Dimagi Deploy Tools Documentation

Release 0.5a

Anton de Winter

July 15, 2012

CONTENTS

Relevant Documentation:

Contents:

CHAPTER

ONE

DIMAGI DEPLOY TOOLS MODULES

Contents:

1.1 Django Module

Django deployment module. Creates necessary folders for housing project source code clones, deploys, submodule checkouts

1.1.1 Django Module Settings

```
DJANGO_GIT_REPO_URL = "git://your/repo.git" #django project repo url
DJANGO_STAGING_GIT_BRANCH = "develop"
DJANGO_STAGING_SERVER_NAME = "staging.some_project.com"
DJANGO_PRODUCTION_GIT_BRANCH = "master"
DJANGO_PRODUCTION_SERVER_NAME = "production.some_project.com"
DJANGO_GUNICORN_PORT = '9010'
DJANGO_LOCALSETTINGS_LOCAL_PATH = "localsettings.py" #Path to localsettings on local machine
```

A dictionary you can use if you want to treat localsettings as a template. # This dict get's autopopulated with other useful keys: # * project_root # * www_root - holds the log_dir and code_root # * log_dir # * code_root - where your django code ends up # * project_media - django media folder (in your code_root by default) ((usually used by collectstatic)) # * project_static - django static files folder location, also in code_root by default # * virtualenv_root # * services - path where things like apache.conf and supervisor.conf end up.

DJANGO_LOCALSETTINGS_TEMPLATE_DICT = { } DJANGO_LOCALSETTINGS_REMOTE_DESTINATION = "some_folder/localsettings.py" #RELATIVE TO THE CODE_ROOT ON REMOTE MACHINE DJANGO_LOCALSETTINGS_NO_TEMPLATE = True #Set to false if you want to treat localsettings.py as a template

modules.django.**bootstrap**(*deploy_level='staging'*)
Creates initial folders, clones the git repo. DOES NOT DEPLOY()

modules.django.**clone_repo**()
clone a new copy of the git repository

modules.django.**collectstatic**()
run collectstatic on remote environment. ASSUMES YOU ALREADY HAVE ALL REQUIRED PACKAGES AND VIRTUALENV INSTALLED.

modules.django.**deploy**(*deploy_level='staging'*)
deploy code to remote host by checking out the latest via git

modules.django.**setup_dirs**()
create (if necessary) and make writable uploaded media, log, etc. directories

modules.django.**start**()
Does nothing in this module

modules.django.**stop**()
Does nothing in this module

modules.django.**upload_localsettings**()
Uploads your django settings from your local machine to host

1.2 OS Module

OS Level Operations.

Creates required users

Installs base packages

Misc OS level config

This module set env.user to your settings.SUDO_USER, be sure to have that set to a good value in your settings file!

1.2.1 OS MODULE SPECIFIC SETTINGS

```
OS_PACKAGE_LIST_PATH_REDHAT = "yum_packages.txt" #each package on a new line in this file
OS_PACKAGE_LIST_PATH_UBUNTU = "apt_packages.txt" #each package on a new line in this file
```

modules.os.**bootstrap**(*deploy_level='staging'*)
Installs all packages listed in the OS packages list file(s) specified in settings.

modules.os.**deploy**(*deploy_level='staging'*)
Does the same thing as bootstrap. Installs all packages listed in the OS packages list file(s) specified in settings.

modules.os.**install_packages**()
Install packages, given a list of package names

modules.os.**start**()
Does nothing in this module

modules.os.**stop**()
Does nothing in this module

1.3 Packages Module

Deals with pip requirements. (Initial installing, refreshing, etc)

1.3.1 Packages Module Settings

```
PACKAGES_PIP_REQUIREMENTS_PATH = "pip_requires.txt" #The path to pip_requires file ON THE LOCAL Machi
```

modules.packages.**bootstrap**(*deploy_level='staging'*)
Performs initial install of virtualenv, then installs listed pip packages specified in settings file

modules.packages.**deploy**(*deploy_level='staging'*)
Installs all packages listed in the pip packages list file(s) specified in settings.

modules.packages.**install_packages**()
Install packages, given a list of package names

modules.packages.**setup_virtualenv**()
Initially creates the virtualenv in the correct places (creating directory structures as necessary) on the remote host. If necessary, installs setup_tools, then pip, then virtualenv (packages)

modules.packages.**start**()
Does nothing in this module

modules.packages.**stop**()
Does nothing in this module

modules.packages.**upload_pip_requires**()
Uploads the pip requirements file to the host machine and sets the relevant env variables

1.4 Supervisor Module

Oh, god. Supervisord.

