DjangoFloor Documentation

Release 0.19.12

Matthieu Gallet

February 06, 2017

Contents

CHAPTER 1

Overview

DjangoFloor helps you to easily create new website with the excellent framework Django. However, Django suffers from different drawbacks:

- websites are complex to deploy in production,
- JavaScript is somewhat hard to use for integrating dynamic parts in a webpage,
- several other libraries are almost required,
- you need a lot of code which will be common to all your projects.

We try to solve these problems:

- a system of settings based on default project settings and local configuration files,
- an easy-to-use signal system allowing to call Python or Javascript from Python or Javascript,
- some common libraries are set as dependencies,
- all common code are in a unique library that will be included into your projects.

Default configuration assumes that you use Redis as secondary database, alongside a more classical SQL like *PostgreSQL* or *MySQL*. Everything is ready to use Celery for background tasks.

Installing / Upgrading Instruction on how to install this package

Tutorial Start here for a quick overview

Using the demo Use the demo provided in the source

Global architecture Explain the global architecture of a production deployment

Included dependencies What are all these included dependencies?

Integrate your settings How to integrate your own views into a DjangoFloor project

Signals DjangoFloor Signals, or how to call Python and JS code from Python or JS with the same syntax

Settings DjangoFloor settings system

Deployment How to deploy a production website

Packaging your application How to create packages for Linux distributions

Documentation generation Use DjangoFloor to generate a complete documentation of your project

Python API Documentation The complete API documentation, organized by modules

Javascript and HTML code Documentation for the JavaScript functions

CHAPTER 2

Full table of contents

2.1 Installing / Upgrading

2.1.1 Environment

DjangoFloor is compatible with Python 2.7, 3.2, 3.3, 3.4 and 3.5. We strongly recommend to create a virtualenvironment and the use of *virtualenvwrapper*. Go to the doc to discover how to use it.

```
mkvirtualenv djangofloor
```

2.1.2 Simple installation

The simplest way is to use pip:

```
pip install djangofloor
```

You can also install with optional dependencies:

```
pip install djangofloor[websocket,scss]
```

2.1.3 Installing from source

If you prefer install directly from the source:

```
git clone https://github.com/d9pouces/django-floor.git
cd django-floor
python setup.py install
```

2.1.4 Upgrading

Again, the easiest way is to use pip:

```
pip install djangofloor --upgrade
```

2.2 Using the demo

First, you must download and unzip the source on Github.

```
curl -L -o django-floor.zip https://github.com/d9pouces/django-floor/archive/master.zip
unzip django-floor.zip
cd DjangoFloor/demo
mkvirtualenv -p `which python2.7` demo
python setup.py develop
demo-manage migrate
demo-manage collectstatic --noinput
demo-manage runserver
```

And open http://localhost:9000/test.html in your favorite browser.

If you want to test with Celery, please install a Redis server (it should listen on 127.0.0.1:6379) and set *USE_CELERY* to *True* in *demo/defaults.py*. In another shell:

```
workon demo
demo-celery worker
```

If you want to test websockets:

```
workon demo
pip install djangofloor[websocket]
demo-manage runserver
```

2.3 Tutorial

This tutorial shows you how to create a new DjangoFloor-based website.

Basically, there are two ways for creating a new project: from scratch, or with StarterPyth.

2.3.1 Creating a new project from scratch

Let assume that our great idea is called *myproject*.

As always, we start by creating a new virtualenv:

```
mkvirtualenv myproject -p `which python2.7`
pip install djangofloor
```

We can create required files and directories:

```
mkdir -p myproject/myproject/tests myproject/myproject/static myproject/myproject/templates
cd myproject
touch myproject/models.py
echo "__version__ = '1.0.0'" > myproject/__init__.py

cat << EOF > myproject-manage.py
from djangofloor.scripts import manage
manage()
EOF
cat << EOF > myproject-gunicorn.py
from djangofloor.scripts import gunicorn
gunicorn()
```

```
EOF
cat << EOF > myproject-celery.py
from djangofloor.scripts import celery
celery()
EOF
cat << EOF > myproject-uwsgi.py
from djangofloor.scripts import uwsgi
uwsgi()
EOF

cat << EOF > setup.py
# -*- coding: utf-8 -*-
from setuptools import setup, find_packages
from myproject import __version__ as version
entry_points = {'console_scripts': ['myproject-manage = djangofloor.scripts:manage',
                                    'myproject-celery = djangofloor.scripts:celery',
                                    'myproject-uswgi = djangofloor.scripts:uswgi',
                                    'myproject-gunicorn = djangofloor.scripts:gunicorn']}
setup(
    name='myproject',
    version=version,
    entry_points=entry_points,
    packages=find_packages(),
    include_package_data=True,
    install_requires=['djangofloor'],
)
EOF

cat << EOF > myproject/defaults.py
FLOOR_PROJECT_NAME = 'myproject'
FLOOR_INSTALLED_APPS = ['myproject', ]
EOF
```

That's it!

Let's start playing :-):

```
python myproject-manage.py config
python myproject-manage.py collectstatic --noinput
python myproject-manage.py migrate
python myproject-manage.py runserver
```

Open your favorite browser and explore http://localhost:9000/.

2.3.2 Creating a new project with Starterpyth

It's a bit simpler:

```
mkvirtualenv myproject -p `which python2.7`
pip install starterpyth
starterpyth-bin
[some questions...]
cd myproject
python myproject-manage.py config
python myproject-manage.py collectstatic --noinput
python myproject-manage.py migrate
python myproject-manage.py runserver
```

2.4 Global architecture

A complete production server is based on several components:

- pure web server: apache or nginx (or any equivalent),
- main database: mysql or postgresql (or any SQL database supported by Django),
- secondary database, for session and cache: redis,
- daemon controler: systemd or supervisor, allowing to automatically start (or restart) services,
- application server: gunicorn or uwsgi,
- a celery worker,
- some regular tasks (cleaning sessions, backup).

The webserver must handle three kinds of paths:

- static and media files (directly handled),
- websockets (passed to uwsgi in websocket mode, only if you use websockets),
- classical Django views (served in reverse-proxy mode).

The complete architecture is incomplete without other classical development tasks:

- unitary/non-regression testing (with classical tools like tox),
- documentation (and DjangoFloor generates a customized documentation),
- packages (pure-Python packages or Debian ones),
- backup tasks (information on how to backup your data are given in the documentation),
- a configuration cleanly separated from the code (DjangoFloor provides such a mechanism).

2.5 Included dependencies

Here is a list of the dependencies, with a short description. All these dependencies are plain Python, they can be installed without compiling C extensions.

2.5.1 Django

Ok, you should already know Django ;)

2.5.2 Gunicorn

Gunicorn 'Green Unicorn' is a Python WSGI HTTP Server for UNIX. It's a pre-fork worker model ported from Ruby's Unicorn project. The Gunicorn server is broadly compatible with various web frameworks, simply implemented, light on server resources, and fairly speedy.

Gunicorn is in pure Python, very simple to use and allow an easy production deployment.

2.5.3 Bootstrap3

DjangoFloor comes with Bootstrap3 and JQuery. Bootstrap3 is the most popular HTML, CSS, and JS framework for developing responsive, mobile first projects on the web You obtain nice pages out-of-box. The drawback is of course that many websites have the same look.

Required dependencies:

- django-bootstrap3.

2.5.4 Django-Debug-Toolbar

Django-Debug-Toolbar is a configurable set of panels that display various debug information about the current request/response.

Required dependencies:

- django-debug-toolbar.

2.5.5 CSS and JS files

Pipeline is an asset packaging library for Django, providing both CSS and JavaScript concatenation and compression, built-in JavaScript template support, and optional data-URI image and font embedding.

Required dependencies:

- django-pipeline,
- rcssmin (directly included in djangofloor.df_pipeline to ease installation - the official package require the compilation of a C extension).

By default, *PIPELINE_ENABLED* is set to *False*, as some bugs can appear in JS. Several JS minimifiers exist, even pure-Python ones :

- *jsmin*,
- *slimit*,
- *css-html-js-minify*.

2.5.6 Redis

Traditionnal SQL databases (PostgreSQL or MySQL, for example) are very good at storing your persistent data. However, there are several usages for which such databases are not recommended:

- caching web pages,
- broker for Celery,
- websockets (currently only with Python 2.6/2.7),
- session storage.

All these usages can be fullfilled by some other technology (for example, AMQP as Celery brocker, memcached for... cache and your default SQL engine for sessions). Redis is performant and very easy to use, and it limits the number of different systems in your architecture.

Several dependencies allow to properly use Redis:

- django-redis,

- django-redis-cache.

If you can use C extensions, you should also install (and use) django-redis-sessions-fork.

