Garment Documentation

Release 0.1

Evan Borgstrom

March 25, 2014

A collection of fabric tasks that roll up into a single deploy function. The whole process is coordinated through a single deployment configuration file named `deploy.conf`

Garment was written to be flexible enough to deploy any network based application to any number of hosts, with any number of roles, yet still provide a convention for the deployment process and take care of all of the routine tasks that occur during deployment (creating archives, maintaining releases, etc).

Currently Garment only supports applications that use GIT as their SCM, but it could easily be extended to support others.

Contents:

CHAPTER 1

Using Garment

Garment is a library of tasks to be used from within a **'Fabric file'**_.

1.1 Installing Garment

To install garment we recommend using pip to install it into a virtualenv:

```
pip install garment
```

Once installed you will be able to use garment in your `fabfile.py`

1.2 Creating the Fabric file

At the root of your project create a file named `fabfile.py` and add the following line to it:

```
from garment import *
```

You can now add any other tasks you want into your `fabfile.py` file, using all the power of Fabric.

1.3 Create your configuration

See the *Deployment Strategy* documentation to understand how garment approaches the deployment process and then see the *Deployment Configuration* documentation for information on how to build a configuration.

1.4 Deploying your application

Now that you have installed, setup and configured garment you can use Fabric to run a deployment. Let's say you wanted to deploy to staging, you would run:

```
deploy staging
```

The above uses the `deploy` script we ship with garment. You can also invoke Fabric directly:

```
fab deploy:staging
```

1.5 Runtime commands

Garment provides a number of functions for managing your project's deployment, to see all of them ask Fabric for a list:

```
fab -l
```

1.6 Including or Excluding steps

Sometimes you may want to only run some of the steps in your deployment. For example, perhaps you're iterating over a new piece of functionality and want to share your designs with other team members but these changes don't require anything other than deploying the files (ie. you can skip the DB migrations, environment update, requirement installation, etc). To accomplish this you can use either the `include` or `exclude` parameters to the `deploy` function. They are mutually exclusive and you can only use one of them.

Both `include` and `exclude` take a comma separated list of step ids. As Fabric also uses commas you must quote & escape them:

```
fab deploy:staging,exclude='virtualenv\,install\,django_db'
```

CHAPTER 2

Deployment Strategy

This document serves to outline the approach to deployment that Garment takes, and the associated conventions it defines based on this approach. See the *Deployment Configuration* documentation for how you configure your deployments.

2.1 Environments

Environments are containers for your various deployment scenarios, you might have just a `production` environment, or you may have more complex setups that involve multiple environments (`qa`, `staging`, etc).

Their names are free form and you can call them anything you like that suits your deployment needs.

2.2 Releases

Releases refer to an archive of an application that is identified by a time stamp (ISO8601), the GIT reference and the user specification that deployed the release:

```
20131228T202753-4634f44-evan@pungi.local
```

Releases are maintained in a designated folder on each of the Hosts and a configurable number of historical releases are kept around for easy rollbacks should a deployment go awry.

Each Host keeps a full copy of the repository. On each deployment the local copy of the repository is updated using `git pull`.

To create a release the `git archive` command is used. We also support adding files from the submodules in a project.

2.3 Hosts

Hosts are the targets that you wish to deploy your application to. They can be assigned roles that can be used to only run certain Steps on certain Hosts when running the Stages.

2.4 Roles

Roles are arbitrary items that you create that can be used to control which of the different Stages you define run on which Hosts.

There is a special role named `all` that is used for targeting all hosts by internal Garment operations (like sending the release). You do not need to assign any roles to your hosts if you don't need them to accomplish your deployments.

2.5 Stages

Stages make up the workflow of the deployment process and are named as follows:

- **before**
- **after**
- **rollback**
- **cleanup**

Steps within the Stages are defined as an ordered list and run in the order that they are defined.

Inside each Step is some configuration about the execution environment (working directory, command prefixes, environment variables, etc) and then an ordered list of shell commands that will be executed.

All commands in a Stage must succeed for the Stage to be considered a success, if a Step fails then deployment stops.

Currently no clean up is done on failed deployments to ensure that you have the opportunity to investigate the failure manually while implementing your config. Your deployment shouldn't ever fail after that point so if it does leaving the machine in the broken state allows you to fix it properly.

2.5.1 Deployment preparation

When a release is being deployed Garment will first prepare the release by generating a name, as described above, and then making an archive of the repository into the releases directory under this new name.

This is done entirely automatically and can only be influenced through the *Base configuration*.

2.5.2 Before stage

The **before** stage is run once the release is prepared and it allows you to prepare the application for becoming the running release. It also allows you to do any backup related tasks that can be used during the **rollback** stage.

If your application does not require any special preparation you do not need to specify any tasks in the before stage and it can be omitted.

Examples of things that are done in the **before** stage:

- Installing dependencies (virtualenv, rvm, composer, etc)
- Run database migrations
- Managing/preparing static content
- Backing up the current environment
- Backing up the database

2.5.3 Updating current symlink

Now that the application is prepared the symlink that is used to represent the active deployment version is updated to point to our new release.

