Argyle Documentation

Release 0.2.1

Mark Lavin

April 15, 2013

CONTENTS

Argyle is a collection of Fabric utilities for Django deployment. This project is still in very early phases.

CHAPTER

ONE

REQUIREMENTS

- Fabric >= 1.1
- Jinja2 >= 2.3

These tasks were written primarily for deployments on Ubuntu Linux. Your mileage may vary with other operating systems and flavors of Linux.

CHAPTER

TWO

DOCUMENTATION

Additional documentation on using argyle is available on Read The Docs.

CHAPTER

THREE

QUESTIONS OR ISSUES?

If you have questions, issues or requests for improvements please let me know on Github.

CHAPTER

FOUR

CONTENTS

4.1 Motivation

We love Fabric for deployments. We also use it for our server provisioning scripts. However, I found myself repeatedly writing the same Fabric tasks over and over again without a good way to reuse them from project to project. This was leading to a lot of wasted time and copy-paste errors.

While there are a number of existing projects out there for Django deployments with Fabric, I found most to be very opinionated about project/server layout. Unfortunately I am also very opinionated about server layout. I also found that they did not scale up well to multiple servers (separate app and db servers, multiple app servers behind a load balancer).

In no particular order my goals in creating this project are:

- Standardize common Fabric tasks for server configuration and deployment
- Provide sane defaults for configurations with the ability to override
- Remain as agnostic as possible for directory layout
- Works well for single server and muli-server deployments

4.1.1 Related Projects

There are some similar projects. If this project does work for you then I would recommend you check out:

1. woven
2. django-fab-deploy
3. django-fab
4. django-fabtastic

I apologize in advance if you felt I've left out your project.

4.2 Base Functions

`argyle.base` contains functions which extend the Fabric api to make the rest of the Argyle framework work.

sshagent_run(*cmd*)

The base ssh library paramiko and hence Fabric does not support SSH Agent forwarding. However, using agent forwarding can be helpful for accessing private repositories (such as hg/git) using ssh without having to generate

and allow the ssh key for the server on the repository. To use this function you must enable SSH forwarding on your system.:

```
ForwardAgent yes
```

This code is based on code from Lincoln Loop. For more info you can see the Fabric issue and paramiko issue for adding support for agent forwarding.

Note: This issue has been fixed in Fabric 1.4.

---

upload_template (*filename*, *destination*, *context=None*, *use_sudo=False*, *backup=True*, *mode=None*)
Fabric comes with support for uploading files using a template. See fabric.contrib.files.upload_template. With this you can use either Python string formatting or Jinja2. Argyle uses this same idea but with a few differences:

1. All templates use Jinja2.
2. The current Fabric environment is always passed into the template context.
3. You can pass list of template names. The first matched template will be used.

Argyle ships with templates which are loaded using jinja2.PackageLoader. You can override these template by defining *env.ARGYLE_TEMPLATE_DIRS* as a tuple of template locations.

4.3 System Functions

`argyle.system` contains functions for managing system packages and users. Most of these system functions require sudo permissions. They should be run with a user with sufficient permission on the remote server.

install_packages (**packages*)
This function installs a list of packages using `apt-get`. You can use this function from both the command-line:

```
fab -H 33.33.33.10 install_packages:nginx
fab -H 33.33.33.10 install_packages:git-core,subversion,mercurial
```

or from another Fabric task

```
def install_python_packages():
    package_list = [
        'python2.6', 'python-all-dev', 'python-setuptools'
    ]
    install_packages(*package_list)
```

install_packages_from_file (*file_name*)
A common use case for installing packages is to install a list of packages from a file. `install_packages_from_file` is a thin wrapper around `install_packages` which takes a filename and installs the listed packages. This file should contain single package name per line. The file is read from the local filesystem not the remote.

update_apt_sources ()
Not much to say here. It just runs `apt-get update` to reload the sources.

upgrade_apt_packages ()
Again this is pretty simple. It runs a `apt-get upgrade` on the remote system.

add_ppa (*name*, *update=True*)
Adds a personal package archive (PPA) and updates the sources. This requires that python-software-properties is installed on the system.

add_ppas_from_file(*file_name*)
A common use case for adding PPAs is to add them from a list of PPAs from a file. `add_ppas_from_file` is a thin wrapper around `add_ppa` which takes a filename and adds the listed PPAs. This file should contain single package name per line. The file is read from the local filesystem not the remote.

add_apt_source(*source*, *key=None*, *update=True*)
Adds a source to /etc/apt/sources.list. There is an optional `key` parameter which is the url to the key. If given it will be fetched and added via `apt-key add`.