Some settings are related to Redis:

- **all values:**

```
REDIS_HOST = 'localhost'  # valid by default
REDIS_PORT = 6379  # valid by default
```

- **Celery:**

```
USE_CELERY = True
CELERY_TIMEZONE = '{TIME_ZONE}'  # valid by default
BROKER_DB = 13
BROKER_URL = 'redis://{REDIS_HOST}:{REDIS_PORT}/{BROKER_DB}'  # valid by default
CELERY_APP = 'djangofloor'  # valid by default
CELERY_CREATE_DIRS = True  # valid by default
```

- **cache:**

```
CACHES = {
    'default': {
        'BACKEND': 'redis_cache.RedisCache',
        'LOCATION': '{REDIS_HOST}:{REDIS_PORT}',
    },
}
```

- **sessions:**

```
SESSION_ENGINE = 'redis_sessions.session'
SESSION_REDIS_PREFIX = 'session'  # valid by default
SESSION_REDIS_HOST = '{REDIS_HOST}'  # valid by default
SESSION_REDIS_PORT = '{REDIS_PORT}'  # valid by default
SESSION_REDIS_DB = 10  # valid by default
```

- **websockets emulation (if you cannot use native websockets):**

```
WS4REDIS_EMULATION_INTERVAL = 1000  # (in ms, you should not set it below 500 or 1,000)
WEBSOCKET_URL = '/ws/'  # valid by default
```

- **websockets:**

```
FLOOR_USE_WS4REDIS  # should automatically set to `True`
WEBSOCKET_URL = '/ws/'  # valid by default
WS4REDIS_DB = 15
WS4REDIS_CONNECTION = {'host': '{REDIS_HOST}', 'port': '{REDIS_PORT}', 'db': WS4REDIS_DB, }
WS4REDIS_EXPIRE = 0  # valid by default
WS4REDIS_PREFIX = 'ws'  # valid by default
WS4REDIS_HEARTBEAT = '--HEARTBEAT--'  # valid by default
WSGI_APPLICATION = 'ws4redis.django_runserver.application'  # valid by default
WS4REDIS_SUBSCRIBER = 'djangofloor.df_ws4redis.Subscriber'  # valid by default
FLOOR_WS_FACILITY = 'djangofloor'  # valid by default
```

2.5.7 Websockets

Currently, only Python 2.6/2.7 allow to use websockets.

These dependencies are required:

- django-websocket-redis,
- gevent,
- uwsgi.

2.5.8 Celery

Celery is an asynchronous task queue/job queue based on distributed message passing. It is focused on real-time operation, but supports scheduling as well. The execution units, called tasks, are executed concurrently on a single or more worker servers using multiprocessing, Eventlet, or gevent. Tasks can execute asynchronously (in the background) or synchronously (wait until ready).

This dependency is required:

- celery.

You must launch a Celery worker:

```
djangofloor-celery --dfproject myproject worker
myproject-celery worker
```

2.5.9 Authentication

django-allauth is an integrated set of Django applications addressing authentication, registration, account management as well as 3rd party (social) account authentication.

This dependency is required:

- django-allauth.

2.6 Integrate your settings

In a standard Django project, you add your own settings in a single *settings.py* file. With DjangoFloor, you rather use a *defaults.py* file, which is merged with defaults settings offered by DjangoFloor. You can use it exactly as the traditionnal *settings.py* file.

The only difference stands for the *INSTALLED_APPS* setting (see below).

2.6.1 Project name

This settings is essentially decorative. Set *FLOOR_PROJECT_NAME* to your project name. It will be shown on the default index page.

2.6.2 Other Django applications

Since DjangoFloor comes with several applications, you should rather use *FLOOR_INSTALLED_APPS* in `myproject.defaults` to add your extra applications. The actual *INSTALLED_APPS* will be equal to `djangofloor.defaults.INSTALLED_APPS + myproject.defaults.OTHER_ALLAUTH myproject.defaults.FLOOR_INSTALLED_APPS`.

2.6.3 URL configuration

DjangoFloor comes with several configured URLs (check `djangofloor.root_urls` to see them). Add your own URL into a `list`, for example in file *myproject/urls.py*:

```
my_urls = [
    (r'^test.html$', 'myproject.views.test'),
]
```

Then define the setting *FLOOR_URL_CONF* to `myproject.urls.my_urls`. Now, you can define all your views, as explained in the doc.

2.6.4 Index view

The website index can be defined with the *FLOOR_INDEX* setting.

2.7 Signals

Communications between Python world and Javascript one is often awful, with a lot of boilerplate. The main goal of my implementation of signals is to really limit this code and to easily integrate Celery and websockets.

In a standard web application, you have four kind of codes:

- JavaScript (can call Django views or communicate with websockets)
- Python code in Django views (receives a HttpRequest as argument and returns a HttpResponse, can call Celery tasks)
- Python code in Celery tasks (receives some arguments and often return None, can only call other tasks, cannot communicate with Django, websocket or JS)
- Python code in the websocket world (can call some Celery tasks but cannot perform any blocking call or call Django views)

In DjangoFloor, a signal is a simple name (a unicode string). You connect some code (JS or Python) to this name and then you can call this new signal (with some arguments).

2.7.1 Defining a signal

We want to create a signal named *demo.my_signal*, and we want to display all the arguments provided when this signal is called.

Python example (create a file called *signals.py* in one of the *INSTALLED_APPS*:

```
from djangofloor.decorators import connect
@connect(path='demo.my_signal')
def my_signal(request, arg1, arg2, arg3):
    [ some interesting code ]
    print('blablabla', arg1, arg2, arg3)
```

A lot of computation, and this code must be used through Celery?:

```
from djangofloor.decorators import connect
@connect(path='demo.my_signal', delayed=True)
def my_signal(request, arg1, arg2, arg3):
```

```
    [ some interesting code ]
    print('blablabla', arg1, arg2, arg3)
```

Do not forget to run a Celery worker! You can allow non-connected users to trigger the signal:

```
from djangofloor.decorators import connect
@connect(path='demo.my_signal', auth_required=False)
def my_signal(request, arg1, arg2, arg3):
    [ some interesting code ]
    print('blablabla', arg1, arg2, arg3)
```

JavaScript example:

```
df.connect('demo.my_signal', function (options) { alert('blablabla' + options.arg1 + options.arg2 + o
```

Ok, that is enough to connect any code to a signal.

2.7.2 Calling a signal

There are three cases:

- you can use websockets with *django-websocket-redis* (currently requiring Python 2),
- you cannot use websockets but you have a Redis server,
- you cannot use websockets, nor you have a Redis server.

With websockets

This is the simplest case: if you want to call this code in Python:

```
from djangofloor.tasks import call, WINDOW
call('demo.my_signal', request, WINDOW, arg1='argument', arg2='other value', arg3=42)
```

And in Javascript?:

```
df.call('demo.my_signal', {arg1: 'argument', arg2: 'other value', arg3: 42})
```

Any Python or Javascript can call any Python or JavaScript signal, with (almost) the same syntax. If more than one code is connected to the same signal, then all codes will be called (both JS and Python).

This mode is activated if *settings.FLOOR_USE_WS4REDIS* is *True*. By default, *djangofloor* tries to import *ws4redis* before settin
but you can override this behaviour.

sharing signals

When Javascript calls a signal, it is called on client-side and on server-side (if required). When Python calls a signal, there are more possibilities to propagate it with the *sharing* argument:

- only to Python with *sharing=None*,
- to Python and only to the original Javascript window with *sharing=WINDOW*,
- to Python and only to the original Javascript session with *sharing=SESSION*,
- to Python and to all connected users with *sharing=BROADCAST*,
- to Python and to the original user with *sharing=USER* (he can receive multiple instances of this signal with multiple browser windows),

- to Python and to a limited set of users with *sharing={USER: ['username1', 'username2',]}*.

You can prevent Python code from calling the javascript side of calls *call('demo.my_signal', request, None, **kwargs)*

Note: sometimes, signals are to properly propagated to client when using USER or SESSION.

Without Redis nor websockets

This is the hardest case. You can still you signals through standard HTTP requests:

- Javascript can call both Javascript and Python code,
- Python code called from JS can only return a list of dict *{'signal': 'signal.name', 'options': kwargs}* (see example below), which will be called in JS
- you cannot use delayed Python signals (because they require Celery),

So, the only valid patterns are:

- Javascript that calls Javascript
- Javascript that calls Python that returns several Javascript calls

```
@connect(path='demo.test_form')
def test_form(request, form):
    form = SerializedForm(SimpleForm)(form)
    if form.is_valid() and form.cleaned_data['first_name']:
        return [{'signal': 'df.messages.info', 'options': {'html': 'Your name is %s' % form.cleaned_
    return [{'signal': 'df.messages.error', 'options': {'html': 'Invalid form. You must provide your
```

Without websockets but with Redis

In addition of the degraded mode without Redis, using a Redis server allows to more things:

- use delayed (Celery) tasks,
- call Javascript signals anywhere from Python code (including in delayed tasks) by activating a regular polling from JS. However, you can only activate JS signals on a given session (probably the one that sent the first *SignalRequest*).

You activate a regular polling by setting *WS4REDIS_EMULATION_INTERVAL* to a positive value. This interval is in milliseconds! Do not set it below 500 or 1,000 if you do not want to flood your webserver. Leave it to 0 to desactivate this behaviour. With this polling, you can emulate an almost complete websocket behaviour (with celery tasks sending signals to the client).

By default, this polling is also deactivated for anonymous users, even if you set it to a positive value.

If *WS4REDIS_EMULATION_INTERVAL* looks like *my_package.my_module.my_callable* or if it is callable, then it will be called with a *django.http.HttpRequest* as argument, and it must return a non-negative integer (interval in milliseconds).

2.7.3 Notes

- all signals defined in files *signals.py* of each app listed in INSTALLED_APPS are automatically taken into account (if some signals are defined elsewhere, you must import their modules into a *signals.py*),
- You can prevent specific Python code from being called by JS:

```
@connect(path='demo.my_signal', allow_from_client=False)
```

- Several functions (both JS and Python) can be connected to the same signal,
- Python calls require a *request*, which can be either a standard *django.http.HttpRequest* or a *djangofloor.decorators.SignalRequest*. *SignalRequest* propagates the username and the session key from call to call and is provided from a JS key.
- Required JS files (jquery, ws4redis and *js/djangofloor.js*) are defined in *settings.PIPELINE_JS*

2.8 Settings

IMHO, dealing with Django settings is a nightmare for at least two reasons. Even if you are building a custom website, some settings are common to both your developper instance and your production instance, and others settings are different. So, *you must maintain two differents files with lots of common parts.*

Moreover, setting file (with secret data like database passwords) is expected to be in your project. *You mix versionned files (the code) and non-versionned files (the settings), and any reinstallation can overwrite your settings.*

With DjangoFloor, you can forget these drawbacks. You always use *djangofloor.settings* as Django settings, but this module does not contain any settings. This module can smartly merge settings from three different sources:

- default DjangoFloor settings (*djangofloor.defaults*),
- default settings for your wonderful website (*myproject.defaults*, for settings like INSTALLED_APPS, MIDDLEWARES, and so on),
- local settings, specific to an instance (*[prefix]/etc/myproject/settings.py*) for settings like database infos,
- you can even define settings in a more traditionnal way, with .ini files.

