Policy Study Documentation

Matthew Jaffee

September 17, 2014

Contents

1 Overview

This is an application for studying how users read and interpret password policies. It tracks users, allowing new users to be created, and old users to sign in.

The purpose of the application is to have users complete a set of questions about a password policy using information in a password policy description document. Each user is assigned several password policies and is asked to complete the same questionnaire for each policy.

When the questionnaire is complete, statements of a BNF grammar are generated which formally describe the password policy.

Most aspects of the applications behavior are configurable - there is an administrative interface which allows several parameters to be changed, such as the number of policies assigned to each user, and email addresses which comments are sent to. It is also possible to add and remove admin accounts, and change their passwords.

Other administrative functionality includes viewing raw database information and testing the policy questionnaire.

1.1 Dependencies

The policy study is a Flask application http://flask.pocoo.org/. It makes use of the Jinja2 templating engine http://jinja.pocoo.org/docs/ for html templates, and MongoDB http://www.mongodb.org/ for a datastore. Pymongo http://api.mongodb.org/python/2.7rc0/ is used for communicating with MongoDB.

On start up, the policy study reads in an XML file (specified in config.py) and uses xmltodict https://github.com/martinblech/xmltodict to convert the XML to a Python dictionary for internal manipulation.

1.2 Datastore

There are currently 4 'collections' in the database. Their formats look like the following:

1.2.1 user

```
{
"isadmin": boolean,
"date_created": isoformat date,
"done_demographic_survey: boolean,
"name": string,
"hash_name": string describing hash func for username if not admin and pass if admin,
"password_hash": string,
"policies": list of policies for this user,
"demographic_survey": {"field": "value, ....},
}
```

1.2.2 config

```
{
"policies_per_user": int,
"comments_emails": list
"policies": list [["title", "file"], ["title2", "file"]],
"policies_used": {"title": int, "title2": int },
}
```

1.2.3 questionnaire

```
{
```

```
"username": string,
"policy_name": string,
"date_created": isoformat date,
"state": string ["new"|"draft"|"completed"]
"questions": {
  "1": "1.A", (select one question)
  "20.1": ["20.1.A", 20.1.G"], (select multi question)
  "13.1.A.a": 22, (numerical cloze question)
  "13.1.B.c": "second(s)", (select one cloze / memo / text)
}
}
```

1.2.4 logs

```
{u'bnf_version': u'kevinV1',
 u'policy_name': None,
 u'qaversion': u'30oct2013v1',
 u'timestamp': 1395435599.534258,
 u'type': u'http',
 u'user_agent_browser': u'chrome',
 u'user_agent_language': None,
 u'user_agent_os': u'linux',
 u'user_agent_version': u'32.0.1700.123',
 u'user_type': u'participant',
 u'xmlversion': u'022114'}
```

1.3 Overall instruction flow

On application start up, `runserver.py` calls `views.initialize`. The `initialize` function drives the functions in `elements.py`, loading the XML file, parsing it, and preparing it for use by the application. When this is complete, `runserver.py` calls `app.run` which starts the webserver, and then the application is listening for requests from user's web browsers.

When a user requests a page, the request is routed to a particular function in `views.py` based on the `@app.route` decorators that you will see above many of the functions. Many of the functions also have an `@login_required` or `@admin_required` decorator which ensure that a user is logged in before displaying that page, or is an admin respectively.

Once a request is routed, the logic in the view function executes. Most functions end with something like `return render_template('page.html',`

`extradata=extradata)` This finds the `page.html` template file (under the `templates/` subdirectory) and potentially passes some extra data to it which jinja will use to fill out values in the template and return an actual html page to the user. Aside from templates, flask will also serve files from the `static/` subdirectory, which consists of CSS, Javascript, and the PDFs of password policies.

1.3.1 questionnaire

The most complex view/template is the policy questionnaire. Because of it's highly dynamic and configurable nature, the questionnaire page is set up as a single view and template which is flexible enough to display any "page" of the questionnaire.

`elements.py` pre-processes the XML file to get it into a format which is easily usable by the `questionnaire_page` template. The template consists of a series of Jinja macros which call eachother to render all the main parts of the page.

2 Scripts

The application comes with a set of scripts for generating reports, backing up the data store - switching to a clean datastore, etc.

