Virtstrap Documentation

Release 0.3.2

Reuven V. Gonzales

March 14, 2012

CONTENTS

virtstrap makes your life easier, by providing a simple and repeatable bootstrapping process. It was inspired by pip+virtualenv and buildout, but seeks to create a unified and standard way that won't scare away any new python developers and will make old pythonista's lives a bit easier. All with the wonderful convenience of the MIT License.

With virtstrap, setting up your development environment is as simple as this:

```
$ vstrap init
```

This will make a new virtualenv in a directory `.vs.env` and a file called `quickactivate` in your current working directory. To use this virtualenv just type:

```
$ source quickactivate
```

Python development should be this simple. That's the goal of this project. To make setting up a project for python as simple as developing the project itself.

CHAPTER

ONE

CURRENT FEATURES

- Provides a standard location for virtualenv
- Provide a quick and simple way to activate the current environment
- Generate a requirements file much like a Gemfile.lock
- Provide a simple plugin system

CHAPTER

TWO

FUTURE FEATURES

- Allow for arbitrary environment variables to be set

CHAPTER

THREE

USER GUIDE

3.1 Virtstrap Installation Guide

3.1.1 Recommended method

Pip is the recommended method for installing virtstrap. Simply do:

```
$ pip install virtstrap
```

If you only have easy_install, you should install pip before continuing and execute the command above.

Once it has completed all the correct dependencies should be installed.

3.1.2 Installation from source

Due to the complexity of installing from source. It is highly discouraged at this time. However, this should hopefully work for most people

First grab the code:

```
$ git clone git://github.com/ravenac95/virtstrap-suite.git
```

Install the code:

```
$ make install
```

If you'd like to be able edit the code use this command instead:

```
$ make install-develop
```

But if you're ready to do that you may want to look at *Contributing to virtstrap* to get started.

3.2 Virtstrap Quickstart Guide

First, make sure you have virtstrap installed. If you do not, head on over to the *Virtstrap Installation Guide*.

3.2.1 Simplest virtstrap example

After virtstrap has been installed a command, vstrap, will be available on your command line. You can create an virtstrap enabled project just by doing the following:

```
$ mkdir myproject
$ cd myproject
$ vstrap init
```

This creates a virtualenv in the directory `myproject/.vs.env` and a bash script at `myproject/quickactivate`.

Finally, do:

```
$ source quickactivate
```

You now have a virtualenv for `myproject`.

3.2.2 Virtstrap with basic configuration

In the previous section we created the most simple type of virtstrap environment possible. However, without any configuration files virtstrap is a bit anemic. So let's start a simple configuration file to go along with the previous example.

In your favorite editor start a file called `VEfile` in your myproject directory (mine is vim):

```
$ vim VEfile
```

Let's say you'd like to grab two packages: Armin Ronacher's wonderful Flask micro web framework, and Kenneth Reitz's amazing requests HTTP library. Put the following inside `VEfile`:

```
requirements:
  - flask
  - requests: '>=0.10'
```

Save your file and run this command in your shell:

```
$ vstrap install
```

This command runs the installation portion of the init command. Doing `vstrap init` would have had the same effect. The `install` command skips some of the steps involved in `init`

After the command completes it's work, you will now have the latest version of *flask* and any *requests* package greater than version 0.10 inside your virtual environment. In order for this to happen, virtstrap converted the requirements defined in VEfile to a pip requirements. Pip then takes over and installs all of the requirements.

In the future, the VEfile will also generate a file called VEfile.lock which will contain the exact versions of the packages you just installed. This file like, Ruby Bundler's Gemfile.lock, should be added into your repository to create a truly repeatable project environment.

3.2.3 Repeatable environments. Because it matters

A repeatable environment is the main goal of virtstrap. As such let's take a look at exactly how that all works.

First let's get rid of the virtstrap environment. *VEfile and VEfile.lock are not deleted*:

```
$ vstrap clean
```

This brings an almost bare directory, save the configurations defined in VEfile.lock and VEfile. Finally do:

```
$ vstrap init
```

Your project is now brought us back to the state before we ran `vstrap clean`. The implication of this is that say you and Bob are working on this project together. Instead of emailing you and asking you about all the dependencies or manually creating a virtualenv and running a pip requirements all bob has to do is type the following inside his cloned project directory:

```
$ vstrap init
```

Now you're both ready to go. Beautiful isn't it :-)?

3.3 The VEfile

The VEfile is a central point of virtstrap. It allows you to define project metadata, requirements, and eventually options for plugins. The VEfile is a YAML file that uses some unique conventions to define the configuration.