Default DjangoFloor settings are overriden by your project defaults, which are overriden by local settings. Run *myproject-manage config* to display all configuration files, in order of precedence:

- Python local configuration: [...]/etc/archeolog_server/settings.py (defined in environment by DJANGOFLOOR_PYTHON_SETTINGS)
- INI local configuration: [...]/etc/archeolog_server/settings.ini (defined in environment by DJANGOFLOOR_INI_SETTINGS)
- Default project settings: [...]/[...]/defaults.py (defined in environment by DJANGOFLOOR_PROJECT_DEFAULTS)
- Other default settings: [...]/[...]/djangofloor/defaults.py

Existing files are displayed in blue, missing files are displayed in red. All Python settings files have the same syntax as the traditionnal Django settings file.

However, any string setting can use reference other settings through *string.Formatter*. An example should be clearer:

DjangoFloor default settings:

```
LOCAL_PATH = '/tmp/'
MEDIA_ROOT = '{DATA_PATH}/media'
STATIC_ROOT = '{DATA_PATH}/static'
LOG_ROOT = '{DATA_PATH}/logs'
```

your project default settings (which override DjangoFloor settings):

```
MEDIA_ROOT = '{DATA_PATH}/data'
```

your local settings (which have the top priority):

```
LOCAL_PATH = '/var/www/data'
STATIC_ROOT = '/var/www/static'
```

in the Django shell, you can check that settings are gracefully merged together:

```
>>> from django.conf import settings
>>> print(settings.LOCAL_PATH)  # overriden in local settings
/var/www/data
>>> print(settings.MEDIA_ROOT)  # overriden in project defaults but not in local settings
/var/www/data/data
>>> print(settings.STATIC_ROOT)  # overriden in local settings
/var/www/static
>>> print(settings.LOG_ROOT)  # reference LOCAL_PATH in DjangoFloor, which is overriden
/var/www/data/logs
```

2.8.1 How to use it?

Since your settings are expected to be in *myproject.defaults* and in *[prefix]/etc/myproject/settings.py*, DjangoFloor must to guess your project name *myproject*. There are three ways to let it know *myproject*:

- use one of the provided commands *djangofloor-celery*, *djangofloor-manage*, *djangofloor-gunicorn* or *djangofloor-uwsgi* with the option *–dfproject myproject*
- *export DJANGOFLOOR_PROJECT_NAME=myproject* before using *djangofloor-celery*, *djangofloor-manage*, *djangofloor-gunicorn* or *djangofloor-uwsgi*
- copy *djangofloor-celery*, *djangofloor-manage*, *djangofloor-gunicorn* or *djangofloor-uwsgi* as *myproject-[celery|manage|gunicorn|uwsgi*.

You can change *myproject.defaults* to another value with the environment variable *DJANGOFLOOR_PROJECT_SETTINGS* You can specify another local setting files with the option *–dfconf [path/to/settings.py]*

If you run *[prefix]/bin/myproject-manage*, then local settings are expected in *[prefix]/etc/myproject/settings.py*. If you run directly from the source (without installing), local settings are expected in *working_dir/my_project_configuration.py*.

2.8.2 And Pycharm (or other IDEs)?

PyCharm (and, I guess, many other IDEs) has built-in support for the Django framework and is able to use the settings module for a better auto-completion. However, it is not able to use such a complex system.

DjangoFloor can generate a merged settings file for you:

```
myproject-manage config --merge > pycharm_settings.py
```

Then you can use this file as Django settings in PyCharm.

2.8.3 Notes

- Only settings in capitals are taken into account.
- interpolation of settings is also recursively processed for dicts, lists, tuples and sets.
- If you have a settings MY_SETTING and another called MY_SETTING_HELP, the latter will be used as help for *manage.py config*.

2.8.4 Full list of settings

DjangoFloor define a few new settings.

- *FLOOR_INDEX*: django view your the website index,
- *FLOOR_INSTALLED_APPS*: list of extra Django apps (including yours),
- *FLOOR_PROJECT_NAME*: your project name,
- *FLOOR_URL_CONF*: your extra URL configuration,
- *FLOOR_FAKE_AUTHENTICATION_USERNAME*: set it to any username you want (allow to fake a HTTP authentication, like Kerberos). Only for debugging purposes,
- *FLOOR_FAKE_AUTHENTICATION_GROUPS*: set it to the names of the groups you want for the fake user. Only for debugging purposes,
- *FLOOR_WS_FACILITY*: websocket facility for the signal implementation,
- *FLOOR_USE_WS4REDIS*: automatically set if you installed ws4redis. If you manually change it:
  - add *ws4redis* in *INSTALLED_APPS*
  - add *ws4redis.context_processors.default* in *TEMPLATE_CONTEXT_PROCESSORS*
  - set *WSGI_APPLICATION* to *ws4redis.django_runserver.application*
- *FLOOR_DEFAULT_GROUP_NAME*: name of the default group for newly created users (when authenticated by the reverse proxy). Leave it to *None* to avoid this behavior.
- *USE_SCSS*: use SCSS scss compiler with Pipeline
- *LOCAL_PATH*: the base directory for all data,
- *BIND_ADDRESS*: the default bind address for the runserver command, or for gunicorn,
- *REDIS_HOST* and *REDIS_PORT*: this is self-explained,
- *THREADS*, *WORKERS*, *MAX_REQUESTS*: all these settings are related to gunicorn
- *REVERSE_PROXY_IPS*: the IPs of your reverse proxy, allowing authenticating users by the *REMOTE_USER* header

2.8.5 Using flat config files

If your application has a few settings available to the end-user (typically the coordinates of the database), you can also put them into a .ini file. However, this require a mapping between the option in the .ini file and the settings.

This dictionnary is expected in the file *myproject/iniconf.py*, with a single variable named *INI_MAPPING* which is a list of `djangofloor.iniconf.OptionParser`. For example:

```
INI_MAPPING = [
    OptionParser('DATABASE_ENGINE', 'database.engine'),
    OptionParser('DATABASE_NAME', 'database.name'),
    OptionParser('DATABASE_USER', 'database.user'),
    OptionParser('DATABASE_PASSWORD', 'database.password'),
    OptionParser('DATABASE_HOST', 'database.host'),
    OptionParser('DATABASE_PORT', 'database.port'),
]
```

In this case, DjangoFloor will look for a file *[prefix]/etc/myproject/settings.ini* with a section *database*, with the options *engine*, *name*, *user*, *password*, *host* and *port*:

```
[database]
host = localhost
user = my_user
password = my_secret_password
engine = django.db.backends.postgresql_psycopg2
```

The exact expected filename is always given by the command *myproject-manage config*.

2.8.6 Setting classes

DjangoFloor define a few special classes:

- Use *djangofloor.utils.DirectoryPath* to reference a directory. It takes a directory path (`str`) as argument. This string can reference any other existing setting (e.g., *DirectoryPath("{LOCAL_PATH}/static")*). The referenced directory will automatically be created on startup.
- Use *djangofloor.utils.FilePath* to reference a file. It takes a file path (`str`) as argument. This string can reference any other existing setting (e.g., *FilePath("{LOCAL_PATH}/data/database.db")*). The parent directory of the referenced file will automatically be created on startup.
- Use *djangofloor.utils.SettingReference* to reference another setting. It takes another setting name as argument.

```
A_SETTING = 0
B_SETTING = SettingReference('C_SETTING')
C_SETTING = A_SETTING
```

  If *A_SETTING* is overidden in another config file, *C_SETTING* is still equal to *0* while *B_SETTING* will always be equal to *A_SETTING*.

- Use *djangofloor.utils.ExpandIterable* to include a list in another one. It takes another setting name as argument.

```
A_SETTING = [0, 1, 2]
B_SETTING = [ExpandIterable('C_SETTING'), 3, 4]
```

  B_SETTING will be equal to *[0, 1, 2, 3, 4,]*. If *A_SETTING* is overidden, then all elements of *A_SETTING* will be included before *3, 4* in *B_SETTING*.

- Use *djangofloor.utils.CallableSetting* to define a setting on startup. It takes a callable as argument. This callable takes a dict as argument: keys are parsed setting names and values are their values.

```
A_SETTING = True
B_SETTING = False
C_SETTING = CallableSetting(lambda dict_: dict_['A_SETTING'] or dict_['B_SETTING'])
D_SETTING = None
```

  The callable used by *C_SETTING* can only use *A_SETTING* and *B_SETTING*.

```
A_SETTING = True
B_SETTING = False
C_SETTING = CallableSetting(lambda dict_: dict_['A_SETTING'] or dict_['B_SETTING'], 'D_SETTING')
D_SETTING = None
E_SETTING = None
```

  The callable used by *C_SETTING* can now use *D_SETTING*, as its name is given to *CallableSetting*.

2.8.7 Parsing order

Settings names are sorted before being parsed, but if a setting references another one, the latter will be imported.

```
A_SETTING = 0
B_SETTING = '{C_SETTING}'
C_SETTING = 2
```

Of course, *B_SETTING* will be equal to *"2"* on startup. The parsing order will be *A_SETTING*, *C_SETTING* and *B_SETTING* (since *B_SETTING* requires *C_SETTING*).