Installs Supervisord on the remote machine. Uses the standard config file produced by the `echo_supervisord_conf` command. This standard config is modified to point to your custom supervisord.conf template (that actually has the commands used for stopping/starting/etc a process).

1.4.1 Supervisor Module Settings

```
SUPERVISOR_TEMPLATE_PATH = "templates/supervisorctl.conf"  #PATH ON LOCAL DISK, RELATIVE TO THIS FIL
SUPERVISOR_INIT_TEMPLATE = "templates/supervisor-init"  #PATH ON LOCAL DISK, Relative to THIS file.
SUPERVISOR_DICT = {
    "gunicorn_port": DJANGO_GUNICORN_PORT,
    "gunicorn_workers": 3
}
```

modules.supervisor.**bootstrap**(*deploy_level='staging'*)
Installs supervisor, creates required directories (if they don't exist). Points supervisord.conf to look in the correct folder for service info

modules.supervisor.**setup_dirs**()
create (if necessary) and make writable uploaded media, log, etc. directories

modules.supervisor.**upload_sup_template**()
Uploads the supervisor template to the server (while populating the template)

1.5 Web Module

Install/Configure Apache for your static (or other web) file serving needs.

1.5.1 Web Module Settings

```
WEB_HTTPD = "apache" #apache2 or nginx
WEB_CONFIG_TEMPLATE_PATH = "templates/my_apache.conf"  #This is the path to the template on the LOCA

#Params that are provided to the template.
#Note: The following env variables are also inserted
#into this dictionary:
#code_root
#log_dir
#project  (the project name)
#virtualenv_root
WEB_PARAM_DICT = {
    "HOST_PORT" : 80 #Primary port for hosting things
}
```

modules.web.**bootstrap**(*deploy_level='staging'*)
Sets up log directories if they don't exist, uploads the apache conf and reloads apache.

modules.web.**deploy**()
Uploads the httpd conf (apache or nginx in future) to the correct place.

modules.web.**run_apache_command**(*command*)
Runs the given command on the apache service

modules.web.**setup_dirs**()
create (if necessary) and make writable uploaded media, log, etc. directories

modules.web.**upload_apache_conf**()
Upload and link Supervisor configuration from the template.

1.6 Utils Module

Utility functions

(e.g. Determining the remote system OS)

You should ALWAYS include this module in your deployment. It contains functions that are used by other modules.

modules.utils.**bootstrap**(*deploy_level='staging'*)
Does nothing in this module.

modules.utils.**develop**(*deploy_level='staging'*)
Does nothing in this module.

modules.utils.**start**(*deploy_level='staging'*)
Does nothing in this module.

modules.utils.**stop**(*deploy_level='staging'*)
Does nothing in this module.

modules.utils.**try_import**(*module_name*)
Import and return *module_name*.

```
>>> try_import("csv")
<module 'csv' from '...'>
```

Unlike the standard try/except approach to optional imports, inspect the stack to avoid catching ImportErrors raised from **within** the module. Only return None if *module_name* itself cannot be imported.

```
>>> try_import("spam.spam.spam") is None
True
```

modules.utils.**what_os**()
Returns a string indicating the Host OS. Currently Only supports 'redhat' and 'ubuntu'

CHAPTER

TWO

THE CORE FABRIC FILE

This Fabric file ('fabfile') goes through your settings file to determine what modules you've included, and allows one to call the same command on each of these modules.

Calling `deploy()`, for example, will result in each respective module's own `deploy()` method being called.

2.1 Operating Methods

Each module has the following operating methods

- `bootstrap` - usually only called once during the life of the server. Performs initial installation actions (such as installing packages, uploading templates, configuring of settings)
- `deploy` - Called regularly. This action updates the code/software/data that the module acts on. For example: a deploy call in the django module would result in the latest code being checked out from your git repository and your settings.py (or localsettings.py) file being updated.
- `stop` - Usually used only by service type modules. For example in the web module, this would *stop* the httpd service
- `start` - Usually used only by service type modules. As above, but would *start* the httpd service.

2.2 The settings file

A module's operating methods (bootstrap, deploy, start, stop) are special in that they each take care to initialize the fabric env object correctly for their module.

The settings.py file for Deploy Tools allows the user to customize how the fabric env object is configured. Each module has some unique fields that can be specified in the settings file.

