vagrant-django Documentation

Release 0.6.2

Alex Church

Nov 28, 2018

Contents

The building blocks for a Vagrant environment for Django development.

CHAPTER 1

Introduction

Included are shell provisioning scripts and sample configuration files allowing the construction of a Vagrant guest machine designed to support either full Django projects or the development of single Django apps for packaging and distribution.

The scripts are also designed to be run independently of Vagrant in order to provision production environments that match those used in development. See *Usage in Production*.

While various aspects of the provisioned environment are configurable, some are not. Therefore, it may not be suitable for all projects. In particular, the locations of various important directories (such as the Vagrant synced folder) and the system users used for various tasks are fixed.

Be sure to check out the *Features* and *Limitations and Restrictions* documentation.

1.1 How to use

1. Copy the `provision/` directory into your project.
2. Copy the included `Vagrantfile` or add `provision/scripts/bootstrap.sh` as a shell provisioner in your existing `Vagrantfile`, specifying the project name. The included `Vagrantfile` is pretty basic, but it can be used as a foundation. See *Vagrantfile* for details.
3. Modify the example `provision/env.sh` file. See *env.sh* for details.
4. Add/modify any further configuration files to `provision/conf/`. See *Configuring the environment* for details on what further customisation options are available.
5. Add any project-specific provisioning steps to a `provision/project.sh` file. See *Project-specific Provisioning* for details.
6. Add `provision/env.sh` (and any other necessary config files) to your `.gitignore` file, or equivalent. Environment-specific configurations should not be committed to source control.
7. `vagrant up`

In production environments, a few additional steps are necessary. See *Usage in Production* for details.

Note: When running a Windows host and using VirtualBox shared folders, `vagrant up` must be run with Administrator privileges to allow the creation of symlinks in the synced folder. See *Windows Hosts* for details.

1.2 Re-provisioning

The provisioning scripts can be re-run on existing environments to update them with any changes.

- Any newly-added provisioning steps will be run.
- Dependency packages will be updated if the specified versions have changed (e.g. in `requirements.txt` or `package.json`).
- Config files in `provision/conf` will be re-copied.
- Existing software will NOT be updated (the scripts do not run `apt-get upgrade`). This step will need to be run manually if required. **Note: This is particularly important when provisioning a new environment.**
- `env.py` will NOT be overwritten if it exists. This allows it to be modified as necessary (either changing existing settings or adding new ones) without those changes getting replaced. As such, if the file *needs* rewriting (e.g. provisioning has been updated to change what it writes to `env.py`), it should be deleted first.

1.3 Upgrading

When upgrading to a new version of `vagrant-django`, do not replace the entire `provision/` directory - that will wipe out any customised configuration files, templates, etc. The `provision/scripts/` subdirectory is not designed to be customised, so it can safely be replaced as a whole. Modifications/additions to files in other subdirectories will be specified in the release notes, and can be updated individually.

CHAPTER 2

Features

The following features are available in the environment constructed by the included provisioning scripts.

Several features may only apply in a production or development environment. This is differentiated based on the *DEBUG* setting in the `env.sh` file.

2.1 Well-defined project structure

The provisioning process creates a well-defined directory structure for all project-related files.

The root of this structure is `/opt/app/`.

The most important subdirectory is `/opt/app/src/`. This is the project root directory, and the target of the Vagrant synced folder. Subsequently, `/opt/app/src/provision/` contains all the provisioning scripts.

Some of the other useful directories in this structure are:

- `/opt/app/conf/`: For storage of configuration files such as `nginx.conf` and gunicorn's `conf.py`. Such files are copied here instead of being referenced directly from within `provision/conf/` so they may be modified without affecting the committed source files.
- `/opt/app/logs/`: For storage of log files output by supervisor, etc.
- `/opt/app/media/`: Target for Django's `MEDIA_ROOT`.
- `/opt/app/static/`: Target for Django's `STATIC_ROOT` (in production environments).

The final directory is `/opt/app/ln/`. This directory is primarily used to simplify the process of configuring the server. It acts as a container for shortcut symlinks to various project-specific files and directories (i.e. those that contain the project name). It is designed to allow using known paths in config files, without forcing customisation in projects that would not otherwise need it. Using the shortcuts in the `ln` directory, default config files that work out-of-the-box can be provided (such as `provision/conf/supervisor/production_programs/gunicorn.conf`). Otherwise, such files would require the modification of a series of paths to include the project name.

2.2 Locked down user access

SSH access is locked down to the custom `webmaster` user created during provisioning. SSH is available via public key authentication only - no passwords. In a development environment, only the `webmaster` and `vagrant` users are allowed SSH access. In a production environment, only `webmaster` is granted access. No other users, including `root`, can SSH into the machine.