add_sources_from_file(*file_name*, *update=True*)
A wrapper around `add_apt_source` which parses a list of source/key pairs from a file. The format is:

```
deb http://example.com/deb lucid main (http://example.com/key)
```

create_user(*name*, *groups=None*, *key_file=None*)
This is used to create a new user on the remote server. You can optionally pass a list of groups and the user will be added to them. In addition you can pass the location of a key file. If given the public key will be added to the newly created user's authorized_keys. All users are created without passwords.

service_command(*name*, *command*)
This is a task for calling service commands (such as init.d). This takes the name of the service and the command to run:

```
fab -H 33.33.33.10 service_command:apache2,reload
```

By default this will call *sudo /etc/init.d/name command*. You can configure this by setting *env.ARGYLE_SERVICE_COMMAND_TEMPLATE*.

```
from fabric.api import env

env.ARGYLE_SERVICE_COMMAND_TEMPLATE = u'invoke-rc.d %(name)s %(command)'
```

`start_service`, `stop_service` and `restart_service` are wrappers around `service_command` to call start, stop and restart commands for a particular service. As such they are also impacted by setting `ARGYLE_SERVICE_COMMAND_TEMPLATE`.

4.4 Nginx Functions

Tasks for configuring sites running under the Nginx webserver.

remove_default_site()
Nginx installs with a default server listening on 80 defined in `/etc/nginx/sites-enabled/default.conf`. This task removes that configuration.

upload_nginx_site_conf(*site_name*, *template_name=None*, *context=None*, *enable=True*)
This task uploads a new configuration to `/etc/nginx/sites-available/<site_name>`. This looks for a template named `nginx/<site_name>.conf` and if not found uploads the default `nginx/site.conf` unless `template_name` is given. By default this site configuration will be enabled `/etc/nginx/sites-enabled/`.

enable_site(*site_name*)
Enables a site in `/etc/nginx/sites-available/<site_name>` and links it to `/etc/nginx/sites-enabled/<site_name>`.

disable_site(*site_name*)
Disables a site in `/etc/nginx/sites-enabled/` by the name. The configuration in `/etc/nginx/sites-available/` is not touched.

4.5 Postgres Functions

Tasks for managing clusters, databases, users and configurations for a Postgres server.

create_db_user(*username*, *password=None*, *flags=None*)
Creates a database user and sets a password if given. You can pass additional creation flags with the `flags` parameter.

db_user_exists(*username*)
Return True if the database user already exists.

excute_query(*query*, *db=None*, *flags=None*, *use_sudo=False*, ***kwargs*)
Execute a SQL query on the remote server. You can specify the DB and additional flags with the `db` and `flags` parameter. If `use_sudo` is True then this will be executed as the `postgres` user. Additional `**kwargs` are passed to `sudo` or `run`.

create_db(*name*, *owner=None*, *encoding=u'UTF-8'*, *template='template1'*)
Creates a new database with a given owner (if given) and encoding. You can specify which database is copied to create the new one with the `template` parameter. Using `'template0'` as the template will allow creating a database with a different encoding from the default, which can't be done from `template1`.

db_exists(*name*)
Return True if the database already exists.

upload_pg_hba_conf(*template_name=None*, *pg_version=None*, *pg_cluster='main'*, *restart=True*)
Uploads a configuration to `/etc/postgresql/<version>/<cluster>/pg_hba.conf` from a template. If not given the Postgres version will be detected on the server. The default template name is `postgres/pg_hba.conf`.

reset_cluster(*pg_cluster='main'*, *pg_version=None*, *encoding=u'UTF-8'*, *locale=u'en_US.UTF-8'*)
Drops and restores a given cluster. This is mainly used for provisioning a new server to ensure the cluster has the desired default encoding.

4.6 Supevisord Functions

Tasks for configuring processes which will be managed by Supervisord.

supervisor_command(*command*)
This is a simple way to execute a `supervisorctl` command on the remote server similar to the `service_command()`.

upload_supervisor_app_conf(*app_name*, *template_name=None*, *context=None*)
Uploads a configuration to `/etc/supervisor/conf.d/<app_name>.conf` for managing a new process. If the `template_name` is not given it will look for templates named `supervisor/<app_name>.conf` and if not found it will use the `supervisor/base.conf` included with this project. The `app_name` parameter is passed in the context and additional context can be provided with the `context` parameter.

The default `supervisor/base.conf` is shown below.

```
[program:{% block name %}{{ app_name }}{% endblock %}]
{% block main %}
command={% block command %}{% endblock %}
{%- if directory -%}directory={{ directory }}{% endif %}
{%- if run_user -%}user={{ run_user }}{% endif %}
{% endblock %}
{% block logging %}
```

```
stdout_logfile={{ log_dir|default('/var/log') }}/%(program_name)s.log
redirect_stderr=true
stderr_logfile={{ log_dir|default('/var/log') }}/%(program_name)s.error.log
{% endblock %}
{% block additional %}
{% endblock %}
```

remove_supervisor_app(*app_name*)
Deletes the `/etc/supervisor/conf.d/<app_name>.conf` configuration.

upload_celery_conf(*command='celeryd'*, *app_name=None*, *template_name=None*, *context=None*)
A wrapper around `upload_supervisor_app_conf()` for managing a Celery process such as `celeryd` or `celerybeat`. The `app_name` defaults to the `command` and both are pass in the context. A default `supervisor/celery.conf` is included which will be used instead of `supervisor/base.conf` if `supervisor/<app_name>.conf` is not found.

