tarbena Documentation

Release tarbena

tarbena

Mar 11, 2022

Contents

CHAPTER 1

Contents:

1.1 Introduction

1.1.1 About the app

Each Town Hall in small towns needs to be introduced to new technologies. So for my people (Tárbena/Alicante) I have decided to make an application that suits their needs.

The sole purpose of `tarbena_app` is to add a *graphical user interface* system functionality and give you the ability to control | everything.

The key features of Tarbena apps are:

- Manage subsidies
- Manage parcels
- Manage keys that belongs to the Town Hall of Tarbena
- Add any other app they think can resolve their daily problems

1.1.2 Requirements

1. Django 1.11.6
2. Python 3.6.5
3. mysqlclient

1.1.3 Get ready (Windows)

- Download python
- Install `pip`. For windows we get it here
- Create an environment variable for pip:

```
setx PATH "%PATH%;C:\Python27\Scripts"
```

- Create an isolated environment for python with virtualenv:

```
pip install virtualenvwrapper-win
mkvirtualenv myproject
```

- Activate the virtualenv:

```
Scripts/activate => windows
source bin/activate => linux
```

- Install `Django` with pip
- Install `MySQL`:

```
For Python 2.7:
Download it here: http://www.codegood.com/download/10/
And with our virtualenv activated we do: easy_install file://C:/Users/ORDENADOR_1/
↪Downloads/MySQL-python-1.2.3.win32-py2.7.exe

For Python 3.6:
Download it here: https://www.lfd.uci.edu/~gohlke/pythonlibs/#mysql-python
32 bits refers to Python version and not to our system
```

- Create Django project and migrate the database:

```
django-admin startproject src
python manage.py migrate
python manage.py startapp subvenciones
```

1.2 Architecture

1.2.1 Configuration

1. Gitignore: local settings
2. Creating settings module with files for different environments: base, local and production
3. Creating project documentation with reStructuredText(.rst) files and Sphinx
4. Multiple requirements files

1.2.2 Apps

- **Authentication** (built-in django.auth): This app handles user signups, login, and logout
- **Profiles**: this app provides additional user profile information

- **Subvenciones**: subsidies management
- **Parcelas**: parcels management with Google Maps
- **Django Honeypot**: admin security
- **Django Admin Interface**: theme for Django Admin Panel
- **smart-selects**: area-ente selects functionality
- **django-notify-x**: django notification system activity

Authentication

Created Login and Signup system in the root project (urls.py, templates, static).

Profiles

When a new user is created, his profile is also created.

Subvenciones

Application to manage subsidies.
Tables:

- Estado
- Colectivo
- Ente
- Area
- Subvencion
- Comment
- Like

Create Subvencion

When I create a new subsidie I add admin functionality in my form so I can add new fields suchs as: estado. Its a Django Popup functionality:
django-admin-popup-functionality

On the other hand, I insert the comment field into the subvencion form with a formset following this guide:
django formset implementation

Here for the formset field (contenido) I used markdown editor (added his configuration into settings):
Django markdown editor

In my custom template the editor didn't work as expected so in my base.html I had to add the following urls:

```
<link href="{% static 'plugins/css/ace.min.css' %}" type="text/css" media="all" rel=
↪"stylesheet" />
<link href="{% static 'plugins/css/semantic.min.css' %}" type="text/css" media="all"␣
↪rel="stylesheet" />
<link href="{% static 'plugins/css/resizable.min.css' %}" type="text/css" media="all"␣
↪rel="stylesheet" />
<link href="{% static 'martor/css/martor.min.css' %}" type="text/css" media="all" rel=
↪"stylesheet" />
<script type="text/javascript" src="{% static 'plugins/js/ace.js' %}"></script>
<script type="text/javascript" src="{% static 'plugins/js/semantic.min.js' %}"></
↪script>
<script type="text/javascript" src="{% static 'plugins/js/mode-markdown.js' %}"></
↪script>
<script type="text/javascript" src="{% static 'plugins/js/ext-language_tools.js' %}">
↪</script>
<script type="text/javascript" src="{% static 'plugins/js/theme-github.js' %}"></
↪script>
<script type="text/javascript" src="{% static 'plugins/js/highlight.min.js' %}"></
↪script>
<script type="text/javascript" src="{% static 'plugins/js/resizable.min.js' %}"></
↪script>
<script type="text/javascript" src="{% static 'plugins/js/emojis.min.js' %}"></script>
<script type="text/javascript" src="{% static 'martor/js/martor.min.js' %}"></script>
```

