StreetSign Documentation

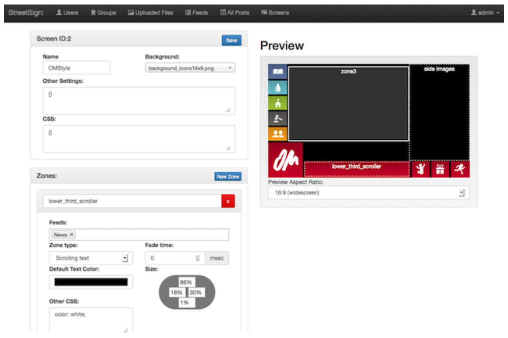
Release 0.5

Daniel Fairhead

May 02, 2017

Contents

StreetSign is a Digital Signage System, build using Python, Flask, and a bunch of *wonderful libraries*, designed to be as simple and friendly as possible.



Screen ID:2

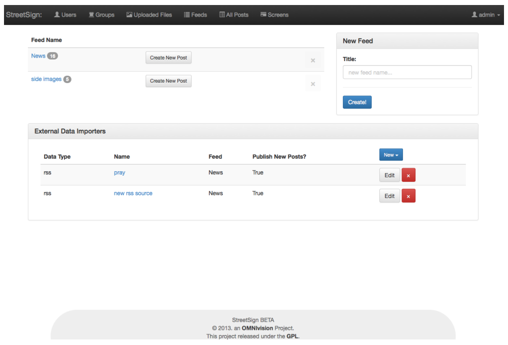
Name: OMStyle
Background: background_icons16x9.png
Other Settings: {}
CSS: {}

Zones:

lower_third_scroller

Feeds: News
Zone type: Scrolling text
Fade time: 0 msec
Default Text Color:
Size: 86% 18% 30% 1%
Other CSS: color: white;

Preview Aspect Ratio: 16:9 (widescreen)



Feed Name		
News 16	Create New Post	×
side images 5	Create New Post	×

New Feed

Title: new feed name...

External Data Importers

Data Type	Name	Feed	Publish New Posts?	
rss	pray	News	True	Edit ×
rss	new rss source	News	True	Edit ×

StreetSign BETA
© 2013. an **OMNIvision** Project.
This project released under the **GPL**.

This documentation is a work in progress.

Contents:

CHAPTER 1

Getting Started

Hi! Welcome to StreetSign! I hope you enjoy working with this system as much as I've enjoyed developing it so far.

Installation instructions are in the Project README file. It should be very easy on any linux/OSX or other unixy type computer.

System Requirements

- Python 2.7 is recommended, but 2.6 should work too.
- Python development headerfiles & capable compiler: *yum install python-devel* on CentOS/RPM based distros, *apt-get install python-dev* on Debian/Ubuntu

It's also recommended to install ImageMagick for creating image thumbnails:

- *yum install ImageMagick* (on CentOS/RPM) *apt-get install imagemagick* (on Debian/Ubuntu)

Installing

Essentially, once you've downloaded or cloned the project:

```
./setup.sh
```

Should download and install a virtualenv, and set up all the dependencies and the initial database (a local sqlite file). Then

```
./run.py
```

To actually run a local test server.

To deploy streetsign for production use, check out the *Deploying Streetsign in Production* guide.

The initial administration user 'admin' is created, with a password of 'password'.

If you need to start again with a fresh database, then delete the *database.db* file, and re-run *./setup.sh*.

This rest of this document hopes to make the different concepts and terminology clear, so that you can be up to speed as quickly as possible.

Basic Overview

StreetSign is a Digital Signage system, that makes it easy to display content on screens & computers around a campus.

It works by having a single 'server', which runs a intranet/web interface which you can connect to with any web browser, which lets content authors create items of news which can then be displayed on any screen.

Each physical screen needs a computer (a cheap Raspberry Pi or similar works fine) connected to it, and to the network, which then connects to the server, and displays the content in organized layouts, which are specified by the designer. These are simple HTML webpages, and simply displayed with a full- screen web browser. (They can also be connected to by smartphones, tablets, or other desktop machines if you so desire.)

Content

These are the basic pieces of content that you work with:

Posts

Individual items of content, be these news items, photos, clocks, whatever, are known as 'posts'. Kind of like on a blog or other news system. There are different types of posts: Pure text posts, Picture posts, and 'Rich' content HTML posts, which can have formatted text, mixing in pictures and so on as well.

Feeds

Posts are grouped into 'Feeds'. A specific post can only be in one feed. This might sound limiting, but don't worry. You can create as many feeds as you want, and assign them specific permissions if you need to.

