django-fab-deploy Documentation

Release 0.8a1

Mikhail Korobov

Jul 22, 2017

Contents

django-fab-deploy is a collection of fabric scripts for deploying and managing django projects on Debian/Ubuntu servers.

CHAPTER 1

Design goals

- Provide (heroku, ep.io, gondor.io, ...)-like experience using your own VPS/server;
- servers should be configured in most standard and obvious way: invent as little as possible;
- developer should be able to customize deployment;
- it should be possible to integrate django-fab-deploy into existing projects;
- try to be a library, not a framework; parts of django-fab-deploy should be usable separately.

CHAPTER 2

Tech overview

- django projects are isolated with virtualenv and (optionally) linux and db users;
- python requirements are managed using pip;
- server interactions are automated and repeatable (the tool is fabric);
- several projects can be deployed on the same VPS;
- one project can be deployed on several servers.

Server software:

- First-class support: Debian Squeeze, Ubuntu 10.04 LTS;
- also supported: Debian Lenny, Ubuntu 10.10;
- the project is deployed with Apache + mod_wsgi for backend and nginx in front as a reverse proxy;
- DB: MySQL and PostgreSQL (+PostGIS) support is provided out of box;
- VCS: hg and git support is provided out of box + it is possible not to store project into VCS.

User Guide

The basic workflow for setting up a new web site is described in this guide. If this workflow doesn't fit for some reason then django-fab-deploy can still be used as a collection of scripts, a lot of them can be used independently.

Prerequisites

1. Clean Debian Lenny, Debian Squeeze, Ubuntu 10.04 or 10.10 server/VPS with root or sudo-enabled user ssh access;
2. working ssh key authentication;

Warning: OpenVZ has serious issues with memory management (VIRT is counted and limited instead of RSS) so a lot of software (including apache2, Java and mysql's InnoDB engine) is nearly unusable on OpenVZ while being memory-wise and performant on XEN/KVM. So please try to avoid OpenVZ or Virtuozzo VPS's, use XEN or KVM or real servers.

Prepare the project

1. Install django-fab-deploy its requirements:

```
pip install django-fab-deploy
pip install jinja2
pip install "fabric >= 1.4"
pip install fabric-taskset
```

2. Create `fabfile.py` at project root. It should provide one or more function putting server details into Fabric environment. Otherwise it's just a standart Fabric's fabfile (e.g. project-specific scripts can also be put here). Example:

```
# my_project/fabfile.py
from fabric.api import env, task

from fab_deploy.project import WebProject
from fab_deploy.utils import update_env
from fab_deploy.django import Django
from fab_deploy.webserver.apache import Apache
from fab_deploy.webserver.nginx import Nginx

apps = dict(django=Django(Nginx(), Apache()))
WebProject(apps=apps).expose_to_current_module()

@task
def my_site():
    env.hosts = ['my_site@example.com']
    env.conf = dict(
        DB_USER = 'my_site',
        DB_PASSWORD = 'password',
        DB_BACKEND = 'mysql',

        # uncomment this line if the project is not stored in VCS
        # default value is 'hg', 'git' is also supported
        # VCS = 'none',
    )
    update_env()

my_site()
```

`apps` dictionary is provided with default values for WebProject. Yes, that is a fallback to previous versions of django-fab-deploy. And there is a simpler syntax for the code above:

```
from fab_deploy.project import WebProject
from fab_deploy.utils import define_host

WebProject().expose_to_current_module()

@define_host('my_site@example.com')
```

```
def my_site():
    return dict(
        DB_USER = 'my_site',
        DB_PASSWORD = 'password',
        DB_BACKEND = 'mysql',
    )

my_site()
```

In order to make things simple set the username in *env.hosts* string to your project name. It should be a valid python identifier. Don't worry if there is no such user on server, django-fab-deploy can create linux user and setup ssh access for you, and it is preferrable to have linux user per project if possible.

Note: There are some defaults, e.g. *DB_NAME* equals to *INSTANCE_NAME*, and *INSTANCE_NAME* equals to username obtained from *env.hosts*.

Read *fabfile.py API* for more details.

---

3. Copy `config_templates` folder from django-fab-deploy to your project root, manually or by running the following command from the project root:

```
django-fab-deploy config_templates
```

Read the configs and adjust them if it is needed. Basic configs are usually a good starting point and should work as-is.

Note: {{ variables }} can be used in config templates (engine is jinja2). They will be replaced with values from *env.conf* on server.

If you change web server config file or *env.conf* variables after initial deployment, apply the changes in web server configs by running

```
fab update_web_servers
```

It will update all remote configs of all apps of your default project.

