VEE

Release 0.1

Mar 22, 2017

Contents

This project aims to encapsulate various package Packages in order to allow for consistent, versioned assembly of an execution environment across a fleet of OS X and Linux hosts.

Eventually, VEE may provide continuous deployment.

The initial goal is to include packages from:

- Homebrew (and linuxbrew);
- the Python Package Index;
- ad-hoc Python packages;
- ad-hoc general packages.

CHAPTER 1

Contents

Installation and Workflows

Basic Installation

```
# Install VEE from GitHub; it will prompt you for install location.
python <(curl -fsSL https://raw.githubusercontent.com/westernx/vee/master/install_vee.
↪py)

# Either add VEE to your environment, or to your .bashrc; the installer
# above defaults to:
export VEE=/usr/local/vee
export PATH=$VEE/src/bin:$PATH

# If working in a group, set permissions:
sudo chown $(whoami) $groupname $VEE
sudo chmod -R g=rwXs,o=rwX $VEE
```

User Workflow

```
# Before your first use of vee, it must be initialized. This command
# will error if already run, with no ill effects.
vee init

# Add the environment repository (replace with your git remote and name),
# and build it.
vee repo clone git@git.westernx:westernx/vee-repo westernx
vee upgrade

# Run some installed commands.
vee exec COMMAND
#   or:
```

```
eval "$(vee exec --export)"
COMMAND
#   or:
vee exec --export >> ~/.bashrc
COMMAND

# Whenever there are changes to the environment repo, you must "update"
# to fetch the changes, and "upgrade" to build the environment.
vee update
vee upgrade
```

Developer Workflow

```
# Specify where you want your dev packages to be, if not in $VEE/dev.
export VEE_DEV=~/dev

# Install a package for development. This must be a package that is
# referred to by the default repository.
vee develop install PACKAGE

cd ~/dev/PACKAGE

# Develop here; use `dev` to run in the dev environment.
dev MY_COMMAND

# Commit your changes to the package.
git commit -am 'What you did to PACKAGE.'

# Commit your changes to the VEE repo.
vee add PACKAGE
vee commit --patch -m 'Did something to PACKAGE.'

# Test locally.
vee upgrade
MY_COMMAND

# Push out the package, and repo.
git push
vee push
```

Manual Workflow

```
# Install some individual packages into the default environment.
# These will be lost upon the next "upgrade".
vee link homebrew+sqlite
vee link homebrew+ffmpeg --configuration='--with-faac'
vee link git+https://github.com/shotgunsoftware/python-api.git --name shotgun_api3
vee link --force https://github.com/westernx/sgmock/archive/master.zip --install-name␣
↪sgmock/0.1
vee link git+git@git.westernx:westernx/sgsession

# Link a few packages into an "example" environment.
vee link -e example examples/basic.txt
```

```
# Execute within the "example" environment.
vee exec -e example python -c 'import sgmock; print sgmock'
```

Core Concepts

Names

Package names are currently assumed to existing within a single namespace, regardless of what type of package they represent. This means that there are potentially severe collisions between a similarly named package in Homebrew and on the PyPI, for instance.

The `--name` argument is provided to allow for manual disambiguation.

In the future, we may add a concept of namespaces, such that Python projects exist within a "python" namespace, Ruby gems within "ruby", and Homebrew/others within "binaries".

Home

VEE's home is where it installs and links environments. It is structured like:

```
builds/         # Where packages are built; largely temporary.
dev/            # Where you work as a developer.
environments/   # The final linked environments you execute in.
installs/       # The installed (post-build) packages.
opt/            # Sym-links to last-installed version of every package.
packages/       # The packages themselves (e.g. tarballs, git repos, etc.).
repos/          # The requirement repositories which drive your environments.
src/            # VEE itself.
```

Environment

An environment is a single "prefix", linked from installed packages. Contains standard top-level directories such as `bin`, `etc`, `lib`, `include`, `share`, `var`, etc..

These are symlinked together using the least number of links possible; a directory tree that only exists within a single package will be composed of a single symlink at the root of that tree.

Since their link structure is then unknown, it is highly advised to not write into an environment.

Packages

Outside of VEE, packages are bundles provided by a remote source which contains source code, or prepared build artifacts. E.g. a tarball, zipfile, or git repository.

Within VEE, the `Package` class is more abstract, representing both abstract requirements and a concrete instance of them. It provides all state required for the various pipelines.

Requirement

A requirement is specification of a package that we would like to have installed in an environment. These are still represented via the `Package` class.

Requirement Specification

Requirements are specified via a series of command-line-like arguments. The first is a URL, which may be HTTP, common git formats, or VEE-specific, e.g.:

- `http://cython.org/release/Cython-0.22.tar.gz`
- `git+git@git.westernx:westernx/sitetools`
- `pypi:pyyaml`

The requirements are further refined by the following arguments:

These may be passed to individual commands, e.g.:

```
vee link pypi:pyyaml --revision=3.11
```

or via a `requirements.txt` file, which contains a list of requirements.