2.9 Deployment

Deploying Django is merely complex if we follow the official guide . In this small guide, we only support the wsgi. WSGI can be used with two application servers: gunicorn and uwsgi, behind a pure webserver: nginx or apache.

Do not forget to read the official doc!

I will only cover the deployment with wsgi through gunicorn (installed as a dependency) or uwsgi (optional). Gunicorn is a pure-Python application, but on the other hand uwsgi is maybe more efficient and allows websockets.

settings due to the reverse proxy:

- ALLOWED_HOSTS = ('IP of the reverse proxy', 'its DNS name',)
- REVERSE_PROXY_IPS = ('IP of the reverse proxy',)
- USE_X_SEND_FILE = True if you use Apache, X_ACCEL_REDIRECT if you use nginx
- USE_X_FORWARDED_HOST = True
- SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')

settings for SSL/HTTPS:

- SECURE_SSL_REDIRECT = True
- SESSION_COOKIE_SECURE = True
- CSRF_COOKIE_SECURE = True
- CSRF_COOKIE_HTTPONLY = True

You can also check cipherli for good Apache/nginx configurations.

settings for HTTP authentication (Kerberos/Shibboleth/SSO/...)

- FLOOR_AUTHENTICATION_HEADER = 'HTTP_REMOTE_USER'

settings for Redis:

- REDIS_HOST = 'localhost'
- REDIS_PORT = '6379'

2.9.1 application

The first thing to do is to create a virtualenv and install your project inside. If your project is uploaded to pypi or to an internal mirror:

```
mkvirtualenv myproject_prod
pip install myproject_prod
```

Otherwise, you can just copy the source (or git clone the project):

```
mkvirtualenv myproject_prod
cd myproject/
python setup.py install
```

Then you can configure it:

```
myproject-manage config
# look on the first lines to check the location of the Python local config file
# create this file and set the right settings (database, LOCAL_PATH, ...)
myproject-manage migrate
# create the database and the tables
myproject-manage collectstatic --noinput
# generate all static files
myproject-manage createsuperuser
# create a first user with admin rights
```

2.9.2 backup

A basic DjangoFloor application is made of different kinds of files:

- the code of your application and its dependencies (you should not have to backup them),
- static files (as they are provided by the code, you can lost them),
- configuration files (you can easily recreate it, or you must backup it),
- database content (you must backup it),
- media files (you also must backup them).

database backup

DjangoFloor comes with a command to dump the database. You can combine it with the *logrotate* utility.

```
cat << EOF > [prefix]/etc/myproject/dbrotate.conf
/backupdirectory/dbbackup.sql.gz {
    rotate 5
    nocompress
    dateext
    dateformat _%Y-%m-%d
    extension .sql.gz
    missingok
}
EOF
mkdir -p [prefix]/var/myproject/

myproject-manage dumpdb | gzip > /backupdirectory/dbbackup.sql.gz && logrotate -s [prefix]/var/mypro
```

The last command should be in crontab to be regularly launched.

media files backup

Media files can be backuped with two distinct strategies:

- generate a single tar.gz archive (takes a lot of disk space),

- synchronize the folder with another one (say, on a NFS) with *rsync*.

A good strategy is to run the rsync command daily with a monthly tar.gz archive:

```
cat << EOF > [prefix]/etc/myproject/mediarotate.conf
/backupdirectory/mediabackup.tar.gz {
    rotate 5
    nocompress
    dateext
    dateformat _%Y-%m-%d
    extension .tar.gz
    missingok
}
EOF
mkdir -p [prefix]/var/myproject/
SRC=`python manage.py config -m | grep MEDIA_ROOT | cut -f 3 -d ' '`
tar -C $SRC -czf /backupdirectory/mediabackup.tar.gz . && logrotate -s [prefix]/var/myproject/mediarc
rsync -arltDE $SRC /backupdirectory/media
```

2.9.3 gunicorn

Gunicorn is an easy-to-use application server:

```
myproject-gunicorn
```

Or, if you wan to daemonize (but you really should prefer to use systemd/supervisor or launchd):

```
myproject-gunicorn -D
```

2.9.4 uwsgi

Since uwsgi requires compilation, it is not installed as DjangoFloor dependency, but it can be installed with pip:

```
pip install uwsgi
```

And then run:

```
myproject-uwsgi
```

2.9.5 Apache

Here is a simple configuration file for your project behind Apache, assuming that LOCAL_PATH is set to "/var/www/myproject" in your settings:

```
<VirtualHost *:80>
    ServerName my.project.com
    Alias               /static/    /var/www/myproject/static/
    Alias               /media/     /var/www/myproject/media/
    ProxyPass           /static/    !
    ProxyPass           /media/     !
    ProxyPass           /           http://localhost:9000/
    ProxyPassReverse    /           http://localhost:9000/
    DocumentRoot        /var/www/myproject/static/
    ServerSignature     off
</VirtualHost>
```

2.9.6 Nginx

Here is a simple configuration file for your project behind Nginx, assuming that LOCAL_PATH is set to "/var/www/myproject" in your settings:

```
server {
    listen       80;
    server_name  my.project.com;
    location /static/ {
        alias            /var/www/myproject/static/;
    }
    location /media/ {
        alias            /var/www/myproject/media/;
    }
    location / {
        proxy_pass       http://localhost:9000/;
        proxy_set_header Host               $host:$proxy_port;
        proxy_set_header X-Real-IP          $remote_addr;
        proxy_set_header X-Forwarded-Host   $host:$proxy_port;
        proxy_set_header X-Forwarded-Server $host;
        proxy_set_header X-Forwarded-For    $proxy_add_x_forwarded_for;
    }
}
```

2.9.7 supervisor

A single config file for Supervisor can handle all processes to launch:

```
PROJECT_NAME=myproject
VIRTUAL_ENV=$VIRTUAL_ENV
USER=www-data
cat << EOF | sudo tee /etc/supervisor.d/$PROJECT_NAME.conf
[program:${PROJECT_NAME}_gunicorn]
command = $VIRTUAL_ENV/bin/$PROJECT_NAME-gunicorn
user = $USER
[program:${PROJECT_NAME}_celery]
command = $VIRTUAL_ENV/bin/$PROJECT_NAME-celery worker
user = $USER
EOF
```

2.9.8 systemd (Linux only)

Most distribution are now using systemd for starting services:

```
PROJECT_NAME=myproject
VIRTUAL_ENV=$VIRTUAL_ENV
USER=www-data

cat << EOF | sudo tee /etc/systemd/system/$PROJECT_NAME-gunicorn.service
[Unit]
Description=$PROJECT_NAME Gunicorn process
After=network.target
[Service]
User=$USER
Group=$USER
WorkingDirectory=$VIRTUAL_ENV
```

```
ExecStart=$VIRTUAL_ENV/bin/$PROJECT_NAME-gunicorn
ExecReload=/bin/kill -s HUP $MAINPID
ExecStop=/bin/kill -s TERM $MAINPID
[Install]
WantedBy=multi-user.target
EOF

cat << EOF | sudo tee /etc/systemd/system/$PROJECT_NAME-celery.service
[Unit]
Description=$PROJECT_NAME Celery worker process
After=network.target
[Service]
User=$USER
Group=$USER
WorkingDirectory=$VIRTUAL_ENV
ExecStart=$VIRTUAL_ENV/bin/$PROJECT_NAME-celery worker
[Install]
WantedBy=multi-user.target
EOF

sudo systemctl restart $PROJECT_NAME-gunicorn
sudo systemctl enable $PROJECT_NAME-gunicorn
sudo systemctl restart $PROJECT_NAME-celery
sudo systemctl enable $PROJECT_NAME-celery
```

2.9.9 launchd (Mac OS X only)

We need to create a config file for each process to launch:

```
PROJECT_NAME=myproject
VIRTUAL_ENV=$VIRTUAL_ENV
cat << EOF > ~/Library/LaunchAgents/$PROJECT_NAME.gunicorn.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd"
<plist version="1.0">
  <dict>
    <key>KeepAlive</key>
    <true/>
    <key>Label</key>
    <string>$PROJECT_NAME-gunicorn</string>
    <key>ProgramArguments</key>
    <array>
      <string>$VIRTUAL_ENV/bin/$PROJECT_NAME-gunicorn</string>
    </array>
    <key>EnvironmentVariables</key>
    <dict>
    </dict>
    <key>RunAtLoad</key>
    <true/>
    <key>WorkingDirectory</key>
    <string>/usr/local/var</string>
    <key>StandardErrorPath</key>
    <string>/dev/null</string>
    <key>StandardOutPath</key>
    <string>/dev/null</string>
  </dict>
</plist>
```

```
EOF
cat << EOF > ~/Library/LaunchAgents/$PROJECT_NAME.celery.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd"
<plist version="1.0">
  <dict>
    <key>KeepAlive</key>
    <true/>
    <key>Label</key>
    <string>$PROJECT_NAME-celery</string>
    <key>ProgramArguments</key>
    <array>
      <string>$VIRTUAL_ENV/bin/$PROJECT_NAME-celery</string>
      <string>worker</string>
    </array>
    <key>EnvironmentVariables</key>
    <dict>
    </dict>
    <key>RunAtLoad</key>
    <true/>
    <key>WorkingDirectory</key>
    <string>/usr/local/var</string>
    <key>StandardErrorPath</key>
    <string>/dev/null</string>
    <key>StandardOutPath</key>
    <string>/dev/null</string>
  </dict>
</plist>
EOF
```

In this case, your project run as the current logged user. Maybe you should use a dedicated user.