The public key to use for the `webmaster` user must be provided via the *PUBLIC_KEY* variable in the `env.sh` file. This will be installed into `/home/webmaster/.ssh/authorized_keys`.

The `webmaster` user is given sudo privileges. In development environments, for convenience, it does not require a password. In production environments, it does. A password is not configured as part of the provisioning process, one needs to be manually created afterwards. When logged in as the `webmaster` user, simply run the `passwd` command to set a password.

Most provisioned services, such as nginx and gunicorn, are designed to run under the default `www-data` user.

Warning: Using the provisioning scripts in a production environment with *DEBUG* set to `1` will leave the `webmaster` user with open `sudo` access, unprotected by a password prompt. This is a Bad Idea.

2.3 Time zone

The time zone can be set using the *TIME_ZONE* setting in the `env.sh` file.

2.4 Firewall

In production environments, and if a *firewall rules configuration file* is provided, a firewall is provisioned using UncomplicatedFirewall.

2.5 Git

Git is installed.

Note: A `.gitconfig` file can be placed in `provision/conf/` to enable configuration of the git environment for the `webmaster` user.

2.6 Ag (silver searcher)

The "silver searcher" commandline utility, `ag`, is installed in the guest machine. `ag` provides fast code search that is better than ack.

Note: An `.agignore` file can be placed in `provision/conf/` to add some additional automatic "ignores" for the command.

2.7 Image libraries

Various system-level image libraries used by Pillow are installed in the guest machine.

To install Pillow itself, it should be included in `requirements.txt` along with other Python dependencies (see *Python dependency installation* below). But considering many of its features require external libraries, and the high likelihood that a Django project will require Pillow, those libraries are installed in readiness.

The exact packages installed are taken from the Pillow "depends" script for Ubuntu, though not all are used.

Installed packages:

- libtiff5-dev
- libjpeg8-dev
- zlib1g-dev
- libfreetype6-dev
- liblcms2-dev

2.8 PostgreSQL

PostgreSQL is installed.

In addition, a database user is created with a username equal to the *project name* and a password equal to *DB_PASS*. A database is also created, also with a name equal to the *project name*, with the aforementioned user as the owner.

The Postgres installation is configured to listen on the default port (5432).

2.9 Nginx

In production environments, nginx is installed.

The `nginx.conf` file used can be modified. Also, the site config can - and must - be modified. See *Configuring nginx* for details.

Nginx is controlled and monitored by *Supervisor*. A default supervisor program is provided, but can be modified. See *Supervisor programs* for details.

2.10 Gunicorn

In production environments, gunicorn is installed.

The `conf.py` file used can be modified. See *Configuring gunicorn* for details.

Gunicorn is controlled and monitored by *Supervisor*. A default supervisor program is provided, but can be modified. See *Supervisor programs* for details.

2.11 Supervisor

Supervisor is installed.

The `supervisord.conf` file used can be modified. See *Configuring supervisor* for details.

Default programs for *Nginx* and *Gunicorn* are provided, but any number of additional programs can be added. See *Supervisor programs* for details.

2.12 Virtualenv

A virtualenv is created using pyenv and its pyenv-virtualenv plugin.

The version of Python used to build the virtualenv can be specified in *versions.sh* using the *BASE_PYTHON_VERSION* setting. If not specified, the system version will be used.

The virtualenv is automatically activated when the `webmaster` user logs in via SSH.

2.12.1 Python dependency installation

If a `requirements.txt` file is found in the project root directory (`/opt/app/src/`), the included requirements will be installed into the virtualenv (via `pip -r requirements.txt`).

In development environments, a `dev_requirements.txt` file can also be specified to install additional development-specific dependencies, e.g. debugging tools, documentation building packages, etc. This keeps these kinds of packages out of the project's primary `requirements.txt`.

2.13 Node.js/npm

If a `package.json` file is found in the project root directory (`/opt/app/src/`), node.js and npm are installed. The version of node.js installed is dictated by the *NODE_VERSION* setting in `versions.sh`.

A `node_modules` directory is created at `/opt/app/node_modules/` and a symlink to this directory is created in the project root directory (`/opt/app/src/node_modules`). Keeping the `node_modules` directory out of the synced folder helps avoid potential issues with Windows host machines - path names generated by installing certain npm packages can exceed the maximum Windows allows.

Note: In order to create the `node_modules` symlink when running a Windows host and using VirtualBox shared folders, `vagrant up` must be run with Administrator privileges to allow the creation of symlinks in the synced folder. See *Windows Hosts* for details.

Note: If a `package.json` file is added to the project at a later date, provisioning can be safely re-run to install node/npm (using the `vagrant provision` command).

2.13.1 Node.js dependency installation

`npm install` will be run in the project root directory.

In production environments, `npm install --production` will be used, limiting the installed dependencies to those listed in the `dependencies` section of `package.json`. Otherwise, dependencies listed in `dependencies` and `devDependencies` will be installed. See the documentation on npm install.