3.3.1 It's just YAML

To understand the VEfile here's a short introduction to it's structure. The following is a valid VEfile:

```
foo: bar
unladen: swallow
python_is: awesome
```

In it's most basic form the VEfile is a simple dictionary or key/value storage. The top most level of keys are considered "sections" and their values can be anything. In the example above the sections are `foo`, `unladen`, and `python_is`. In python this VEfile simply becomes:

```
{'foo': 'bar', 'unladen': 'swallow', 'python_is': 'awesome' }
```

Just remember, you can define any key/value pair you wish in the VEfile and virtstrap will happily ignore any section (key) it doesn't recognize.

3.3.2 Virtstrap Sections

For the sections virtstrap does recognize, it expects particular types of values (although it's still pretty lenient). By design, none of the sections in virtstrap are required. This allows you to use virtstrap without any real specifications. However, once you're done being lazy and not setting up your project's repeatable environment, here are the sections you can set.

- *project_name*: Defines the project name. By default the project name is inferred from a projects root directory name. Set this if you'd like it to ensure consistency no matter where it is located.
- *requirements*: Defines the requirements for the project. This is the most useful section and one that you will probably use most. Requirements are explained in the next section, *The "requirements" Section*
- *plugins*: Defines the plugins just like you would define requirements. The suggested way to define plugins is declaration **Style 1** (explained in *The "requirements" Section*)

Note: The freezing of these requirements will be handled differently in the future. At this time there is no freezing of the plugin requirements, this will be changed soon once a proper solution is determined.

3.3.3 The "requirements" Section

The requirements section of the VEfile allow you to define your project's dependencies. Currently there are three styles of dependency declaration.

1. *Package name* - This is the simplest declaration. All you do is use the package name so your VEfile would look like this:

   ```
   requirements:
     - some_package # Syntax
     - flask        # Example
   ```

2. *Package name with version specification* - This declaration allows you to specify a version or a range of versions. The syntax is similar to defining a just a package name, but it separates the specification string from the package name by a colon. See here:

   ```
   requirements:
     - some_package: "some_spec"  # Syntax
     - flask: ">=0.7"             # Example
     - requests: "<=1.0"          # Another example
   ```

   `some_spec` can be any specification that is allowed by python's distutils.

3. *Package name with urls* - This declaration is the most complex and is meant to be used when you'd like to grab a package from a repository. The syntax may seem verbose for those used to pip's requirement syntax, but it is meant to be read more easily and hopefully more usable as well. See here:

   ```
   requirements:
     - some_package2:              # Syntax for normal urls
       - url_to_package_tar_or_zip

     - some_package1:              # Syntax for VCS
       - vcs_type+url_to_repo      # vcs_type must be git|bzr|hg|svn
       - editable: true            # This is optional and makes
                                   # a package editable

     - requests:                   # Example1 (normal url)
       - https://github.com/kennethreitz/requests/tarball/v0.10.6

     - flask:                      # Example2 (VCS url)
       - git+https://github.com/mitsuhiko/flask
       - editable: true
   ```

   Those familiar with pip will see that the syntax isn't too far off. The basic syntax for urls is one of two different types: the VCS url or a normal url. A VCS url **must** be preceded by a type, which is any of the following: git, hg, bzr, or svn. The normal url must point to a tar, zip, or a local directory.

Here's a full example of a requirements section that installs `flask`, `requests`, `virtstrap-core`, and `virtstrap-local`.

```
requirements:
  - flask
  - requests: ">=0.7"
  - virtstrap-core:
    - git+https://github.com/ravenac95/virtstrap-core.git
    - editable: true
  - virtstrap-local:
    - https://github.com/ravenac95/virtstrap-local/tarball/v0.3.0
```

3.3.4 Profiles

One additional, and powerful, part of VEfile's structure is it's ability to use profiles. In virtstrap, a profile is a particular type of environment you'd like to setup. These types of environments could be something like *development*, *testing*, *staging*, *production*, etc. Virtstrap makes little assumptions about the names you with to use for profiles. The *development* profile is the single exception. Virtstrap will always use the *development* profile if you do not specify a different profile. The reason for this is that most of your time with virtstrap will be spent developing code, so it should be simple.

In order to define profiles, VEfile utilizes YAML's concept of documents. Each document in a YAML file is separated by a `---`. The first document in the VEfile is always the default profile. This profile is always used regardless of the currently chosen profile. Every document after that must define a section `profile` whose value will be used as the profile name. Here's an example of a VEfile that uses profiles:

```
################################################
# This section is the default profile
# it is ALWAYS used. So don't put anything here
# that isn't absolutely necessary on every
# environment
################################################
project: tobetterus

requirements:
  - sqlalchemy
  - flask: ">=0.7"

some_value: foo

--- # This starts a new document (therefore a new profile)
########################################
# This profile is the development profile
# as defined by the section directly
# below this comment
########################################
profile: development

# Lists and dictionaries always append the other profile's data
# when profiles are combined
requirements:
  - ipython

# If it isn't a list or dictionary it's value
# is overridden entirely.
# So the value of some_value if you use the
# development profile will be 'bar'
some_value: bar

---
profile: production

requirements:
  - python-memcached
  - mysql-python
```

The VEfile above defines 3 profiles: *default*, *development*, and *production*.

