The building blocks for a Vagrant environment for Django development.
Included are shell provisioning scripts and sample configuration files allowing the construction of a Vagrant guest machine designed to support either full Django projects (both development and production environments) and the development of single Django apps for packaging and distribution.

The provisioning scripts are idempotent. Re-running the provisioner will not reinstall programs or overwrite config files that already exist, but will update anything where necessary. Therefore, it is safe to run the provisioner on an already established guest machine to perform updates.

**Build modes**

The provisioning scripts support building the Vagrant guest machine in two slightly different ways:

- “project”: For a full Django project. The environment will be suitable for development as well as serving a Django website.
- “app”: For a single Django app.

The documentation of the *available features* indicates which mode each feature applies to - most apply to both. At a glance, the differences are:

- Full project builds install Python dependencies from a `requirements.txt` file, if one can be found. See *Python dependency installation* for details.
- The `env.py` settings file is written for full project builds only.
- App builds always set the `DEBUG` flag to 1.

Which mode is used is specified by the `Vagrantfile`.

**Assumptions and dependencies**

The provisioning scripts assume:
• The guest machine is Ubuntu.

• The code is synced to /vagrant/ in the Vagrant guest machine.

• If a manage.py file exists, it will be at /vagrant/. For full projects, this effectively means that the Django project root directory should be /vagrant/.

• Full Django projects will have a post-1.4 project structure, having a directory with a name equal to the specified project name under the project root directory (i.e. in the guest machine: /vagrant/<project name>/).

Several features are only enabled if a manage.py file is detected:

• Running migrations.

• The shortcut commands, as they are shortcuts to manage.py commands.

The shortcut commands also depend on the django-extensions app. It will need to be installed for them to work. This means including it in either your requirements.txt or dev_requirements.txt file (see Python dependency installation).

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.

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 and build mode. The included Vagrantfile is pretty basic, but it can be used as a foundation. See Vagrantfile for details.


4. Add any project-specific provisioning steps to a provision/project.sh file. See Project-specific provisioning for details.

5. Add any further configuration files to provision/conf/. See User Environment Config Files for details on how these files are applied.

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

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.
The environment of the Vagrant guest machine 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 provisioning scripts. Each of these files is described below.

**Vagrantfile**

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. This can be used with some minor modifications, or the relevant provisioner can be added to a custom Vagrantfile. In either case, the provision/scripts/bootstrap.sh shell provisioner needs to be configured.

```bash
config.vm.provision "shell" do |s|
  s.path = "provision/scripts/bootstrap.sh"
  s.args = ["<project name>" "<build mode>"]
end
```

Two variables are required to be passed to the provisioner:

- **Project name**: The name of the project. This dictates several features of the environment, as described below.
- **Build mode**: The build mode to use, see Build modes for details.

**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 name of the virtualenv 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`.

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`).

**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.

---

**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`.

---

**DB_PASS**

 Required

The password to use for the default database user.

---

**Warning:** Do NOT use the password in the example `env.sh` file.

---

**PUBLIC_KEY**

*Optional*

If given, the public key will be installed into `/home/vagrant/.ssh/authorized_keys` so it may be used to SSH into the Vagrant guest machine.

---

**Note:** No additional system users are created, the key will simply grant SSH login access for the default `vagrant` user.

---

**DEBUG**

*Optional*

This flag controls whether or not the Vagrant guest environment is a development or production environment. A value of 1 indicates a development environment, otherwise (including when it is not specified at all) it indicates a production environment.

When the *build mode* is “app”, there is no real notion of a production environment, so the `DEBUG` flag is always 1.
The flag affects:

- **Python dependency installation**: If 1, a `dev_requirements.txt` file will be used, if present, to install development-only Python dependencies.
- **Node.js dependency installation**: If 1, any `devDependencies` listed in the `package.json` file will be installed along with the listed dependencies.
- **env.py**: If 1, `DEBUG=True` is set, otherwise `DEBUG=False` is.

**TIME_ZONE**

*Optional*

The time zone that the Vagrant guest machine should be set to. 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.

**User Environment Config Files**

*Location: provision/conf/*

Any files found in the `provision/conf/` directory will be copied verbatim into the `vagrant` user’s home directory in the guest machine. This facility can be used to provide config files that affect the logged in user’s shell environment. E.g. `.gitignore` for the configuration of `git`, `.agignore` for additional “ignores” for the silver searcher `ag` command.

*Note:* Files will not be copied if they already exist in the user’s home directory. This means changes to these files on the guest machine will not be overwritten, and also that changes to the files in `provision/conf/` will not be applied (if re-running the provisioning process on an existing guest machine) unless the home directory file is removed.
CHAPTER 3

Features

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

**Custom SSH public key**

A user-defined SSH public key can be provided as the `PUBLIC_KEY` variable in the `env.sh` file. This will be installed into `/home/vagrant/.ssh/authorized_keys`, allowing it to be used to SSH into the guest machine as the `vagrant` user.