You can have a "latest photos from the campus" feed, a "URGENT" feed, a "today's joke" feed, a 'urgent things to do' feed, a 'world news from BBC' feed, or whatever.

A feed can have many different types of posts in it, if you want, and you can have multiple feeds display in the same places, or have the same feeds display in different ways on different screens.

External Data Sources

There are also external data sources, which can open RSS feeds (say the news from the BBC, or the latest pictures you put on tumblr ...) and import those as posts into whichever feed you desire.

Posts, when they come in from an external source, can be set to be published automatically, or not. You can also edit the posts once they've come in, or unpublish or delete them. This is really useful when pulling data from sources that you're not in complete control of, such as a news feed, or twitter, when you want editorial control before they go live on screens.

Displaying it all

It's all well and good to have everything organised nicely, but it's only useful if you can put your content where people can see it...

Publish the posts!

Before the posts will show up anywhere, you will need to set them to be *published*, which is done on the Feed page. You can have different users with permission to do different things - so you can have users who can create posts, but not publish them, and then a manager's user who can publish them after approval.

Screens

Full Reference: *Screen Design*

You can design screens, that is, specially taylored output formats for each display, on the 'screens' area of the web interface. Each screen will resize and stretch to try and fit whichever size physical screen you open it on, but for very different shaped screens (say a tall portrait 9:16 display, and a 4:3 point of sales screen) you will want to design different layouts.

You can set up specific screens for certain areas, say a dining room screen, a lounge area screen, a welcome desk one, and so on.

Screen Zones

Each Screen layout has multiple 'zones' on it. A zone has it's own position on the screen, as well as formatting, fonts, etc. You then tell that zone to display the posts from as many different feeds as you wish.

If you want, you could have a screen with a left zone which had photos from two or three different photo feeds, a right zone which showed mainly textual content about the day's events, and occasional safety notices, and a bottom scrolling 'ticker' type zone which has world news, and a top 'Clock' zone, which mainly shows the clock, except when there are urgent announcements, when it displays those as well.

Dividing content

Because you can select multiple feeds to appear in the same zone, and the same feed can appear on as many screens and zones as you like, you can easily have announcements specific to certain areas, and general ones. So you can have a 'site-announcents' feed, a 'bookstore offers' feed and a 'how much do drinks cost' feed, say, which are all displayed in the foyer, but the book store screen only shows the books and site-wide ones, and the coffee bar only shows the drinks ones, say.

Timing things

It's often convinent, especially at conferences, to have announcments which appear only at certain times of day, such as "what's on next", or "which band is playing in which venue this evening".

Post Lifetime

Each post has a lifetime, which defines when you want it to start appearing on screens. So before a conference, you can set up Post items which give the day's schedule or theme for each day, and set the lifetime on each one to only the day that it is relevant for.

Time Limits

As well as the total lifetime of a post, you can also set limitations on what time of day you want it to be shown. So you might have a "It's lunchtime, kids!" message, which lasts the whole length of the conference, but is only displayed between 12:30 and 1:30.

You can set the time limits for each post to be either "only show this post during certain times" (useful for dinner time annoucements, say) or to "Don't show during these specific times" (useful for frivolous/jokey slides which you don't want up during reflection or meditation times, say).

Permissions

Posts can either be 'published' or not. If they aren't published, then they can't be seen by the outside world, and the screens won't display them. You can give permission to some users to create posts, but not publish them, if you desire. This means you can have content authors who make content for specific feeds, but you can give publish permission on that feed only to certain line-managers or communication directors, who then publish the posts.

Permission to change the layout and design of the actual screens can only be done by "administrators", but still using the web interface. An administrator in this sense may well be your graphic designer, which is fine. You really need to be trusting your graphic designers, as they care a lot more about making things look perfect than anyone else, especially when they are given the tools to do so.

That's it!

Hopefully that gives you a good overview of the system, it's designed to be reasonably easy to work with.

Notes when using the system

There are a few things which it's good to know:

Magic Variables

In HTML and plain text posts, you can put the following "magic variables":

`%%TIME%%` and `%%DATE%%` which will show up on the output screens as the current date and time, respectively.

Note: this time is local to that screen's computer! So if you are using a raspberry pi or similar, and you're on a closed network without internet access, then you'll also need to set up some kind of NTP server too.

You can customize how the DATE or TIME is formatted using standard strftime style formatting tags after the word DATE or TIME Eg:

`The month is: %%DATE%B%%` (will show "July" only)

`<h1>%%DATE%B</br>%Y%%</h1>` (will show "July" and then "2015" on the next line)

It's not recommended to put the current second, as due to the current design, that will only get updated infreqently, and has no way of guarenteing being right.