To use profiles all you have to do is specify the `--profiles` options on the command line interface. You do this like so:

```
$ vstrap [command] --profiles=production,development
```

The line about will use both the production and the development profile. So the list of requirements installed will be `sqlalchemy`, `flask`, `ipython`, `python-memcached`, and `mysql-python`. In addition, if you request for the value `some_value` you will get the value `bar`, but that's only really useful if you're developing a plugin for virtstrap.

3.3.5 VEfile Suggestions

These are some suggestions when creating a VEfile.

- Use spaces instead of tabs (this is pretty much a suggestion for everything you write).
- Use 2 spaces for each tab level. This makes VEfiles a bit easier to read.
- Try not to specify exact versions for requirements in the VEfile. It is most powerful when you do not do that. Virtstrap is able to lock all the requirement versions so you can repeat your environment on each machine.
- Don't specify absolute file URL's. This makes your project less repeatable.

CHAPTER

FOUR

DEVELOPER GUIDE

4.1 Contributing to virtstrap

In order to provide for the an easy setup for the user, virtstrap has been split into 3 different packages. That are all combined into a single repository, virtstrap.

- **virtstrap** - This is the main package that users see. It provides the console script `vstrap` which is the main interface to anything virtstrap related. It also contains the commands that can be used without the presence of a project.
- **virtstrap-core** - This is the core of all of the virtstrap logic. The majority of virtstrap's code is contained in this core package. It is also a dependency for the other two packages.
- **virtstrap-local** - This package contains any commands that can only be used within a project and not throughout the system.

4.1.1 Start developing!

To start contributing to virtstrap is pretty simple. First, fork the repository on github. Once you've done that do the following:

```
$ make develop
$ source quickactivate.sh
```

Now you'll be in a virtualenv made for virtstrap.

4.1.2 Virtstrap Makefile

The virtstrap repository contains a Makefile that has the following commands:

- `develop` - Setup the development environment using an old version of virtstrap
- `testall` - Runs all of the tests in all the packages
- `supportfiles` - Builds the support files and places them into the virtstrap_support folder inside the virtstrap package.
- `install` - Installs virtstrap and virtstrap-core
- `install-develop` - Installs virtstrap and virtstrap-core as development versions (they're editable)

4.2 Plugin Development Quickstart

Note: This part of the documentation is extremely sparse and will be updated very soon.

One of the main goals of virtstrap is to provide a simple way to create plugins that can extend the functions of virtstrap for each project. Plugins can add functionality using two different objects:

- *Command* - A Command is any new command that you'd like to add to the command line interface
- *Hook* - A Hook is essentially an event listener. The hooks listen for events that occur during a command. The default events for almost every command are `before` and `after`. As a convention, commands should declare the events they call.

This quickstart guide goes through the creation of a hook. Eventually this will contain information on creating new commands.

4.2.1 A quick note about virtstrap's structure

In order to understand plugins, you have to understand a bit about how virtstrap works. Virtstrap has multiple packages explained below:

- `virtstrap-core` - The majority of virtstrap's logic is here. The other packages are dependent on this package. You cannot write plugins for this package. If you wish to extend it then refer to *Contributing to virtstrap*.
- `virtstrap` - This is the system wide package that contains the script for the `vstrap` command. It also contains the logic for the `init` and `commands` commands. Plugins written for `virtstrap` apply to the entire system. So the plugins for `virtstrap` should most likely be commands as hooks may cause unwanted functionality on different projects.
- `virtstrap-local` - Contains all the logic pertaining to anything project specific. Plugins written for `virtstrap-local` allow you to vary functionality depending on project. In general, the suggestion is to write plugins for `virtstrap-local`.

So let's start by creating a new `virtstrap-local` plugin.

4.2.2 Creating a virtstrap-local plugin

Like mentioned previously, `virtstrap-local` is the suggested package to extend with plugins. The reason is that your changes won't interfere with other projects on your system and it allows your plugins to be used in a repeatable fashion.

Step 1: Start a new python package

Virtstrap plugins, like buildout recipe's, are simply python packages. The package simple registers a module to an entry point and virtstrap takes over from there. So let's start out by creating a new package for to create our new plugin:

```
$ mkdir virtstrap-new-plugin
$ cd virtstrap-new-plugin
$ mkdir virtstrap_new_plugin
$ touch virtstrap_new_plugin/__init__.py
```

Note: For the time being, virtstrap assumes you're using a unix environment. At this time windows is untested although plans are in the works for a future release.

