Captain’s Log

This document is meant as a loose guide on how to set up a fork of this software.
It will take some minor adjustments to fully match your setup, but this and some Google’ing should get you there.

Some Pyhton & HTML knowledge is assumed. This is not WordPress.

PDF LAST UPDATE: 07 jan 2017 05:42 (BE time)

Chapter 0: Introduction

I’m assuming a Linux environment. If you are developing on Windows, I suggest getting a small VPS to use as a playground.
You can develop on Windows and deploy and debug on there, it will save you a load of pain with Docker and File permissions.
(NTFS won’t be your friend in this story)

Command prefixes:
- `#` Root user on the host machine. (This is usually in the run environment.)
- `$` Normal user on the host machine, in virtual environment. (This is always in development environment.)
- `'name'>` Inside a docker interactive mode, with `'name'` specifying which container. (See §2.1 Commands to remember)

Used software/technologies

- Backend structure
  - Docker (Containerize)
    - Nginx (Static files + front end + HTTPS)
    - Certbot (Renews Let’s Encrypt HTTPS cert)
    - PostgreSQL (Database)
    - Flask (Python 3 web framework)
      - Jinja2 (Template language)
      - …
      - SQLAlchemy (Database models)
- Web frontend
  - Gravatar (Universal User avatars)
  - Markdown (User friendly markup language)
  - Bootstrap 4.0 α5 (HTML, CSS, and JS framework, includes jQuery)
  - Font Awesome (Icon font)
  - emojify.js (Browser side emoji rendering)
  - cdnjs (Content Delivery Framework for all JS & CSS frameworks)
  - SimpleMDE (In browser WYSIWYG markdown editor)

For a full list of Python packages, see the `requirements.txt` file.
There are way too many Flask plugins to list them all here.

Most of the HTML, JS and CSS won’t be explained here.

Acknowledgments

Most sources will be listed where appropriate, along side links to the documentation used.

But some sources are more equal than others, and so deserve a special extra thanks:

- Richard Campen:
  My friend who got me using Python and Flask, and he recommended me:
- The Flask Mega-Tutorial:
  A sometimes outdated but still really helpful tutorial. This was the stepping stone for this app.
- The Bash Hackers Wiki
  For making the custom certbot script possible.

Chapter 1: Software installations

- Install all the required software:
  - Dev environment
    - Git
    - PyCharm (see note below)
    - Python 3, with pip and buildtools
  - Run environment
    - Git
    - Docker
      - Engine: docker engine
      - Compose: docker compose
- Fork and clone this repo in a work folder. (Both dev and run.)
  All commands are now issues from here, unless otherwise noted.
- Create a clean python virtual environment.
  Do not inherit global packages!
- Open the repo in PyCharm
  - You can dismiss the “Dockerfile detection” event.
    We won’t be using PyCharm’s docker integration, it appears to be buggy.
  - Do enable the version control integration, if asked. (Click ‘add VCS root’)
  - Mark the following directories:
    - app/migrations -> Exclude
    - app/venv -> Exclude
    - app -> Sources Root
    - app/uBlog/templates -> Templates Folder
    - app/static -> Resources Root
  - Via Settings -> Project -> Python interpreter, use the virtual environment.
    This may require a PyCharm restart.
  - Setup autodeploy (Optional, rec)
    - Tools -> Deployment -> Configuration…
    - Add a server and fill out the login details (For ssh used sftp)
      Under Root path, set something the user has full access too. eg `/home/server`
    - ‘Mappings’ tab: Add the deployment path.
      (I recommend something under `/home/user/` while developing. Less permission issues that way.)
    - ‘Excluded Paths’:
      - Exclude the ‘Local Path’ `/venv/`
    - Enable ‘Automatic Upload’ and open the ‘Browse Remote Host’ tab (Tools -> Deployment)
    - Do the initial upload.
      - Select the root folder in the Project tab (left)
      - Via Tools -> Deployment -> Upload to …
      - SSH into the server, `cd` into the folder.
        Any command for the run environment are supposed to be run here.

All commands issues on the dev environment are assumed as being executed in the root folder of the git clone.
With the virtual environment activated!

- Check to make sure you didn't import any global packages.
  `$ pip freeze` should output nothing.
- Install the required packages.
  `$ pip install -r app/requirements.txt`

It is important this file it kept up to date, because it gets read by our docker container to know what to install.
If you add a package, make sure you update the file. `$ pip freeze > app/requirements.txt`.
If you do this, know you'll have to restart (rebuild actually) the docker image!