Also I follow this guide to add the dropdown functionality for ente and area:
Dependent dropdown list

Datatables functionality to list subvenciones into the index.html

Parcelas

Django Honeypot

https://github.com/jamesturk/django-honeypot

Django Admin Interface

https://djangopackages.org/grids/g/admin-styling/
https://github.com/fabiocaccamo/django-admin-interface
You can choose your own theme!

smart-select

https://github.com/digi604/django-smart-selects
I use this app for chaining selects (ente-area)

Installation:

```
pip install django-smart-selects
url(r'^chaining/', include('smart_selects.urls')), # into root url's, after admin
```

models.py:

```
from smart_selects.db_fields import ChainedForeignKey
area = ChainedForeignKey(
    Area,
    chained_field="ente",
    chained_model_field="ente",
    show_all=False,
    auto_choose=True,
    sort=True,
    default=''
)
```

Warning: In Lib/site-packages/smart_selects/static/smart_selects/admin/js/chainedfk.js has a problem, all his methods should be defined as object so I copy the new js from here: new chainedfk.js

And I copy it to my root static project so when I git pull to my production server I have it solved:
static/smart-selects/admin/js/chainedfk.js

And finally into my create.html and edit.html template I import them like this:

```
<script type="text/javascript" src="{% static 'smart-selects/admin/js/chainedfk.js'
↪%}"></script>
<script type="text/javascript" src="{% static 'smart-selects/admin/js/chainedm2m.js
↪' %}"></script>
<script type="text/javascript" src="{% static 'smart-selects/admin/js/bindfields.js
↪' %}"></script>
```

My old functionality is from here: old functionality

django-notify-x

https://github.com/v1k45/django-notify-x

```
pip install django-notify-x
INSTALLED_APPS = ('notify',)
url(r'^notifications/', include('notify.urls', 'notifications')),
python manage.py migrate notify
python manage.py collectstatic
```

Warning: `notify` application has in his models the verb to 50 limit character, just change it to TextField instead of CharField.

About the warning you can do:

```
# Lib/site-packages/notify/models.py
verb = models.TextField(verbose_name=_('Verb of the action'))
python manage.py makemigrations
python manage.py migrate
```

Views:

```
from notify.signals import notify
notify.send(self.request.user, recipient=self.request.user, actor=self.object,
                verb='subvención, %s' % (form.cleaned_data.get('nombre')), obj=self.
↪object,
                nf_type='create_subvencion')

Actor: The object which performed the activity.
Verb: The activity.
Object: The object on which activity was performed.
Target: The object where activity was performed.
```

1.2.3 Project commands

To start the Python interactive interpreter with Django, using your `settings/local.py` settings file:

```
python manage.py shell --settings=tarbena.settings.local
```

To run the local development server with your `settings/local.py` settings file:

```
python manage.py runserver --settings=tarbena.settings.local
```

Backup my models:

```
python manage.py dumpdata myapp --indent=2 --output=myapp/fixtures/subsidies.json
python manage.py dumpdata auth --indent=2 --output=myapp/fixtures/auth.json
```

Load data from those backups:

```
python .\manage.py loaddata subsidies.json
```

Export my production database password and then get it or save it in a secure folder in the production server:

```
export MYSQL_PASSWORD=1234
'PASSWORD': os.getenv('MYSQL_PASSWORD'),
Or I can add it to my file and import it like the secret key and the email password.
```

