FrÄndz

(FrÄndz ver. 0.5)

What it is, What it can, How it works

[http://fraendz.sourceforge.net]

Matthias Ihrke[*]

April 28, 2005

Abstract

This paper includes a short, technical introduction to the FrÄndz online system. Implementation details are considered.

[*]E-Mail: mihrke@uni-goettingen.de

Contents

1 What is FrÄndz?

FrÄndz (pronounced like “friends”) is an intimate online meeting-place for good friends. FrÄndz runs with minimal dependencies (only standard python modules are used; no other libraries etc. required). It features a secure user management and a flexible configuration mechanism.
Features in the current version are:

- a username/password based authentification (very secure when run over the https protocol),
- user registration with personal picture,
- an internal messaging system,
- a discussion functionality,
- a news functionality,
- an address book,
- a chat,
- a skinnable layout (i.e. can be changed by every user for himself),
- RSS feeds for new messages (every user), entries in the forum, logbook entries (for admin) and news-entries,
- and internal user-homepages.

The System is implemented 100% in Python[1].
Since FrÄndz operates on plain files and directories, it is probably not suitable for a huge amount of users. It has been tested with about 20 users. However, it is possible to reimplement the data-access functions (exclusively located in the security-module (`fraendz.security`) and the userio-module (`fraendz.userio`) to access a SQLite[2]-database via the PySQL module to allow a higher number of users. In the current version, this has been omitted to keep the number of dependencies very low.
An in-use version of the system information on FrÄndZ as well as a project page for the system covering the API-documentation and news concerning the system can be found under the FrÄndZ-Homepage: http://fraendz.sourceforge.net.
A CVStrac[3]-page with the neweset CVS snapshot and additional information can be found under can be accessed under this URL, too.

[1]http://www.python.org
[2]http://www.sqlite.org
[3]http://www.cvstrac.org

2 Installation

To install FrÄndz, you will need the following:

- a web-server,
- CGI-support on the webserver,
- a Python-Interpreter on the webserver,
- optionally, if you want to enable RSS feeds for the users, you will need to run the Apache Webserver with the htpasswd utility installed.

An installation script is provided under fraendz/tools. It is highly recommended to use this script, because all file and directory permissions will be set conveniently. However, manual instructions will be provided as well.

2.1 Script-based Installation

For a fast and easy installation, simply run the install script, located under the `tools`-section of your copy:

```
tar xvfz fraendz05.tar.gz
cd fraendz05/tools
python install.py
```

This will start the interactive installation production, which was written to be pretty self-explanatory. However, it might be helpful to know, that you have to provide three directories for the script: one outside the scope of the webserver where libraries and user information will be stored, one inside the scope of the web server for static html-pages and images and a cgi-bin directory.
If you should encounter problems after the installation, run the script

```
python setpermissions.py
```

from your newly installed `fraendz/tools` directory.

2.2 Manual Installation

If you encounter problems during the automatic installation, refer to this section as well as section 3 on page 6 for details of the structure of the FrÄndz-package.
This is basically what the script does

1. Create a directory *outside* the scope of the webserver and copy the `doc`, `documents`, `templates`, `fraendz` and `tools` part of the package to this new directory (e.g. if HTML documents on the server lie under `~/public_html/`, create the directory `~/fraendz`).
2. Create a directory *in* the scope of the webserver and copy the contents of `fraendz/htdocs` to this directory (e.g. create `~public_html/fraendz`)

3. Create a directory in the `cgi-bin` directory of your webserver (e.g. `~public_html/cgi-bin`) and copy everything from `fraendz/front-end` to this directory (e.g. create `~public_html/cgi-bin/fraendz`).

4. If RSS is enabled (variable in `config.py` set to 1), copy the `rssfeeds` to the appropriate locations.

5. create world writable directories `fraendz/users` and `fraendz/users/admin`.