Call scripts by executing `python policy_study/script_name.py` followed by whatever commands and options are necessary. Almost all modes of all scripts may take an optional argument of –dbname=<dbname>, the name of the mongo database to operate on. The default database name is "policydb".

2.1 report.py

The report script has 3 modes.

2.1.1 BNF

```
bnf [options] <outputdir>
```

Writes a file per questionnaire to the output directory which contains the bnf statements for that questionnaire.

2.1.2 Report

```
[options] <outfile>
```

Writes a single file which contains all collected log data in CSV format

2.1.3 Report per

`[options] <field> <outputdir>`

Writes one file per unique value of `field` name to the outputdir containing the logs which have that field equal to that value. Writes logs which do not have that field to a file named `None`.

2.2 store_admin.py

The store administration script has 2 modes.

2.2.1 cp

`cp [options] <target_name>`

Copy the given dbname, or the default database to a new name - potentially to save for later reports.

2.2.2 rm

`rm <db_to_remove>`

Drop the named database. Dropping policydb will reset the application to a clean slate.

2.3 mongodump

Mongodump is a script that comes with MongoDB. You should be able to run it simply by typing 'mongodump'. It will write the contents of whatever database it is given to disk. Options for controlling it are available by executing `mongodump --help`.

3 File Layout

3.1 setup.py

Project file for python setuptools. Running `python setup.py install` will download all necessary dependencies to run the application.

3.2 runserver.py

Running `python runserver.py` starts a lightweight development server for testing the application.

3.3 policy_study/

The policy_study directory contains application code and tests.

3.3.1 *.py files

All of the `.py` files that don't start with test_ are python application code.

3.3.2 test_*.py files

Files containing tests which are named after the file they are testing. i.e. `test_utils.py` contains tests for functions in `utils.py`.

3.3.3 templates

The templates directory contains `.html` files which are actually Jinja2 templates. Each of these files corresponds to a page in the policy except for `common.html` which contains common elements which most of the other pages inherit, and `macros.html` which contains Jinja2 macros that the other template files can use.

3.3.4 static

The static directory contains files which the webserver will need to access to serve directly. This includes stylesheets (CSS), Javascript, images/icons, and fonts. All CSS written for the application is in `style.css`. The majority of the javascript is in `questionnaire.js`, although there are a few page specific pieces in `demosurvey_page.js`, `general_comments_page.js`, and `questionnaire_page.js`. All other `.js` and `.css` files are libraries which were pulled in from outside sources.

3.4 documented-input.xml

This is an annotated version of the input xml file which explains what each element is used for.

3.5 resources

The resources directory contains a number of non-code files used by the application.

3.5.1 policies.txt

A text file of the format:

```
Policy_Name filename.pdf
Policy_Name2 filename2.pdf
```

Where `Policy_Name` is the title which will be given to the policy in the application, and filename.pdf is the name of the file to take from the `policies` directory (also under resources).

Each user has a list of the titles associated with them.A persistent mapping from title to filename is stored in the "config" collection in the database. When you change a title, a new entry gets added to the mapping with the new title and the old filename - The old title will continue working as it did before. If you change the filename associated with a particular title, then any user who has that title will now be served the new filename.

To be more succint: Changing a title adds to the title:filename mapping, whereas changing a filename changes the mapping.

3.5.2 polices

Contains policy pdfs, which are specified in policies.txt.

3.6 Input XML file

The input xml file (which, at the time of this writing, is set to `pp_test.xml` in `policy_study/config.py`. See `documented-input.xml` for more information.

4 Moving to production

4.1 Installing and configuring production application

Adapted from: http://vladikk.com/2013/09/12/serving-flask-with-nginx-on-ubuntu/

Create production directory, and chown to yourself for setup convenience

```
sudo mkdir /var/www/policy_study
sudo chown -R mij:mij policy_study
```

Create the python virtualenv and point it to a python 2.7.8 executable you'll have to make sure this python executable is accessible by www-data or whatever user you end up using for uwsgi.

```
cd /var/www/policy_study
virtualenv -p /home/mij/usr/bin/python2.7 venv
source venv/bin/activate
```

Clone the repository

```
git clone mij@localhost:/home/mij/opt/git/policy_study.git .
```

