings.

> **Warning:** Anyone with read access to the clans data directory may be able to log into plans as you. For this reason, it has 700 permissions by default, but *Dropbox does not sync this*.
> It is a good idea to remain logged out until you can do:
>
> ```
> chmod 700 ~/Dropbox/clansdata
> ```
>
> on all computers synced by your Dropbox. Consider using selective sync to limit which computers your login token is stored on.

1.6.5 Using an alternate Plans server

By default, clans communicates with the installation of Plans running at https://www.grinnellplans.com/. It can also talk to other installations, such as one running on your local development server.

The `url` setting in the `[login]` section of `clans.cfg` can be used to change which Plans we are talking to. However, switching this back and forth can have unexpected consequences (for example, when using the newlove extension, it will erase my read state).

It is better to create an entirely separate profile directory, and use the `CLANS_DIR` environment variable to control which one clans uses.

```
$ mkdir localhost.clansprofile
$ nano localhost.clansprofile/clans.cfg
```

You can name this directory whatever you want (It doesn't have to have a `.clansprofile` extension, but this helps me remember what it is). In this new `clans.cfg` file, define the location of the development server and whatever other settings you want to use:

```
[login]
username=baldwint
url=http://localhost/~tkb/plans/
```

Then, to switch between profiles, do

```
$ export CLANS_DIR=path/to/localhost.clansprofile
```

To switch back to the default profile:

```
$ export CLANS_DIR=
```

C

C

G

P

R

S