2.10 Packaging your application

2.10.1 Python

The simplest way for packaging is to use built-in Python tools.

```
cd myproject
workon myproject
python setup.py sdist
```

2.10.2 Debian / Ubuntu

Creating a Debian package for a Djangofloor project, like any other Django project, requires to also package all dependencies.

packaging dependencies

You should use DebTools to create packages for all dependencies in a single command. Creating all packages is quite simple:

1. create a virtualenv and install your application (so all your dependencies are also installed),

2. create a generic *stdeb.cfg* file for all Debian-based distributions,
3. create a *stdeb-distribution.cfg* specific to each distribution,
4. in these files, create a section for each Python package requiring specific treatment,
5. call *multideb -r -v -x stdeb-distribution.cfg*,
6. wait (it takes almost one hour on a slow computer) while all Python modules are packaged.

In the *demo* application, a generic *stdeb.cfg* file and specific files for Debian 7/8 and Ubuntu 14.04+ are provided. The following packages require specific treatment to be packaged as .deb files:

- *djangofloor* (dependencies for Python 3),
- *anyjson* (needs to delete the *tests* directory before packaging),
- *requests-oauthlib* (needs to delete the *tests* directory before packaging),
- *celery* (needs to delete the *docs* directory before packaging),
- *pathlib* (needs to change the *MANIFEST.in* file),
- *msgpack-python* (needs to change the package name and to change the *MANIFEST.in* file),
- *gunicorn* must be installed in version *18.0* on Debian 7.

If you do not have more dependencies, you can juste reuse the files bundled with the *demo* app. Their names should be explicit enough.

packaging your project

Packaging your project cannot be done with the standard *bdist_deb* command, provided by the *stdeb* package:

- static files must be collected,
- configuration file must be prepared in */etc/my_project/settings.ini*,
- configuration file is required for Apache or Nginx,
- configuration file is required for Supervisor or Systemd.

Djangofloor provides a new command *bdist_deb_django* that do all these things for you. It only requires a new section *my_project* in your *stdeb.cfg* file. This section can also be in the files specific to each distribution. In addition to the options used by the original *bdist_stdeb* command, you can add this

```
[my_project]
; a list of processes to run (like the gunicorn process, or the celery worker)
processes = myproject-gunicorn:/usr/bin/myproject-gunicorn
    myproject-celery:/usr/bin/myproject-celery worker
; the frontend server (valid values are currently apache2.2, apache2.4 or nginx)
frontend = nginx
; the process manager (valid values are currently supervisor or systemd)
process_manager = supervisor
; the username used for the process (should be my_project).
username = myproject
; extra post-inst. script, appended to the default one (Python 2 only)
extra_postinst = path_of_postinstallation_script_for_python2.sh
; extra post-inst. script, appended to the default one (Python 3 only)
extra_postinst3 = path_of_postinstallation_script_for_python.sh
; uncommon option
preinst = path_of_preinstallation_script_for_python2.sh
; uncommon option
preinst3 = path_of_preinstallation_script_for_python3.sh
```

```
; uncommon option
postinst = path_of_postinstallation_script_for_python2.sh
; uncommon option
postinst3 = path_of_postinstallation_script_for_python3.sh
; uncommon option
prerm = path_of_preremove_script_for_python2.sh
; uncommon option
prerm3 = path_of_preremove_script_for_python3.sh
; uncommon option; override the default postinst script
postrm = path_of_postremove_script_for_python2.sh
; uncommon option; override the default postinst script
postrm3 = path_of_postremove_script_for_python3.sh
```

Then you can create your .deb:

```
rm -rf `find * | grep pyc$`
python setup.py bdist_deb_django -x stdeb-distribution.cfg
```

Again, you should take a look the files provided with the *demo* application (the same files are used to create .deb packages for the dependencies and for the application).

installing your project

If you cannot use a private mirror, just put all .deb files on your server and run:

```
sudo dpkg -i deb/python3-*.deb  # for a Python3 project
sudo dpkg -i deb/python-*.deb   # for a Python2 project
```

You should configurate your project by tweaking */etc/apache2/sites-available/my_project.conf* and */etc/my_project/settings.ini*.

```
sudo a2ensite my_project.conf
sudo a2dissite 000-default
sudo -u my_project my_project-manage migrate
sudo service supervisor restart
sudo service apache2 restart
```

2.10.3 RedHat / CentOS / Scientific Linux

TODO

2.11 Documentation generation

Documentation is an important part of a software. DjangoFloor provides a basic, templated, documentation that can be automatically generated for your project. Required files for generating Debian packages are also provides.

```
python manage.py gen_dev_files my_root
```

You can easily adapt this basic documentation to your own needs. You just have to create in your app *my_app* a folder named *templates/my_app/dev*. All files in this folder will be templated and written to the given root (most probably .).

Here is the list of the default generated files (you can find them in *djangofloor/templates/djangofloor/dev*.

```
-- debian-7-python3.sh
-- debian-8-python3.sh
-- doc
|   -- Makefile
|   -- make.bat
|   -- source
|       -- conf.py
|       -- configuration.rst
|       -- debian.rst
|       -- index.rst
|       -- installation.rst
-- stdeb-debian-7.cfg
-- stdeb-debian-8.cfg
-- stdeb.cfg
-- debian-8_ubuntu-14.10-15.10-python3.sh
```

Assume that you have the following tree in *my_app/templates/my_app/dev*:

```
-- doc
    -- source
        -- extra_file.rst
        -- index.rst
```

Your folder and DjangoFloor's one will be merged according to these rules:

- any file existing in only one of these folder (like *doc/source/conf.py* or *doc/source/extra_file.rst*) will be written,
- if a file is defined in both folders (like *doc/source/index.rst*), then your file overrides DjangoFloor's one,
- an overriden file with an empty content will be ignored (so you can make some default files ignored).

All files are templated using the Django template system. You can override only parts of the default files.

```
{% extends 'djangofloor/dev/doc/source/index.rst' %}
{% block description %}
This is a description of a project
{% endblock %}
```

If a filename ends with '_tpl', then this suffix is entirely ignored and the file. This prevents scripts (like *setup.py*) to compile them.

2.12 Python API Documentation

Here is the documentation of the complete API.

2.12.1 djangofloor.backends

Authentication backend used for HTTP authentication

Check `django.contrib.auth.backends.RemoteUserBackend` for a more detailed explanation.

Automatically add a specified group to newly-created users. The name of this default group is defined by the setting FLOOR_DEFAULT_GROUP_NAME. Set it to *None* if you do not want a default group for new users.

class djangofloor.backends.**DefaultGroupRemoteUserBackend**

configure_user(*user*)
Configures a user after creation and returns the updated user.

By default, returns the user unmodified; only add it to the default group.

2.12.2 djangofloor.context_processors

define ContextProcessors

The only ContextProcessor defined add some common variables related to DjangoFloor.

djangofloor.context_processors.**context_base**(*request*)
Provide the following variables to templates when you RequestContext:

- 'df_remote_authenticated': *True* if the user has been authenticated by *djangofloor.middleware.RemoteUserMiddleware*.
- 'df_project_name': name of your project, as defined in settings.FLOOR_PROJECT_NAME,
- 'df_user': the user (`django.contrib.auth.models.AbstractUser`),
- 'df_language_code': settings.LANGUAGE_CODE (`str`),
- 'df_user_agent': the User Agent or '' if not defined in the request (`str`),
- 'df_index_view': the default view name (`str`)
- 'df_window_key': a random value, unique to the window and allowing to identify successive JS calls (`str`)

Parameters **request** (`django.http.HttpRequest`) – a HTTP request

Returns a dict to update the global template context

Return type `dict`

2.12.3 djangofloor.decorators

Connect Python code to the DjangoFloor signal system

Define the decorator for connecting Python code to signals, a *djangofloor.decorators.SignalRequest* which can be easily serialized and is lighter than a `django.http.HttpRequest`, and some code to convert serialized data sent by Javascript to something useful in Python.

Using the decorator

The decorator *djangofloor.decorators.connect()* let you connect any Python function to a signal (which is only a text string). Signals are automatically discovered, as soon as there are in files *signals.py* in any app listed in the setting *INSTALLED_APPS* (like *admin.py* or *models.py*).

To use this decorator, create the file *myproject/signals.py*:

```
from djangofloor.decorators import connect
@connect(path='myproject.signals.demosignal')
def my_function(request, arg1, arg2):
    print(arg1, arg2)
```

Serialization/deserialization

Since all arguments must be serialized (in JSON), only types which are acceptable for JSON can be used as arguments for signals.

Using Python3

Python3 introduces the notion of function annotations. DjangoFloor can use them to deserialize received data sent by JS:

```
@connect(path='myproject.signals.demosignal')
def my_function(request, arg1: int, arg2: float):
    print(arg1, arg2)
```

The annotation can be any callable, which raise ValueError (in case of error ;)) or a deserialized value. DjangoFlor define several callable to handle common cases:

- checking if a string matches a regexp: *djangofloor.decorators.RE*,
- checking if a value is one of several choices: *djangofloor.decorators.Choice*,
- handle form data sent by JS as a plain Django form.

class djangofloor.decorators.**Choice**(*values*, *caster=None*)
used to check if a value is among some valid choices.

Example (requires Python 3.2+), for a function that can only two values:

```
    @connect(path='myproject.signals.test')
    def test(request, value: Choice([True, False])):
        pass
```

Your code wan't be called if value is not True or False.

Parameters **caster** – callable to convert the provided deserialized JSON data before checking its validity.

class djangofloor.decorators.**RE**(*value*, *caster=None*, *flags=0*)
used to check in a string value match a given regexp.