Note about LetsEncrypt: You'll need a domain tied to the IP of your run environment.
If you, like me at home, can't reach your own external IP from inside the network, (Thanks Telenet…)
you can add the domain to your hosts file so that you can use the actual full domain to test your site.
This will make sure that you'll get the green lock when your browser accepts the certificate.

If you want a domain name, I recommend Versio. They don't have all of the TDL's, but they are super cheap.
If you do want to use them, I have a referral link here ;-).

Note on PyCharm
If you can, get the professional edition.
It has things like auto-upload on save, which make debugging on a VPS a whole lot easier.
You can request a free license as a student, or for open source projects.

Chapter 2: Docker

We are using docker to containerize all of our software.
This makes deploying and scaling easy if its ever required.
While you don't need to do this, I'm assuming you are.

If you are running windows: Welcome to your worst nightmare; File permissions.

Run environment assumed.

Make sure the Docker service is running.
`# docker ps` shows the list of running docker instances.

Have a look in the `docker-compose.yml` file.
The `Fixme` notes indicate where you should look and see if settings are appropriate for your environment.

The reason why there are so many host->container volumes instead of plain named volumes is for easier debug access.
Flask supports live code updating, and it detects changes though the mounting barrier of docker.
If this wasn't the case, you would have to rebuild and restart the entire composition for every single change made.
It also helps you not getting blocked by LetsEncrypt for going over the 5 certs/domain/week limit.

You will need to adapt the `settings.env` file and make a `passwords.env` file to provide the required configuration.
Also edit the email templates in `/app/config/`, unless you want people to get messages with my URLs.

I highly recommend using `tmux` to split your terminal so you can have one running the docker-compose on

top.
And have one shell ready to jump into any running container below.
(Scrolling up in tmux is done with 'Control-b' + '['. This is called copy mode. Hit 'q' to exit it.)

If something errors and not all of the containers are started, use `# docker ps` to see what all is running.
Then, with the IDs listed here, you can stop any that did start: `# docker stop <id [more ids ...]>`.

The container ids will be a hexadecimal string, the names something like `thebeerarchive_app_1`.
I'll be referring to the containers by there names inside the `docker-compose.xml` file.

Commands to remember

Run environment assumed.

- To build everything: `# docker-compose build`
- To run everything: `# docker-compose up`
- Your new 2 best friends:
  - Build and start all containers: `# docker-compose build && docker-compose up`
  - Jump into a running container with a shell: `# docker exec -it <container name> bash`
- End all the containers by Control-Cing in the `docker-compose` terminal.
- End an interactive session by `exit` (or Control-D, for EndOfFile).

Chapter 2: A virgin start

The fist start requires all the container images to be downloaded, and the custom ones to be build.
This

- Start all containers.
  This will take a while the first time.
  Every container's output stream will get a prefix in a different color.
- If you browse to your domain now, should give you a `sqlalchemy.exc.ProgrammingError` error page.
  Do this, to verify that everything started up correctly.
  - If you have no connection at all, look at `nginx` 's output.
  - The HTTPS should be valid, if not, restart all containers.
    If the HTTPS still isn't valid, look at `certbot` 's output.
    It may indicate that it can't reach your server.
    Port forwarding or firewall rules are a good first guess if something is wrong.
- Try and initialize the database with `app> python -m uBlog db upgrade`.
  The result should be something along these lines, if not, look at the output of the containers:
  - Did the `db` container say this anywhere?
    `PostgreSQL init process complete; ready for start up.`
  - Is `db` complaining about file permissions being wrong?
- Open the domain again. You should get a 404.
- Register an account. The fist account will become the Überadmin.
  No verification email will be send.
  You will be redirected to create a first page, which will always the the root page.

Upgrade log:

```
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> 4701d5186146, ...
```

Chapter 3: Using the website

You should be on the ‘Edit New Page’-page right now.

- Give you index page a title eg `Home`. This the name by which they will appear in the menu.
- The page name eg `index`, is the URL of the page. The has to be unique and alphanumerical and underscore only.
- After you make a basic page, you’ll be redirected to `/admin`.
- You should probably make yourself a brewer, but it’s not required.
- Make a new page with name `/terms`, it is linked to in the footer.

Pages

Every page has a name, a title and content.
The name is the page’s URL and must thus be unique and allows word characters (`[A-z_]`) only.
A page does not have to appear in the top menu, if if you set it to do so, it will be sorted based on ID.