6. Open the the config-module in your installation (e.g. `~fraendz/lib/config.py`).

   a) Have a look at the variables (all in upper case) and adapt them according to your configuration.

   b) It is probably sufficient to adapt the following variables (see listing 1:

Listing 1: fraendz.config

```
SYSTEM_ROOT='/Users/thias/web/internal/'
# cgi-bin root
CGI_ROOT=SYSTEM_ROOT+'htdocs/cgi-bin/'
# library
LIB_DIR=SYSTEM_ROOT+'lib/'
# image directory
PIC_ROOT=SYSTEM_ROOT+'pics/'
# user-directories
USER_DIRS=SYSTEM_ROOT+'users/'
# central user-control file
USER_FILE=SYSTEM_ROOT+'.users'
# template directory
TEMPLATE_DIR=SYSTEM_ROOT+'templates/'

## here all locations for web-access
WEB_ROOT='/internal/'
# cgi-bin root
WEB_CGI_ROOT='/internal-bin/'
# image directory
WEB_PIC_ROOT=WEB_ROOT+'pics/'
```

7. to finish the installation, the fraendz-modules located in the `fraendz/fraendz` directory must be made accessible to the front-end scripts. There are generally two possibilities to do this,

   a) you find some way of storing the fraendz/fraendz folder in a standard python-module position,

   b) building a file called "path" in your fraendz-cgi directory, that includes information about the location of the module (downside: possible security issue, as this file is world-readable and available to the webserver)

3 Structure of the FrÄndz-package

The structure of a FrAendZ package is as follows (assumed by the installation script)

```
fraendz[ver]
 |
 ---- doc - documentation
 |
 ---- documents - some static documents
 |
 ---- fraendz - the python modules!
 |
 ---- front-end - the front-end cgi's (without rss)
 |
 ---- htdocs - static html's and pics
 |
 ---- rssfeeds - cgi's for the RSS support
 |
 ---- templates - tpl's (skins)
 |
 ---- tools - installation and administration tools
```

The fraendz system is divided into front-end scripts (the actual cgi-scripts called by the webserver) and underlying library scripts in `fraendz/fraendz`. The front-end scripts basically call functions from the library modules to create the dynamic webpages for the user.
This is a list of currently available front-end scripts and their function:

User-functionality

`adminlogin.py`	prepares the login field for adminable users
`adressbook.py`	shows an addressbook of all currently registered users
`chat.py`	wrapper for the chat Java-Applet; currently not used (see htmlchat.py)
`forum.py`	provides the forum functionality
`homepage.py`	shows user homepages and allows for editing of the user's own homepage
`htmlchat.py`	the currently used chat-client; an inline HTML object with automatic refresh is used (CAUTION: not supported by all browsers!)
`message.py`	provides the internal-messaging functionality
`news.py`	provides the news-system
`portal.py`	startpage after login and navigation portal
`register.py`	possibility to change user's login information (and password)
`showhtml.py`	wrapper to display Webpages that lie outside the scope of the webserver
`showimg.py`	wrapper to display images that lie outside the scope of the webserver

Admin-functionality

`adminportal.py`	start and navigation page for adminable users after login as admin
`chat_management.py`	management of the chat (not yet implemented!)
`forum_management.py`	management of the forum (not yet implemented!)
`news_management.py`	management of the news system (not yet implemented!)
`parameter.py`	displays a list of all defined FrÄndz-variables by parsing the source code at runtime
`run.py`	lists all scripts and extracts the possible parameters and offers a webinterface to run these scripts with arbitrary parameter values (very useful for testing purposes)
`showscript.py`	displays the source code of one of the scripts
`usermanagement.py`	useradd, userdel and userkick from the webinterface

RSS-feeds

`logfilefeed.py`	displays the last few entries in the logfile as RSS feed
`messagefeed.py`	displays user's messages in the inbox as RSS feed
`forumfeed.py`	displays the last entries made to the forum as RSS feed
`newsfeed.py`	displays the latest news in the system as RSS feed

The underlying modules will be discussed in the next sections.

4 Implementation

I tried to implement the system in a way to allow a topmost flexibility. All major functions (such as access to stored data, security related functions or definitions) are implemented in the modules of the fraendz-package. The design (look and feel) of the system is separated from the implementation by an own template system (see section 4.2). Finally, the scripts read directly on the files and directories, hence allowing a very flexible building of the webpages (simply removing or providing a file changes the page, no database update is required).
In the code, I follow an own naming convention of using upper cases for all global variables (constants).