Install dependencies

```
python setup.py install
```

Edit `policy_study/config.py` to point to the correct resource files and database. I added the `BASE_DIR` line and modified the other 4 - leave the rest of the file the same

```
BASE_DIR="/var/www/policy_study/"
DBNAME = "production_policydb"
POLICY_DIR = BASE_DIR + "resources/policies/"
POLICY_FILE = BASE_DIR + "resources/policies.txt"
INPUT_FILE = BASE_DIR + "resources/pp_test.xml"
```

Install uwsgi

```
sudo apt-get install build-essential python-dev
pip install uwsgi
```

Install nginx

```
sudo add-apt-repository ppa:nginx/stable
sudo apt-get update && sudo apt-get upgrade
sudo apt-get install nginx
sudo /etc/init.d/nginx start
```

Configure nginx

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

Create nginx conf file `/etc/nginx/conf.d/policy_study_nginx.conf`

```
server {
    listen      80;
    server_name localhost;
    charset     utf-8;
    client_max_body_size 75M;

    location / { try_files $uri @policy_study; }
    location @policy_study {
        include uwsgi_params;
        uwsgi_pass unix:/var/www/policy_study/policy_study_uwsgi.sock;
    }
}
```

Create uwsgi ini file `/var/www/policy_study_uwsgi.ini`

```
[uwsgi]
#application's base folder
base = /var/www/policy_study

#python module to import
app = policy_study
module = %(app)

home = %(base)/venv
pythonpath = %(base)

#socket file's location
socket = /var/www/policy_study/%n.sock

#permissions for the socket file
chmod-socket    = 644

#the variable that holds a flask application inside the module imported at line #6
callable = app

#location of log files
logto = /var/log/uwsgi/%n.log
```

Create and chown uwsgi log directory.

```
sudo mkdir -p /var/log/uwsgi
sudo chown -R mij:mij /var/log/uwsgi
```

Configure uwsgi to run as a background processes using uwsgi emperor. Edit uwsgi emperor conf `/etc/init/uwsgi.conf`

```
description "uWSGI"
start on runlevel [2345]
stop on runlevel [06]
respawn
env UWSGI=/var/www/policy_study/venv/bin/uwsgi
env LOGTO=/var/log/uwsgi/emperor.log
exec $UWSGI --master --emperor /etc/uwsgi/vassals --die-on-term --uid www-data --gid w
```

Finish configuring emperor

```
sudo mkdir /etc/uwsgi && sudo mkdir /etc/uwsgi/vassals
sudo ln -s /var/www/policy_study/policy_study_uwsgi.ini /etc/uwsgi/vassals
sudo chown -R www-data:www-data /var/www/policy_study
```

Start it up

```
sudo start uwsgi
```

4.2 Updating Production Code Base

4.2.1 Pull down the code

```
cd /var/www/policy_study
sudo git pull
```

If you encounter problems at this step (merge conflicts, "you must commit your changes", that sort of thing), it is probably because of the changes made to config.py (see the install instructions). You can do `git status` and `git diff` to see what your changes are, then:

```
git stash
git pull
# and optionally
git stash apply
# or you can just edit config.py and add your changes back in
```

4.2.2 It may be necessary to restart uwsgi - depending on the code change

```
sudo stop uwsgi
sudo start uwsgi
```

5 Running Multiple Instance for Testing purposes

You can run multiple instance of the policy study (with different input files, and data stores) simultaneously.

5.1 Instructions

5.1.1 Check out a fresh copy of the codebase.

5.1.2 Edit runserver.py

Replace the line

```
app.run(debug=True, host='0.0.0.0')
```

with a line like:

```
app.run(debug=True, host='0.0.0.0', port=5001)
```

You can replace 5001 with anything from 1000 to 65536, as long as it doesn't conflict with an already running service. If the server fails to start, try changing the number.

5.1.3 Edit `policy_study/config.py`

Replace the line

```
DBNAME = "policydb"
```

with a line like:

```
DBNAME = "mickyTestingSection2"
```

Once again, with each further instance, you just have to keep changing to a different (unique) name.

5.1.4 Start it up as usual

Just remember to navigate to the appropriate instance - if you were going to http://policyserver.nist.gov:5000 before, now you need to go to http://policyserver.nist.gov:5001