Example (requires Python 3.2+), for a function that can only handle a string on the form 123a456:

```
    @connect(path='myproject.signals.test')
    def test(request, value: RE("\d{3}a\d{3}")):
        pass
```

Your code wan't be called if value has not the right form.

Parameters

- **value** (*str*) – the pattern of the regexp
- **caster** (*callable* or *None*) – if not *None*, any callable applied to the value (if valid)
- **flags** (*int*) – regexp flags passed to *re.compile*

class djangofloor.decorators.**RedisCallWrapper**(*fn*, *path=None*, *delayed=False*, *allow_from_client=True*, *auth_required=True*)

prepare_kwargs(*kwargs*)

register(*path*)

class djangofloor.decorators.**SerializedForm**(*form_cls*)
Transform values sent by JS to a Django form.

Given a form and a `list` of `dict`, transforms the `list` into a `django.http.QueryDict` and initialize the form with it.

```
>>> class SimpleForm(forms.Form):
...     field = forms.CharField()
...
>>> x = SerializedForm(SimpleForm)
>>> form = x([{'field': 'object'}])
>>> form.is_valid()
True
```

How to use it with Python3:

```
@connect(path='myproject.signals.test')
def test(request, value: SerializedForm(SimpleForm), other: int):
    print(value.is_valid())
```

How to use it with Python2:

```
@connect(path='myproject.signals.test')
def test(request, value, other):
    value = SerializedForm(SimpleForm)(value)
    print(value.is_valid())
```

On the JS side, the easiest way is to serialize the form with JQuery:

```
<form onsubmit="return df.call('myproject.signals.test', {value: $(this).serializeArray(), other
    <input name='field' value='test' type='text'>
</form>
```

class djangofloor.decorators.**SignalRequest**(*username*, *session_key*, *user_pk=None*, *is_superuser=False*, *is_staff=False*, *is_active=False*, *perms=None*, *window_key=None*)
Store the username and the session key and must be supplied to any Python signal call.

Can be constructed from a standard `django.http.HttpRequest`.

Parameters

- **username** (`str`) – should be User.username
- **session_key** (`str`) – the session key, unique to a opened browser window (useful if a user has multiple windows)
- **user_pk** (`int`) – primary key of the user (for easy ORM queries)
- **is_superuser** (`bool`) – is the user a superuser?
- **is_staff** (`bool`) – belongs the user to the staff?
- **is_active** (`bool`) – is the user active?
- **perms** (`list`) – list of "app_name.permission_name" (optional)
- **window_key** (`str`) – a string value specific to each opened browser window/tab

classmethod from_request(*request*)
return a *djangofloor.decorators.SignalRequest* from a `django.http.HttpRequest`.

If the request already is a *djangofloor.decorators.SignalRequest*, then it is returned as-is (not copied).

Parameters **request** (django.http.HttpRequest or *djangofloor.decorators.SignalRequest*) – standard Django request

Returns a valid request

Return type *djangofloor.decorators.SignalRequest*

classmethod from_user(*user*)

Create a *djangofloor.decorators.SignalRequest* from a valid User. The SessionKey is set to *None*

Parameters **user** (django.contrib.auth.models.AbstractUser) – any Django user

Return type *djangofloor.decorators.SignalRequest*

has_perm(*perm*)

return true is the user has the required perm.

```
>>> r = SignalRequest('username', perms=['app_label.codename'])
>>> r.has_perm('app_label.codename')
True
```

Parameters **perm** – name of the permission ("app_label.codename")

Returns True if the user has the required perm

Return type bool

perms

set of all perms of the user (set of "app_label.codename")

template_perms

dict of perms, to be used in templates.

Example:

```
{% if request.template_perms.app_label.codename %}...{% endif %}
```

to_dict()

Convert this *djangofloor.decorators.SignalRequest* to a dict which can be provided to JSON.

Returns a dict ready to be serialized in JSON

Return type dict

class djangofloor.decorators.**ViewWrapper**(*fn*, *path=None*)

register(*path*)

djangofloor.decorators.**connect**(*fn=None*, *path=None*, *delayed=False*, *allow_from_client=True*, *auth_required=True*)

Decorator to use in your Python code. Use it in any file named *signals.py* in a installed Django app.

```
@connect(path='myproject.signal.name', allow_from_client=True, delayed=False)
def function(request, arg1, arg2, **kwargs):
    pass
```

Parameters

- **fn** (callable) – the Python function to connect to the signal
- **path** (unicode or str) – the name of the signal
- **delayed** (bool) – should this code be called in an asynchronous way (through Celery)? default to *False*
- **allow_from_client** (bool) – can be called from JavaScript? default to *True*
- **auth_required** (bool) – can be called only from authenticated client? default to *True*

Returns a wrapped function

Return type callable

2.12.4 djangofloor.defaults

Default values for your Django project

Define all DjangoFloor default settings. The goal is to define valid settings out-of-the-box.

2.12.5 djangofloor.exceptions

Common DjangoFloor exceptions

Define several common exceptions, which can be gracefully handled by DjangoFloor. You should raise these exceptions, or create new exceptions which derive from these ones. They help to display helpful messages to end-user.

exception djangofloor.exceptions.**ApiException**(*msg=None*)
Base exception, corresponding to a bad request from the client.

default_msg

http_code = 400

exception djangofloor.exceptions.**AuthenticationRequired**(*msg=None*)

default_msg

http_code = 401

exception djangofloor.exceptions.**InternalServerError**(*msg=None*)

default_msg

http_code = 500

exception djangofloor.exceptions.**InvalidOperation**(*msg=None*)

default_msg

http_code = 400

exception djangofloor.exceptions.**InvalidRequest**(*msg=None*)

default_msg

http_code = 400

exception djangofloor.exceptions.**NotFoundException**(*msg=None*)

default_msg

http_code = 404

exception djangofloor.exceptions.**PermissionDenied**(*msg=None*)

default_msg

http_code = 403

2.12.6 djangofloor.log

Log handler that send emails

Overrides django.utils.log.AdminEmailHandler to avoid flooding admin with many e-mails.

If an e-mail with the same object has already been sent in the last 10 minutes, then nothing is done. This duration can be configured (argument *min_interval*).

class djangofloor.log.**FloorAdminEmailHandler**(*include_html=False*, *email_backend=None*, *min_interval=600*)

An exception log handler that emails log entries to site admins.

If the request is passed as the first argument to the log record, request data will be provided in the email report.

send_mail(*subject*, *message*, **args*, ***kwargs*)

2.12.7 djangofloor.middleware

Several middleware, for production or debugging purposes

class djangofloor.middleware.**BasicAuthMiddleware**

Basic HTTP authentication using Django users to check passwords.

process_request(*request*)

class djangofloor.middleware.**FakeAuthenticationMiddleware**

Only for dev/debugging purpose: emulate a user authenticated by the remote proxy.

Use *settings.FLOOR_FAKE_AUTHENTICATION_USERNAME* to create (if needed) a user and authenticate the request. Only works in *settings.DEBUG* mode and if *settings.FLOOR_FAKE_AUTHENTICATION_USERNAME* is set.

group_cache = {}

process_request(*request*)

class djangofloor.middleware.**IEMiddleware**

required for signals tight to a window Add a HTTP header for Internet Explorer Compatibility. Ensure that IE uses the last version of its display engine.

process_request(*request*)

process_template_response(*request*, *response*)

class djangofloor.middleware.**PyScssCompiler**(*verbose*, *storage*)

compile_file(*infile*, *outfile*, *outdated=False*, *force=False*)

match_file(*filename*)

output_extension = u'css'

class djangofloor.middleware.**RCSSMinCompressor**(*verbose*)

static compress_css(*css*)

class djangofloor.middleware.**RemoteUserMiddleware**
Like django.contrib.auth.middleware.RemoteUserMiddleware but:

- can use any header defined by the setting *FLOOR_AUTHENTICATION_HEADER*,
- add a *df_remote_authenticated* attribute to the request (*True* if the user has been authenticated via the header)

header = u'HTTP_REMOTE_USER'

original_process_request(*request*)

process_request(*request*)

2.12.8 djangofloor.scripts

"Main" functions for Django, Celery, Gunicorn and uWSGI

Define "main" functions for your scripts using the Django *manage.py* system or Gunicorn/Celery/uWSGI.

djangofloor.scripts.**celery**()

djangofloor.scripts.**gunicorn**()
wrapper around gunicorn. Retrieve some default values from Django settings.

Returns

djangofloor.scripts.**load_celery**()
Import Celery application unless Celery is disabled. Allow to automatically load tasks

djangofloor.scripts.**manage**()
Main function, calling Django code for management commands. Retrieve some default values from Django settings.

djangofloor.scripts.**set_env**()
Determine project-specific and user-specific settings and set several environment variable, update sys.argv and return the name of current djangofloor project.

1.determine the project name

if the script is {xxx}-[gunicorn|manage][.py], then the project_name is assumed to be {xxx} if option –project {xxx} is available, then the project_name is assumed to be {xxx}

2.determine project-specific settings

project-specific settings are expected to be in the module {xxx}.defaults Can be overridden by the DJANGOFLOOR_PROJECT_SETTINGS environment variable.

3.determine user-specific settings

if option –conf_file {yyy} is available, the {yyy} is used as user-specific settings file else if {xxx}_configuration.py exists in working directory, then it is used. else if [prefix]/etc/{xxx}/settings.py exists, then it is used. Can be overridden by the DJANGOFLOOR_USER_SETTINGS environment variable.