4.1 Configuration - `fraendz.config`

The module `fraendz.config` includes the general configuration of the system. This module is included in all other modules and must hence not be imported anywhere else.
All absolute declarations (of pathes or ip-addresses) are made here. A change in this file will effect the whole system.

4.2 Templates - `fraendz.template`

For an easy way of changing the layout of the system, I implemented an own way of using templates rather than mixing HTML and Python. Templates are stored in the `fraendz/templates` directory and include the suffix `.tpl`.
Templates are generally pure HTML-code, but can include dynamical elements (variables or other templates) which is indicated by wrapping the upper-case variable name in curly braces and double daggers (e.g. {#USERNAME#}). Each `tpl`-file consists of several template-parts. Each part is surrounded by

```
%%begin{<template-name>}
%%end{<template-name>}
```

tags. When a template should include another template, the syntax looks as follows:

```
{#template::<primary-name>:<part-name>#}
```

where the last part is optionally (if omitted, the default 'main' will be used).
Care should be taken not to define circular inclusions! If one template includes another template which in turn includes the first template, this would result in an infinite recursion!
The design is realized from the front-end script, by calling fraendz.template's function `getTemplate()` (see listing 2).

Listing 2: fraendz.template.getTemplate()

```
def getTemplate(template, s=locals()):
        """
        * new function to provide the template-system
        * same method as in earlier version, but more
          sophisticated
        * e.g. recursive template includings possible
        - see documentation for details
        """
        import os, re
        # default template part is main
        if len(template.split(':')) < 2:
                template += ':main'
        try: file, part = template.split(':')
        except: return TEMPLATE_ERROR

        """
        open and read out template file
        and get the desired part in the template file
        (see documentation for details)
        -- this is a fallback mechanism, if the desired template
           is not found in the skin directory, the default skin's
           template is used
        """
        if s.has_key('SKIN'):
                try:
                        f = open(os.path.join(s['SKIN'], file+'.tpl'), 'r')
                        content = f.read()
                        f.close()
                        partcontent = re.search('%%begin\{'+part+'\}'+\
                                        '(?P<temp>.*)%%end\{'+part+'\}',\
                                        content, re.DOTALL).group('temp')
                except:
                        f=open(os.path.join(TEMPLATE_DIR, file+'.tpl'), 'r')
                        content=f.read()
                        f.close()
                        partcontent = re.search('%%begin\{'+part+'\}'+\
                                        '(?P<temp>.*)%%end\{'+part+'\}',\
                                        content, re.DOTALL).group('temp')
        else:
                try:
                        f=open(os.path.join(TEMPLATE_DIR, file+'.tpl'), 'r')
                        content=f.read()
                        f.close()
                        partcontent = re.search('%%begin\{'+part+'\}'+\
                                        '(?P<temp>.*)%%end\{'+part+'\}',\
                                        content, re.DOTALL).group('temp')
                except:
                        return TEMPLATE_ERROR

        ### substitute variables in partcontent
        found=re.findall('\{\#[A-Za-z1-9\_]+\#\}', partcontent)
        for f in found:
                try:
                        # call eval with dictionaries, defining the scope
                        eval(f[2:-2], s, globals())
                except:
                        continue
                partcontent=re.sub(f, str(eval(f[2:-2], s, globals())),\
                                partcontent)

        ### substitute links to other templates
        found=re.findall('\{\#template::[A-Za-z1-9\_\:]+\#\}', partcontent)
        for f in found:
                temp = f[2:-2]
                try: name = temp.split('::')[1]
                except: return TEMPLATE_ERROR
                ### RECURSION
                substitute = getTemplate(name, s)
                partcontent = re.sub(f, substitute, partcontent)

        return partcontent
```

The substitution of the placeholders in the template file is done through regular expressions. The evaluation is done in the scope of a user-supplied dictionary and the `globals()` dictionary. Thus, the variables corresponding to the placeholder in the template file can be specified either in the template module itself, in a module that is imported by the template module or in the front end script that sends in the dictionary to the `getTemplate()`-function (see for example listing 3).