Post Sizing/Scaling

When posts are displayed on the output screens, they will automatically be scaled to fit in the zone that they're displayed in. If you're really struggling to get text big enough, there's a good chance that you simply have too much text to fit it all into that zone.

Also, if you have text which is "Title 1" (`<h1>...</h1>` for the HTML junkies) it can display as different sizes in different posts, as each post is scaled independently.

The HTML "rich text" posts are intentionally somewhat limited. If you want to have a post where the *design* is important, not just the textual content of the post, they you should use an external graphic design package, such as Inkscape (free), Adobe Illustrator (expensive), PixelMator (good, not too expensive, mac only'), and then post as an Image type.

CHAPTER 2

Screen Design

"Screens" are where your posts actually get displayed.

A "Screen" output is actually a URL on the streetsign server which you can point a browser on client machines at.

Backgrounds

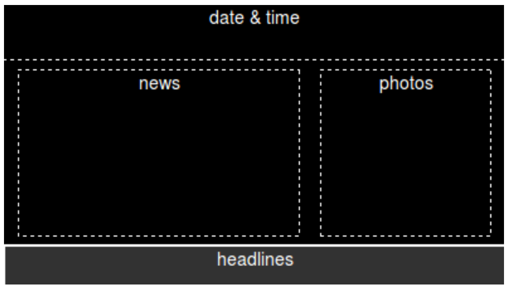
Are any normal web image file (jpg, png, gif...) and stored in the main 'uploaded files' directory.

The picture will automatically scale to try and fill the screen as best as it can. This is usually what you want, but if you are using a design with exact spacing, it can be useful to use the *Aspect Ratio* over-ride to force the screen to display it correctly.

Zones

Each 'zone' is a block in your Screen which can display Posts from multiple Feeds.

The usual layout is a scrolling "headlines" zone at the bottom, a "date and time" zone at the top, and a textual news zone on one side, and pictures on the other.



The reason that each zone can show multiple feeds is so that you can separate your data "semantically". You can have a "Campus News" feed, a "Staff-only" feed, a "World News" feed, a "daily photos" feed, and so on. Then in your staff room(s), the screen can show posts from all of those, but in the public cafeteria not show the staff only feeds, and so on.

Types of Zone Animation

Zones can either be "Fade-in, Fade-out" type, or "scrolling". Scrolling text is currently limited to right-to-left (sorry Arabic & other RTL language groups - I'm happy to work on it if there is a demand, just let me know). The scrollspeed can be over-ridden using the *scrollspeed* option.

The fade-in/fade-out time can be set per zone on the screen editor. This is a value in milliseconds.

Size

These 4 values are usually percentages, and are "distance from this side of the screen". So setting them all to 0 would mean a zone that filled the screen completely.

Fonts

There are a few default commonly available font families listed, but you can also upload your own .TTF fonts into the *user_files/fonts* directory, and they should then appear for you to use.

Other CSS

You can use standard CSS settings such as *text-shadow*, *font-weight* etc. in here to further style your zones.

Screen 'client aliases'

As well as the Layouts described above, you can also set up "client aliases" which can be changed to point at different layouts, with different options, without the URL (and so the client configuration) needing to be changed. These aliases are edited on the 'screens' section, and can be set to appear on the front 'dashboard' as well, if you choose.

Screen "over-ride" Options

Sometimes it's useful to be able to force certain settings on a screen view. These over-ride options can be set by passing them in the URI query string:

```
http://streetsign-server/screens/basic/Default?options=go_here&more=can_to!
```

One useful combination for some lower powered machines is:

```
?fadetime=0&scrollspeed=20
```

These can all be set as part of a client alias, which would result in a simpler URL such as:

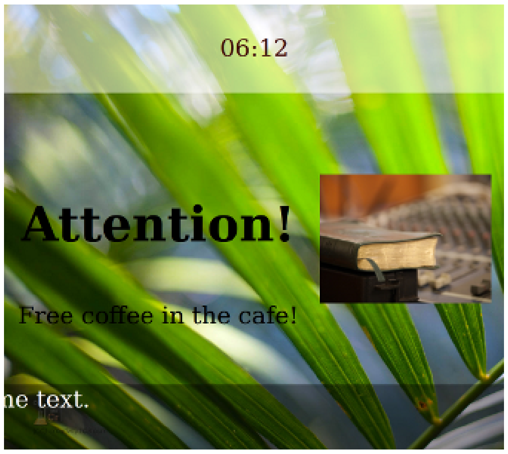
```
/client/dining-hall
```

for instance.