Save my `SECREY_KEY` in a secure file in the production server:

```
>>> from django.core.signing import Signer
>>> signer = Signer()
>>> value = signer.sign('My string')
>>> value
'My string:GdMGD6HNQ_qdgxYP8yBZAdAIV1w'
```

1.2.4 Multiple requirements files

- **base.txt**: place the dependencies used in all environments

- **local.txt**: place the dependencies used in local environment such as debug toolbar
- **production.txt**: place the dependencies used in production environment
- **ci.txt** (continuous integration): the needs of a continuous integration such as django-jenkins or coverage

1.2.5 Admin Documentation

https://docs.djangoproject.com/en/1.11/ref/contrib/admin/admindocs/

```
pip install docutils
```

1.2.6 git-flow

The main branches

- **Master**
- **Develop**

I consider `origin/master` to be the main branch where the source code of HEAD always reflects a `production-ready state`.

I consider `origin/develop` to be the main branch where the source code of HEAD always reflects a state with the latest delivered development changes for the next release. Some would call this the `integration branch`.

Note:

When the source code in the develop branch reaches a stable point and is ready to be released, all of the changes should be merged back into `master` somehow and then tagged with a release number.

Therefore, each time when changes are merged back into master, this is a new production release by definition. We tend to be very strict at this, so that theoretically, we could use a Git hook script to automatically build and roll-out our software to our production servers everytime there was a commit on master.

Supporting branches

The different types of branches we may use are:

- **Feature branches**
- **Release branches**
- **Hotfix branches**

Feature branches

Comes from `develop` and must merge back into `develop`.
Branch naming convention: anything except `master`, `develop`, `release-*`, or `hotfix-*`

Feature branches (or sometimes called topic branches) are used to develop new features for the upcoming or a distant future release. When starting development of a feature, the target release in which this feature will be incorporated may well be unknown at that point. The essence of a feature branch is that it exists as long as the feature is in development, but will eventually be merged back into develop (to definitely add the new feature to the upcoming release) or discarded (in case of a disappointing experiment).

Feature branches typically exist in developer repos only, not in origin.

Creating a feature branch

```
$ git checkout -b myfeature develop
Switched to a new branch "myfeature"
```

Incorporating a finished feature on develop

```
$ git checkout develop
Switched to branch 'develop'

$ git merge --no-ff myfeature
Updating ea1b82a..05e9557
(Summary of changes)

$ git branch -d myfeature
Deleted branch myfeature (was 05e9557).

$ git push origin develop
```

Note: The `--no-ff` flag causes the merge to always create a new commit object, even if the merge could be performed with a fast-forward. This avoids losing information about the historical existence of a feature branch and groups together all commits that together added the feature.

Release branches

Comes from `develop` and must merge back into `develop` and `master`.
Branch naming convention: `release-*`

Release branches support preparation of a new production release.

Creating a release branch

Release branches are created from the develop branch. For example, say version 1.1.5 is the current production release and we have a big release coming up. The state of develop is ready for the "next release" and we have decided that this will become version 1.2 (rather than 1.1.6 or 2.0). So we branch off and give the release branch a name reflecting the new version number:

```
$ git checkout -b release-1.2 develop
Switched to a new branch "release-1.2"

$ ./bump-version.sh 1.2
Files modified successfully, version bumped to 1.2.

$ git commit -a -m "Bumped version number to 1.2"
[release-1.2 74d9424] Bumped version number to 1.2
1 files changed, 1 insertions(+), 1 deletions(-)
```

After creating a new branch and switching to it, we bump the version number. Here, bump-version.sh is a fictional shell script that changes some files in the working copy to reflect the new version. (This can of course be a manual change—the point being that some files change.) Then, the bumped version number is committed.

