django-duke-client Documentation

Release dev

Maxime Haineault

Sep 27, 2017

Contents

Contents:

Django duke client aims provide a turnkey development environment for django and make django development easier and faster.

CHAPTER 1

Overview

Contents

What is Django Duke client ?

Django duke client aims provide a turnkey development environment for django and make django development easier and faster. He set up a number of strategies to reduce setup time. It can also install, delete, or edit external sources easy, and quickly through a single file.

TODO..

Feature overview

- Uses and installs its own modules (for example, python, django), and thus avoids the potential for conflict.
- shortcuts
- deployment simple, configurable, and fast
- custom possible by overwriting files, functions moduls ...

CHAPTER 2

Screenshots

Contents

duke commands

duke tartproject

duke init

Dev. environment commands

buildout

dev or develop

django

Dev. environment shortcuts

runserver

syncdb

CHAPTER 3

Installation

Contents

Official releases

Official releases will eventually be available from PyPI.

Download the .zip distribution file and unpack it. Inside is a script named `setup.py`. Enter this command:

```
python setup.py install
```

...and the package will install automatically.

Development version

Alternatively, you can get the latest source from our git repository:

```
git clone git://github.com/h3/django-duke-client.git
```

Add the resulting folder to your PYTHONPATH or symlink the `dukeclient` directory inside it into a directory which is on your PYTHONPATH, such as your Python installation's `site-packages` directory.

You can verify that the application is installed by typing the following command in a terminal:

```
$: duke help
```

When you want to update your copy of the source code, run `git pull` from within the `django-duke-client` directory.

Caution: The development version may contain bugs which are not present in the release version and introduce backwards-incompatible changes.

If you're tracking master, keep an eye on the recent Commit History before you update your copy of the source code.

CHAPTER 4

Getting started tutorial

Contents

Project layout

The django duke client tries to be independent as possible in term of project layout. However a minimal structure is required for it to work properly.

This is the absolute minimal project layout to initialize duke:

```
project-root-folder/
  - setup.py
```

When the project is built, it looks like this:

```
project-root-folder/
  - bootstrap.py
  - buildout.cfg
```

```
  - dev.cfg
  - setup.py
  + .duke/
    + bin/
    + develop-eggs/
    + eggs/
    + parts/
    + src/
  + projectname.egg-info
  + projectname/
    - settings.py
    - local_settings.py
    + conf/
        + settings/
            - default.py
            - dev.py
```

Starting a project from scratch

Creating a new project from scratch is easy as:

```
user@host$ duke startproject my-project-name
user@host$ cd my-project-name/
user@host$ ls
README.rst  setup.py
```

The *setup.py* file is the only required file for a new project. The README.rst is created only for convenience. The next step is to edit the setup file according to your needs.

Initializing your project

Next we need to initialize duke on this project. Which can be done like so:

```
user@host$ duke init myprojectname
Installing bootstrap.py..
Installing default buildout.cfg
Installing default base.cfg
Installing default dev.cfg
Installing default prod.cfg
Initializing zc.buildout
Creating directory '/tmp/my-project-name/.duke/bin/'.
Creating directory '/tmp/my-project-name/.duke/parts/'.
Creating directory '/tmp/my-project-name/.duke/eggs/'.
Creating directory '/tmp/my-project-name/.duke/develop-eggs/'.
Generated script '/tmp/my-project-name/.duke/bin/buildout'.
Installing dev hooks
Done!
```

As you can see, the init command setup and configure *buildout* for the project and put most of the stuff in a folder name *.duke/*. **This folder should not be added to your VCS.** It is meant to be recreated easily.

Caution: It's important to understand the difference between *my-project-name* and *myprojectname*. The first is only the folder containing your project. Its name doesn't really matters. If you are using SVN you should probably use trunk as folder name to match SVN folder naming conventions.

On the other side, *myprojectname* is your real django project name. Duke will create it automatically only if there isn't already a project of that name in the folder.

Once the initialization done, django duke automatically enters development mode (which can be done by typing *duke dev* in your project folder).

You know when you are in development mode when your shell prompt is prefixed with a project name like this:

```
user@host|myprojectname:~/.../trunk/my-project-name$ ls
bootstrap.py  buildout.cfg  dev.cfg  prod.cfg  README.rst  setup.py
```

You can see django duke created different configuration files which will be covered later in the documentation.