This is useful in situations where `vagrant ssh` is not supported out of the box, and you already have an alternate SSH client with an existing private/public key pair in operation. E.g. Using PuTTY under Windows.

**Time zone**

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

**Git**

Git is installed in the guest machine.

**Note:** A `.gitconfig` file can be placed in `provision/conf/` to enable configuration of the git environment for the `vagrant` user.
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. This can be used, for example, to exclude documentation from the search. A sample .agignore file is included.

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. 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

PostgreSQL

PostgreSQL is installed in the guest machine.

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).

Virtualenv

A virtualenv with a name equal to the project name is created in the guest machine, at /home/vagrant/. virtualenvs/<project name>. This virtualenv is automatically activated when the vagrant user SSHs into the machine.

Python dependency installation

If using the “project” build mode, the provisioner will look for a requirements.txt file defined in the project root directory (/vagrant/ in the guest machine). If found, these requirements will be installed into the virtualenv. This is designed to allow installation of Python packages required in a production environment. It is not suitable for
“app” builds, as apps should specify their dependencies in other ways, so they can be identified and installed along with the app.

For both “project” and “app” build modes, and where \texttt{DEBUG} is \texttt{1}, the provisioner will also look for a \texttt{dev_requirements.txt} file, also in the project root directory. If found, these requirements will also be installed into the virtualenv. This is designed to enable specification of Python packages required during development, that are \textit{not} required for the project/application to run in production. An example might include \texttt{sphinx}, for documentation generation.

### Node.js/npm

\texttt{Node.js} and \texttt{npm} are installed in the guest machine, if a \texttt{package.json} file is found in the project directory (\texttt{/vagrant/} in the guest machine). If a \texttt{package.json} file is added to the project at a later date, provisioning can be safely re-run to perform this step (using the \texttt{vagrant provision} command).

A \texttt{node_modules} directory is created at \texttt{/home/vagrant/node_modules/} and a symlink to this directory is created in the project root directory (\texttt{/vagrant/node_modules}). Keeping the \texttt{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.

\textbf{Note:} In order to create the \texttt{node_modules} symlink when running a Windows host and using VirtualBox shared folders, \texttt{vagrant up} must be run with Administrator privileges to allow the creation of symlinks in the synced folder. See \textit{Windows hosts} for details.

### Node.js dependency installation

\texttt{npm install} will be run in the project root directory.

If \texttt{DEBUG} is not set to \texttt{1}, \texttt{npm install --production} will be used, limiting the installed dependencies to those listed in the \texttt{dependencies} section of \texttt{package.json}. If it is \texttt{set} to \texttt{1}, \texttt{dependencies} and \texttt{devDependencies} will be installed. See the documentation on \texttt{npm install}.

### Running migrations

If a \texttt{manage.py} file is found in the project root directory, the management command \texttt{manage.py migrate} will be run after the virtualenv is built and activated, Postgres is installed and the database created.

\textbf{Note:} In order for \texttt{manage.py migrate} to execute, Django must have been installed via \texttt{requirements.txt} or \texttt{dev_requirements.txt} and the \texttt{DATABASES} setting in \texttt{settings.py} must be correctly configured.

### env.py

\textit{Only available when using the “project” build mode}

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

3.8. \textit{Node.js/npm}
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.

```python
# Using env.py
from . import env
ev.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:

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

The environ dictionary will always contain each of the following key/values:

- **DEBUG**: Will be True if `DEBUG` is set to 1, False otherwise (including when it is not defined at all).
- **DB_USER**: Set to the value of the `project name`.
- **DB_PASSWORD**: Set to the value of `DB_PASS`.
- **TIME_ZONE**: Set to the value of `TIME_ZONE`.
- **SECRET_KEY**: Automatically generated when the `env.py` file is first written. More secure than the default provided by Django’s `startproject`, this version containing 128 characters from an expanded alphabet, chosen pseudorandomly using Python’s `random.SystemRandom().choice`.

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

**Shortcut commands**

The following shell commands are made available for convenience:

- **shell+**: Simply a shortcut to `manage.py shell_plus`. Assumes installation of `django-extensions`, which defines the `shell_plus` command.
- **runserver+**: A shortcut to `manage.py runserver_plus`. It takes a port number as a required first argument, using it to call `manage.py runserver_plus 0.0.0.0:<port>`. Any further arguments provided will also be added to the `runserver_plus` command call. It has the following additional features:
  - Calls `manage.py clean_pyc` prior to calling `runserver_plus`.
  - Automatically restarts the runserver, after a 3 second delay, if it exits. This avoids the need to babysit the runserver - if an error occurs that causes it to exit, it will automatically restart. It will keep trying to get going until the error is fixed, without you needing to interact with it. Note that `clean_pyc` is not called between automatic restarts.

Assumes installation of `django-extensions`, which defines the `runserver_plus` and `clean_pyc` commands.
**Project-specific provisioning**

In addition to the above generic provisioning, any special steps required by your individual project 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.

---

**Note:** Project-specific provisioning is performed prior to the installation of Python and Node.js dependencies, so additional system libraries required by these dependencies can be installed.