2.14 Multiple Python versions and tox support

The base Python version (used to create the virtualenv under which all relevant Python processes for the project will be run) and additional versions of Python can be specified in `versions.sh`, via the *BASE_PYTHON_VERSION* and *PYTHON_VERSIONS*, respectively.

All specified Python versions are installed with pyenv. The pyenv global command is used to provide system-wide access to all installed versions, with the following priority:

- *PYTHON_VERSIONS*, in the order they are defined
- The specified *BASE_PYTHON_VERSION*, if there is one and if it doesn't already appear in `PYTHON_VERSIONS`
- The system Python

For example:

```
# The following settings...
BASE_PYTHON_VERSION='3.6.4'
PYTHON_VERSIONS=('2.7.14' '3.5.4')

# ... yield the command:
pyenv global 2.7.14 3.5.4 3.6.4 system
```

If you want the specified base version to appear somewhere specific among the list of versions, include it explicitly in `PYTHON_VERSIONS`:

```
# The following settings...
BASE_PYTHON_VERSION='3.6.4'
PYTHON_VERSIONS=('3.6.4' '2.7.14' '3.5.4')

# ... yield the command:
pyenv global 3.6.4 2.7.14 3.5.4 system
```

This support is most useful when using tox to test your code under multiple versions of Python.

2.15 env.py

Several of the *env.sh* settings are designed to eliminate hardcoding environment-specific and/or sensitive settings in Django's `settings.py` file. Things like the database password, the `SECRET_KEY` and the `DEBUG` flag should be configured per environment and not be committed to source control.

12factor recommends these types of settings be loaded into environment variables, with these variables subsequently used in `settings.py`. But environment variables can be a kind of invisible magic, and it is not easy to simply view the entire set of environment variables that exist for a given project's use. To make this possible, an `env.py` file is written by the provisioning scripts.

This ordinary Python file simply defines a dictionary called `environ`, containing settings defined as key/value pairs. It can then be imported by `settings.py` and used in a manner very similar to using environment variables.

```
# Using env.py
from . import env
env.environ.get('DEBUG')

# Using environment variables
import os
os.environ.get('DEBUG')
```

The `environ` dictionary is used rather than simply providing a set of module-level constants primarily to allow simple definition of default values:

```
env.environ.get('DEBUG', False)
```

The default `environ` dictionary will contain the following key/values:

- DEBUG: Will be True if *DEBUG* is set to `1`, False if it is set to `0`.
- DB_USER: Set to the value of the *project name*.
- DB_PASSWORD: Set to the value of *DB_PASS*. Automatically generated by default.
- TIME_ZONE: Set to the value of *TIME_ZONE*.
- SECRET_KEY: Set to the value of *SECRET_KEY*. Automatically generated by default.

If a specific project has additional sensitive or environment-specific settings that are better not committed to source control, it is possible to modify the way `env.py` is written such that it can contain those settings as well, or at least placeholders for them. See *Customising env.py* for more details.

Note: The `env.py` file should not be committed to source control. Doing so would defeat the purpose!

2.16 Project-specific provisioning

In addition to the above generic provisioning, any special steps required by individual projects can be included using the `provision/project.sh` file. If found, this shell script file will be executed during the provisioning process. This file can be used to install additional system libraries, create/edit configuration files, etc.

For more information, see the *Project-specific Provisioning* documentation.

2.17 Shortcut commands

The following shell commands are made available on the system path for convenience:

- `pull+`: For git users. A helper script for pulling in the latest changes from origin/master and performing several post-pull updates. It must be run from the project root directory (`/opt/app/src/`). Specifically, and in order of operation, the script:
  - Runs `git pull origin master` as the `www-data` user
  - Runs `python manage.py collectstatic` (production environments only), also as the `www-data` user
  - Checks for differences in requirements.txt[#]
  - Asks to install from requirements, if any differences were found

- Runs `pip install -r requirements.txt` if installing was requested
- Checks for unapplied migrations (using Django's `showmigrations` management command)
- Asks to apply the migrations, if any were found
- Runs `python manage.py migrate` if applying was requested
- Runs `python manage.py remove_stale_contenttypes` if using Django 1.11+
- Restarts gunicorn (production environments only)

#: When first run, `pull+` detects differences between the `requirements.txt` file as it existed *before* the pull vs *after* the pull. Even if no differences are found, the installed packages may still be out of date if an updated `requirements.txt` was pulled in prior to running the command. After the first run, it stores a temporary copy of `requirements.txt` any time updates are chosen to be installed. It can then compare the newly-pulled file to this temporary copy, enabling it to detect changes from any pulls that took place in the meantime as well. However, if the requirements are updated manually (outside of using this command), it will detect differences in the files even if the installed packages are up to date.