Here are all the options, and how to use them:

Aspect Ratio

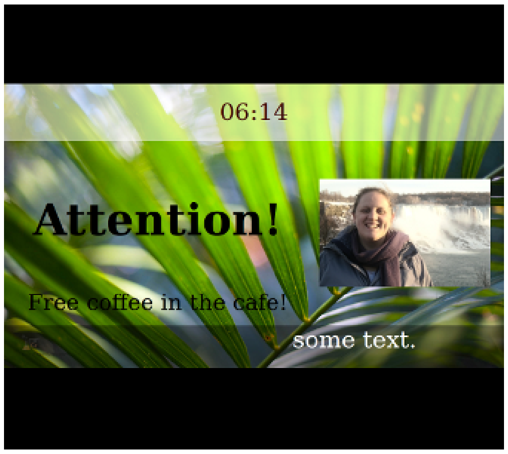
By default each screen will display at 'full screen', stretching all the zones out to fill the screen. This is usually what you want, but not always.



`forceaspect` lets you force the aspect ratio of the screen. This will usually end up letterboxing on the screen. This is really useful when you are using the same Screen on 4:3 projectors as well as 16:9 displays, or when you are testing / designing a view for one aspect ratio while using a screen with a different one

```
http://streetsign-server/screens/basic/Default?forceaspect=1.7777
```

for instance will force the aspect ratio to 1.7777 - which is 16:9.



By default, the picture will then be centered vertically inside the browser window.

`forcetop` lets you then force the top of the active picture to whereever you want. so:

```
http://streetsign-server/screens/basic/Default?forceaspect=1.7777&forcetop=0
```

will force a 16:9 image at the top of the display, rather than centering it vertically.

Fade Time

You can over-ride all zone fade times using the `fadetime` option.

```
http://streetsign-server/screens/basic/Default?fadetime=0
```

for instance will disable fading between posts. This is very useful for underpowered clients, where you want to turn off effects and fading, but don't want to disable the effects for other more powerful clients which are also using that screen URL. Fadetime is an integer value in milliseconds. So 2000 is 2 seconds, 20000 is 20 seconds, 200 is .2 of a second, and so on.

Scroll Speed

You can over-ride the scrolling text speed for all scrolling zones using the `scrollspeed` option:

```
http://streetsign-server/screens/basic/Default?scrollspeed=30
```

Will scroll quite a lot slower.

The default speed is '17', lower numbers are faster, and higher numbers are slower. Why 17? Well, it just seemed like a reasonable compromise speed that looks decent in most places.

CHAPTER 3

StreetSign Admin's guide

Here's first a guide on how to set up streetsign to play with. For full proper deployment (for real live production usage) check the *Deploying Streetsign in Production* page.

Instalation

StreetSign requires python 2.7, imagemagick (to generate thumbnails). If you are installing on a fresh server, you may need to install `python-headers` or `python27-dev` or whatever your distribution calls it.

(On a stock OSX computer, it doesn't need anything.)

Once you've cloned or otherwise downloaded StreetSign and put it where you want it to be, you need to run the setup script:

```
./setup.sh
```

which will create a python virtualenv in `.virtualenv`, and install all the libraries and other requirements into there.

Running it.

To run the server in 'production mode' you can use the built in `waitress` web server:

```
./run.py waitress
```

Which will spin up a version which should be totally capable for small deployments (up to a hundred screens or so, I guess). If you are going to be on a public network, then it's advised that you run streetsign - either with waitress or another WSGI server of your choice - behind nginx or another reverse proxy, which should keep things a bit saner. If you're on a public network, also remember to set up your reverse proxy to use SSL, so that you aren't sending log-in credentials around in plaintext.

If you want to run anther WSGI server, remember the virtualenv that streetsign is using, so any scripts you write need to use the python found in there (`.virtualenv/bin/python`). You can use pip from in there to install any pypi packages you need too (`.virtualenv/bin/pip install gunicorn`, say).

Some links:

- The official flask deployment docs
- The waitress server docs

Users

When you first install, the `setup.sh` script will create some default users for you. The admin user has the default password `password`. You should change this as soon as possible.

Since some things display differently for admins compared to 'normal' users, it's probably a good idea if you are supporting other users, to create a 'normal' user for yourself as well. Then if someone is finding something confusing, you can check from that non-admin user quickly.

Password hashes and moving the database

The user passwords are stored in the database hashed using two salts - an individual salt per password (stored in standard passlib style in the password field) and also with the site-wide "secret". This "secret" is generated automatically when you run the setup script, and is stored in the `config.py` file. (It's also used by flask for encrypting session data, and so should NEVER be stored in a repository, or shared outside deployment.)