For example, in the *Django Module* the following settings can be configured:

```
DJANGO_GIT_REPO_URL = "git://your/repo.git" #django project repo url
DJANGO_SUDO_USER = "some_user"
DJANGO_STAGING_GIT_BRANCH = "develop"
DJANGO_STAGING_SERVER_NAME = "staging.some_project.com"
DJANGO_PRODUCTION_GIT_BRANCH = "master"
DJANGO_PRODUCTION_SERVER_NAME = "production.some_project.com"
DJANGO_GUNICORN_PORT = '9010'
```

(Have a look at the settings.py.example file found in the root of this project for an example deployment setup).

In addition to providing values for each module's settings, you also need to specify which modules deploy_tools should use, by modifying the `MODULES` field in settings.py:

```
#Deploy modules we'll be utilizing in this project
MODULES = ["modules.os",
           "modules.web",
           "modules.util",
           "modules.django",
           "modules.packages"]
```

As with django, it's possible to author your own modules and plug them into this list.

2.3 Other Usage Examples

2.3.1 Triggering a specific module

Occasionally you may want to perform an operating method on a single module. Doing so is straightforward:

```
$ fab production deploy:django
```

This causes the Django module to run it's `deploy` operating method for a production-level remote host. Where `deploy` is the operating method that we want to run, and `django` is the module we would like to run it on.

2.3.2 Trigger a module specific action

Sometimes modules have additional extra methods that don't fit with/fall into the bootstrap/deploy/start/stop paradigm.

Deploy_tools makes it possible to run these specific commands while still utilizing the overall configuration settings in *settings.py*:

```
$ fab production run_command:do_foo,module=django,extra1=bar,extra2=bees
```

This will cause the *django* module (specified with the `module=...` arg to execute `do_foo`. Additional arguments, `extra1` and `extra 2` in this case, are passed on to the special command (`do_foo`) within the module (*django*).

2.4 Fabfile Methods

`fabfile.`**`bootstrap`**(*module=None*)
Runs the bootstrap command for each module.

If 'module' argument is specified, this command will only be run for that specific module.

Bootstrapping is usually associated with initial setup or the module being called upon for the first time.

`fabfile.`**`deploy`**(*module=None*)
Runs the deploy command for each module.

If 'module' argument is specified, this command will only be run for that specific module.

'Deployment' is usually associated with a refreshing/updating of content/data

`fabfile.`**`production`**()
use production environment on remote host

`fabfile.`**`run_command`**(*cmd*, *module=None*, **args*, ***kwargs*)
Calls the given command on a single specified module (specified with the 'mod' argument.

fabfile.**start**(*module=None*)
Runs the start command for each module.

If 'module' argument is specified, this command will only be run for that specific module.

'Starting' is usually associated with services that can be started/stopped. (Useful for restarting services manually)

fabfile.**stop**(*module=None*)
Runs the stop command for each module.

If 'module' argument is specified, this command will only be run for that specific module.

'Stopping' is usually associated with services that can be started/stopped. (Useful for restarting services manually)

CHAPTER

THREE

INTRODUCTION

Deploy Tools is a python package that harnesses Fabric to create a modular deployment system. Deploy Tools has a collection of modules that all operate in a similar way allow you to perform bulk operations on a variety of different areas of your infrastructure.

Deploy Tools is intended to be a very simple Chef or Puppet: capable of being quite flexible without the massive learning curve.

The core elements of Deploy Tools are:

- **Modules** - Fabric scripts that implement a certain set of functions (see *Operating Methods*)
- The main **fabfile** - Initializes the actions to be performed on each module
- your **settings.py** file - determines how each module behaves on the remote system.

3.1 Installation

There are two options for getting this project set up:

- Add this repo as a submodule to your existing project.
- Install it on your python path.

Henceforth, this document will assume it is a submodule in some larger project.

3.2 Usage

1. Ensure that your setting.py is configured. Look at settings.py.example for... an example.
2. From the command line simply run:

   ```
   $ fab production bootstrap deploy stop start
   ```

There is no next step. Your server should now live and ready for action!

3.2.1 Details:

The command from step 2 will cause deploy_tools to do bootstrap within each module, refresh all respective code (deploy), stop all services then start all services (a.k.a restart).

- `fab` is the command that invokes deploy tools. See the Fabric Documentation for more information

- `production` indicates that the remote host is a production level machine (as opposed to staging).
- `bootstrap deploy stop start` are all the operating methods we would like to perform.

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

f

m