CHAPTER 3

Limitations and Restrictions

While various aspects of the provisioned environment are configurable, some are not. The following are some of the limitations and restrictions the provisioning scripts are subject to.

3.1 Target OS

The provisioning scripts have only been tested on Ubuntu Linux, specifically 16.04 Xenial Xerus.

While some versions have been tested in Ubuntu 16.04 production environments (outside of Vagrant), recent and in-development versions will probably only have been tested via Vagrant, using the "bento/ubuntu-16.04" box.

3.2 apt-get upgrade

The provisioning scripts do NOT run `apt-get upgrade`. They avoid this specifically so that re-provisioning does not trigger updates to installed packages beyond the scope of provisioning (i.e. system packages that provisioning didn't install in the first place).

The scripts *do* run `apt-get update`, so the packages they do install are the latest repository versions at the time of installation.

It is incumbent on the user to run `apt-get upgrade`, especially for a newly provisioned system. **This is particularly important in production environments**.

3.3 Python

Python (either 2 or 3) is required to be installed on the *unprovisioned* system. This is due to its use generating random strings, which is potentially one of the first things the provisioning scripts do (if `env.sh` settings such as `DB_PASS` and `SECRET_KEY` are not given).

3.4 Directory structure

The provisioning process creates the `/opt/app/` directory to store most things related to the project.

The provisioning scripts and various configuration files expect this directory, and its subdirectories, to exist and contain the relevant files.

See *Well-defined project structure* for a description of this structure.

3.5 Users

The provisioning process creates a `webmaster` user. This is the only user with SSH access and is granted `sudo` privileges. See *the feature documentation* for more details.

The `webmaster` user is placed in the `www-data` group.

File ownership of almost everything under `/opt/app/` is `www-data:www-data`. Various services, such as nginx and gunicorn, are configured to run under `www-data`.

3.6 Windows Hosts

If using Virtualbox as a provider for Vagrant under Windows, the synced folders will be handled by Virtualbox's "shared folders" feature by default. When creating symlinks in this mode, which the provisioning scripts do when installing Node.js (see *Node.js/npm*), requires Administrator privileges. Specifically, `vagrant up` needs to be run from a command prompt with Administrator privileges.

This can be done by right-clicking the command prompt shortcut and choosing "Run as administrator", then running `vagrant up` from that command prompt.

Alternatively, the Windows `.cmd` script found here can be used to automatically launch a command prompt with Administrator privileges requested from UAC, opened to a given development directory, ready for `vagrant` commands to be issued. See the script's comments for details on usage.

CHAPTER 4

Configuring the environment

The environment of the Vagrant guest machine (or production server) provisioned by these scripts is designed to provide everything necessary for developing and hosting Django-based projects with minimal configuration. However, several configuration files are recognised and utilised by the scripts.

4.1 Vagrantfile

Location: project root (`/opt/app/src/`)

The use and feature set of the `Vagrantfile` are beyond the scope of this documentation. For more information on the file itself, see the Vagrant documentation.

An example `Vagrantfile` is included, but an entirely custom one can be used. In either case, the following features are of note:

- **The provisioner** The `provision/scripts/bootstrap.sh` shell provisioner needs to be included and configured.

  ```
  config.vm.provision "shell" do |s|
      s.path = "provision/scripts/bootstrap.sh"
      s.args = ["<project_name>"]
  end
  ```

  `<project_name>` should be replaced with a suitable name for the project. It dictates multiple features of the environment. *See below* for details.

- **Synced folder** The type of synced folder used is not important, however the following aspects are:

  - The location of the folder on the guest machine must be `/opt/app/src/`. Various provisioning scripts and included config files expect the project's source to be found there.
  - The owner and group should be `www-data`. Various other files and directories will have their owners/groups set to `www-data`, and certain included config files (such as the supervisor programs for nginx and gunicorn) run programs under `www-data`.

- **The box** While not necessarily a requirement, the most recent versions of the provisioning scripts have only been tested on "bento/ubuntu-16.04".

4.1.1 Project name

The name of the project is used by the provisioning scripts for the following:

- The name of the default PostgreSQL database created.
- The name of the default PostgreSQL database user created.
- The location of the `env.py` Python settings file: `<project root>/<project name>/env.py`. It is assumed this is the directory containing `settings.py`.
- The name of the nginx site config file (placed in `/etc/nginx/sites-available/` and linked to in `/etc/nginx/sites-enabled/`).

This means that the name given must be valid for each of those uses. E.g. names incorporating hyphens should use underscores instead (use `project_name` instead of `project-name`).

4.2 env.sh

Location: `provision/env.sh`

The primary configuration file is `env.sh`. It is simply a shell script that gets executed by the provisioning scripts to load the variables it contains. Each of the variables is discussed below. An example file is included.