What this means is that if you move a database.db file from one installation to anonther, you will also need to bring the same config.py (or, at least, copy the SECRET from there.)

Housekeeping & removing old content

By default, streetsign will tag content that has a lifetime which ended over a week ago as "archived". This means it no longer shows up on the interface for anyone except admin users. After a month, it will be deleted, including any uploaded images.

This "housekeeping" should take milliseconds to run as long as it's run regularly, so each screen view will fire a request to the server to do it once an hour or so. It and can also be triggered through the interface by hitting the "housekeeping" button on the "All Posts" page, or on the front page (Dashboard).

If you want to ensure that this runs every hour or so, you can use standard unix cron, or any other task scheduling program.

- `HTTP POST` to `/posts/housekeeping`

so if you're using cron:

```
0 * * * * nobody curl -d "" 'http://streetsign_url/posts/housekeeping' > /dev/null
```

should do it.

For automatically updating content from external feeds, again, screen views will automatically do this once a minute, but you can also trigger it manually (or via cron) with a

- `HTTP POST` to `/external_data_sources/`

If you are on a public network, and worry about DOS issues, then realistically, you should be running behind a revese proxy such as nginx. With nginx you can add restrictions on what URLS are accessble by any IP address, so you can limit these addresses to only be accessed by the machine with cron, for instance.

Server Time

You may well have your main server running in one timezone (say GMT), but actually be using the signs in another time zone. By default, clients will all use their own local time zone, and the server uses the server time. There is a configuration option you can set in `config.py`:

```
TIME_OFFSET=60
```

for example, which will offset post lifetimes, etc, by an hour. (Minutes are used so that half-hour-off timezones are supported).

CHAPTER 4

Deploying Streetsign in Production

How to deploy a 'production-ready' streetsign installation.

Dependencies

First you need to install the python headers (for compiling some extra modules), imagemagick (to generate thumbnails), and pip for installing other python modules, and git for downloading streetsign itself.

On Debian/Ubuntu Server, this will be:

```
sudo apt-get install python-pip python-dev imagemagick git
```

On CentOS 6.7, its:

```
sudo yum install python-devel python-pip ImageMagick git
```

User/Group

Streetsign, as every other service, should really run as it's own user, for security's sake

```
sudo useradd streetsign
```

Which will also create a new group for it.

Installation path

As per the LSB, probably the best place for public facing services to install their data is `/srv/`. So we should create that directory, and install streetsign there:

```
sudo mkdir /srv/streetsign
sudo chown -R streetsign:streetsign /srv/streetsign
```

Actually Installing it

We'll use git to get the latest version, and set it up as normal:

```
cd /srv/streetsign
sudo su streetsign
git clone https://bitbucket.org/dfairhead/streetsign-server.git .
./setup.sh
```

Test it's all ready to go

This step is technically un-needed, but probably a good idea. While still `su`'d as streetsign:

```
./run.py waitress
```

and then from a web browser, browse to that server's IP at port 5000. If you don't know the server IP:

```
ifconfig |grep 'inet addr:'
```

Note that often servers may have a firewall (e.g. IPTables, or similar) blocking port 5000.

And then you can `exit` from the streetsign user.

Configure streetsign to start on system-boot

Unfortunately, this is different on practically every linux distribution, and even different between Ubuntu 14 and Ubuntu 15, for instance.

There are startup files in the streetsign source, in the `deployment` folder.

systemd systems (Ubuntu 15.x, CentOS 7, Debian Jessie, etc)

If you're on a systemd based linux (Such as Ubuntu 15.x), then copy the `deployment/systemd/streetsign.service` file to `/var/systemd/system`, edit it to make sure it's all correct for your system (which it should be, if you've followed the above instructions):

```
sudo cp /etc/streetsign/deployment/systemd/streetsign.service /var/systemd/system/
```

And then tell enable the service:

```
sudo systemctl enable streetsign
```

And then you can actually start it up:

```
sudo systemctl start streetsign
```

If it's all running quite happily, then cool. If you want to test that it does actually start on boot, feel free to reboot the server and see what happens.

Logs for streetsign can then be found using the normal systemd logging utils:

```
journalctl -u streetsign.service
```

(Recent) upstartd systems (Ubuntu 14.x, etc)

Copy the streetsign upstart configuration file to `/etc/init`:

```
sudo cp /srv/streetsign/deployment/upstart/streetsign.conf /etc/init/
```

And then you should edit /etc/init/streetsign.conf to make sure it's all correct for your system. If you've followed the above instructions, then it should be.