4.standard Django settings module is djangofloor.settings.

of course, you can always leave out djangofloor's settings system and override DJANGO_SETTINGS_MODULE

djangofloor.scripts.**uwsgi**()

2.12.9 djangofloor.tasks

Calling DjangoFloor signals

Public functions

Define the *df_call* function for calling signals and its shortcut *call*. Can activate the test mode, allowing to retain signal calls (simplifying tests). Activate this mode with *set_test_mode(True)* and fetch called signals with *pop_called_signals()*.

Internal functions

Define several Celery tasks, get signals encoders/decoders (JSON by default) and a function for automatically discover all signals.

djangofloor.tasks.**call**(*signal_name*, *request*, *sharing=None*, ***kwargs*)

Call a signal and all the three kinds of receivers can receive it:

- standard Python receivers
- Python receivers through Celery (thanks to the *delayed* argument)
- JavaScript receivers (through websockets)

This is a shortcut for *djangofloor.tasks.df_call* but that forbids several signal argument names (*signal_name*, *request* and *sharing*. Directly use *djangofloor.tasks.df_call* if you want to use any of the argument names, or if you want to specify more options (like wait some time before executing code). Example:

```
from djangofloor.tasks import call, SESSION
from djangofloor.decorators import connect

def any_function(request):
    call('myproject.signal_name', request, sharing=SESSION, arg1="arg1", arg2=42)

@connect('myproject.signal_name')
def signal_name(request, arg1, arg2):
    print(arg1, arg2)
```

Parameters

- **signal_name** (str) –
- **request** (*djangofloor.decorators.SignalRequest* or django.http.HttpRequest) – initial request, giving information about HTTP sessions and its user

- **sharing** –
  - *None*: does not propagate to the JavaScript (client) side
  - *WINDOW*: only to the browser window that initiated the original request
  - *USER*, *SESSION*, *BROADCAST*: propagate to the request user, only to its current session, or to all currently logged-in users
  - {'users': ['username1', 'username2'], 'groups': ['group1', 'group2'], 'broadcast': True} (or any subset of these keys)
  - *RETURN* return result values of signal calls to the caller
- **kwargs** – arguments for the receiver

`djangofloor.tasks.`**df_call**(*signal_name*, *request*, *sharing=None*, *from_client=False*, *kwargs=None*, *countdown=None*, *expires=None*, *eta=None*)

Call a signal and all the three kinds of receivers can receive it:

- standard Python receivers
- Python receivers through Celery (thanks to the *delayed* argument)
- JavaScript receivers (through websockets)

Do not use it directly, you should prefer use the *call* function.

Parameters

- **signal_name** (`str`) –
- **request** (*djangofloor.decorators.SignalRequest* or `django.http.HttpRequest`) – initial request, giving information about HTTP sessions and its user
- **sharing** –
  - *None*: does not propagate to the JavaScript (client) side
  - *WINDOW*: only to the browser window that initiated the original request
  - ***USER*, *SESSION*, *BROADCAST*: propagate to the request user, only to its current session,** or to all currently logged-in users
  - **{'users': ['username1', 'username2'], 'groups': ['group1', 'group2'], 'broadcast': True}** (or any subset of these keys)
  - *RETURN* return result values of signal calls to the caller
- **from_client** – True if this call comes a JS client
- **kwargs** – arguments for the receiver
- **countdown** (`int`) – Wait *countdown* seconds before actually calling the signal. Check Celery doc
- **eta** (`datetime.datetime`) – Wait until *eta* before actually calling the signal. Check Celery doc
- **expires** (`datetime.datetime` or `int`) – Wait until *eta* before actually calling the signal. Check Celery doc

Returns if sharing != *RETURN*: return *None* else: call *djangofloor.tasks.df_call* on each element of the call result

djangofloor.tasks.**get_signal_decoder**(**args*, ***kwds*)
return the class for decoding signal data to JSON. The result is cached.

Only import *settings.FLOOR_SIGNAL_DECODER* and cache the results.

djangofloor.tasks.**get_signal_encoder**(**args*, ***kwds*)
return the class for encoding signal data to JSON. The result is cached.

Only import *settings.FLOOR_SIGNAL_ENCODER* and cache the results.

djangofloor.tasks.**import_signals**(**args*, ***kwds*)
Import all *signals.py* files to register signals.

djangofloor.tasks.**pop_called_signals**()
return the list of called signals with their requests and arguments when *test_mode* is set to *True*.

Returns list of *(signal_name, request, sharing, kwargs)*

Return type list

djangofloor.tasks.**set_test_mode**(*test=True*)
Activate (or deactivate) test mode, allowing to gather all signals calls (instead of actually calling them) :param test: :type test: bool

2.12.10 djangofloor.utils

Classes and functions used for the DjangoFloor settings system

Define several helpers classes and internal functions for the DjangoFloor settings system, allowing to merge settings from different sources. This file must be importable while Django is not loaded yet.

class djangofloor.utils.**CallableSetting**(*value*, **required*)
Require a function(kwargs) as argument, this function will be called with all

```
>>> SETTING_1 = True
>>> SETTING_2 = CallableSetting(lambda x: not x['SETTING_1'])
```

In *local_settings.py*

```
>>> SETTING_1 = False
```

In your code:

```
>>> from django.conf import settings
```

Then *settings.SETTING_2* is equal to *True*

You can provide a list of required settings that must be available before the call to your function.

get_value(*merger*)

class djangofloor.utils.**DirectoryPath**(*value*)
Represent a directory that must be created on startup

get_value(*merger*)

class djangofloor.utils.**DjangoFloorConfig**(*value*)
Base class for special setting values. When a setting is a *djangofloor.utils.DjangoFloorConfig*, then the method *get_value(merger)* is called for getting the definitive value.

get_value(*merger*)
Return the intepreted value :param merger: merger object, with all interpreted settings :type merger: *djangofloor.utils.SettingMerger* :return: :rtype:

class djangofloor.utils.**ExpandIterable**(*value*, *func=None*)
Allow to import an existing list inside a list setting. in *defaults.py*:

```
>>> LIST_1 = [0, ]
>>> LIST_2 = [1, ExpandIterable('LIST_1'), 2, ]
>>> DICT_1 = {0: 0, }
>>> DICT_2 = {1: 1, None: ExpandIterable('DICT_1'), 2: 2, }
```

In case of dict, the key is ignored when the referenced dict is expanded. In *local_settings.py*

```
>>> LIST_1 = [3, ]
>>> DICT_1 = {3: 3, }
```

In your code:

```
>>> from django.conf import settings
```

Then *settings.LIST_2* is equal to *[1, 3, 2]*. *settings.DICT_2* is equal to *{1: 1, 2: 2, 3: 3, }*.

class djangofloor.utils.**FilePath**(*value*)
Represent a file, whose parent directory should be created on startup

get_value(*merger*)

class djangofloor.utils.**Path**(*value*)

get_value(*merger*)

class djangofloor.utils.**SettingMerger**(*project_name*, *default_settings_module_name*, *project_settings_module_name*, *user_settings_path*, *djangofloor_config_path*, *djangofloor_mapping*, *doc_mode=False*, *read_only=False*)
Load different settings modules and config files and merge them.

all_ini_options
Return an OrderedDict of list of available options in the .ini configuration file.

- Keys correspond to the section names
- Values are list of OptionParser, with help_str and default_values attributes set

Returns

Return type

static ensure_dir(*path_*, *parent_=True*)
Ensure that the given directory exists

Parameters

- **path** – the path to check
- **parent** – only ensure the existence of the parent directory

get_setting_value(*setting_name*)
import the required settings from user-specific settings, or project-specific settings, or django-floor settings. Also add it to globals(), so this function is idempotent.

Parameters **setting_name** – name of the setting to import

Returns the imported setting :)

static **import_file**(*filepath*)
import the Python source file as a Python module.

Parameters **filepath** (str) – absolute path of the Python module

Returns

load_settings()

load_settings_providers()
Load the different sources of settings and set the corresponding attributes :return: *None* :rtype: NoneType

parse_setting(*obj*)
Parse the object for replacing variables by their values.

If *obj* is a string like "THIS_IS_{TEXT}", search for a setting named "TEXT" and replace {TEXT} by its value (say, "VALUE"). The returned object is then equal to "THIS_IS_VALUE". If *obj* is a list, a set, a tuple or a dict, its components are recursively parsed. If *obj* is a *djangofloor.utils.DirectoryPath* or a *djangofloor.utils.FilePath*, required parent directories are automatically created and the name is returned. Otherwise, *obj* is returned as-is.

Parameters **obj** – object to analyze

Returns the parsed setting

post_process()
Perform some cleaning on settings:

- remove duplicates in *INSTALLED_APPS* (keeps only the first occurrence)

process()

class djangofloor.utils.**SettingReference**(*value*, *func=None*)
Reference any setting object by its name. Allow to reuse a list defined in another setting file.

in *defaults.py*:

```
>>> SETTING_1 = True
>>> SETTING_2 = SettingReference('SETTING_1')
```

In *local_settings.py*

```
>>> SETTING_1 = False
```

In your code:

```
>>> from django.conf import settings
```

Then *settings.SETTING_2* is equal to *False*

get_value(*merger*)

djangofloor.utils.**guess_version**(*defined_settings*)
Guesss the project version. Expect __version__ in *your_project/__init__.py*

Parameters **defined_settings** (dict) – all already defined settings (dict)

Returns

Return type str

djangofloor.utils.**import_module**(*name*, *package=None*)
Import a module.

The 'package' argument is required when performing a relative import. It specifies the package to use as the anchor point from which to resolve the relative import to an absolute import.

djangofloor.utils.**walk**(*module_name*, *dirname*, *topdown=True*)
Copy of os.walk(), please refer to its doc. The only difference is that we walk in a package_resource instead of a plain directory. :type module_name: basestring :param module_name: module to search in :type dirname: basestring :param dirname: base directory :type topdown: bool :param topdown: if True, perform a topdown search.