Beers

New beers can be made by all brewers. Brewers can only edit there own beer pages, and can only post to there own beers.
Every beer needs a unique name, but it can contain spaces and special characters since the URL is based on the ID.

Unlisted beers appear only for other brewers. This allows posts to be made and saved over a period of time, but the
published date will be the time of first creation.

A beer or post can be edited without notice for 10 minutes after its first saved.
After that it will show a “Last edited some time ago.” marker.

Posts

A post is an appendage to a beer page, meant to provide updates.

Profiles

Your own profile page shows all information, use the public link to view your profile like someone else would see it.
You can show or hide your email address, and globally disable emoji rendering for your user.

Admins

The admin dashboard has a list of all pages, users and beers.
User permissions can be edited here.
Be careful who you make admin! Admins can appoint other admins, and can also demote other admins!
(Userid 1 is an exception. They can never be demoted.)

An admin can edit all pages and beers & posts. If an edit (or new) button is orange, it indicated that you are editing
a beer or post belonging to someone else. Admins can only erase user’s bio, not edit it.

Markdown quirks

- It is assumed that whomever writes the content has a basic understanding of Markdown and HTML.
- You can't use all HTML elements. No content that could be considered unsafe, such as styles and scripts.
- There may also be differences between SimpleMDE (the content editor) and Python Markdown.
- All markdown rendered by Python is surrounded by a div with class 'md'; this makes jQ/CSS selecting easier.

Special pages

You can't use these as a page urls, except for `/terms`.

- `/` (root) Always the page with ID = 1.
- `/terms` Linked to via the footer link and the link on the register page, should be created ASAP.
- `/login`
- `/logout`
- `/register`
- `/brews`
- `/profile`
- `/admin`

Chapter 4: How it was made

This is a retrospective look at how I ended up making most of the systems.
Also contains links to relevant documentation.

Docker & Docker-Compose

The resulting yml file is a combination of the referenced tutorials & docs,
compose tut 1 was closest to what's needed for this project, but uses the old v1 syntax.
So I consulted the official documentation compose overview & compose file.
The second tutorial's yml file compose tut 2 was used as a starting point for the v2 syntax.

Docker images used:
- db: docker postgres (Official PostgreSQL)
- nginx: docker nginx (Official Nginx)
- certbot: docker debian (Custom Debian)
- app: docker python (Custom Python)

app (Nginx)

Most of the nginx configs are based on nginx configs I have used for years.
All http traffic gets rerouted to https. (As it should, now that https is basically free.)

The `/static/` block serves static files, `/.well-known/` serves files from LetsEncrypt.
Any other requests are routed to `app`. When the Flask internal server is used, it routes to port 5000.
In a production environment this should be changed to using a uwsgi socket.

If any changes are made to config and if you want to reload the configs without restarting the whole docker:
`# docker kill -s HUP $(docker ps -q -f "label=${CERT_LABEL}")` (`CERT_LABEL` is defined in `settings.env`)

`/robots.txt` and `/favicon.ico` are served from the static folder too. Any file starting with `.ht` is blocked.

Any errors outside of `app` are routed to `/static/[404|403|50x].html`.
They are only really used when the `app` container is down, or when the error happens inside of the `static` folder.

Volumes from `app` and `certbot` are mounted, and the container is linked (networked) with `app`.

References: nginx docs (and in particular nginx docs core)

db (PostgreSQL)

Nothing much special here, except that, I would have liked to have the data folder as a Host->Container volume.
This would have made it easier to backup, but because PostgreSQL is *very* picky about file permissions, this made
it impossible to use at all on Windows. Although I later dropped caring about Windows at all, I did start out developing
on 2 machines. A Windows 10 laptop for school and an ArchLinux desktop at home.

I don't recommend removing the external port binding. You are probably better off blocking the port with a firewall rule.
That way you could still SSH tunnel and connect to debug issues. (And connect the Database explorer from PyCharm!)

Named volume `/var/lib/postgresql/data` (`db_volume`) is not host mounted.

certbot (LetsEncrypt)

The is a fully custom Docker image, build on top of Debian (with jessie backports).
Debian with jessie backports is fully supported by certbot: certbot install.

The bash script was made with the help of the certbot user manual and the bash hackers wiki.

This container also does some minor other things, like make sure the `./run/log/nginx` folder exists.

