Platter Documentation

Release 0.1

Armin Ronacher

March 20, 2015

Contents

Platter is a tool for Python that simplifies deployments on Unix servers. It's a thin wrapper around pip, virtualenv and wheel and aids in creating packages that can install without compiling or downloading on servers.

Why would you want to use it?

- fastest way to build and distribute Python packages in an ecosystem you control. With the built-in caching we have seen build time improvements from 400 seconds down to 20 seconds for releases with no version changes on dependencies.
- no need to compile or download anything on the destination servers you distribute your packages to. Everything (with the exception of the interpreter itself) comes perfectly bundled up.
- 100% control over your dependencies. No accidental version mismatches on your servers (this includes system dependencies like setuptools, pip and virtualenv).

You can get the tool directly from PyPI:

```
$ pip install platter
```

To create a platter distribution all you need is this:

```
$ platter build /path/to/your/python/package
```

Once this finishes, it will have created a tarball of the fully built Python package together will all dependencies and an installation script. You can then take this package and push it to as many servers as you want and install it:

```
$ tar -xzf package-VERSION-linux-x86_64.tar.gz
$ cd package-VERSION-linux-x86_64
$ ./install.sh /srv/yourpackage/versions/VERSION
$ ln -sf VERSION /srv/yourpackage/versions/current
```

CHAPTER 1

Documentation Contents

1.1 Why Platter?

Platter is not the first software of it's kind that tries to help you with Python deployments. The main difference between platter and alternative solutions is that platter tries to diverge as little from common deployment scenarios and by providing the highest amount of stability and speed possible. Changes to the Python packaging infrastructure will not affect platter based deployments.

See also the the *why-not* headline for some differences with alternatives.

Platter also places a lot of emphasis on automation. Both the build and the installation process provides a lot of helpers for automatic usage through assisting tools.

1.1.1 Platter Operation

Platter distributions are based on Python wheels. It creates wheels for all dependencies of a Python package (including the package itself) and bundles it together with a installation script. That script then can create a brand new virtualenv and installs all dependencies into it.

This ensures that both the version of the system dependencies (setuptools, pip and wheel) as well as the versions of your own packages are 100% predictable. It never uses any packages that naturally come with the target operating system.

For as long as the platter distribution is installed on a compatible version of Unix it will install correctly without having to download or compile any Packages.

1.1.2 Supporting Automated Deployments

Platter supports the creation of automated deployments. You can use platter to create a python distribution on your build server, then download the tarball and distribute it across all target machines.

You only need to ensure that you use the same major version of Python on all machines (for instance 2.7.x or 3.4.x).

1.1.3 Why Not . . . ?

Platter is hardly the first package that tries to help with deployments. And it's also not the last one that there will be. In fact, there is a good chance Platter might not be the tool for you.

Pex

A popular deployment tool for Python is Twitter's pex. Platter and pex have very little in common other than that they are both intended for deploying things. Pex can be compared to jar files in Java. They contain an application in its entirety together with a virtualenv and provide various ways to interact with the contained application.

Pex is perfect for things such as command line applications that are written in Python, but also for various deployment scenarios that go above that.

Platter on the other hand isn't anywhere this fancy. Platter has two primary goals: be fast and be simple. Platter acknowledges that there are more things in an application than Python code and things that can execute from zipfiles. As such upon installation it just places a virtualenv on the file system and anything contained within works as normally. This is very useful when an application also needs to ship other files (such as config files, static media files, node.js modules etc.). Everything just ends up on the filesystem and is within an arm's reach.

venv-update

An alternative approach to fast deployment's is Yelp's venv-update. It tries to make things fast by figuring out the least amount of changes necessary to a virtualenv. This approach works rasonably well but causes problems if you want to move a virtualenv around. For instance it's not ideal if you want to have a version specific installation for quick rollbacks.

Some testing also does not reveal a noticable performance improvement of *venv-update* over platter.

Docker

Platter and Docker are good friends, but one does not replace the other. It makes a lot of sense to install a platter distribution into a docker container but it's probably not the best idea to use Docker alone. The reason for this is that Platter allows you to isolate the process of building and deploying, keeping the final server clean of unnecessary development dependencies (compilers etc.). It also means that you can disconnect your final deployment container entirely from the internet for security reasons. From the start.