When provisioning is first run, it will most likely modify this file. Some of the settings below generate defaults if no value is provided, and that default will get written back to the file so the same value will be used in the case of re-provisioning. Some additional settings may also be written to this file - these are convenience settings used internally by the provisioning process and should not be modified.

Note: The settings contained in `env.sh` are sensitive and/or environment-specific, and thus should not be committed to source control.

Note: Several of these settings affect `env.py`. See *env.py* for the virtues of using these values over values hardcoded in `settings.py`.

4.2.1 DEBUG

Required

This flag controls whether or not to provision a development or production environment. A value of `1` indicates a development environment, a value of `0` indicates a production environment.

This flag affects numerous aspects of the environment. For a breakdown of the features only available in production environments (when the flag is `0`), see *Usage in Production*.

This value is also written to `env.py` so it may be imported into `settings.py` and used for Django's `DEBUG` setting. A value of `1` is written as `True`, a value of `0` is written as `False`.

4.2.2 PUBLIC_KEY

Required

This public key will be installed into `/home/webmaster/.ssh/authorized_keys` so it may be used to SSH into the provisioned environment as the `webmaster` user.

4.2.3 TIME_ZONE

Optional

The time zone that the provisioned environment should use. Defaults to "Australia/Sydney".

This value is also written to `env.py` so it may be imported into `settings.py` and used for Django's `TIME_ZONE` setting.

4.2.4 SECRET_KEY

Optional

A value for the Django `SECRET_KEY` setting. If provided as an empty string, or left out of the file altogether, a default random string will be generated. This generated value is more secure than the default provided by Django's `startproject` - containing 128 characters from an expanded alphabet, chosen using Python's `random.SystemRandom().choice`.

If a default value is generated, it will be written back to this file so the same value can be used in the case of re-provisioning.

This value is also written to `env.py` so it may be imported into `settings.py` and used for Django's `SECRET_KEY` setting.

4.2.5 DB_PASS

Optional

The password to use for the default database user. If provided as an empty string, or left out of the file altogether, a default 20-character password will be generated.

If a default value is generated, it will be written back to this file so the same value can be used in the case of re-provisioning.

This value is also written to `env.py` so it may be imported into `settings.py` and used as a database password in Django's `DATABASES` setting.

4.2.6 ENV_PY_TEMPLATE

Optional

The template to use when writing the `env.py` file, as a file path relative to `provision/templates/`. Defaults to `env.py.txt`. A default template file is provided at `provision/templates/env.py.txt`.

See *Customising env.py* for more details on using custom `env.py` templates.

4.3 versions.sh

Location: `provision/versions.sh`

This file contains the versions of various packages to be installed during provisioning. Like `env.sh`, it is simply a shell script that gets executed by the provisioning scripts to load the variables it contains. Unlike `env.sh`, this file *should* be committed to source control. All environments should install the same versions of the software they use.

The included `versions.sh` comes with acceptable default values for all variables. It will not require modification unless the default values are unsuitable for your project.

4.3.1 BASE_PYTHON_VERSION

The "base" Python version is the version that will be used to create the virtualenv under which all relevant Python processes for the project will be run. It can be left blank in order to use the operating system's standard version.

If specified, it must be the full version string, e.g. "2.7.14", "3.6.4", etc. In addition, it must be a version recognised and usable by pyenv. Pyenv is used to automate the process of downloading and installing the specified version of Python, and using it to build the virtualenv (via its pyenv-virtualenv plugin).

4.3.2 PYTHON_VERSIONS

An array of Python versions to install, e.g. to use with tox for testing under multiple versions. It can be left empty to install no additional versions of Python on the provisioned system. If specified, each version should be a full version string, such as "2.7.14", "3.6.4", etc. For example:

```
PYTHON_VERSIONS=('2.7.14' '3.5.4' '3.6.4')
```

Pyenv is used to automate the download and installation of the specified versions.

These versions are installed *in addition* to any *base version*, but the same base version can be included in the list in order to control its position in the version priority list used with the `pyenv global` command. If the base version is *not* included in the list, it will be added to the end of it for the purposes of the `pyenv global` command. See the *feature documentation* for more details.

4.3.3 NODE_VERSION

The version of node.js to install. Only the major version should be specified - the latest minor version will always be used.

Installation is performed by first installing the relevant Nodesource apt repo, using a script from the Nodesource binary distribution repository on GitHub. Therefore, the version must correspond to a installation script provided by Nodesource.

Note: Regardless of this version setting, node.js will only be installed if a `package.json` file is present in the root directory of your project.

4.4 Configuring the firewall

Only applicable in production environments

Location: `provision/conf/firewall-rules.conf`

In production environments, the existence of the `provision/conf/firewall-rules.conf` file determines whether a firewall will be configured. A default file is provided, so be sure to remove it if no firewall is desired. The default file also defines a default set of useful firewall rules, namely:

- Allowing incoming traffic on port 22, for SSH connections
- Allowing incoming traffic on ports 80 and 442, for web traffic

Any modifications to these rules or additions to them should be done in the `firewall-rules.conf` file. Each line in the file simply needs to be a valid argument sequence for the `ufw` command. Refer to the manual for details on the `ufw` command syntax.