Volumes (used by nginx):
- `/etc/letsencrypt`: host folder `./run/le`
- `/var/log`: host folder `./run/log`
- `/var/run/docker.sock`: host file `/var/run/docker.sock`, *setup dependent*!
- `/usr/bin/docker`: host file `/usr/bin/docker`, *setup dependent*!

Other references: docker certbot

app (Python Flask)

Another custom image, though it didn't really have to be.
The reason for using a custom image is to force a rebuild when the `requirements.txt` is changed.
Without using a custom image, docker doesn't seem to always detect that a rebuild is required.
(Particularly on Windows.)

To make debugging easy, I set flask to run directly instead of via uwsgi.
This allows for on-the-fly python edits *if* the container's files are also updated live.
To do this I mount the app directory where it would normally have been copied on build of `app`.
Now the only thing that require a relaunch (and rebuild) of the `app` container are changed dependencies.

This should be changed when the app is deployed in production.

The basic layout of the flask app is based on the flask mega tutorial, and my friend's advice.
There are 2 modules, `config` and `uBlog`.

Volumes:
- `/usr/src/app`: host folder `./app/`

The app is linked (networked) with `db`.

config

Mostly reads from the environment, to keep all configuration in one place.
The email templates are defined here though.

uBlog

Most of the systems used here are based on the flask mega tutorial, and my friend's advice. Again.

Templates

The templates are based on the [jinja2] language.

There is only one base HTML file, and it actually contains quite a lot of default code.
Because of template inheritance, and the ability to call the super block and include its code.
You can both easily extend as override default behaviour.

You will have to edit the templates to adapt the app to your own needs.

Some notes:

- In `base.html`:
  - Block `content`:
    - `self.title()`:
      Self refers to the template itself, and title to the title block.
      So the content of the title block is inserted here.
  - Block `script`:
    - Emojis can be disabled by the user. They are rendered via JS, and optionally disabled here.
    - Markdown markup is applied here, since its easier to do it via JS than changing the existing markdown parsers.
    - The navbar's active element (if any) is selected here, because, again its easier than doing in in python.
  - There is an extra script block that tells people not to touch the js console.
- In `admin.html`:
  - Show all pages, so even the ones that arn't in the menu can be found.
  - All registered accounts with no confirmed email older than 24h can be manually removed here.
    (This happens at 36h automatically.)
  - All users management happens here.
    - Promoting to admin (or demoting)
    - Promoting to brewer (or demoting)
    - Banning (and unbanning) (Cannot ban other admins, you have to demote them first.)
    - Manually activating an un-activated user.
- In any of the `edit_xxx.html` files:
  Template inheritance is used to expand on the existing tags, like the `head` and `script` blocks.
  Originally I used a template block inside a `script` tag, but then PyCharm does not recognise that its inside

a
script tag when that block is overwritten. This results in no language injection (syntax highlighting and more).
- Pages include an edit button when appropriate. If an admin edit button is shown, its orange.
- Any form is secured using WTForms's build in hidden_tag().
- `error.html` is used for any error as part of the site, not any programming error.
  So any raised/returned exceptions like '403 Forbidden' or '404 Not Found' is caught.
- `macros.html` is not directly used as a template, but imported as a lib.
  It contains the SimpleMDE script, because its used in all of the `edit_xxx.html` templates.

Because of Jinja2's auto-escaping, you need to indicate when a string is already processed HTML.
Otherwise it would be escaped. Example of this: The HTML markdown output for any of the pages is out thought the jinga filter `safe` which prevents is from being escaped.

__init__.py

TIL: you can escape in markdown.

General setup. Sets up all of the 'master object' instances, such as:
- `app` : Flask (God. Also reads configuration)
- `db` : Flask-SQLAlchemy (Database connection)
- `migrate` : Flask-Migrate (Database version management)
- `manager` : Flask-Script (See `__main__.py` below.)
- `lm` : Flask-Login (User session handling)
- `gravatat` : Flask-Gravatar (User avatars)
- `scheduler` : APScheduler (Background tasks)
- `mail` : Flask-Mail (Email)

As you can see, most of these are specific Flask plugins. Some, like SQLAlchemy, have an underlying framework
that is independent of Flask, but the use of the plugin makes its use a lot easier.

This file includes the helpers, views, models, forms and tasks AFTER all of the master objects are initialized.
It also contains the catch-all handler for errors, because of a bug in Flask.
And lastly it starts the scheduler before first request is processed.