Your command line prompt also has been changed. It now includes your project name so you always know in which sandbox you are working on. It also indicate if you are in a Subversion or Git repository. This is all customizable.

Building your project

At this point you need to edit *buildout.cfg* to add the requirements you need and buildout your project:

```
user@host|myprojectname:~/.../trunk/my-project-name$ buildout
Getting distribution for 'mr.developer'.
warning: no files found matching 'README.txt'
Got mr.developer 1.19.
Getting distribution for 'buildout.dumppickedversions'.
Got buildout.dumppickedversions 0.5.
Getting distribution for 'elementtree'.
zip_safe flag not set; analyzing archive contents...
Got elementtree 1.2.6-20050316.
mr.developer: Creating missing sources dir /tmp/my-project-name/src.
mr.developer: Queued 'djangodukerecipe' for checkout.
mr.developer: Cloned 'djangodukerecipe' with git.
Develop: '/tmp/my-project-name/src/djangodukerecipe'
Develop: '/tmp/my-project-name/.'
Getting distribution for 'zc.recipe.egg'.
Got zc.recipe.egg 1.3.2.
Getting distribution for 'z3c.recipe.scripts'.
Got z3c.recipe.scripts 1.0.1.
Unused options for buildout: 'downloads-directory'.
Installing _mr.developer.
Generated script '/tmp/my-project-name/.duke/bin/develop'.
Installing python.
Getting distribution for 'simplejson'.
zip_safe flag not set; analyzing archive contents...
simplejson.tests.__init__: module references __file__
Got simplejson 2.3.2.
Generated interpreter '/tmp/my-project-name/.duke/bin/python'.
Installing djangodev.
Generated script '/tmp/my-project-name/.duke/bin/djangodev'.
Generated script '/tmp/my-project-name/.duke/bin/djangodev.wsgi'.
```

Once buildout has been run for the first time, you'll see new files in your project folder:

```
user@host|myprojectname|svn:~/.../trunk/my-project-name$ ls -a
bootstrap.py  buildout.cfg  dev.cfg  .duke  myprojectname/
my_project_name.egg-info/  prod.cfg  README.rst  setup.py  src/
```

Start working !

At this point you can start working on your django project:

```
user@host|myprojectname|svn:~/.../trunk/my-project-name$ cd projectname/
user@host|myprojectname|svn:~/.../trunk/my-project-name$ django syncdb
user@host|myprojectname|svn:~/.../trunk/my-project-name$ django runserver
```

You don't need to type *python manage.py*, there is a short cut named *django*. In fact there is many useful shortcuts for django:

- dbshell
- dumpdata
- loaddata
- runserver
- shell
- syncdb

To see the full list of available commands type *duke help*.

Customization

You can tweak your development environment quite alot.

To do so, simply type this command:

```
user@host$ duke customize
Copying setup.py to ~/.duke/templates/
Copying profile to ~/.duke/templates/
Copying bootstrap.py to ~/.duke/templates/
Copying gitignore to ~/.duke/templates/
Copying buildout.cfg to ~/.duke/templates/
Copying project_conf.yml to ~/.duke/templates/
Copying dev to ~/.duke/templates/
Copying env to ~/.duke/templates/
Copying duke_conf.yml to ~/.duke/templates/
Copying base.cfg to ~/.duke/templates/
Copying svnignore to ~/.duke/templates/
Copying dev.cfg to ~/.duke/templates/
```

Now any modification made to files copied in *~/.duke/templates/* will take precedence over those used normally by duke.

If you want to change the command prompt, you will need to modify *~/.duke/templates/profile*.

If there is not enough options for your taste, you can tweak *~/.duke/templates/env*. Be warned that it might put your bashfu to test.

Note that you will need to restart your environment for the changes to take effect.

To do so, simply hit Ctrl+D (or exit) and retype *duke dev*.

Finally, resist the temptation of editing files in *.duke/bin/* as they are recreated each time you run the buildout command. Per project configuration is not supported as now, but it should be sufficiently easy to implement to be supported sooner than later.