Making changes to this file and re-provisioning via `vagrant provision` will enact the changes.

4.5 Configuring nginx

Only applicable in production environments

4.5.1 nginx.conf

Location: `provision/conf/nginx/nginx.conf`

In production environments, this file is copied to `/opt/app/conf/nginx/nginx.conf` as part of the provisioning process. The provided nginx supervisor program references that location when providing a config file to the `nginx` command.

A default file is provided which requires no configuration out of the box.

The only aspect of the default configuration to note is that it passes access and error logs through to be written and rotated by supervisor.

Making changes to this file and re-provisioning via `vagrant provision` will enact the changes. Alternatively, on-the-fly changes can be made to the copied file, simply restarting nginx via `supervisorctl restart nginx` to make them effective.

Note: On-the-fly changes to the copied file will not survive re-provisioning. Any changes made to this file should be duplicated in `provision/conf/nginx/nginx.conf`.

4.5.2 Site config

Location: `provision/conf/nginx/site`

In production environments, this file is copied to `/etc/nginx/sites-available/<project_name>`, and symlinked into `sites-enabled`, as part of the provisioning process.

A default file is provided which **does require minimal configuration**: setting the `server_name` directive.

The default configuration contains a single server context for port 80, with three location contexts:

- `/static/`: Directly serving static content out of `/opt/app/static/`.
- `/media/`: Directly serving media content out of `/opt/app/media/`.
- `/`: Proxying to gunicorn via a unix socket.

Making changes to this file and re-provisioning via `vagrant provision` will enact the changes. Alternatively, on-the-fly changes can be made to the copied file, simply restarting nginx via `supervisorctl restart nginx` to make them effective.

Note: On-the-fly changes to the copied file will not survive re-provisioning. Any changes made to this file should be duplicated in `provision/conf/nginx/site`.

4.6 Configuring gunicorn

Only applicable in production environments

Location: `provision/conf/gunicorn/conf.py`

In production environments, this file is copied to `/opt/app/conf/gunicorn/conf.py` as part of the provisioning process. The provided gunicorn supervisor program references that location when providing a config file to the `gunicorn` command.

A default file is provided which requires no configuration out of the box.

The default configuration binds to nginx via a unix socket and passes error logs through to be written and rotated by supervisor.

Making changes to this file and re-provisioning via `vagrant provision` will enact the changes. Alternatively, on-the-fly changes can be made to the copied file, simply restarting gunicorn via `supervisorctl restart gunicorn` to make them effective.

Note: On-the-fly changes to the copied file will not survive re-provisioning. Any changes made to this file should be duplicated in `provision/conf/gunicorn/conf.py`.

4.7 Configuring supervisor

4.7.1 supervisord.conf

Location: `provision/conf/supervisor/supervisor.conf`

This file is copied directly into `/etc/supervisor/supervisord.conf` as part of the provisioning process.

A default file is provided which requires no configuration out of the box.

The only aspect of the default configuration to note is that it makes the supervisor socket file writable by the `supervisor` group. The `supervisor` group itself is added during provisioning, and the `webmaster` user is added to it, enabling the `webmaster` user to interact with `supervisorctl` without needing `sudo`.

Making changes to this file and re-provisioning via `vagrant provision` will enact the changes. Alternatively, on-the-fly changes can be made to the copied file, simply restarting supervisor via `service supervisor restart` to make them effective.

4.7.2 Supervisor programs

Location: `provision/conf/supervisor/dev_programs/` or `provision/conf/supervisor/production_programs/`

A separate set of supervisor program files is used in development and production environments. In either case, though, the entire contents of the relevant `*_programs` directory is copied into `/etc/supervisor/conf.d/` as part of the provisioning process.

Default programs are provided for running nginx and gunicorn in production environments. Neither program should require any configuration out of the box.

Making changes or additions to program files and re-provisioning via `vagrant provision` will enact the changes.

4.8 Configuring the user's shell environment

Location: `provision/conf/user/`

Any files found in the `provision/conf/user/` directory will be copied directly into the `webmaster` user's home directory. This facility can be used to provide config files that affect the logged in user's shell environment. E.g. `.gitconfig` for the configuration of *git*, or additional shortcut scripts under the `bin` subdirectory.

Note: Files will not be copied if they already exist in the user's home directory. This means local changes to these files will not be overwritten, and also that changes to the files in `provision/conf/user/` will not be applied when re-provisioning unless the home directory file is removed.

Note: Any files present in the `provision/conf/user/bin/` directory will be marked as executable when they are copied, and will be available on the system path.