Listing 3: snippet from portal.py

```
scope=defineDefaultScope(user=user, code=code)
### user Adminable??
if userAdminable(user):
        scope['ADMIN_LINE'] = getTemplate('portal:admin', scope)
else: scope['ADMIN_LINE'] = ''

### create SKIN-list
scope['LIST_SKINS']=''
for skin in SKINS:
        scope['LIST_SKINS'] += getTemplate('portal:list_skins',\
                        {'SKIN_NAME':skin})

# print Welcome-message and table of contents...
print getTemplate('portal', scope)
```

All static variables are defined in `template.py`, but since the security system (see section 4.3) demands a variable nature of all links (user authentification must be present, which changes with each login), these must be defined from the front end script by calling the `template.defineDefaultScope()` function.
At the present point I'm not sure anymore that this is a clever way to implement this functionality, as the local variable scope is crowded with unimportant variable-definitions. I think about exporting this function to another file (or maybe even in a database) that is included when actually running `getTemplate()`.

4.3 Security - `fraendz.security`

The security-module (`fraendz.security`) deals with security related issues. Because all potentially sensitive data is stored outside the scope of the webserver, it can only be accessed through a CGI-script that provides a dynamically generated webpage of this information.
There is a central file (`fraendz/.users`) in which username and encrypted password for each user is stored. When a user logs in the system via the login page, the unencrypted password is sent to `portal.py` which immediately encrypts it. Because of this procedure, at least the login procedure must be handled using the `https`-protocol[4].
With each login, a time stamp is set up in the users directory (`fraendz/users/<user>`). This stamp is valid for a predefined period of time (30 minutes by default), which ensures that the pages cannot be accessed even if the user forgets to log out. This stamp is intermingled with the users password an some nonsense letters to produce a code that must be delivered along with the username to each and every script (see listing 4).

[4]In contrast to the http-protocol, the https-protocol encrypts all data before sending it over the internet.

Listing 4: intermingling of password and username in `security.py`

```
def encodePw(user, pw):
        """
        * takes the password and the user stamp and combines it
          to a cryptic code which is used to forward the user
        """
        from random import choice

        stamp = str(getStamp(user))

        code = ''
        # create 5 random letters
        for i in range(RANDOM_LETTERS):
                code += choice(ALPHABET)
        # append the password
        code += pw
        # append the delimiter
        code += DELIMITER
        # append stamp
        code += stamp
        code += DELIMITER
        # more random letters
        for i in range(RANDOM_LETTERS):
                code += choice(ALPHABET)
        return code
```

This code is unique for each user-session.
That means, that before a script sends information to the user, the delivered security information is processed, to check if the user has a valid registration (see listing 5).

Listing 5: standard security checks performed in `security.standardSecurityChecks()`

```
form=cgi.FieldStorage()
print 'Content-type: text/html\n'

### ABORT
# is the script called with the correct parameters?
if not 'stamp' in form.keys() or not 'user' in form.keys():
        printHTMLPart('upper_empty')
        printHTMLPart('illegal')
        printHTMLPart('lower_empty')
        sys.exit()
code = form["stamp"].value
user=form["user"].value

### LOGIN
# is the user registered?
pw, stamp = decodePw(code)
if checkIfUserRegistered(user, pw):
        printHTMLPart('upper_empty')
        printHTMLPart('notregistered')
        printHTMLPart('lower_empty')
        sys.exit()
if checkStamp(user, stamp):
        printHTMLPart('upper_empty')
        printHTMLPart('illegal')
        printHTMLPart('lower_empty')
        sys.exit()
if checkStampTime(user, stamp):
        printHTMLPart('upper_empty')
        printHTMLPart('timeout')
        printHTMLPart('lower_empty')
        sys.exit()
```

However, the storage of the sensitive information outside the scope of the webserver has its downside. It gets more difficult to access complete documents that have been uploaded by the users (e.g. their personal picture). Therefore wrapper scripts (`showimg.py` and `showhtml.py`, see listing 6 for the code for image presentation) are provided that allow the display of this information.