Don't hesitate to share your improvements with me ! :)

References

setup.py	http://www.buildout.org/docs/tutorial.html
Buildout	http://www.buildout.org/docs/ http://pypi.python.org/pypi/zc.buildout/1.5.2
djangorecipe	http://pypi.python.org/pypi/djangorecipe/0.99
z3c.recipe.scripts	http://pypi.python.org/pypi/z3c.recipe.scripts
mr.developer	http://pypi.python.org/pypi/mr.developer
Django	https://docs.djangoproject.com/
django/buildout	http://jacobian.org/writing/django-apps-with-buildout/

CHAPTER 5

Development mode

Contents

Introduction

One of the key feature of django duke is to provide a sandboxed development environment which provide some shortcuts and utilities to make it easier to work with django.

To activate the development environment on a project managed by duke, simply go in the project folder and type *duke dev*:

```
user@host$: cd my-project-name/
user@host$: duke dev
user@host|myprojectname:~/.../trunk/my-project-name$:
```

Once the development environment is activated, the shell prompt should be prefixed with the project's name to indicate that you are working within a sandboxed environment.

The distinction is important because the development environment extends your shell with new commands and does some magic to make sure you are working within the sandbox.

For example, if you type the command *python* in dev mode, the Python interpreter executed isn't the system wide python interpreter (usually */usr/bin/python*). Instead it will call the python interpreter sandboxed in *my-project-name/.duke/bin/python*.

The environment also provide shortcuts and commands to ease the development process.

Commands & shortcuts

Command	Description
buildout	Run buildout to build or rebuild your environment
dbshell	Run the database shell (alias for *django runserver*)
dev	Run the duke development environement
python	Run the python environement
runserver	Run the dev server (alias for *django runserver*)
shell	Run a python shell* (alias for *django shell*)
syncdb	Synchronize database (alias for *django syncdb*)
init	Create the duke environement (run buildout after using init)
startproject	Create a new django project
customize	Copy the config of duke to ~/.duke (see *Customisation*)
help	Print all the commandes

* if ipython is install, ipython will be ruen instead of python

CHAPTER 6

Project configuration

Contents

Introduction

Project configuration is made with the buildout configuration files. By default there is only two *cfg* files; *buildout.cfg* and *dev.cfg*.

It is possible to create stage specific configuration by adding *cfg* files named after the stage name which extends *buildout.cfg*.

For example, if I have a stage named *prod* on which I want to configure cron jobs, I simply have to create a *prod.cfg* file in which I put the required configuration.

At deploy time, duke will use *prod.cfg* instead of *buildout.cfg*.

buildout.cfg

The main configuration is *buildout.cfg*, it should be complete and functional stand alone as this is the configuration used in production.

dev.cfg

This configuration file is used only for development, it extends *buildout.cfg*.

You can extend individual configurations keys like so:

```
[buildout]
extends = buildout.cfg

# extend
eggs +=
 ipython
```

If you wish to overwrite it instead, simply remove the + sign.

Configurations

[duke]

Directive	Default	Description
django	${buildout:directory}/.duke/bin/django	Shortcut to django executable
cron	${buildout:directory}/cron/	Path where cron jobs script are stored

[buildout]

Directive	Default	Description
allowed-eggs-from-site-packages	PIL, MySQL-python, ...	Use this directive to tell buildout which system wide package it can use*
auto-checkout	djangoduk-erecipe	List of modules sources to auto checkout
develop	.	List of editable modules to install with develop
eggs	none	List of eggs to install (project requirements)
exec-sitecustomize	false	Normally the Python's real sitecustomize module is not processed
extensions	mr.developer	Buildout extensions to load
include-site-packages	true	We allow site packages unless allowed-eggs-from-site-packages is specified
index	http://pypi.python.org	HTTP URL of pypi (default) or a pypi mirror
newest	false	Check for new packages versions
parts	python, django, scripts	Buildout parts to run (ex: python, djangodev)
unzip	true	Zipped eggs make debugging more difficult and often import more slowly
versions	versions	Freeze eggs or sources to specific versions
sources	sources	This specifies the name of a section which lists the repositories
sources-dir	src	This specifies the directory where your package sources will be placed
auto-checkout	src	This specifies the names of packages which should be checked out during buildout. Packages already checked out are skipped. You can use * as a wildcard for all packages in *sources*
always-checkout	false	This defaults to false. If it's true, then all packages specified by auto-checkout and currently in develop mode are updated during each buildout run. If set to force, then packages are updated even when they are dirty instead of asking interactively.
always-accept-server-certificate	false	If it's true, invalid server certificates are accepted without asking (for subversion repositories)

- If allowed-eggs-from-site-packages is an empty list, then no eggs from site-packages are chosen, but site-packages will still be included at the end of path lists.