---

4. Create `config.server.py` near your project's `settings.py`. This file will become `config.py` on server. Example:

```
# my_project/config.server.py
# config file for environment-specific settings

DEBUG = False
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': '{{ DB_NAME }}',
        'USER': '{{ DB_USER }}',
        'PASSWORD': '{{ DB_PASSWORD }}',
    }
}
```

Then create `config.py` for local development. Import config in project's `settings.py`:

```
# Django settings for my_project project.
# ...
from config import *
# ...
```

`config.py` trick is also known as `local_settings.py` (make sure `config.py` is ignored in your VCS if one is used).

Note: {{ variables }} can be used in `config.server.py`. They will be replaced with values from *`env.conf`* on server.

If you change `config.server.py` or *`env.conf`* variables after initial deployment, apply the changes to `config.server.py` by running

```
fab apps.django.update_config
```

for default apps configuration. Or more generic

```
fab apps.{{ django_app_name }}.update_config
```

---

5. Create `reqs` folder at project root. This folder should contain text files with pip requirements.

   You can get basic/example `reqs` folder by running

   ```
   django-fab-deploy example_reqs
   ```

   One file is special: `reqs/all.txt`. This is the main requirements file. List all project requirements here one-by-one or (preferrable) by including other requirement files using "-r" syntax.

   There is also

   ```
   django-fab-deploy generate_reqs
   ```

   command. It creates `reqs` folder with `all.txt` file containing a list of currently installed packages (obtained from `pip freeze`).

The project should look like that after finishing steps 1-5:

```
my_project
    ...
    config_templates <- this folder should be copied from django-fab-deploy
        apache.config
        django_wsgi.py
        hgrc
        nginx.config

    reqs             <- a folder with project's pip requirement files
        all.txt      <- main requirements file, list all requirements in this file
        active.txt   <- put recently modified requirements here
        ...          <- you can provide extra files and include them with '-r' syntax␣
↪in e.g. all.txt

    config.py        <- this file should be included in settings.py and ignored in .
↪hgignore
    config.server.py <- this is a production django config template (should be␣
↪ignored too!)
    fabfile.py       <- your project's Fabric deployment script
```

```
settings.py
manage.py
```

Note: django-fab-deploy does not enforce this layout; if it doesn't fit for some reason (e.g. you prefer single pip requirements file or django project in subfolder or you use django >= 1.4), take a look at *Custom project layouts*.

The project is now ready to be deployed.

Prepare the server

Note: It is assumed that you would manage imports in fabfile.py appropriately. E.g. for command "fab system.{{commmand}}" to work "from fab_deploy import system" would be added, for command "fab db.{{command}}" - "from fab_deploy import db", and so on.

1. If the server doesn't have sudo installed (e.g. clean Lenny or Squeezy) then install sudo on server:

   ```
   fab system.install_sudo
   ```

   Note: Fabric commands should be executed in shell from the project root on local machine (not from the python console, not on server shell).

2. If there is no linux account for user specified in *env.hosts* then add a new linux server user, manually or using

   ```
   fab system.create_linux_account:"/home/kmike/.ssh/id_rsa.pub"
   ```

   You'll need the ssh public key. *create_linux_account* creates a new linux user and uploads provided ssh key. Test that ssh login is working:

   ```
   ssh my_site@example.com
   ```

   SSH keys for other developers can be added at any time:

   ```
   fab system.ssh_add_key:"/home/kmike/coworker-keys/ivan.id_dsa.pub"
   ```

3. Setup the database. django-fab-deploy can install mysql and create empty DB for the project (using defaults in your default host function):

   ```
   fab db.mysql.install
   fab db.mysql.create_db
   ```

   `mysql.install` does nothing if mysql is already installed on server. Otherwise it installs mysql-server package and set root password to *env.conf.DB_ROOT_PASSWORD*. If this option is empty, mysql_install will ask for a password.

   `mysql.create_db` creates a new empty database named *env.conf.DB_NAME* (it equals to *env.conf.INSTANCE_NAME* by default, which equals to the user from *env.hosts* by default). `mysql.create_db` will ask for a mysql root password if *DB_USER* is not 'root'.

Note: If the DB engine is not mysql then use appropriate commands.

4. If you feel brave you can now run `fab full_deploy` from the project root and get a working django site.

Warning: django-fab-deploy disables 'default' apache and nginx sites and takes over 'ports.conf' so apache is no longer listening to 80 port.

If there are other sites on server (not managed by django-fab-deploy) they may become unaccessible due to these changes.

`fab full_deploy` command:

- installs necessary system and python packages;
- configures web-servers for all applications of your project;
- creates virtualenv;
- uploads project to the server;
- runs `python manage.py syncdb` and `python manage.py migrate` commands on server.