4.9 Customising env.py

Location: `provision/templates/env.py.txt`

If a specific project has additional sensitive or environment-specific settings that are better not committed to source control, it is possible to modify the way `env.py` is written such that it can contain those settings, or at least placeholders for them.

The `env.py` file is written by taking a template and replacing placeholders with settings from `env.sh`. The default template lives in `provision/templates/env.py.txt`.

This template can be extended or replaced to produce a custom `env.py` file. `env.py` is just a Python file, so any custom template needs to generate valid Python code. Other than that, there is no limitation on what can be included in the `env.py` file, though it is recommended it remain a simple key/value store, with as little logic as possible.

Note: The `env.py` file will not be overwritten once it is created, so if the template is modified, the existing file will need to be removed prior to re-provisioning if a new file is to be generated.

4.9.1 Placeholders

The default template contains placeholders for the following settings: `DEBUG`, `SECRET_KEY`, `TIME_ZONE`, `PROJECT_NAME` and `DB_PASSWORD`.

These placeholders share the name of the setting, prefixed with a dollar sign. E.g. the placeholder for the `DEBUG` setting is `$DEBUG`.

When the `env.py` file is written, any occurrence of these placeholders within the template will be replaced with that setting's actual value.

A custom `env.py` template can use as many additional placeholders for these settings as necessary.

On its own, just customising the template cannot inject *additional* settings. But it can define the structure, and all the keys, that are necessary - such that viewing the `env.py` file shows all the values that need to be provided.

The following shows the default `env.py` template compared to an example that modifies the structure and adds an additional entry for an API key that isn't known at the time of provisioning, but needs to be added afterward.

```
# Default template
environ = {
    'DEBUG': $DEBUG,
    'SECRET_KEY': r'$SECRET_KEY',
    'TIME_ZONE': '$TIME_ZONE',
    'DB_USER': '$PROJECT_NAME',
    'DB_PASSWORD': r'$DB_PASSWORD'
}

# Example custom template
environ = {
    'DEBUG': $DEBUG,
    'SECRET_KEY': r'$SECRET_KEY',
    'TIME_ZONE': '$TIME_ZONE',
    'DATABASE': {
        'NAME': '$PROJECT_NAME',
        'USER': '$PROJECT_NAME',
        'PASSWORD': r'$DB_PASSWORD'
    },
    'API_KEY': r'<replace_this>'
}
```

4.9.2 Injecting additional settings

If a project has other settings that are generated as part of the provisioning process, such as a random password or key, it is convenient to also be able to inject it into the `env.py` file. Customising the template allows defining a key, but injecting the generated value itself cannot be done through the custom template alone.

That's where *project-specific provisioning* comes in.

The custom template simply needs to provide a placeholder that can be identified for replacement. As per the main settings, a unique name prefixed with a dollar sign works well. E.g. `$MY_CUSTOM_VALUE`. Then, in `project.sh`, add the following:

```
sed -i -r -e "s|\\\$MY_CUSTOM_VALUE|$MY_CUSTOM_VALUE|g" "/opt/app/src/project_name/
↪env.py"
```

The following shows a custom template that includes extra entries for credentials generated for RabbitMQ, installed and configured as per the project-specific provisioning *example*.

```
# Example custom template
environ = {
    'DEBUG': $DEBUG,
    'SECRET_KEY': r'$SECRET_KEY',
    'TIME_ZONE': '$TIME_ZONE',
    'DB_USER': '$PROJECT_NAME',
```

(continues on next page)

(continued from previous page)

```
    'DB_PASSWORD': r'$DB_PASSWORD',
    'RABBIT_USER': '$PROJECT_NAME',
    'RABBIT_PASSWORD': r'$RABBIT_PASSWORD'
}
```

CHAPTER 5

Project-specific Provisioning

Individual projects will usually require some additional provisioning that isn't included in these generic provisioning scripts. The `provision/project.sh` file provides support for this. If found, this shell script file will be executed during the provisioning process. Execution happens:

- after the project directory structure under `/opt/app/` is generated, allowing additions to be made to it
- after the `env.py` file is written, allowing it to be modified
- before Python and Node.js dependencies are installed, allowing required system libraries to be installed first

Some common uses for `project.sh` are:

- installing additional software and services
- altering system configuration files
- modifying the `env.py` file with additional generated settings
- generating the necessary media directory structure under `/opt/app/media/` (any subdirectories specified in `FileField`/`ImageField` `upload_to` arguments will need to exist before any upload is attempted)

5.1 Accessing `env.sh` settings

Any setting present in `env.sh` can be loaded into `project.sh` and can be used to control the provisioning done within. This includes any custom settings that may be added specifically for this process to use. Simply include the following at the top of the file:

```
source /tmp/env.sh
```

Note: `/tmp/env.sh` is a temporary copy of `env.sh` written when provisioning starts, simplifying the provisioning scripts' references to it.