[python]

interpreter	Name of the Python interpreter (default *python*)
extra-paths	List of paths to add to the PYTHONPATH. Note that you must add paths of modules installed from sources here. The path should look like this: *${buildout:directory}/src/mptt*

[django]

Directive	Default	Description
extra-paths	*${python:extra-paths}*	
settings	*settings*	Name of the django settings module
wsgi	false	
project	false	The project name

[sources]

Example:

```
[sources]
django = git git://github.com/django/django.git
django-mptt = git git://github.com/django-mptt/django-mptt.git branch=reodering_test
django-fiber = git://github.com/ridethepony/django-fiber.git update=true
```

Supported source kinds: svn, hg, git, bzr, darcs, cvs, and fs.

When adding new sources, don't forget to also add them in to the *extra-paths* of the *[python]* section and the *auto-checkout* in the *[buildout]* section.

[versions]

Example:

```
[versions]
django=1.4
PIL=1.7.1
```

Working with sources

If you work with source packages You need to edit tree configs.

Tell buildout to checkout the package every time:

```
[buildout]
auto-checkout +=
 django
```

Then specify the source URL:

```
[sources] # svn, hg or git
django = git git://github.com/django/django.git
```

Finally, add it to the environment's *PYTHONPATH* like this:

```
[python]
extra-paths +=
 ${buildout:directory}/src/django
```

CHAPTER 7

Deploying

Contents

With fabric

Caution: This is pretty much alpha stuff, it might change a lot in the future.

Currently the duke client only offer some useful fabric tasks for standard django deployment.

Project Configurations

To use it, simply create a file named *fabfile.py* in the root directory of your project (where your *setup.py* file is).

The file content should look like this:

```
import os

from dukeclient.fabric.utils import get_role, get_conf, get_project_path
from dukeclient.fabric.tasks import *

LOCAL_PATH = os.path.dirname(os.path.abspath(__file__))

env.roledefs.update({
    'demo': ['user@demo.host.com'],
    'prod': ['user@production.host.com:5555'],

    # Not required, but can be useful if you want to invoke commands
    # on multiple servers at once.
    'http_servers': ['user@production.host.com:5555', 'user@demo.host.com'],
})

env.site = {
    'domain':   'mysite.com',
    'package':  'mysite.com',
    'project':  'mysite',
    'repos':    'svn://svn.myserver.com/mysite.com/trunk/mysite.com/',
}

env.roleconfs = {

    # This is an example of how you can deploy on Plesk
    'prod': {
        'hosts': env.roledefs['prod'],
        'user': 'username',
        'group': 'usergroup',
        'document-root': '/var/www/vhosts/%(domain)s/httpdocs/',
        'vhost-conf': '/var/www/vhosts/%(domain)s/conf/vhost.conf',

        # Most commands uses an event system which will run scripts
        # at specific times.
        'on-code-sync': [],
        'on-code-sync-done': [],
        'on-apache-reload': [
            # You can run scripts before and after most of the available
            # commands. In this case we tell Plesk to reload its vhost
            # configuration for mysite.com
            '/usr/local/psa/admin/sbin/websrvmng --reconfigure-vhost --vhost-name=
↪%(domain)s',
        ],
        'on-apache-reload-done': [],

        # If mod_python is installed on your Apache server, you'll need
        # virtualenv or you will go insane. Really.
        'virtualenv': True,
    },

    # This example show a more basic Apache deployment
    'demo': {
        'hosts': env.roledefs['demo'],
        'document-root': '/var/www/vhosts/demo.%(domain)s/',
        'media-root':  '/var/www/vhosts/demo.%(domain)s/%(domain)s/%(project)s/media/
↪',
        'static-root': '/var/www/vhosts/demo.%(domain)s/%(domain)s/static/',
```

```
        'vhost-conf': '/etc/apache2/sites-enabled/demo.%(domain)s',
        'virtualenv': True,
        'user': 'www-data',
        'group': 'www-data',
        'on-deploy-done': [
            'ln -sf /var/www/vhosts/demo.%(domain)s/%(domain)s/%(project)s/media/ /
↪var/www/vhosts/demo.%(domain)s/media',
        ],
    },

}
```

Deployment configurations

Deployment configurations must be stored in a directory named *deploy/* in the root directory of your project.