Project sources will be available under `~/src/<INSTANCE_NAME>`, virtualenv will be placed in `~/envs/<INSTANCE_NAME>`.

Working with the server

django-fab-deploy provides additional commands that should be useful for updating the server:

1. Source changes are deployed with `fab_deploy.deploy.push()` command:

```
fab push
```

Another example (deploy changes on 'prod' server, update pip requirements and perform migrations in one step:

```
fab prod push:pip_update,migrate
```

2. Update web servers configuration:

```
fab update_web_servers
```

3. Update some app configuration (`config.server.py` for django or `production.ini` for pyramid):

```
fab apps.{{ app_name }}.update_config
```

where `app_name` actually is a key in apps dictionary.

4. Requirements are updated with `fab_deploy.virtualenv.pip_update()` command. Update requirements listed in reqs/active.txt:

```
fab update_r
```

Update requirements listed in reqs/my_apps.txt:

```
fab update_r:my_apps
```

5. Remotely change branch or revision (assuming *env.conf.VCS* is not 'none'):

```
fab up:my_branch
```

Full list of commands can be found *here*.

Customization guide is also worth reading.

Customization

Custom deployment scripts

django-fab-deploy is intended to be a library, not a framework. So the preferred way for customizing standard command is to just wrap it or to create a new command by combining existing commands:

```
# fabfile.py
from fab_deploy import *
from fab_deploy.utils import run_as_sudo
import fab_deploy.deploy

@run_as_sudo
def install_java():
    run('aptitude update')
    run('aptitude install -y default-jre')

def full_deploy():
    install_java()
    fab_deploy.deploy.full_deploy()
```

`fab_deploy.deploy.push()` accepts callable 'before_restart' keyword argument. This callable will be executed after code uploading but before the web server reloads the code.

An example of 'fab push' customization

```
# fabfile.py
from fab_deploy import *
import fab_deploy.deploy

@inside_src
def rebuild_docs():
    with cd('docs'):
        run ('rm -rf ./_build')
        run('make html > /dev/null')

def push(*args):

    # run local tests before pushing
    local('./runtests.sh')

    # rebuild static files before restarting the web server
    def before_restart():
        manage('collectstatic --noinput')
        manage('assets rebuild')

    # execute default push command
```

```
    fab_deploy.deploy.push(*args, before_restart=before_restart)

    # rebuild developer documentation after pushing
    rebuild_docs()
```

Custom project layouts

User Guide describes standard project layout:

```
my_project
    ...
    config_templates <- this folder should be copied from django-fab-deploy
        ...

    reqs             <- a folder with project's pip requirement files
        all.txt      <- main requirements file, list all requirements in this file
        active.txt   <- put recently modified requirements here
        ...          <- you can provide extra files and include them with '-r' syntax␣
↪in e.g. all.txt

    config.py        <- this file should be included in settings.py and ignored in .
↪hgignore
    config.server.py <- this is a production django config template
    fabfile.py       <- your project's Fabric deployment script
    settings.py
    manage.py
```

django-fab-deploy does not enforce this layout. Requirements handling, config templates placement, local settings file names and project source folder can be customized using these options:

- *env.conf.PROJECT_PATH*
- *env.conf.LOCAL_CONFIG*
- *env.conf.REMOTE_CONFIG_TEMPLATE*
- *env.conf.CONFIG_TEMPLATES_PATHS*
- *env.conf.PIP_REQUIREMENTS_PATH*
- *env.conf.PIP_REQUIREMENTS*
- *env.conf.PIP_REQUIREMENTS_ACTIVE*

Example

Let's configure django-fab-deploy to use the following layout:

```
my_project
    hosting                 <- a folder with server configs
        staging             <- custom configs for 'staging' server
            apache.config   <- custom apache config for staging server

        production          <- custom configs for 'production' server
            apache.config
            nginx.config

        apache.config       <- default configs
```

```
        django_wsgi.py
        nginx.config

    src                     <- django project source files
        apps
            ...

        local_settings.py   <- local settings
        stage_settings.py   <- local settings for staging server
        prod_settings.py    <- local settings for production server

        settings.py
        manage.py

    requirements.txt        <- single file with all pip requirements
    fabfile.py              <- project's Fabric deployment script
```

It uses subfolder for storing django project sources, single pip requirements file and different config templates for different servers in non-default locations.