Finishing a release branch

When the state of the release branch is ready to become a real release, some actions need to be carried out. First, the release branch is merged into `master` (since every commit on `master` is a new release by definition, remember). Next, that commit on master must be tagged for easy future reference to this historical version. Finally, the changes made on the release branch need to be merged back into `develop`, so that future releases also contain these bug fixes.

```
$ git checkout master
Switched to branch 'master'

$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)

$ git tag -a 1.2
```

Note: You might as well want to use the -s or -u <key> flags to sign your tag cryptographically.

To keep the changes made in the release branch, we need to merge those back into develop, though. In Git:

```
$ git checkout develop
Switched to branch 'develop'

$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)
```

This step may well lead to a merge conflict (probably even, since we have changed the version number). If so, fix it and commit.

Now we are really done and the release branch may be removed, since we don't need it anymore:

```
$ git branch -d release-1.2
Deleted branch release-1.2 (was ff452fe).
```

Hotfix branches

Comes from `master` and must merge back into `develop` and `master`.
Branch naming convention: `hotfix-*`

Hotfix branches are very much like release branches in that they are also meant to prepare for a new production release, albeit unplanned. They arise from the necessity to act immediately upon an undesired state of a live production version. When a critical bug in a production version must be resolved immediately, a hotfix branch may be branched off from the corresponding tag on the master branch that marks the production version.

The essence is that work of team members (on the `develop` branch) can continue, while another person is preparing a quick production fix.

Creating the hotfix branch

Hotfix branches are created from the master branch. For example, say version 1.2 is the current production release running live and causing troubles due to a severe bug. But changes on develop are yet unstable. We may then branch off a hotfix branch and start fixing the problem:

```
$ git checkout -b hotfix-1.2.1 master
Switched to a new branch "hotfix-1.2.1"
$ ./bump-version.sh 1.2.1
Files modified successfully, version bumped to 1.2.1.
$ git commit -a -m "Bumped version number to 1.2.1"
[hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1
1 files changed, 1 insertions(+), 1 deletions(-)
```

Don't forget to bump the version number after branching off! Then, fix the bug and commit the fix in one or more separate commits.

```
$ git commit -m "Fixed severe production problem"
[hotfix-1.2.1 abbe5d6] Fixed severe production problem
5 files changed, 32 insertions(+), 17 deletions(-)
```

Finishing a hotfix branch

When finished, the bugfix needs to be merged back into `master`, but also needs to be merged back into `develop`, in order to safeguard that the bugfix is included in the next release as well. This is completely similar to how release branches are finished.

First, update master and tag the release.

```
$ git checkout master
Switched to branch 'master'

$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Summary of changes)

$ git tag -a 1.2.1
```

Note: You might as well want to use the -s or -u <key> flags to sign your tag cryptographically.

Next, include the bugfix in develop, too:

```
$ git checkout develop
Switched to branch 'develop'

$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Summary of changes)
```

The one exception to the rule here is that, *when a release branch currently exists, the hotfix changes need to be merged into that release branch, instead of** `develop`. Back-merging the bugfix into the release branch will eventually result in the bugfix being merged into develop too, when the release branch is finished. (If work in develop immediately requires this bugfix and cannot wait for the release branch to be finished, you may safely merge the bugfix into develop now already as well.)

Finally, remove the temporary branch:

```
$ git branch -d hotfix-1.2.1
Deleted branch hotfix-1.2.1 (was abbe5d6).
```

Note:

This work-flow guide I brought it from:
https://nvie.com/posts/a-successful-git-branching-model/
http://aprendegit.com/que-es-git-flow/

1.3 Secret files

When deploying don't save the secret files into the project. Save them into a safe place and ignore them with gitignore. Things such as: database password, `SECRET_KEY`, email password, etc.

When I use:

```
install -r requirements/production.txt
```

Check that in base.txt I have no debug toolbar and it's in local.txt and also check that I have everything from development server with:

```
pip freeze -r requirements/base.txt
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search