1.2 Quickstart

To create platter packages you need an installation of Python 2.7. Note that platter does support the creation of Python 3 packages but itself is running on 2.x only.

Platter can be installed into a virtual environment with `pip`:

```
$ virtualenv venv
$ ./venv/bin/pip install platter
```

It's recommended to install platter into its own virtualenv as it has its own dependencies that might otherwise interfere with your system. However all packages build with platter are themselves created in a separate virtualenv.

1.2.1 Building Platter Packages

In order to create a platter package you need a setuptools based distribution. This means you need to have a `setup.py` file for your package. If you do not have one, consult the setuptools documentation for more information.

To then create a distribution all you need to do is to invoke `platter build` with the path to your package:

```
$ platter build /path/to/yourpackage
```

This will download all dependencies, compile all extension modules and pack them up. The resulting artifact will be created in a folder called *dist* in the current directory.

Alternatively you can also instruct platter to not create a final tarball and to instead just create a folder with all files:

```
$ platter build --format=dir /path/to/yourpackage
```

1.2.2 Installing Platter Packages

Once you have created such a platter package you can distribute it to different servers and install it there. Inside the tarball there is an install script `install.sh` which will install the platter package into a fresh and isolated virtualenv. Note that virtualenv itself is packaged up together in the platter tarball and the system version will *not* be used.

To install the package you can do something like this:

```
$ tar -xzf package-VERSION-linux-x86_64.tar.gz
$ cd package-VERSION-linux-x86_64
$ ./install.sh /srv/yourpackage/versions/VERSION
$ ln -sf VERSION /srv/yourpackage/versions/current
```

Note that platter tarballs have a lot of support for automatic deployments. For more information see *Automation with Platter*.

1.3 Customizing Builds

The default behavior of platter is to create a package that contains the following structure:

```
yourapp-<VERSION>-<PLATFORM>/
    PACKAGE
    VERSION
    PLATFORM
    info.json
    install.sh
    data/
        yourapp-<VERSION>-<PLATFORM>.whl
        yourdependency-<VERSION>-<PLATFORM>.whl
        virtualenv.py
        ...
```

For your package and all of the dependencies a wheel is created and placed in the data folder. Next to the data folder there are some useful files that contain meta information that is useful for automation (see *Automation with Platter*).

The package is build out of the *setup.py* file that you created for your project.

1.3.1 Virtualenv and Wheel Pinning

The version of virtualenv, setuptools and wheel that is used for building this is automatically discovered by default. It can however be explicitly provided on the command line in case you encounter a bug with the current version or the upgrade is incompatible with what you expect:

```
$ platter build --virtualenv-version VER ./package
$ platter build --wheel-version VER ./package
```

1.3.2 Specifying The Python Interpreter

By default the interpreter that is used in the virtualenv of platter is used. If you want to build for a different version (for example Python 3) you can provide it explicitly:

```
$ platter build -p python3.4 ./package
```

1.3.3 Passing pip Options

By default pip will execute without any extra arguments when building wheels. There are two ways to pass extra arguments to pip. The first is to set environment variables. These will also be used by the pip process that platter launches. The second option is to pass them on the command line. For instance if you want to change the pip cache you can use this command:

```
$ platter build --pip-option='--cache-dir=.cache' ./package
```

1.3.4 Extra Requirements

By default the dependencies are pulled from the `setup.py` file. In some circumstances it is a good idea to define extra dependencies in a requirements file. This is useful for instance if you have optional dependencies like database drivers that only apply for the production deployment but are not a strict requirement for the package itself.

In that case the `--requirements` (or `-r`) flag comes in useful. It can point to a requirements file:

```
$ platter build -r requirements.txt ./package
```

1.3.5 Custom Build Scripts

While platter is perfectly capable of creating Python distributions, it might encounter problems if you also want to ship other things with your application that are not native to the Python ecosystem. A good example for this is your application also wants to install some node-js modules into the virtualenv for instance.

In this case you can provide a custom pre-build or post-build script that is executed before or after the regular build and before packaging up. It can add additional data to the archive and also emit commands that end up in the install script.