fabfile.py:

```
from fab_deploy.utils import define_host

# Common layout options.
# They are separated in this example in order to stay DRY.
COMMON_OPTIONS = dict(
    PROJECT_PATH = 'src',
    LOCAL_CONFIG = 'local_settings.py',
    PIP_REQUIREMENTS = 'requirements.txt',
    PIP_REQUIREMENTS_ACTIVE = 'requirements.txt',
    PIP_REQUIREMENTS_PATH = '',
)

@define_host('user@staging.example.com', COMMON_OPTIONS)
def staging():
    return dict(
        REMOTE_CONFIG_TEMPLATE = 'stage_settings.py',
        CONFIG_TEMPLATES_PATHS = ['hosting/staging', 'hosting'],
    )

@define_host('user@example.com', COMMON_OPTIONS)
def production():
    return dict(
        REMOTE_CONFIG_TEMPLATE = 'prod_settings.py',
        CONFIG_TEMPLATES_PATHS = ['hosting/production', 'hosting'],
    )
```

Example 2: django 1.4 layout

Django 1.4 presents a new project layout. It can be used e.g. this way:

```
my_project
    my_project
        config_templates
            ...
        reqs
```

```
            ...
        ...
        config.py
        config.server.py
        settings.py

    fabfile.py
    manage.py
    ...
```

fabfile.py:

```
from fab_deploy.utils import define_host

@define_host('user@example.com')
def staging():
    return dict(
        CONFIG_TEMPLATES_PATHS=['my_project/config_templates'],
        LOCAL_CONFIG = 'my_project/config.py',
        REMOTE_CONFIG_TEMPLATE = 'my_project/config.server.py',
        PIP_REQUIREMENTS_PATH = 'my_project/reqs/',
    )
```

fabfile.py API

Overview

- Write a function populating *env.hosts* and *env.conf* for each server configuration.
- Call *update_env()* at the end of each function.
- It is possible to reduce boilerplate by using *define_host* decorator:

```
from fab_deploy import *

@define_host('my_site@example.com')
def my_site():
    return {
        # ...
    }
```

- In order to specify configuration the fab commands should use, run the configuring function as a first fab command:

```
fab my_site mysql_install
```

- In order to make configuration default call the configuring function at the end of `fabfile.py`:

```
from fab_deploy import *

def my_site():
    env.hosts = ['my_site@example.com']
    env.conf = {
        # ...
    }
```

```
    # ...
    update_env()

my_site()
```

This way it'll be possible to run fab commands omitting the config name:

```
fab mysql_install
```

Configuring

fab_deploy.utils.**update_env**()
Updates *env.conf* configuration with some defaults and converts it to state._AttributeDict (that's a dictionary subclass enabling attribute lookup/assignment of keys/values).

Call `update_env()` at the end of each server-configuring function.

```
from fabric.api import env, task
from fab_deploy.utils import update_env

@task
def my_site():
    env.hosts = ['my_site@example.com']
    env.conf = dict(
        DB_USER = 'my_site',
        DB_PASSWORD = 'password',
    )
    update_env()
```

env.**hosts**
A list with host string. Example:

```
env.hosts = ['user@example.com']
```

See fabric docs for explanation.

User obtained from this string will be used for ssh logins and as a default value for *env.conf.INSTANCE_NAME*.

Note: multiple hosts are supported via multiple config functions, not via this option.

Warning: Due to bug in Fabric please don't use `env.user` and `env.port`. Put the username and non-standard ssh port directly into host string.

env.**conf**
django-fab-deploy server configuration.

All *env.conf* keys are available in config templates as jinja2 template variables.

env.conf.**INSTANCE_NAME**
Project instance name. It equals to username obtained from *env.hosts* by default. INSTANCE_NAME should be unique for server. If there are several sites running as one linux user, set different INSTANCE_NAMEs for them.

env.conf.**SERVER_NAME**
Site url for webserver configs. It equals to the first host from *env.hosts* by default.

env.conf.**DB_NAME**
Database name. It equals to *env.conf.INSTANCE_NAME* by default.

env.conf.**DB_USER**
Database user. It equals to 'root' by default.

env.conf.**DB_PASSWORD**
Database password.

env.conf.**DB_ROOT_PASSWORD**
Database password for a 'root' user. django-fab-deploy will ask for mysql root password when necessary if this option is not set.

env.conf.**SUDO_USER**
User with sudo privileges. It is 'root' by default. Use *create_sudo_linux_account* in order to create non-root sudoer.

env.conf.**PROCESSES**
The number of mod_wsgi daemon processes. It is a good idea to set it to number of processor cores + 1 for maximum performance or to 1 for minimal memory consumption. Default is 1.

env.conf.**THREADS**
The number of mod_wsgi threads per daemon process. Default is 15.

Note: Set *env.conf.THREADS* to 1 and *env.conf.PROCESSES* to a bigger number if your software is not thread-safe (it will consume more memory though).