Listing 6: `showimg.py` - displays an image that is stored outside the webservers scope

```
#!/usr/bin/env python
"""
showimg.py - part of fraendz

  * returns an image as to be displayed by the webbrowser
        -> can be accessed by other cgi-scripts via
           <IMG src='showimg.py?user=username&img=imgname'>
  * is necessary, because pics lie in user dirs, which are not
    accessible by normal links

PARAMETERS:
        user, stamp     - standard
        img             - 'imagename.ext'
        whichuser       - 'username'            # whose pic?
"""

import cgi, sys, os.path
from fraendz.security import *
from fraendz.config import *
if DEBUG: import cgitb; cgitb.enable()  # debug

form=cgi.FieldStorage()
# extracts form values
try:
        user = form['user'].value
        img = form['img'].value
        code = form['stamp'].value
except:
        print 'Error - wrong form-values'
        sys.exit()

# security checks
pw, stamp = decodePw(code)
if checkIfUserRegistered(user, pw) or checkStamp(user, stamp)\
                or checkStampTime(user, stamp):
        print 'Error - username/password not correct'
        sys.exit()

# whose picture?
if 'whichuser' in form.keys():
        username = form['whichuser'].value
else:
        username = user

# display the image
root, ext = os.path.splitext(img)
try:
        imgcontent = open(os.path.join(USER_DIRS, username, img), 'rb')
except:
        print 'Error opening of img not possible'
        sys.exit()

# content-line
print 'Content-type: image/%s\n'%ext[1:]
print imgcontent.read()
imgcontent.close()
```

4.4 Storage - `fraendz.userio`

As mentioned before, user data is stored in a directory structure. Each user owns a home-directory under `fraendz/users`, where all information provided by the user is stored. The functions in `userio.py` provide access to this information.
The functions in this module must be primarily reimplemented when switching to a database.

4.5 Chat - `fraendz.chatserver`

The chat server affords somehow the most sophisticated code. The server listens on a given port for incoming TCP connections (`socket.py` is used for the networking) and handles requests from the client. At the moment only three request types are implemented, these being

1. POST
2. GET and
3. QUIT.

After the connection between client and server has been established (see listing 7), the server waits for one of the above keywords and delivers or receives the information.

Listing 7: snippet of `chatserver.py` that handles the request after a connection has been established

```
def handler(csocket, s):
        """
        called as a thread for each connection
        """
        global quit
        request = csocket.recv(BUFFER)
        if not request in ALLOWED_REQUESTS:
                s.log("wrong request: %s, closing connection\n"%request)
                csocket.close()
        elif request == 'POST':
                username = csocket.recv(BUFFER)
                if not username in ALLOWED_USERS:
                        s.log("user '%s' not allowed, closing"+\
                                "connection\n"%username)
                        csocket.close()
                else:
                        msg = csocket.recv(BUFFER)
                        s.update(username, msg)
                        csocket.send(CONFIRM)
        elif request == 'GET':
                csocket.sendall(s.getMessages())
        elif request == 'STATUS':
                pass
        elif request == 'QUIT':
                s.log('QUITTING because of user request...')
                quit = 1

        csocket.close()
```

Each connection is handled in a separate thread, so that the server can continue to listen to incoming requests. The requests are stored as socket objects in a queue which is processed one after another (see listing 8).

Listing 8: snippet of `chatserver.py` that shows the threaded processing

```
1  def listen(s):
2      global conList
3      while 1:
4          csocket = s.listen()
5          if csocket:     # connection accepted
6              conList.append(csocket)
7          else: pass
8
9
10 ### main program
11 global conList, quit
12 quit = 0
13 conList = []
14
15 def main():
16 #if __name__ == '__main__':
17     # initialize ChatServer object
18     s = ChatServer()
19     thread.start_new_thread(listen, (s,))
20     while 1:
21         if conList:
22             thread.start_new_thread(handler, (conList.pop(), s))
23         if s.timeout():
24             s.log('TIMEOUT\n')
25             break
26         if quit:
27             break
28     # shutdown server
29     s.close()
```

In line 19, a thread is created that listens at the fixed port for the whole time the server is running (line 1 to 7 implement the `listen()` function). Than the main loop is entered, in which the queue (`conList`) is processed. A thread is created for each item in the queue (line 22).
The chatserver is implemented as a class `ChatServer`, that provides the communication functionality.
Since the chatserver is needed only on rare occasions, the server should only be started, when a user enters the chatroom. Therefore, the client checks with every request if the server is running or not, and if not starts it up (see listing 9).