You can now start the service, to test it's all working OK:

```
sudo start streetsign
```

And it should automatically run on boot as well. To stop that, you can edit the `/etc/init/streetsign.conf` file, and put a # in front of `start on runlevel [2345]`.

The streetsign log file can be found with the rest of the upstart log files at:

```
/var/log/upstart/streetsign.log
```

SysV (initscript) systems (CentOS 6.x, etc.)

There's a basic (hopefully OK) init script in `deployment/init`, which should work on many other systems. So just copy it in:

```
sudo cp /srv/streetsign/deployment/init/streetsign /etc/init.d/
```

and then turn it on with whatever your OS uses for that. On CentOS, for instance:

```
service streetsign start
```

will start it running. To make it run on system boot, it's:

```
chkconfig --add streetsign
```

Getting Streetsign on to Port 80

If streetsign is going to be 'public facing', and so you want it to be running on the regular HTTP port 80, or over HTTPS, then it's best to run a 'reverse proxy' in front of it.

The most popular options are NGiNX and Apache.

nginx

Install nginx:

```
sudo apt-get install nginx
```

Or on CentOS:

```
yum install nginx
```

copy the basic streetsign configuration file in:

```
sudo cp /srv/streetsign/deployment/nginx/streetsign /etc/nginx/sites-available/
```

on CentOS, it's to `/etc/nginx/conf.d/streetsign.conf`:

```
sudo cp /srv/streetsign/deployment/nginx/streetsign /etc/nginx/conf.d/streetsign.conf
```

Edit it with whatever settings you wish.

Enable it (Debian Only):

```
sudo ln -s /etc/nginx/sites-available/streetsign /etc/nginx/sites-enabled/
```

And if streetsign is the only thing you're using nginx for, and you don't need the default welcome page, turn that off:

```
sudo rm /etc/nginx/sites-enabled/default
```

And of course, restart nginx:

```
sudo service nginx restart
```

Apache

Apache is pretty easy to install:

```
sudo apt-get install apache2
```

or:

```
sudo yum install httpd
```

is usually enough. There's a default configuration file to put streetsign on its own virtualhost in the `deployment/apache` folder. If streetsign is the only site running behind apache here, then that configuration file may be enough. Usually, however, you'll need to modify the VirtualHost / Server Name / other settings a bit yourself.

You will need the apache `mod_proxy` and `proxy_http` modules enabled. On Debian based systems:

```
sudo a2enmod proxy proxy_http
```

on others you need to check in your apache config (usually `/etc/httpd/conf/httpd.conf` or similar) that the modules are enabled. These two lines (wherever they are) need to be uncommented:

```
LoadModule proxy_module module/mod_proxy.so
LoadModule proxy_http_module module/mod_proxy_http.so
```

Or similar.

You can then copy in the config file. On Debian based systems:

```
sudo cp /srv/streetsign/deployment/apache/streetsign.conf /etc/apache2/sites-
↪available/
```

Or on CentOS:

```
sudo cp /srv/streetsign/deployment/apache/streetsign.conf /etc/httpd/conf.d
```

Edit it to have the settings you need, and enable it. (Debian only):

```
sudo a2ensite streetsign
```

And if you want to, disable the default apache welcome-page/site:

```
sudo a2dissite 000-default
```

Finally, restart apache:

```
sudo service apache2 restart
```

and it should all be working.

CentOS Notes: (Esp. SELinux)

CentOS has SELinux installed often, and is locked down pretty hard. You will probably need to allow the HTTPD to make outgoing connections, and also to access files in the */srv/streetsign/streetsign_server/static* folders.

(All of the following commands are as root.)

First install semanage:

```
yum install policycoreutils-python
```

Then open up HTTPD to have outgoing-network access (to the actual python server):

```
/usr/sbin/setsebool httpd_can_network_connect 1
```

And to make that permanent:

```
/usr/sbin/setsebool -P httpd_can_network_connect 1
```

Then give read access for HTTPD to the `/srv/streetsign/streetsign_server/static` and all subdirectories:

```
semanage fcontext -a -t httpd_sys_content_t "/srv/streetsign/streetsign_server/
↪static(/.*)?"
```

And apply the policies:

```
restorecon -Rv /srv/streetsign
```

CHAPTER 5

StreetSign Developer Documentation

This will contain the documentation for developers, either working *on* StreetSign, or making plugins/other software to work with it.