Virtualhost

Virtual host files a threated as template, so you don't have to adjust them every time you change a configuration.

The naming convention is *<role>.vhost*. So if you have a *demo* and a *prod* role, your vhost files should be name *demo.vhost* and *prod.vhost*.

Here's an example of a standard Apache/WSGI vhost configuration file:

```
<VirtualHost *:80>
    ServerAdmin max@motion-m.ca
    DocumentRoot %(document-root)s
    ServerName %(project)s.d.motion-m.ca
    ErrorLog /var/log/apache2/%(package)s.d.motion-m.ca-error_log
    CustomLog %(project)s.d.motion-m.ca common
    Options FollowSymLinks
    WSGIPassAuthorization On
    WSGIScriptAlias / %(document-root)s%(package)s/%(project)s/wsgi.py
    WSGIDaemonProcess %(project)s user=www-data group=www-data processes=5 threads=1
    WSGIProcessGroup %(project)s
    Alias /static/ %(document-root)sstatic/
    Alias /media/ %(document-root)smedia/
    <Directory %(document-root)smedia/>
        Order deny,allow
        Allow from all
        AllowOverride None
    </Directory>
    <Directory %(document-root)sstatic/>
        Order deny,allow
        Allow from all
        AllowOverride None
    </Directory>
</VirtualHost>
```

Settings.py

The settings.py files can be automatically overwritten with a settings.py template.

For example, to set your project's settings on a role named *demo* you would start by creating a file named *deploy/demo_settings.py*.

Now every time you deploy your code, the file *deploy/demo_settings.py* gets copied over *myproject/local_settings.py*, overriding any other settings set elsewhere.

Here's an example which defines the default database backend:

```
from %(project)s.conf.settings.default import *

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.%%s' %% "mysql", # Read the caution below !
        'NAME': '%(project)s_demo',
        'USER': '%(project)s',
        'PASSWORD': '*********',
    }
}
```

Caution: Make sure to scape all the modulos by duplicating them like this: *%%*.

Since I use Advanced String Formating to replace the template variables you have to be careful when using a modulo (%). If you don't escape it Python will think it has to insert a token there and will most likely throw an exception at your next *buildout*.

Usage

Deploying

On *demo*:

```
fab -R demo full_deploy
```

On *prod*:

```
fab -R prod full_deploy
```

On both:

```
fab -R http_servers full_deploy
```

Updating

```
fab -R prod deploy
```

Caution: The *deploy* command will not update externals

Other commands

Other commands will eventually be documented properly .. meanwhile you can list them all using the *fab -l* command.

Per role configurations

Sometimes you want to tweak configurations depending on which role the project is running on.

To accomplish this, simply create a *cfg* file named after the role and make it extend the *buildout.cfg* file.

The next time buildout will be run on this role, it will find the file and use it instead of *buildout.cfg*.

Here's an example of how one could set a cron job on the production server:

prod.cfg:

```
[buildout]
extends = buildout.cfg
parts += django-cleanup

[django-cleanup]
recipe = z3c.recipe.usercrontab
times = @monthly
command = ${buildout:directory}/.duke/bin/django cleanup
```

Development roadmap

In the long term a *django duke master* will be created. The scope of the functionalities isn't yet fixed, but it's main purpose will be to act as a deployment server. It will hold servers and projects configurations and allow easy deployment using the *duke* command.

There is several advantages of using centralized deployment instead of a distributed deployment strategy (with fabric). But the most important advantage for us is to be able to assign deployment rights to developers without giving them actual access to the production servers.

When centralized deployment will be implemented, we will probably move to other nice to have features like scheduled deployment and continous integration and maybe even a plugin architecture for things like website monitoring, project management and such.