This is done entirely automatically and can only be influenced through the *Base configuration*.

2.5.4 After stage

Like the **before** stage the **after** stage allows you to further prepare the application for becoming the running release. You can also preform any backup tasks you'd like as well.

If your application does not require any special preparation you do not need to specify any tasks in the before stage and it can be omitted.

Examples of things that are done in the **after** stage:

- Restarting application servers

2.6 Operations

Operations are the different workflows that Garment exposes. The following describes each of the operations and specifies which stages are run and in which order.

2.6.1 Deploy

When you ask Garment to deploy it does the following:

1. Prepares the release
2. Runs the **before** stage
3. Makes the new release the current release
4. Runs the **after** stage

2.6.2 Rollback

When you ask Garment to rollback it does the following:

1. Runs the **rollback** stage
2. Makes the rollback release the current release
3. Runs the **after** stage

CHAPTER 3

Deployment Configuration

The whole deploy process is configured through a file named `deploy.conf` in your project root. This is a YAML file that describes your environment, hosts that are involved, and how tasks are run on these hosts.

3.1 Environments

An environment is defined through the key at the root level of your `deploy.conf`:

```
#
# My project's deployment configuration
#

production:
  # configure our production deployment

qa:
  # configure our qa deployment
```

When you invoke a deployment with `fab` you will pass the name of the environment you want to deploy as the argument to the `deploy` task:

```
fab deploy:production
```

Tip: Garment installs a script named `deploy` that is a shortcut for running `fab deploy:...`. If you were to run `deploy production` it would be the same as running `fab deploy:production`

3.2 Base configuration

There are some release related configuration values that you must define at the base of your environment.

- `repo_url` - This is the git URL that will be used as the `origin` for the bare repository created on each host. This is required.
- `git_ref` - This is the git reference that will be used for the deployment in this environment. A branch or tag, for example: `master`. This is required.
- `deploy_dir` - This is the base directory that should be used for all of the deployment directories. Example: `/home/user/deploy`. This is required.

- `source_dir` - This holds the checked out git repository. This is optional and defaults to `{deploy_dir}/source`.
- `releases_dir` - This is the file system path that is a directory and will hold all of the releases. This is optional and defaults to `{deploy_dir}/releases/`.
- `current_symlink` - This is the file system path that garment will use as the "current" path. This path will always point to the latest release in the releases directory. This is optional and defaults to `{deploy_dir}/current`.
- `keep_releases` - This should be set to the number of historical releases you want to keep on each host. This is optional and defaults to `10`.
- `forward_agent` - This is `True` or `False` (default) and determines if Fabric should forward your local agent connection when deploying.

3.3 Hosts

Hosts are defined inside an item named `hosts` at the root of the environment configuration. The Host definition is the SSH connection string used for the host. You can also define their roles if you will be using roles:

```
# My deployment configuration
production:
  ...
  hosts:
    deploy@hostname.domain.tld:
    deploy@db-hostname.domain.tld:
      roles: ['db']
```

The roles are completely arbitrary and up to you to define and name in what ever way suits your project.

3.4 Variables

You can define variables to be used throughout the configuration. They are defined in a list and can be built up on top of each other, for example if you define a variable named `foo` you can then reference that variable when you define the next variable, `bar`.

Variables are defined under the `variables` entry inside an environment:

```
# My deployment configuration
production:
  ...
  variables:
    - foo: 'Hello'
    - bar: '{foo} World'
    - baz: '{bar} from Garment'
```

`baz` would be equal to "Hello World from Garment"

As illustrated above you reference variables using the standard Python named string formatting syntax. Each variable is passed all previous variables as keyword args using the Python string .format() method.

You can access any variables that have been previously defined as well as any of the top level configuration items in the environment. For example you could set a variable's value using the `current_symlink` config item:

```
variables:
  - appdir: '{current_symlink}/myapp'
```

3.5 Stages

Stages are named **before**, **after** and **rollback**. They are all optional. They use the YAML list of dictionaries syntax to define the Steps within them. Each step starts with defining a list and then continues the standard dictionary syntax:

```
# My deployment configuration
production:
  ...
  stages:
    before:
      - id: my_step
        roles: ['app']
        cd: ~/current/
        commands:
          - prep_environment

      - id: second_step
        roles: ['app']
        commands:
          - prep_database
          - migrate_database

    after:
      - id: after_step
        roles: ['app']
        commands:
          - restart_app_server
```

Each Step is made up of an `id`, a list of `roles` and a list of `commands`. Stages can also contain the following extra configuration items:

- **cd** - Change to the specified directory prior to executing the `commands`
- **prefix** - Prefix a command onto all the other commands, for example you could use this to use `sudo` to activate a Python virtualenv.
- **shell_env** - A YAML dictionary of items to inject into the shell as variables.