What we just did is create the scaffolding for a new python package. Next, we need to create a `setup.py` file. Open up your favorite editor (mine is vim) and copy this code (you can edit this if you are actually going to use this):

```
from setuptools import setup, find_packages
import sys, os

version = '0.0.0-dev'

setup(
    name='virtstrap-new-plugin',
    version=version,
    description="A new virtstrap plugin",
    long_description="""A new virtstrap plugin""",
    classifiers=[],
    keywords='virtstrap',
    author='John Doe',
    author_email='someone@someemail-place.com',
    url='',
    license='MIT',
    packages=find_packages(exclude=['ez_setup', 'examples', 'tests']),
    include_package_data=True,
    zip_safe=False,
    install_requires=[
        #'virtstrap-local',
    ],
    entry_points={
        'virtstrap_local.plugins': [
            'plugin = virtstrap_new_plugin.plugin',
        ]
    },
)
```

Step 2: Write the plugin!

In general the best way to get something new out of virtstrap is to use hooks to listen for events that commands issue. Commands will almost always issue the `before` and `after` event. The easiest way to determine what events are available is to use the event attribute of a command.

Here is a simple example:

```
from virtstrap import hooks

@hooks.create('install', ['after'])
def new_install_hook(event, options, **kwargs):
    print "I appear after install has completed!"
```

So what did this all do? Let's break it down!

1. First we need to import virtstrap.hooks which provides the `virtstrap.hooks.create` decorator.
2. Next we use the `virtstrap.hooks.create` decorator to define what command event we'd like to listen to. It requires two arguments.
   (a) The name of the command that you'd like to listen to - in our case the `install` command

(b) A list of events you'd like to listen for - in this case the `after` event.

3. Now we create the actual hook logic in the decorated function `new_install_hook`. The decorator expects the decorated function to accept two arguments and an arbitrary set of keyword arguments:
   (a) `event` - This is the current event that is being processed
   (b) `options` - This is an object representing the command line argument options. It should be used as read-only.
4. Finally we just print a message to the user (using an unrecommended method of output... more on this later)

The above example is the bare minimum you'd need to write to create a hook. Really, it's pretty lame. It simply prints the statement `I appear after install has completed` upon receiving the `after` event from the `install` command. However, let's do something a tad more interesting:

```
from virtstrap import hooks
from virtstrap.log import logger

@hooks.create('install', ['after'])
def new_install_hook(event, options, project=None, **kwargs):
    logger.info('The current path of the project is %s' % project.path())
```

There are three changes here.

1. We added an import on the second line to `virtstrap.log.logger`. This is the recommended way to output to the user. It allows virtstrap to log any messages, but also display pertinent messages to the user depending on the verbosity settings.
2. The `new_install_hook` function now has a different argument list. We've added `project=None` to the list of arguments. The project argument is passed in by `install` command's `after` event. The project object is an abstraction to the current project's directory and configuration information.
3. Finally on the last line we use the `logger`'s `info` method to display the current path of the project to the user.

There, that's a bit more interesting. However, it still does almost nothing. Let's do something crazy - like initialize a git repository!

Here goes:

```
from virtstrap import hooks
from virtstrap.log import logger
from virtstrap.utils import call_subprocess

@hooks.create('install', ['after'])
def new_install_hook(event, options, project=None, **kwargs):
    logger.info('Initializing a git repository for project at %s'
                % project.path())
    call_subprocess(['git', 'init', project.path()], show_stdout=False)
```

WOO! Finally, we're getting somewhere. This is what just happened:

1. We import `virtstrap.utils.call_subprocess`. This allows us to call a subprocess. It just makes dealing with subprocesses a tad bit easier. For now, you'll just have to trust it.
2. The next major change we introduce is running `call_subprocess` on the last line of code. What this line does is creates a git repository in your project root. Granted, this isn't really that useful after the install command has been run but it is definitely more interesting than printing out useless strings.

Step 3: Using the plugin

In order for you to use this plugin let's test it out with a new project.

Do the following in any directory you wish to use:

```
$ mkdir test-project
$ cd test-project
```

Next create a VEfile:

```
$ vim VEfile
```

Place this inside of it

```
plugins:
  - virtstrap-new-plugin:
    - file://PATH_TO_PLUGIN
```

Just replace `PATH_TO_PLUGIN` with the actual path to the plugin's directory.

Step 4: Init the project. Watch the magic happen

Finally, from within your new project directory do this:

```
$ vstrap init
```

You should see the following:

```
... (normal virtstrap messages)
Initializing a git repository for project at SOME_DIRECTORY
... (more messages)
```

Now you can do this:

```
$ git status
```

And it you should see that there's a git repository in your current directory!

Recap: This is just an example and not a useful one.

As stated previously, this example isn't very useful in a real project. If you'd like to see a useful example of this type of plugin checkout virtstrap-ruby-bundler