Listing 9: `startupChatServer()` provided by the security module

```
def startupChatServer(user):
    """
    check if the chat-server is running, if not start it
    """
    import socket, os
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        s.connect((CHAT_SERVER_ADDRESS, CHAT_SERVER_PORT))
    except:
        failure = os.popen(LIB_DIR+'chatserver.py')
        if failure: return 1
    s.send('STATUS')
    s.close()
    return 0
```

In order for this to work, the server must run in background daemon-mode. Line 10 in listing 9 simply executes the `chatserver.py` script. If this happens, the code in listing 10 is executed, starting the chatserver in daemon mode using the `os.fork()` function.

Listing 10: daemon wrapping in `chatserver.py`

```
### daemon-wrapping
if __name__ == '__main__':
        try:
                pid = os.fork()
                if pid > 0:
                        sys.exit(0) # exit first parent
        except OSError, e:
                print >>sys.stderr, "fork #1 failed: %d (%s)" %\
                                (e.errno, e.strerror)
                sys.exit(1)

         # decouple from parent environment
        os.chdir("/"); os.setsid(); os.umask(0)

        # do second fork
        try:
                pid = os.fork()
                if pid > 0:
                        # exit from second parent, print eventual PID before
                #       print "Daemon PID %d" % pid
                        sys.exit(0)
        except OSError, e:
                print >>sys.stderr, "fork #2 failed: %d (%s)" %\
                                (e.errno, e.strerror)
                sys.exit(1)

        main() # start the daemon main loop
```

The server shuts down either when the timeout is reached, or a `QUIT` request is sent by a client.
The relevant parts of the client can be found in listing 11.

Listing 11: `htmlchat.py` - relevant parts

```
startupChatServer(user)

### -- STARTING REAL STUFF
if 'chatfield' in form.keys():
        """
        display the inline-object chatfield
        """

        scope['CHAT_CONTENT'] = getCurrentChatContent(user)#.split('\n')
        printHTMLPart('chat_field', scope)
else:
        """
        display the whole chat-form
        """
        printHTMLPart('upper', scope)
        scope['CHAT_FIELD'] = WEB_CGI_ROOT+'htmlchat.py?user=%s&stamp=%s\
                &chatfield=1'%(user, code)

        if 'post' in form.keys():
                if postChatEntry(user, form['post'].value):
                        print "<H3>Error, did not post your message!</H3>"
        printHTMLPart('chat_wrap', scope)
        printHTMLPart('lower', scope)
```

The client script `htmlclient.py` can be called with a different set of parameters. If the parameter `chatfield` is included, only an inline frame-object is displayed. This part of the script is called repeatedly by a self-refreshing webpage that is provided by the rest of the script. Following this method, only the frame object instead of the complete webpage must be refreshed, thus saving time and providing a more coherent design.

5 Administration - `fraendz.admin`

Since FrÄndz was designed for a relatively small and intimate circle of friends, there is no possibility for users to sign up themselves. Therefore, a webinterface for easy administration is provided.
Once a user has been labeled as "adminable" (using the tool `adminsettings.py`), this user will find a link to the admin-area.

5.1 Tools

In the directory `fraendz/tools`, I provided a couple of tools that should ease the administration of the system. This is a list of all currently available tools and their functions:

`adminsettings.py`	set the administrator password and which users are adminable
`build_fortunefile.py`	build a file of some quotations for the use of `template.fortune()` (the original program is not used as to keep down the dependencies)
`get_variables.py`	prepares a list in latex or HTML of all currently available fraendz-variables by reading out the source code at runtime
`install.py`	installation script (interactive)
`setpermissions.py`	helper to set the permissions
`useradd.py`	add a user
`userdel.py`	delete a user

The tool `useradd.py` provides the possibility of adding a new user, while `userdel.py` does the opposite. When removing users, care should be taken, that if the complete user directory of this user is removed, the complete functioning of the system could be impaired if the user has currently undertaken activities in the system (the forum and the news for example retrieve information about the author from the users directory). In future versions this will be fixed.

6 Future Plans

- move all users-stuff to a SQLite database,
- provide transparent database use, so that either SQLite or MySQL can be used,
- provide an english translation of the templates,
- extend FrÄndZ to an online-service, so that people can “apply” for an own FrÄndZ environment via internet (multiple databases → SQLite!).