---

env.conf.**OS**
A string with server operating system name. Set it to the correct value if autodetection fails for some reason. Supported operating systems:

- lenny
- squeeze
- maverick

env.conf.**VCS**
The name of VCS the project is stored in. Supported values:

- hg
- git
- none

Default is 'hg'.

VCS is used for making project clones and for pushing code updates. 'none' VCS is able to upload tar.gz file with project sources on server via ssh and then extract it. Please prefer 'hg' or 'git' over 'none' if possible.

One can write custom VCS module and set *env.conf.VCS* to its import path:

```
env.conf = dict(
    # ...
    VCS = 'my_utils.my_vcs',
)
```

VCS module should provide 'init', 'up', 'push' and 'configure' functions. Look at `fab_deploy.vcs.hg` or `fab_deploy.vcs.none` for examples.

`env.conf.`**`HG_BRANCH`**
Named hg branch that should be active on server. Default is "default". This option can be used to have 1 repo with several named branches and run different servers from different branches.

`env.conf.`**`GIT_BRANCH`**
Git branch that should be active on server. Default is "master". This option can be used to run different servers from different git branches.

`env.conf.`**`PROJECT_PATH`**
Path to django project (relative to repo root). Default is ''. This should be set to a folder where project's manage.py reside.

`env.conf.`**`LOCAL_CONFIG`**
Local django config file name. Default is 'config.py'. Common values include 'local_settings.py' and 'settings_local.py'. This file should be placed inside *env.conf.PROJECT_PATH*, imported from settings.py and excluded from version control.

Note: Default value is not set to one of widely-used file names by default (e.g. 'local_settings.py') in order to prevent potential data loss during converting existing project to django-fab-deploy: this file is overwritten on server during deployment process; it is usually excluded from VCS and contains important information.

`env.conf.`**`REMOTE_CONFIG_TEMPLATE`**
The name of file with remote config template. Default is 'config.server.py'. This file should be placed inside *env.conf.PROJECT_PATH*. It will become *env.conf.LOCAL_CONFIG* on server.

`env.conf.`**`CONFIG_TEMPLATES_PATHS`**
An iterable with paths to web server and other config templates. Default is `['config_templates']`.

`env.conf.`**`PIP_REQUIREMENTS_PATH`**
Default is 'reqs'. This path is relative to repo root.

`env.conf.`**`PIP_REQUIREMENTS`**
The name of main requirements file. Requirements from it are installed during deployment. Default is 'all.txt'.

`env.conf.`**`PIP_REQUIREMENTS_ACTIVE`**
The name of pip requirements file with commonly updated requirements. Requirements from this file are updated by `fab_deploy.virtualenv.pip_install()` and `fab_deploy.virtualenv.pip_update()` commands when they are executed without arguments.

`fab push:pip_update` command also updates only requirements listed here.

Default is 'all.txt'.

You can put any other variables into the *env.conf*. They will be accessible in config templates as template context variables.

Writing custom commands

While django-fab-deploy commands are just Fabric commands, there are some helpers to make writing them easier.

`fab_deploy.utils.`**`inside_project`**(*func*)
Decorator. Use it to perform actions inside remote project dir (that's a folder where `manage.py` resides) with virtualenv activated:

```
from fabric.api import run, task
from fab_deploy.utils import inside_project

@task
@inside_project
def cleanup():
    # the current dir is a project source dir and
    # virtualenv is activated
    run('python manage.py cleanup')
```

fab_deploy.utils.**inside_src**(*func*)
Decorator. Use it to perform actions inside remote source dir (repository root) with virtualenv activated.

fab_deploy.utils.**run_as_sudo**(*func*)
Decorator. By default all commands are executed as user without sudo access for security reasons. Use this decorator to run fabric command as user with sudo access (*env.conf.SUDO_USER*):

```
from fabric.api import run, task
from fab_deploy import utils

@task
@utils.run_as_sudo
def aptitude_update():
    run('aptitude update')
```

fab_deploy.utils.**define_host**(*host_string*, *defaults=None*)
This decorator populates *env.hosts*, *env.conf* and calls *update_env()*:

```
from fab_deploy.utils import define_host

@define_host('my_site@example.com')
def my_site():
    return {
        'DB_USER': 'my_site',
        'DB_PASSWORD': 'password',
    }
```

Decorated function should return a dict with desired *env.conf* values.

There is no need to wrap function in @fabric.api.task decorator because *define_host* will do it for you.

define_host also accepts a dict with default values:

```
from fab_deploy.utils import define_host

DEFAULTS = dict(
    PROCESSES = 3,
    VCS = 'git',
)

@define_host('my_site@example.com', DEFAULTS)
def my_site():
    return {
        'DB_USER': 'my_site',
        'DB_PASSWORD': 'password',
        'PROCESSES': 2,
    }

# env.conf will contain PROCESSES=2 and VCS='git'.
```

Reference

Note: This is auto-generated API reference. Don't expect much from it.