CHAPTER 8

Cheatsheet

Contents

Duke commands

Note: An <argument> ending with -r means the command accepts Regular Expressions

duke startproject <project-name> | Create a new project from scratch	
duke init <project_name>	Initialize a django project*
duke dev	Start development environment

*** Duke will create the project if it doesn't exists. It will also start the** developent environment when done.

Dev commands

buildout	Build or rebuild development env.

buildout | Build or rebuild development env. |

CHAPTER 9

Customization

Intitialisation

You can tweak your development environment quite alot.

To do so, simply type this command:

```
user@host$ duke customize
Copying setup.py to ~/.duke/templates/
Copying profile to ~/.duke/templates/
Copying bootstrap.py to ~/.duke/templates/
Copying gitignore to ~/.duke/templates/
Copying buildout.cfg to ~/.duke/templates/
Copying project_conf.yml to ~/.duke/templates/
Copying dev to ~/.duke/templates/
Copying env to ~/.duke/templates/
Copying duke_conf.yml to ~/.duke/templates/
Copying base.cfg to ~/.duke/templates/
Copying svnignore to ~/.duke/templates/
Copying dev.cfg to ~/.duke/templates/
```

Files

In the same shell:

```
user@host$ cd ~/.duke/templates
user@host$ ls
base.cfg  bootstrap.py  buildout.cfg  dev  dev.cfg  duke_conf.yml  env  gitignore␣
↪profile  project_conf.yml  setup.py  svnignore
```

The files *profile* and *env* are used to personalize your prompt, or to add commands (alias) .

Customize prompt

If you want to modify your duke prompt, (simply) you have to edit *profile*.

Here the default *profile* file:

```
# Typing "--settings=projectname.settingsfile" is annoying.
DJANGO_SETTINGS_MODULE=settings
SEP="|"
ENDCHAR="$ "
DUKE_DIRTRIM=2
DUKE_DJANGO_STR="django:"
DUKE_SVN_STR="svn:"
DUKE_GIT_STR="git:"
DUKE_PS1="${NO_COLOR}\u@\h${SEP}${BOLD_CYAN}\$(__in_project)${CYAN}%(project_name)s$
↪{NO_COLOR}${SEP}${BOLD_YELLOW}\$(__vcs_status)${YELLOW}\w${NO_COLOR}${ENDCHAR}"
```

will produce the following prompt:

```
user@host|projectname|svn:~/.../path/in/project$
```

You can change de value of the variable to personalize your prompt.

If it's not enough, you can edit the *env* file.