__main__.py

Because of Flask-Script, running `app> python -m uBlog` will allow you to use several management related thing:
- `shell` will start a python interpreter with all of the flask context.
Useful for debug, but I prefer to use Flask's in browser console when I need it.
- `runserver` starts the flask internal development server. (Used as startup command in our docker composition.)
- `db` has sub-commands: (list not complete)
- `migrate` create a new migration file. **Use this when making changes to the models.**
- `upgrade` pushes any new migrations to the database. **Use after checking the migration file you made.**

forms.py

Contains all of the WTForm objects, which is again used through a Flask integration plugin (Flask-WTF)
Although in this case, that plugins base class is wrapped by another plugin WTForms-Alchemy.
It makes the forms automagically accept fields from a database mode, and applies the right verifiers.
Two important reasons why I used it:

- The `Unique` filter that does DB a lookup for you
- Transferring data to and from the form fields from the model instance.

helpers.py

Contains a collection of helpful misc functions:

- `load_user` : LoginManager requires to know how to fetch User objects from the DB.
- `context_processor` : Data loaded for each template render. (All beers & pages, for the menu bars for example)
- Filter `date` : pretty print datetime object
- Filter `nl2br` : Turn `[\r]\n` into `<br/>\n`
- Filter `timedelta` : Turn time(delta) into human readable text, with the Humanize package.
- Test `older` : Test if a timestamps is older than x time from now (or a second timestamp). x is set via kwargs.
- `make_markdown` :
  - Turns (Github Flavoured) markdown into HTML.
  - Cleans out any unsafe HTML
  - Wraps it in a div with class 'md', for easier CSS and JS selection.
  - Can use a set string for empty input. (Used in erased user bio.)
- Filter `markdown` : Uses `make_markdown` as a filter, also wraps the result, so no extra `safe` filter is required.
- `login_required` wrapper: Enforces a view to be login only, also blocks banned users.
- `login_admin_required` wrapper: Enforces a view to be admin.

models.py

Contains all database models. Uses SQLAlchemy and SQLAlchemy-Utils.
The extra utils package is used for the EmailType and PasswordType.

The PasswordType uses Python's operator overloading to allow the password to be stored properly in the database without
any extra effort on the programmer's side at all. You can assign the password field with the plain text data from the
form, but it will never be stored like that. It will be converted to a salted hash, to your specifications in the field declaration. Comparing the password field with plain text input data from a form automatically does the required salt
and hash operations to check the validity. If the password encoding algorithm has been upgraded since the last login,
the password will be re-salted and hashed to make sure the strongest possible algorithm is being used.
This is pure genius.

The use of the Utils package does require you to allow auto coercion, as per sqlalchemy utils doc.

The User class is required to have the 3 `is_xxx` properties, so its compatible with Flask-Login's `current_user` object.

The relations between the different classes are declared with backrefs which, for example, cause all Post objects to
have an 'author' field. The relationships are 'dynamic', aka the DB lookup is only preformed when the field is accessed.

The `server_default` argument is primarily used to make sure migrations happen correctly, for example: When a column is

added, marked as not nullable, the table would have to be empty to preform the migration. Otherwise it would result in
invalid data. The `server_default` here fixes this. The regular `default` is used when new instances are created, and
can thus be different.

The `json` field on User is another special one, because it allows non structured data to be stored in an structured DB.
This is however specific to PostgeSQL only! (The JSONB type is anyway, but the regular JSON doesn't allow for some neat
tricks like filtering based on json keys/values INSIDE of the `WHERE` clause of a query.) The type has to be wrapped in
a MutableDict, because otherwise it doesn't automatically update the column when pushing the object to the DB.
See fine print of sqlalchemy postgres dialect.

tasks.py

Contains all scheduler jobs:
- `remove_stale_users` removes any non activated users older than 3 days.
- `remove_stale_reset_tokens` removes any expired password reset token.

This makes use of some of that sweat JSON dependent SQL row filtering.

views.py

Contains all routes (views) and 2 extra handlers:
- `any_error`: Handles all exceptions and errors like 404 and 403, not jinja template errrors.
- `before_any_request`: Closes session of any user who got banned.

Some notes:
- Route `/` pulls up page with id `1`.
- Any single path url (`/<name>`) is handled as a page, if no more specific or complex route exist.
- All of the `/api` routes are meant as AJAX endpoints for the `/admin` page.

If you got here, and read all of it, you deserve a cookie and a break.