Example with all extra items:

```
# database migration & static assets
- id: django
  roles: ['app']
  cd: '{pythonpath}'
  prefix: '{activate}'
  shell_env:
    PYTHONPATH: '{pythonpath}'
    DJANGO_SETTINGS_MODULE: '{settings}'
  commands:
    - django-admin.py syncdb
    - django-admin.py migrate
    - django-admin.py collectstatic --noinput
```

3.6 Extending items

Often times when building deployment configurations you will find yourself repeating the same variables & stages. Garment configuration allows for one environment to extend another through the use of the `extends` keyword so that

you can leverage reusability to keep your configuration concise and error free.

Complete Django Example:

```
#
# Deployment configuration
#

staging:
  forward_agent: True
  repo_url: git@myhost.tld:myrepo.git
  git_ref: develop
  deploy_dir: /home/staging/deploy

  hosts:
    staging@myhost.tld:
      roles: ['app']

  variables:
    - home: '/home/staging'
    - virtualenv: '{home}/virtualenv'
    - activate: 'source {virtualenv}/bin/activate'
    - pythonpath: '{current_symlink}/myapp'
    - settings: 'myapp.settings.staging'
    - logdir: '{home}/logs/application/'

  stages:
    before:
      - id: dirs
        roles: ['app']
        commands:
          - 'mkdir -p {logdir}'

      - id: virtualenv
        roles: ['app']
        commands:
          - '[ ! -d {virtualenv} ] && virtualenv {virtualenv} || echo "virtualenv exists"'
          - 'rm -f virtualenv/lib/*/no-global-site-packages.txt'

      - id: install
        roles: ['app']
        prefix: '{activate}'
        shell_env:
          PYTHONPATH: '{pythonpath}'
          DJANGO_SETTINGS_MODULE: '{settings}'
        commands:
          - 'pip install -r {release_dir}/requirements/production.txt'

      - id: django
        roles: ['app']
        prefix: '{activate}'
        shell_env:
          PYTHONPATH: '{pythonpath}'
          DJANGO_SETTINGS_MODULE: '{settings}'
        commands:
          - django-admin.py syncdb
          - django-admin.py migrate
          - django-admin.py collectstatic --noinput
```

```
    after:
      - id: restart
        roles: ['app']
        prefix: '{activate}'
        commands:
          - 'supervisorctl restart gunicorn'


preview:
  extends: staging

  deploy_dir: /home/preview/deploy

  hosts:
    preview@my-host.tld:
      roles: ['app']

  variables:
    - home: '/home/preview'
    - settings: 'myapp.settings.preview'

  stages:
    after:
      - id: contrived
        roles: ['app']
        commands:
          - 'echo "Just a silly example"'

      - id: restart
        roles: ['app']
        prefix: '{activate}'
        commands:
          - 'supervisorctl restart gunicorn'
          - 'echo "PREVIEW HAS BEEN RESTARTED"'
```

Here you can see that the `preview` environment has specified `extends: staging` as an option. When the configuration loader sees this it will merge the configuration from the `preview` environment together with the `staging` environment.

The `hosts` are not copied during the merge so you **always** need to specify hosts in an extended environment.

The `variables` and `stages` are fully merged in the same order. That means if you have a variable named `home` in the base environment and its the 2nd variable defined when the `home` variable from the new extended environment is merged in it will also been the 2nd variable defined when the variable resolution is applied to the configuration. Anything defined in an extended environment that is not defined in a base environment will be appended. In the above example it means that even though the `contrived` step was defined before the `restart` step when the config is fully resolved the `contrived` step will actually run after the `restart` step because the `restart` step overrode the restart step from the `staging` environment.

The above example shows two extra steps being added to the `after` stage, but in reality they are not needed and have been added purely to explain how the config loaded merges items. If you remove the two extra steps you can see that the configuration for preview becomes quite concise, less than 10 lines.

CHAPTER 4

Vagrant wrapper

Garment provides a simple interface to utilize Fabric on your Vagrant instances. It does this by providing you an object named `vagrant` that you can import into your `fabfile.py` that exposes `run` & `sudo` methods that work on the Vagrant instance of the current working directory.

4.1 Usage

You can import the `vagrant` object from the `garment.vagrant` package and then use it's methods within your tasks.

Here's a basic example:

```
from fabric.api import task

from garment.vagrant import vagrant

@task
def uptime():
    vagrant.run('uptime')
```

This would produce output similar to the following:

```
[1279]  fab uptime
[localhost] local: vagrant ssh-config > /var/folders/_v/195_x2f97vv_y4yrgvtf_2t00000gp/T/tmpAbg0_7
[default] Executing task 'vagrant_run'
[default] run: uptime
[default] out:  17:23:43 up  3:03,  2 users,  load average: 0.19, 0.18, 0.21
[default] out:

Done.
Disconnecting from vagrant@127.0.0.1:2222... done.
```

4.2 How it works

THe interface simply uses Fabric's `local` command to first dump the ssh config via the `vagrant` command, then modifies Fabric's environment and finally it uses the `execute` method to run your command on the `default` host, which is what Vagrant names the box by default.

When it's done it restores Fabric's environment and removes the temporary file used to hold the ssh config.

4.3 TODO / Improvements

- Auto detect hostname from the `ssh-config` output