[source] links are most useful.

Warning: django-fab-deploy is still at early stages of development and API may change in future.

Django

Deployment

fab_deploy.system.**create_linux_account**
Creates linux account, setups ssh access and pip.conf file.

Example:

```
fab create_linux_account:"/home/kmike/.ssh/id_rsa.pub"
```

fab_deploy.system.**create_sudo_linux_account**
Creates linux account, setups ssh access and adds the created user to sudoers. This command requires root ssh access.

fab_deploy.system.**ssh_add_key**
Adds a ssh key from passed file to user's authorized_keys on server.

fab_deploy.system.**ssh_add_root_key**
Adds a ssh key from passed file to root's authorized_keys on server.

fab_deploy.system.**install_sudo**
Installs sudo on server.

Virtualenv/pip

MySQL

Working with crontab

fab_deploy.crontab.**set_content**
Sets crontab content

fab_deploy.crontab.**add_line**
Adds line to crontab. Line can be appended with special marker comment so it'll be possible to reliably remove or update it later.

fab_deploy.crontab.**puts_content**
Shows current crontab

`fab_deploy.crontab.`**`remove_line`**
Removes a line added and marked using add_line.

`fab_deploy.crontab.`**`update_line`**
Adds or updates a line in crontab.

`fab_deploy.crontab.`**`add_management`**
Adds django management command to crontab.

- `when` - crontab's 'when' part (m h dom mon dow)
- `command` - django management command (with all options)
- `marker` - unique marker for future command updating or removing

Example:

```
$ fab crontab_add_management:"0 0 * * *","cleanup"
```

Web servers

Test suite

django-fab-deploy test suite executes fab commands against VirtualBox virtual machines. Full test suite can take a very long time to run (e.g. about 25 minutes for 4mbps broadband, the exact time depends heavily on internet connection speed): all operations are really performed.

VM is rolled back to a clean state or an appropriate snapshot before each test.

This approach is quite extreme but I believe it's the only way to make sure deployment system works: actually execute the deployment scripts against concrete servers.

Preparations

django-fab-deploy requires latest fabtest and mock packages for running tests and (optionally) coverage.py for test coverage reports:

```
pip install -U fabtest
pip install 'mock==0.8'
pip install coverage
```

Please follow instructions for fabtest package in order to prepare OS image. django-fab-deploy tests have 1 additional requirement: root user should have '123' password (fabtest example VM images are configured this way).

Running tests

Pass VM name (e.g. Squeeze) to runtests.py script:

```
cd fab_deploy_tests
./runtests.py <VM name or uid> <what to run>
```

<what to run> can be `misc`, `deploy`, `all`, `prepare` or any value acceptable by `unittest.main()` (e.g. a list of test cases).

Some tests require additional prepared snapshots in order to greatly speedup test execution. But there is a chicken or the egg dilemma: these snapshots can be only taken if software works fine for the VM (at least tests are passed). So there is a very slow `prepare` test suite that ensures preparing will work.

1. make sure slow tests are passing:

```
./runtests.py "VM_NAME" prepare
```

2. prepare snapshots:

```
./preparevm.py "VM_NAME"
```

3. tests can be run now:

```
./runtests.py "VM_NAME" all
```

Note: Tests asking for user input (usually for password) should be considered failed. They mean django-fab-deploy was unable to properly setup server given the root ssh access.

Note: Mercurial can't preserve 0600 file permissions and ssh is complaining if private key is 0644. So in order to run tests change permissions for the `fab_deploy_testskeysid_rsa` to 0600:

```
chmod 0600 fab_deploy_tests/keys/id_rsa
```

Coverage reports

In order to get coverage reports run:

```
cd fab_deploy_tests
./runcoverage.sh <VM name or uid> <what to run>
```

html reports will be placed in `htmlcov` folder.

Related work

There are great projects aiming the same goal. Many of them are listed here: http://djangopackages.com/grids/g/deployment/

Make sure you've read the following document if you are upgrading from previous versions of django-fab-deploy:

CHANGES

dev (TBA)

Major changes:

- All tasks are now new-style fabric tasks;
- postgres (+ postgis) support;

- db backends and vcs backends are now fully reusable and extendable via subclassing (they use https://github.com/kmike/fabric-taskset);
- modules are moved (apache and nginx become webserver.apache and webserver.nginx, mysql become db.mysql);
- task prefixes are removed in favor of namespaces (so e.g. `apache_restart` becomes `apache.restart`);
- new port management facilities based on port-for.