It's assumed that you're reasonably familiar with StreetSign the end user web interface and basic terminology (Post, Feed, Screen, Zone, etc). If not, see the *Getting Started*

Project Structure Overview

The main "everything" is kept in the folder (& python package) `"streetsign_server"`, which is a WSGI application, and can be treated as such. Within `streetsign_server`:

__init__.py

The basic app definition, which pulls in everything else that it needs.

models.py

All the peewee ORM database models are defined in here. These are fairly smart models, containing as much business logic as makes sense to keep in them.

Full Reference: *streetsign_server.models*

user_session.py

A bunch of helpful functions for dealing with the user session & authentication stuff. The basics are kept in the `User`, `Group`, and `UserSession` objects defined in `models.py`, but the functions here help make life easier and shorter. The function `login(username, password)` for example, attempts to log in with those credentials, and if it can, then it creates the appropriate `UserSession` database items, and adds useful items to the session cookie.

Full Reference: *streetsign_server.user_session*

views/

The views package contains all of the actual "endpoints" of the web application. So the functions which generate the pages you see when you use the web interface, the screen rendering, etc. It is mostly split out into submodules.

Full Reference: *streetsign_server.views*

logic/

The logic package is where more complex logic is being moved to from the views package. Once things start becoming more complex than simply pulling things from the database and rendering it, and the logic is application specific rather than model specific, then it should go in here. There is more logic than really should be in *views/*, and in general, it should be ported across to here.

Full Reference: *streetsign_server.logic*

static/

The standard Flask static assets folder.

static/lib/

Contains external libraries (jQuery, knockout, etc).

static/screens/

Is for all the css & javascript used by the front end screen rendering.

static/user_files/

Is the default location for uploaded user files. This can be configured to some degree in the root of the tree `config.py`.

static/style.css

As much as possible, I've kept with vanilla twitter bootstrap, as it looks perfectly good enough. Anything application specific is in here. Of course, this is talking only about the web interface, not about the output screen rendering, which doesn't reference this file at all.

static/main.js

All the basic functions which all of the back end needs. Such as rendering "flashed" notices, etc.

static/model_zones.js

The screen editor is a reasonably complex beast, and uses the wonderful knockout.js library to keep everything sane. This file contains the zones knockout model, and functions to control it all (creating the preview, etc). This file is *not used at all* in the output screen rendering. It's a totally separate kettle of fish.

static/post_times_editor.js

The post time restrictions editing is also a bit fiddly, so all the clobber for that is kept in here.

"Lets Minify And Join all the Javascripts!!!"

Don't be silly. This is a local network application, not expecting zillions of concurrent users from around the globe. There is no point whatsoever in adding any complexity like that.

Other files in the root directory

Starting with the file layout:

`run.py` - run the basic development web server, or waitress stand alone WSGI server.

`setup.sh` - downloads all needed python packages, including virtualenv, and installs them into a local virtualenv called, very creatively, .virtualenv. Also initialises the database, if it doesn't exist. If you totally stuff up the database, then you can simply delete it and run this script again.

`database.db` - the sqlite database, generated by `setup.sh`, normally.

`db.py` - a very simple database shell.

API/Urls

Can be found in *API/Urls*

How Post Types work

TODO

How External Data Types Work

TODO

How the 'Screens' Work

Full Reference: *Screen Design*

How Different Libraries are used

TODO

CHAPTER 6

StreetSign Developer Reference

This is the mainly autodoc generated documentation from the source code docstrings.

API/Urls

This is a list of the basic API/URLs that streetsign uses, and can be used to generate other applications (such as a native hardware accelerated client, for instance.)

All URLs are given in Flask style, so `<stuff>` denotes a variable, part of the URL that changes depending on what you're requesting. `<int:blah>` means only accept integer values for `blah`, etc.

`/screens/<template>/<screenname>`

Returns the data about this screen, including which zones are defined in it, which feeds are attached to those zones, etc. Usually this is called with the `basic` template, which renders the post in place, using their post type javascript renderers. This is the main view that normal screen outputs will use.

`/screens/json/<int:screen_id>`

Returns the JSON details about a screen. Which zones it has, CSS, which zones have what feeds attached, etc.

To save bandwidth, you can call:

`/screens/json/<int:screen_id>/<md5sum>`

with the md5 that was previously given in the screen JSON data, and the server will respond with either ONLY the same MD5sum and screen id, or else with a new MD5sum, and complete new screen JSON data (and id).

`/screens/posts_from_feeds/<[list,of,feed,ids]>`

Given a json type list of feed ids (`[1,3,2,9,21]`, say), return the JSON of all posts which are currently active.

Note that for some web servers/requests/proxy systems, you will have to URL encode the list. For example: `/screens/posts_from_feeds/%5B1%2C2%2C%5D` rather than `/screens/posts_from_feeds/[1,2]`. Most web browsers, and most good HTTP request libraries should do this automatically for you, however.