`requirements.txt`

The requirements file may also include:

- Headers, which are lines formatted like `Header:  Value`, e.g.:

```
Name: WesternX
Version: 0.43.23
Vee-Revision: 0.1-dev+4254bc1
```

- Comments beginning with `#`;
- Basic control flow, starting with `%`, e.g.:

```
# For the Shotgun cache:
% if os.environ.get('VEEINCLUDE_SGCACHE'):
    git+git@git.westernx:westernx/sgapi --revision=6da9d1c5
    git+git@git.westernx:westernx/sgcache --revision=cd673656
    git+git@git.westernx:westernx/sgevents --revision=a58e61c5
% endif
```

Environment Repository

An environment repository is a git repository which contains (at a minimum) a *requirements.txt* file.

They are managed by the cli_vee_repo command.

Execution Environment

VEE_EXEC_ENV
: A comma-delimited list of environment names that were linked into the current environment. If you actually use an environment repository, this will likely contain `"NAME/BRANCH"` of that repo. Each entry here will have a corresponding entry in *VEE_EXEC_PATH* as well.

VEE_EXEC_REPO
: A comma-delimited list of environment repository names that were linked into the current environment.

VEE_EXEC_PATH
A colon-delimited list of paths that are scanned to assemble the current environment.

VEE_EXEC_PREFIX
The first path scanned to assemble the current environment. This is mainly for convenience.

Pipelines

The package processing pipelines consist of a series of steps, the handlers of which are determined as the pipeline is processed. This enables us to defer the build type of a package (e.g. a Python distribution) until after it has been downloaded and extracted.

There are to main pipelines currently in VEE: the *Install Pipeline* and *Develop Pipeline*.

All pipelines must start with an "init" step, which must be indempodent, and may normalize user-specified data on the package (e.g. normalize the URL).

Install Pipeline

The install pipeline is the primary pipeline of VEE, and is reponsible for installing the packages that are used in the default runtime. It's steps are:

"init"

Normalizes user-specified data. This step **MUST** be indempodent, as it may run more than once in common usage. It may also set the package name/path.

"fetch"

The package is retrieved and placed at *Package.package_path*. This step should be idempotent (and so is assumed to cache its results and may freely be called multiple times).

"extract"

The package's contents ("source") are placed into *Package.build_path* (which is usually a temporary directory).

"inspect"

An opportunity to check meta-data and determine self-described dependencies. This step may also set build and install names/paths.

"build"

The source is built into a build "artifact".

"install"

The build artifact is installed into *Package.install_path*.

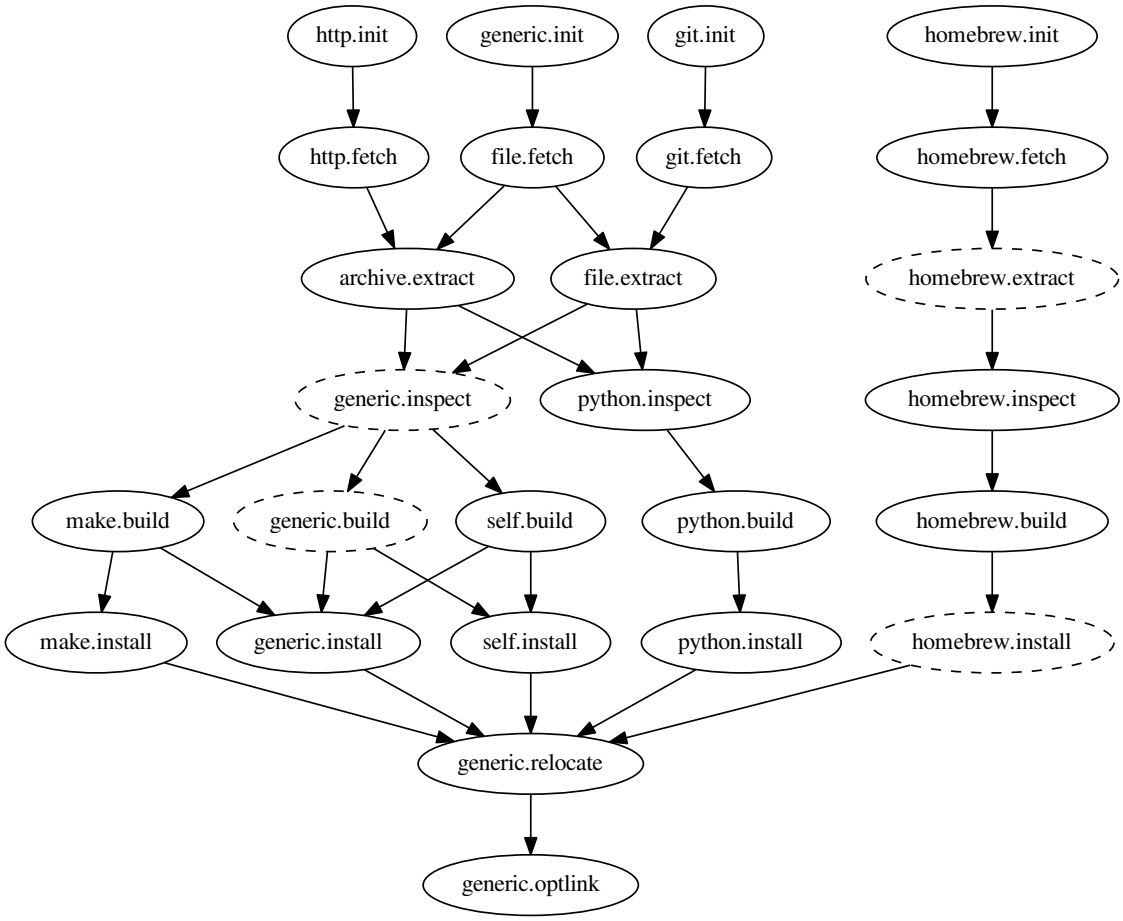
"relocate"