2.12.11 djangofloor.views

Default index and views tied to the signal system

djangofloor.views.**get_signal_calls**(*request*, **args*, ***kwargs*)
Regularly called by JS code when websockets are not available. Allows Python code to call JS signals.

The polling frequency is set with *WS4REDIS_EMULATION_INTERVAL* (in milliseconds).

Return all signals called by Python code as a JSON-list

djangofloor.views.**index**(*request*)

djangofloor.views.**read_file_in_chunks**(*fileobj*, *chunk_size=32768*)
read a file object in chunks of the given size.

Return an iterator of data

Parameters

- **fileobj** –
- **chunk_size** (*int*) – max size of each chunk

djangofloor.views.**robots**(*request*)

djangofloor.views.**send_file**(*filepath*, *mimetype=None*, *force_download=False*)
Send a local file. This is not a Django view, but a function that is called at the end of a view.

If *settings.USE_X_SEND_FILE* (mod_xsendfile is a mod of Apache), then return an empty HttpResponse with the correct header. The file is directly handled by Apache instead of Python. If *settings.X_ACCEL_REDIRECT_ARCHIVE* is defined (as a list of tuple (directory, alias_url)) and filepath is in one of the directories, return an empty HttpResponse with the correct header. This is only available with Nginx.

Otherwise, return a StreamingHttpResponse to avoid loading the whole file in memory.

Parameters

- **filepath** – absolute path of the file to send to the client.
- **mimetype** – MIME type of the file (returned in the response header)
- **force_download** – always force the client to download the file.

Return type *StreamingHttpResponse* or *HttpResponse*

djangofloor.views.**signal_call**(*request*, **args*, ***kwargs*)
Called by JS code when websockets are not available. Allow to call Python signals from JS. Arguments are passed in the request body, serialized as JSON.

Parameters

- **request** – Django HTTP request
- **signal** (str) – name of the called signal

djangofloor.views.**signals**(*request*, **args*, ***kwargs*)
Generate a HttpResponse to register Python signals from the JS side

2.12.12 djangofloor.templatetags.djangofloor

Template tags specific to DjangoFloor

Define a few useful template tags, currently only for the default Django template system.

class djangofloor.templatetags.djangofloor.**MediaNode**(*varname=None*, *path=None*)

classmethod handle_simple(*path*)

djangofloor.templatetags.djangofloor.**df_inivalue**(*value*)

djangofloor.templatetags.djangofloor.**df_underline**(*value*, *kind=u'='*)

djangofloor.templatetags.djangofloor.**df_urlparse**(*value*, *component=u'hostname'*)

djangofloor.templatetags.djangofloor.**df_window_key**(*context*)

djangofloor.templatetags.djangofloor.**do_media**(*parser*, *token*)
Joins the given path with the MEDIA_URL setting.

Usage:

```
{% media path [as varname] %}
```

Examples:

```
{% media "myapp/css/base.css" %}
{% media variable_with_path %}
{% media "myapp/css/base.css" as admin_base_css %}
{% media variable_with_path as varname %}
```

djangofloor.templatetags.djangofloor.**fontawesome_icon**(*name*, *large=False*, *fixed=False*, *spin=False*, *li=False*, *rotate=None*, *border=False*, *color=None*)

djangofloor.templatetags.djangofloor.**media**(*path*)

2.12.13 djangofloor.management.commands.config

Display the current config

Display the current loaded config. Can also generate settings.py (with *-m*) or settings.ini config files (with *–ini*).

class djangofloor.management.commands.config.**Command**(*stdout=None*, *stderr=None*, *no_color=False*)

add_arguments(*parser*)

config_file()

display(*all_keys*)

Parameters **all_keys** (dict) – dictionnary of all default settings (djangofloor's and project's ones), without local settings

display_header(*djangofloor_default_conf*, *project_default_conf*)

force_text(*value*)

handle(**args*, ***options*)

lazy_cls

merge(*djangofloor_default_conf*, *project_default_conf*)

show_config(*kind*, *env_variable*, *path*)

2.12.14 djangofloor.management.commands.dumpdb

Dump the content of the databases to stdout or to a file.

Currently only works for sqlite, mysql and postresql.

class djangofloor.management.commands.dumpdb.**BaseDumper**(*name*, *db_options*)
base class for a given database engine. An instance is returned by *Command.get_dumper()*

dump(*filename*)
dump the content of the database to *stdout* or to a *file*.

If *filename* is given, its parent directory must already exist.

Parameters **filename** (str or *None*) – filename, or None if the content is must be dumped to *stdout*

Returns *None*

class djangofloor.management.commands.dumpdb.**Command**(*stdout=None*, *stderr=None*, *no_color=False*)
Dump the content of one (or more) database for backup to stdout or to a file.

Just call the pg_dump/mysql_dump/... tools, allowing you to forget questions about their syntax. Currently only works for sqlite, mysql and postresql databases.

- If no database is given, only the default one is dumped,
- if multiple databases are given, all their dumps will be dumped and merged to stdout!
- if *–filename* is given, the content is dumped to this file (without compression),
- if *–filename* is given (say, "backup.sql") and multiple databases are given, the written file will be "backup-default.sql", "backup-other.sql", and so on).

add_arguments(*parser*)

static get_dumper(*name*, *db_options*)
return a valid dumper for the given database, based on the *ENGINE* key.

Parameters

- **name** (str) – name of the database (one of the keys in the *DATABASES* setting)
- **db_options** (dict) – dictionnary (one of the values in the *DATABASES* setting)

Return type *BaseDumper*

handle(**args*, ***options*)
execute the command

help = 'Dump the content of one (or more) database'

class djangofloor.management.commands.dumpdb.**MySQL**(*name*, *db_options*)
dump the content of a MySQL database, with *mysqldump*

dump(*filename*)

dump_cmd_list()

Returns

Return type list of str

get_env()
Extra environment variables to be passed to shell execution

class djangofloor.management.commands.dumpdb.**PostgreSQL**(*name*, *db_options*)
dump the content of a PostgreSQL database, with *pg_dump*

dump_cmd_list()

get_env()
Extra environment variables to be passed to shell execution

class djangofloor.management.commands.dumpdb.**SQLite**(*name*, *db_options*)
copy the SQLite database to another file, or write its content to *stdout*

dump(*filename*)

2.12.15 Build a .deb package with binaries and default config

Require a *stdeb.cfg* at the root of your project.

2.13 Javascript and HTML code

2.13.1 Prerequisites

First, you must include the following JavaScript files:

- *jquery.min.js*,
- *bootstrap.min.js* (ok, it is only required if you use Bootstrap3),
- *js/djangofloor.js*,
- *ws4redis.js* (required for using websockets).

You HTML also requires some code for messages, probably somewhere on the top of your page:

```
<div id="messages"></div>
```

2.13.2 Javascript API

DjangoFloor offers an object *df* with basically two methods:

- *df.connect('signal_name', function)*, where *function* is a function taking an object as argument.
- *df.call('signal_name', options)*, where *options* is an object whose attributes are the arguments.

2.13.3 Form submission

You should check the documentation of *djangofloor.decorators.SerializedForm*.

2.13.4 Connected signals

DjangoFloor provides some signals out of the box, but they only works when you use Bootstrap3.

df.messages.warning

Display a message to the user in a warning box (call "df.messages.show" with *level* ="warning").

- *html*: HTML content of a message to display
- *duration* (optional): default to 10000 ms: duration for which the message must be displayed. Set it to 0 to keep it.

JavaScript example:

```
df.call('df.messages.warning', {html: "This is a message"});
```

Python example:

```
from djangofloor.tasks import call
def my_view(request):
    call('df.messages.warning', request, html="This is a message")
```

df.messages.info

Display a message to the user in an info box (call "df.messages.show" with *level* ="info").

- *html*: HTML content of a message to display
- *duration* (optional): default to 10000 ms: duration for which the message must be displayed. Set it to 0 to keep it.

df.messages.error

Display a message to the user in an error box (call "df.messages.show" with *level* ="danger").

- *html*: HTML content of a message to display
- *duration* (optional): default to 10000 ms: duration for which the message must be displayed. Set it to 0 to keep it.

df.messages.success

Display a message to the user in a success box (call "df.messages.show" with *level* ="success").

- *html*: HTML content of a message to display
- *duration* (optional): default to 10000 ms: duration for which the message must be displayed. Set it to 0 to keep it.

df.messages.show

Display a message to the user.

- *html*: HTML content of a message to display
- *level* (optional): default to "warning". Can be "default", "warning", "info", "success" or "danger"
- *duration* (optional): default to 10000 ms: duration for which the message must be displayed. Set it to 0 to keep it.

df.modal.show

Display a modal window. Successive calls replace the content of the modal by the last content.

- *html*: HTML content
- *width* (optional): width (example: "1200px")

df.modal.hide

Hide the modal window (no argument).

df.redirect

Redirect the browser URL to the URL.

- *url*: URL

df.messages.hide

Remove displayed messages.

- *id* (optional): id of the message to remove. If not provided, all messages are removed.

df.notify.warning

Show a Growl-like notification (by default on the top-right of the screen). See all settings on Bootstrap-Notify. Settings and options are merged in the same dict.

JavaScript example:

```
df.call('df.notify.warning', {message: "This is a message", delay: 1000, allow_dismiss: true});
```

Python example:

```
from djangofloor.tasks import call
def my_view(request):
    call('df.notify.warning', request, message="This is a message", title="Notification: ")
```

df.notify.info

Show a Growl-like notification with info style.

df.notify.error

Show a Growl-like notification with error style.

df.notify.success

Show a Growl-like notification with success style.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

d

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W