5.2 Generating random strings

A helper utility exists for generating random strings, such as those used for passwords. The same utility is used to generate the database password and the Django `SECRET_KEY` setting when they are not provided. It uses Python, specifically `random.SystemRandom().choice()`, to pseudo-randomly generate a string of characters. The length of the string to generate is passed in. The alphabet used is a fixed set of letters, numbers and special characters, with several problem-causing characters excluded (such as quotes).

E.g. Generating a 12 character string:

```
MY_RAND_STR=$("$PROVISION_DIR/scripts/utils/rand_str.sh" 12)
```

Note: `$PROVISION_DIR` is a setting that can be loaded as per *Accessing env.sh settings* above.

5.3 Writing settings back to env.sh

Sometimes it is useful to write values back to `env.sh` so the same value can be read again in the event of re-provisioning. This is particularly important if *generating random strings*. A simple utility exists for doing exactly that. If the given variable name exists in `env.sh`, it is replaced. If it does not already exist, it is added to the end of the file.

E.g. To write a value stored in `$MY_VAR` to a variable called `SOME_VALUE` in `env.sh`:

```
"$PROVISION_DIR/scripts/utils/write_var.sh" 'SOME_VALUE' "$MY_VAR" "$PROVISION_DIR/
↪env.sh"
```

Note: `$PROVISION_DIR` is a setting that can be loaded as per *Accessing env.sh settings* above.

5.4 Full example

The following example demonstrates a custom `project.sh` file that:

- loads settings from `env.sh`
- installs and configures project-specific software - the RabbitMQ message broker
- generates a random password
- writes the generated password back to `env.sh`, to avoid generating a new one on re-provisioning
- injects the generated password into `env.py`, assuming a *custom template*

```
#!/usr/bin/env bash
# project.sh

# Source provisioning settings
source /tmp/env.sh

#
# Install and configure RabbitMQ
```

(continues on next page)

(continued from previous page)

```
#

# Generate a password if necessary, and write it back to env.sh
if [[ ! "$RABBIT_PASSWORD" ]]; then
    $RABBIT_PASSWORD=$("$PROVISION_DIR/scripts/utils/rand_str.sh" 12)
    "$PROVISION_DIR/scripts/utils/write_var.sh" '$RABBIT_PASSWORD' "$RABBIT_PASSWORD"
↪"$PROVISION_DIR/env.sh"
fi

# Install rabbitmq and create a user with the password
apt-get -qq install rabbitmq-server
rabbitmqctl add_user "$PROJECT_NAME" "$RABBIT_PASSWORD"

# Replace the env.py placeholder for the password
sed -i -r -e "s|\\\$RABBIT_PASSWORD|$RABBIT_PASSWORD|g" "$SRC_DIR/$PROJECT_NAME/env.py
↪"
```

CHAPTER 6

Usage in Production

The provisioning scripts are designed to be run independently of Vagrant in order to provision production environments that match those used in development. While provisioning is not as simple as `vagrant up`, it is very straightforward.

6.1 Production-specific features

There are several features that are only available in production environments. These include:

- *Firewall*
- *Nginx*
- *Gunicorn*

In addition, the following features behave differently when in a production environment:

- SSH access: the `vagrant` user is not on the list of SSH allowed users
- Python dependencies: only `requirements.txt` is considered, `dev_requirements.txt` is ignored
- Node dependencies: in `package.json`, only `dependencies` is considered, `devDependencies` is ignored
- The `pull+` *shortcut command* performs additional steps

6.2 Configuration

Due to the additional features supported in production environments, some additional configuration may be required. The following are some of the things to consider:

- *Configuring the firewall*
- *Configuring nginx*
- *Configuring gunicorn*

- *Configuring supervisor*

Of particular importance is the nginx site config. It **must** be modified to, at least, provide the `server_name` directive.

6.3 Provisioning

Provisioning in a production environment is not quite as simple as `vagrant up`, it requires a few more steps:

1. Create the `/opt/app/src` directory.
2. Copy the project source, including provisioning files into `/opt/app/src`. The provisioning files should be at `/opt/app/src/provision`. The easiest way to do this is probably to clone your git repository, if you use one.
3. Manually invoke the provisioning bootstrap script **as root**, passing it the name of the project:

```
$ /opt/app/src/provision/scripts/bootstrap.sh project_name_here
```

There are several final steps that automated provisioning does not take care of. This may be because they are unsafe to include in the provisioning process (e.g. in the event of re-provisioning), or because user input is required.

- `sudo apt-get upgrade` (see the *limitations documentation* for more details)
- In order to have sudo privileges, a password needs to be created for the `webmaster` user. When logged in as the `webmaster` user, simply run the `passwd` command to set a password.