The default `supervisor/celery.conf` is shown below.

```
{% extends "supervisor/base.conf" %}

{% block name %}{{ command }}{% endblock %}

{% block command %}{{ bin_dir|default('/usr/local/bin') }}/{{ command }} {{ args }}{% endblock %

{% block additional %}
stopwaitsecs=60
{% endblock %}
```

upload_gunicorn_conf(*command='gunicorn'*, *app_name=None*, *template_name=None*, *context=None*)
A wrapper around `upload_supervisor_app_conf()` for managing a Gunicorn process such as `gunicorn` or `gunicorn_django`. The `app_name` defaults to the `command` and both are pass in the context. A default `supervisor/gunicorn.conf` is included which will be used instead of `supervisor/base.conf` if `supervisor/<app_name>.conf` is not found.

The default `supervisor/gunicorn.conf` is shown below.

```
{% extends "supervisor/base.conf" %}

{% block name %}{{ app_name|default('gunicorn') }}{% endblock %}

{% block command %}{{ bin_dir|default('/usr/local/bin') }}/{{ command|default('gunicorn') }} {{
```

4.7 RabbitMQ Functions

Tasks for managing vhosts, users and configurations for RabbitMQ.

rabbitmq_command(*command*)
This is a simple way to execute a `rabbitmqctl` command on the remote server similar to the `service_command()`.

create_user(*username*, *password*)
Creates a new RabbitMQ user with the given name and password.

create_vhost(*name*)
Creates a new vhost on the remote RabbitMQ server.

set_vhost_permissions (*vhost*, *username*, *permissions='".*" ".*" ".*"'*)
Grants the given permissions to the user on the given vhost. By default all permissions are granted.

upload_rabbitmq_environment_conf (*template_name=None*, *context=None*, *restart=True*)
Uploads the RabbitMQ environment configuration to `/etc/rabbitmq/rabbitmq-env.conf`. This looks for a template named `rabbitmq/rabbitmq-env.conf` if the `template_name` is not given. A default for this template is not given.

upload_rabbitmq_conf (*template_name=None*, *context=None*, *restart=True*)
Uploads the RabbitMQ configuration to `/etc/rabbitmq/rabbitmq.config`. This looks for a template named `rabbitmq/rabbitmq.config` if the `template_name` is not given. A default for this template is not given.

4.8 NPM Functions

Tasks for installing, updating and removing packages installed with NPM, the package manager for Node.JS. New in version 0.2.

npm_command (*command*)
Execute any `npm` command on the remote server. This requires that `npm` is installed.

npm_install (*package*, *flags=None*)
Install a given package from `npm`. The `flags` parameter can be used to pass flags such as `--tag`, `--force`, `--global` or `--link`.

npm_uninstall (*package*)
Uninstall a package.

npm_update (*package*)
Update a package to the latest version.

4.9 Settings

Below are the available settings for configuring Argyle. Each of these settings can be used by setting them in the Fabric environment env dictionary.

4.9.1 ARGYLE_SERVICE_COMMAND_TEMPLATE

This settings configures the behavior of the system `service_command()` for the stop/start/restart tasks. This should be a string with takes the formatting parameters `name` and `command`.

Default: `u'/etc/init.d/%(name)s %(command)s'`

4.9.2 ARGYLE_TEMPLATE_DIRS

By default Argyle loads various configuration templates from the templates directory inside the argyle module. However, if you wish to override, extend or include additional templates you can include additional directories using this setting. Jinja will look for templates in any directories included here first before loading from the default template directory. See the `upload_template()` command for additional information on the usage.

Default: `()`

4.10 Release Notes

4.10.1 v0.2.1 (Released 2012-08-25)

- Fixed group permissions for newly created user's ssh directory

4.10.2 v0.2 (Released 2012-08-03)

Mostly a cleanup release. Added docs which were previously missing and a full test suite.

Features

- Tasks for managing Node packages with npm
- Added unittest suite
- Various bug fixes/cleanup

Bug Fixes

- Fixed bug with creating a user with no groups
- Fixed bug when using upload template with a list of filenames
- Fixed incorrect error case when detecting Postgres version

Backwards Incompatible Changes

- The default contexts for the supervisor templates have been updated slightly but were never documented

4.10.3 v0.1 (Released 2012-02-17)

Initial public release

CHAPTER

FIVE

INDICES AND TABLES

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