`/screens/post_types.js`

Returns all the various JSON rendereres that are needed for drawing posts to a screen zone.

streetsign_server.models

Where all the Peewee ORM models live.

Useful functions

Users and Groups

Login Stuff

Most of the time, these shouldn't be used directly, but instead use the functions from *streetsign_server.user_session*

And the Invalid Password Exception:

Posts and Feeds

Screens and Output

Misc

streetsign_server.user_session

Higher level functions for working with users, logging in, sessions, etc.

streetsign_server.views

The views (HTTP end points) of the web interface, and the Screen Displays.

Utilites

Users and Authentication

The Screen Output Views

Feeds and Posts

User uploaded files

streetsign_server.logic

When the views start becoming overly complex, as much as possible the logic should be split out to sane sizes functions over here in logic.

streetsign_server.post_types

Post types are the different kinds of posts. So plain text posts, rich html posts, image posts, etc. Each type is defined in its own module here, so it's easy to add your own.

As well as the `.py` module, there's also usually a `.screen.js` javascript file which contains the output screen javascript for rendering it again at the other end, and a `.form.html` file, which is the template used by the individual module to render the form for editing that type of post.

Any other bits and pieces you need for this specific post type should be kept here with the module.

Post Type Generic Module Overview

As each type of post is different, and has different data storage requirements in the database, and doesn't have to be queried against, there is no point in having some complex database schema to cope with all possibilities.

Instead, there are columns in the database for storing things common to all posts, and which are queried against, such as lifetime, etc. There is also a CharField textual field which stores post-type-specific data in it as JSON.

Each post-type module doesn't need to do the json dumping and loading, that's done further up the chain. Instead, the module functions are handed standard python dicts (`{"thing": "value"}`), to work with. When they return those dicts, they get bundled back into JSON and stuffed in the database.

So you will need to be careful not to try and store things which aren't JSONable.

Inside each module,

You need the following functions:

`form`(*data*)
: return the html for the form for editing this kind of post. `data` is, of course, the data last saved and to be displayed in the form.

`receive`(*data*)
: when the form is submitted, this function will be send the request.form data, which you should now extract the relevant info from, sanitize!!, and then this function should return the data you want to store in the database about this post.

display(*data*)
: if you need to do anything to the data before it gets sent on to the screen (as json) to be displayed, this is the place to do it.

screen_js()
: return the javascript object used to render this item on the screen client side.

Rendering Screen Zones

And finally

There is also a 'weird' experimental web-hook post type included. If you find it confusing for your users, feel free to remove it.

streetsign_server.external_source_types

External data, such as RSS feeds, weather, news, etc, are really useful things to be able to automatically add in to your signage system.

External Sources are set up on the feeds page, and deliver into various feeds, as you decide.

The different kinds of sources are defined here, in pretty much the same manner as post types. Read that documentation, and have a look at the rss post kind source, and it should make sense.

The main external Source Types:

The RSS Source Type

CHAPTER 7

External Libraries

TODO (much more detail, and author/licence info)

None of this would be even remotely possible without the amazing open source community, and the following fantastic libraries that I've been able to use:

Flask

Flask is cool Python (Micro) web framework

Peewee

Simple, splendid Python ORM

Flask-Peewee

Joining flask and peewee together with ease

Sqlite

Simple, Fast single-file database system.

jQuery

Better DOM for javascript in the browser

Also these jQuery plugins:

jQuery.transit

Use CSS transitions for faster smoother animation.

chosen

Very sweet multiselect boxes.

knockout.js

For doing interactive object modelling and user-interface stuff which would be horrible in javascript + html otherwise.

knockout.mapping

Which turns non-schema/undefined messy JSON objects into knockout object arrays which can be assigned, modified, and played with.

Twitter Bootstrap

I'm not a front-end CSS wizard, and this helps the admin interface look clean and professional with so little work from me. It really does wonders.

bootstrap-wysihtml5 & wysihtml5

Very nice, simple HTML5 rich text editor, with bootstrap integration

bootstrap-datetimepicker

Nice looking reasonably easy to use date & time picker controls. Really, browsers should have this stuff built-in already.

Pylint

For making it so much easier to keep the code base consistent and clean.

pylint git commit hook

forcing me to keep it clean.

Bleach

Stripping and cleaning HTML is so easy with this great library!

FeedParser

Parsing XML (RSS & Atom) feeds has never been so easy!

Passlib & py-bcrypt

There is no reason to keep passwords insecurely...

Waitress

The simplest WSGI server ever...

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index