Shared libraries are relocated to link against existant libraries (in case they are not already relocatable, and their dependencies are not in the same location in all environments).

"optlink"

The *Package.install_path* is linked into `$VEE/opt`, for user convenience.

The built-in pipeline looks like:



Develop Pipeline

"init"

Same as above.

"develop"

Prepare the package for running in the development environment. Prepare any generated scripts, perhaps perform a build, and identify any environment variables to set in order to include this package in the runtime environment.

Names and Paths

There are a series of `*_name` attribute of a `Package`. They are set from `Requirement` attributes, or self-determined on request via `Package._assert_names(build=True, ...)`.

There are a series of `*_path` properties on a `Package`. They usually incorporate the corresponding name, but don't have it. They are set from `Package._assert_paths(build=True, ...)`.

> **Warning:** It is very important that an API consumer only every assert the existence of names or paths that they are about to use. This allows for the determination of some of the names (especially `install_name` and `install_path`) to be deferred as long as possible so that they may use information revealed during the earlier of the build pipeline.

The `*_name` attributes exist only for the construction of paths; API consumers should only ever use the `*_path` properties:

`Package`**.package_path**
: The location of the package (e.g. archive or git work tree) on disk. This must always be correct and never change. Therefore it can only derive from the requirement's specification.

`Package`**.build_path**
: A (usually temporary) directory for building. This must not change once the package has been extracted.

`Package`**.install_path**
: The final location of a built artifact. May be `None` if it cannot be determined. This must not change once installed.

`Package`**.build_subdir**
: Where within the build_path to install from. Good for selecting a sub directory that the package build itself into.

`Package`**.install_prefix**
: Where within the install_path to install into. Good for installing packages into the correct place within the standard tree.

Automatic Building

Most packages are inspected to determine which style of build to use. Unless otherwise stated, they will also use an automatic install process as well. The base styles (in order of inspection) are:

`. vee-build.sh`

If a `vee-build.sh` file exists, it will be sourced and is expected to build the package. A few environment variables are passed to assist it:

- `VEE`
- `VEE_BUILD_PATH`
- `VEE_INSTALL_NAME`

- VEE_INSTALL_PATH

The script may export a few environment variables to modify the install process:

- VEE_build_subdir
- VEE_install_prefix

python setup.py build

If a setup.py file exists, the package is assumed to be a standard distutils-style Python package. The build process is to call:

```
python setup.py build
```

and the install process will be (essentially) to call:

```
python setup.py install --skip-build --single-version-externally-managed
```

EGG-INFO or *.dist-info

If an EGG-INFO or *.dist-info directory exists, the package is assumed to be a prepared Python package (an Egg or Wheel, respectively), and no further build steps are taken. The install process will be modified to install the package contents into lib/python2.7/site-packages.

./configure

If a configure file exists, it will be executed and passed the install path:

```
./configure --prefix={package.install_path}
```

This continues onto the next step...

make

If a Makefile file exists (which may have been constructed by running ./configure), make will be called.

Automatic Installation

Unless overridden (either by the package type, or the discovered build type (e.g. Python packages have their own install process)), the contents of the build path are copied to the install path, like:

```
shutils.copytree(
    os.path.join(pkg.build_path, pkg.build_subdir)),
    os.path.join(pkg.install_path, pkg.install_prefix))
)
```

An optional --hard-link flag indicates that the build and install should be hard-linked, instead of copied. This results in massive time and space savings, but requires the packages to be well behaved.

Caveats

`make install`

Since we cannot trust that the standard `make; make install` pattern will actually install into a prefix provided to `./configure`, we do not run `make install`.

An optional `--make-install` flag signals that it is safe to do so.

`python setup.py install`

Instead of running `python setup.py install`, we break it into `python setup.py build` and `python setup.py install --skip-build`.

Some packages may not like this much.

Command-Line Interface

Python API

Warning: Very incomplete.

class `vee.home.`**`Home`**(*root=None*, *repo=None*)

The starting point of everything VEE.

Parameters

- **`root`** (*`str`*) – The root directory of the home; defaults to `$VEE`.
- **`repo`** (*`str`*) – The name of the default repository; defaults to `$VEE_REPO`.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

v

Index