Caution: all that you make in *profile* overwrite *env*

Here the default *env* file:

```
. ~/.bashrc
# based on virtualenv's activate
# This file must be used with "source bin/activate" *from bash*
# you cannot run it directly

# Shell colors
BLACK="\[\e[0;30m\]"    BOLD_BLACK="\[\e[1;30m\]"   UNDER_BLACK="\[\e[4;30m\]"
RED="\[\e[0;31m\]"      BOLD_RED="\[\e[1;31m\]"     UNDER_RED="\[\e[4;31m\]"
GREEN="\[\e[0;32m\]"    BOLD_GREEN="\[\e[1;32m\]"   UNDER_GREEN="\[\e[4;32m\]"
YELLOW="\[\e[0;33m\]"   BOLD_YELLOW="\[\e[1;33m\]"  UNDER_YELLOW="\[\e[4;33m\]"
BLUE="\[\e[0;34m\]"     BOLD_BLUE="\[\e[1;34m\]"    UNDER_BLUE="\[\e[4;34m\]"
PURPLE="\[\e[0;35m\]"   BOLD_PURPLE="\[\e[1;35m\]"  UNDER_PURPLE="\[\e[4;35m\]"
CYAN="\[\e[0;36m\]"     BOLD_CYAN="\[\e[1;36m\]"    UNDER_CYAN="\[\e[4;36m\]"
WHITE="\[\e[0;37m\]"    BOLD_WHITE="\[\e[1;37m\]"   UNDER_WHITE="\[\e[4;37m\]"
NO_COLOR="\[\e[0m\]"

parse_git_branch () {
  git branch --no-color 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/\1/'
}

parse_git_dirty () {
  [[ $(git status 2> /dev/null | tail -n1) != "nothing to commit (working directory␣
↪clean)" ]] && echo "*"
}

__vcs_status () {
    if [ -d "$PWD/.svn" ]; then
        echo "$DUKE_SVN_STR"
```

```
    elif [ -n "$(parse_git_branch)" ]; then
        echo "$DUKE_GIT_STR"
    fi
}

# Prefix the command prompt with %(project_name)s
function __in_project {
    if [ "`pwd | xargs basename`" = "%(project_name)s" ] ; then
        echo "$DUKE_DJANGO_STR"
    else
        echo ""
    fi
}

# Duke client default environment variables

DUKE_ENV="%(base_path)s"
DUKE_DIRTRIM=2
CUSTOM_TEMPLATES="~/.duke/templates"
_DUKE_OLD_PATH="$PATH"


# Add bin/ to the executable path to make them available
# without having to type their path and make all scripts
# in it executables.
PATH="$DUKE_ENV/.duke/bin:$PATH"
export PATH
chmod a+x $DUKE_ENV/.duke/bin/*

# unset PYTHONHOME if set
# this will fail if PYTHONHOME is set to the empty string (which is bad anyway)
# could use `if (set -u; : $PYTHONHOME) ;` in bash
if [ -n "$PYTHONHOME" ] ; then
    _DUKE_OLD_PYTHONHOME="$PYTHONHOME"
    unset PYTHONHOME
fi

# This should detect bash and zsh, which have a hash command that must
# be called to get it to forget past commands.  Without forgetting
# past commands the $PATH changes we made may not be respected
if [ -n "$BASH" -o -n "$ZSH_VERSION" ] ; then
    hash -r
fi

# set a fancy prompt (non-color, unless we know we "want" color)
case "$TERM" in
    xterm-color) color_prompt=yes;;
esac

# Django commands

function __django {
    if [ -e "settings.py" ] ; then
        django $@
    else
        echo "Error: You must be within a django project to use this command."
    fi
}
```

```
alias syncdb="__django syncdb"
alias runserver="__django runserver"
alias shell="__django shell"
alias dbshell="__django dbshell"
alias loaddata="__django loaddata"
alias dumpdata="__django dumpdata"

# Duke commands

function __duke {
    if [ -e "buildout.cfg" ] ; then
        $@
    else
        echo "Error: You must be within a duke project to use this command."
    fi
}

# FIXME: The -vv flag is only a dirty hack to workaround a suspected
# threading issue with python. For some reason, on a fast machine with
# multiple cores, buildout hangs randomly. Increasing buildout's output
# solves this issue. If you are still experiencing this problem, consider
# using -vvv for even more output.
# https://github.com/fschulze/mr.developer/pull/76
alias buildout='__duke buildout -c dev.cfg -vv'
alias dev='__duke develop'

# Python commands

# Make sure that while within the dev environment we only
# use the sandboxed python interpreter.
alias python="$DUKE_ENV/.duke/bin/python -S"
alias ipython="$DUKE_ENV/.duke/bin/ipython --autoindent --no-banner --deep-reload"

# Prompt

function __duke_prompt {
    if [ -z "$DUKE_ENV_DISABLE_PROMPT" ] ; then
        _DUKE_OLD_PS1="$PS1"
        _DUKE_OLD_DIRTRIM="$PROMPT_DIRTRIM"
        . profile

        if [ "x" != x ] ; then
            PS1="$PS1"
        elif [ "`basename \"$DUKE_ENV\"`" = "__" ] ; then
            # special case for Aspen magic directories
            # see http://www.zetadev.com/software/aspen/
            PS1="(%(project_name)s$(in_project)) $PS1"
            PROMPT_DIRTRIM="$PROMPT_DIRTRIM"
        else
            PROMPT_DIRTRIM="$DUKE_DIRTRIM"
            PS1="$DUKE_PS1"
        fi
        export PS1
        export PROMPT_DIRTRIM
    fi
}
__duke_prompt
```

In this file you can creat / modify some variable. For exemple if you want toi create a new alias for the django collectstatic commande, you juste have to add this line:

```
alias collectstatic="__django collectstatic"
```