Other changes:

- `DB_USER` default value is removed as per deprecation plan (it will be changed to non-root default in the next release);
- migrate command no longer converts backup errors to warnings;
- example hgrc config is removed from config_templates;
- `crontab.add_management` task;
- better staticfiles example in default nginx config template;
- `ssh_add_root_key` command;
- `install_sudo` no longer fails if `aptitude update` was not called;
- support for MS Windows on development machine is improved;
- `execute_sql` selects current database by default for non-superuser queries.
- memcached is no longer installed by default; add `fab_deploy.system.aptitude_install('memcached')` to your deploy script if you need memcached.

This release has command-line interface that is different from 0.7.x branch because of moved modules and switch to new-style fabric tasks. It is also incompatible in python level because of the same reasons. So custom tasks should be updated: wrap them with `fabric.api.task` decorator + update import paths and task names for tasks from django-fab-deploy.

Server itself should also be upgraded. Django-fab-deploy 0.8 introduced much more powerful port management based on port-for.

In order to upgrade from 0.7.x to 0.8.x new port management:

1. Run 'fab_deploy.system.install_software' task for each of your hosts: add

   ```
   from fab_deploy import system
   ```

   line to your fabfile.py (or fabfile/__init__.py) and execute

   ```
   $ fab <host_function_name> system.install_software
   ```

   for each host.

2. Replace `{{ APACHE_PORT }}` with `{{ PORTS['apache'] }}` in apache and nginx configs (`apache.conf` and `nginx.conf`);
3. Add `Listen 127.0.0.1:{{ PORTS['apache'] }}` directive to the top of your `apache.conf`.
4. Run `apache.install`, `apache.update_config` and `nginx.update_config` tasks for each of your hosts.

0.7.5 (2012-03-02)

- root_password argument for `mysql_create_db` (thanks Michael Brown).

0.7.4 (2012-03-01)

- django-fab-deploy now is compatible with fabric 1.4 (and require fabric 1.4);
- nginx and wsgi scripts are now compatible with upcoming django 1.4; example of django 1.4 project configuration is added to guide;
- shortcut for passing env defaults in `define_host` decorator;
- Ubuntu 10.04 apache restarting fix;
- config_templates/hgrc is removed;
- tests are updated for fabtest >= 0.1;
- `apache_is_running` function.

In order to upgrade install fabric >= 1.4 and make sure your custom scripts work.

0.7.3 (2011-10-13)

- permanent redirect from www.domain.com to domain.com is added to the default nginx config. Previously they were both available and this leads to e.g. authorization issues (user logged in at www.domain.com was not logged in at domain.com with default django settings regarding cookie domain).

0.7.2 (2011-06-14)

- Ubuntu 10.04 (lucid) initial support (this needs more testing);
- backports for Ubuntu 10.04 and 10.10;
- docs are now using default theme;
- remote django management command errors are no longer silinced;
- invoking `create_linux_account` with non-default username is fixed;
- `define_host` decorator for easier host definition;
- default `DB_USER` value ('root') is deprecated;
- default nginx config uses `INSTANCE_NAME` for logs.

In order to upgrade please set `DB_USER` to 'root' explicitly in `env.conf` if it was omitted.

0.7.1 (2011-04-21)

- DB_ROOT_PASSWORD handling is fixed

0.7 (2011-04-21)

- requirement for root ssh access is removed: django-fab-deploy is now using sudo internally (thanks Vladimir Mihailenco);
- better support for non-root mysql users, `mysql_create_user` and `mysql_grant_permissions` commands were added (thanks Vladimir Mihailenco);
- hgrc is no more required;
- 'synccompress' management command is no longer called during `fab up`;
- `coverage` command is disabled;
- `nginx_setup` and `nginx_install` are now available in command line by default;
- `mysqldump` no longer requires project dir to be created;
- home dir for root user is corrected;
- `utils.detect_os` is now failing loudly if detection fails;
- numerous test running improvements.

In order to upgrade from previous verions of django-fab-deploy, install sudo on server if it was not installed:

```
fab install_sudo
```

0.6.1 (2011-03-16)

- `verify_exists` argument of `utils.upload_config_template` function was renamed to `skip_unexistent`;
- `utils.upload_config_template` now passes all extra kwargs directly to fabric's `upload_template` (thanks Vladimir Mihailenco);
- `virtualenv.pip_setup_conf` command for uploading pip.conf (thanks Vladimir Mihailenco);
- `deploy.push` no longer calls 'synccompress' management command;
- `deploy.push` accepts 'before_restart' keyword argument - that's a callable that will be executed just before code reload;
- fixed regression in `deploy.push` command: 'notest' argument was incorrectly renamed to 'test';
- customization docs are added.

0.6 (2011-03-11)

- custom project layouts support (thanks Vladimir Mihailenco): standard project layout is no longer required; if the project has pip requirements file(s) and a folder with web server config templates it should be possible to use django-fab-deploy for deployment;
- git uploads support (thanks Vladimir Mihailenco);
- lxml installation is fixed;
- sqlite deployments are supported (for testing purposes).

If you are planning to migrate to non-default project layout, update the config templates:

- in `apache.config` and `nginx.config`: replace `{{ SRC_DIR }}` with `{{ PROJECT_DIR }}`

- in `django_wsgi.py`: replace `{{ SRC_DIR }}` with `{{ PROJECT_DIR }}` and make sure DJANGO_SETTINGS_MODULE doesn't contain INSTANCE_NAME:

```
os.environ['DJANGO_SETTINGS_MODULE'] = 'settings'
```

0.5.1 (2011-02-25)

- Python 2.5 support for local machine (it was always supported on servers). Thanks Den Ivanov.

0.5 (2011-02-23)

- OS is now auto-detected;
- Ubuntu 10.10 maverick initial support (needs better testing?);
- fabtest package is extracted from the test suite;
- improved tests;
- `fab_deploy.system.ssh_add_key` can now add ssh key even if it is the first key for user;
- 'print' calls are replaced with 'puts' calls in fabfile commands;
- django management commands are not executed if they are not available.

You'll probably want to remove `env.conf.OS` option from your fabfile.

If you're planning to deploy existing project to Ubuntu, add `NameVirtualHost 127.0.0.1:{{ APACHE_PORT }}` line to the top of your `config_templates/apache.conf` or delete the templates and run `django-fab-deploy config_templates` again.

0.4.2 (2011-02-16)

- tests are included in source distribution

0.4.1 (2011-02-14)

- don't trigger mysql 5.1 installation on Lenny

0.4 (2011-02-13)

- `env.conf.VCS`: mercurial is no longer required;
- undeploy command now removes virtualenv.

0.3 (2011-02-12)

- Debian Squeeze support;
- the usage of `env.user` is discouraged;
- `fab_deploy.utils.print_env` command;
- `fab_deploy.deploy.undeploy` command;

- better `run_as` implementation.

In order to upgrade from 0.2 please remove any usages of `env.user` from the code, e.g. before upgrade:

```
def my_site():
    env.hosts = ['example.com']
    env.user = 'foo'
    #...
```

After upgrade:

```
def my_site():
    env.hosts = ['foo@example.com']
    #...
```

0.2 (2011-02-09)

- Apache ports are now managed automatically;
- default threads count is on par with mod_wsgi's default value;
- `env.conf` is converted to _AttributeDict by `fab_deploy.utils.update_env`.

This release is backwards-incompatible with 0.1.x because of apache port handling changes. In order to upgrade,

- remove the first line ('Listen ...') from project's `config_templates/apache.config`;
- remove APACHE_PORT settings from project's `fabfile.py`;
- run `fab setup_web_server` from the command line.

0.1.2 (2011-02-07)

- manual config copying is no longer needed: there is django-fab-deploy script for that

0.1.1 (2011-02-06)

- cleaner internals;
- less constrains on project structure, easier installation;
- default web server config improvements;
- linux user creation;
- non-interactive mysql installation (thanks Andrey Rahmatullin);
- new documentation.

0.0.11 (2010-01-27)

- fab_deploy.crontab module;
- cleaner virtualenv management;
- inside_project decorator.

this is the last release in 0.0.x branch.

0.0.8 (2010-12-27)

Bugs with multiple host support, backports URL and stray 'pyc' files are fixed.

0.0.6 (2010-08-29)

A few bugfixes and docs improvements.

0.0.2 (2010-08-04)

Initial release.

CHAPTER 3

Bug tracker

If you have any suggestions, bug reports or annoyances please report them to the issue tracker at https://bitbucket.org/kmike/django-fab-deploy/issues/new

CHAPTER 4

Contributing

Development of django-fab-deploy happens at Bitbucket: https://bitbucket.org/kmike/django-fab-deploy/

You are highly encouraged to participate in the development of django-fab-deploy. If you don't like Bitbucket or Mercurial (for some reason) you're welcome to send regular patches.

Authors

- Mikhail Korobov (primary author);
- Andrey Rahmatullin (mysql install script, postgresql backups);
- Den Ivanov (python 2.5 compatibility);
- Vladimir Mihailenco (git support, support for custom project layouts, imports overhaul, etc.);
- Ruslan Popov;
- Michael Brown;
- Denis Untevskiy (windows support, bug fixes).

CHAPTER 5

License

Licensed under a MIT license.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index