The script needs to be executable and is invoked with some environment variables. The following environment variables exist:

Variable	Description
`HERE`	The path of the root folder in the archive. This is the folder where the install script ends up in and the parent folder of the data directory. This is where you can place additional metadata for instance. This is also guarnateed to be the working directory of the script.
`DATA_DIR`	The path of the bundled `data` folder in the archive. This is useful when you want to add more data into the data directory.
`SOURCE_DIR`	The path of the source directory. This is the directory of the Python package (the parent folder of the `setup.py` file).
`SCRATCHPAD`	A temporary folder provided for the script which is deleted after the execution. This is useful when you need to temporarily create files.
`INSTALL_SCRIPT`	The path to a auxilary installation script. You can echo install commands to this path and they are added to `install.sh` automatically.
`VIRTUAL_ENV`	The path to the virtual env that has been used for building the package. This can come in useful when you need to start a python interpreter or launch an executable in the venv. Note that the virtualenv is also guarnateed to be active.

The variables `HERE`, `DATA_DIR` and `VIRTUAL_ENV` are also available in the install script.

The post build script can be provided to the build command with the `--postbuild-script` parameter:

```
$ platter build --postbuild-script=build.sh ./package
```

Likewise pre-build scripts can be provided:

```
$ platter build --prebuild-script=prebuild.sh ./package
```

An example build script that ships a `npm` module in the virtualenv can look like this:

```
#!/bin/bash
set -eu

(cd "$DATA_DIR"; npm install --production uglify-js)

cat << "EOF" >> "$INSTALL_SCRIPT"
cp -R "$DATA_DIR/node_modules" "$VIRTUAL_ENV"
ln -s "../node_modules/.bin/uglifyjs" "$VIRTUAL_ENV/bin"
EOF
```

This will install a node executable into the virtualenv and then link the executable into the virtualenv's bin folder. What's piped into the `$INSTALL_SCRIPT` is added as commands to the `install.sh` script. Note that the double quoting of `EOF` (`"EOF"`) disables the interpolation so the variables are expanded at installation time, not at build time!

1.4 Automation with Platter

Platter is built with automation in mind. There are a handful of tools you can use platter together with to make automated deployments a reality.

1.4.1 Automated Building

To automate the building process we recommend Fabric. Fabric can be used to upload the source artifacts on a build server, then invoke the building process there and to fetch down the resulting build artifact. This allows you to trigger a build from any machine even if it does not have the correct architecture or operating system.

You will need `fabric` installed locally and a *fabfile.py* that looks something like this:

```
import os
import tempfile
from fabric.api import task, local, run, cd, get, hosts


@task
@hosts('my-build-server-hostname')
def build(rev='HEAD'):
    # ask git to create an archive for the right version.
    tmp = tempfile.mktemp(suffix='.tar.gz')
    local('git archive "%s" | gzip > "%s"' % (rev, tmp))

    # upload that archive to a temporary folder
    buildtmp = '/tmp/build-%s' % os.urandom(20).encode('hex')
    put(tmp, '%s/src.tar.gz' % buildtmp)

    # In that folder
    with cd(buildtmp):
        # extract the uploaded archive
        run('tar xzf src.tar.gz')
        # and invoke platter to build it
        run('/path/to/venv/bin/platter build .')
        # then download the archive and place it in 'dist'
        local('mkdir -p dist')
        get('dist/*.tar.gz', 'dist')

    # Clean up
    run('rm -rf %s' % buildtmp)
```

This example requires a few things:

1. in this case we use git as source control system. If you use something else you will need to adjust the code accordingly.
2. platter is installed on the build server into `/path/to/venv` into a virtualenv. You can obviously adjust this.

To build the package you can then run this:

```
$ fab build
```

Or to build a specific version:

```
$ fab build:rev=1.0-rc1
```

The resulting build artifact will end up in the *dist* directory next to the fabfile.

1.4.2 Automated Installing

For automated installation the archive can be placed on a server, extracted and installed. The usually recommended way for this is to extract the package to a version specific folder and to then symlink the virtualenv to an alias.

This is easy to accomplish because the tarball generated by platter contains metadata that can be used by tools. For instance it contains a file named `VERSION` with the version number.

Here an example *fabfile.py* which can upload a package to hosts:

```
import os
from fabric.api import task, put, run, cd

@task
def deploy(archive=None):
    # If not archive is provided, we use the 'last' one
    if archive is None:
        archive = os.path.join('dist',
            sorted(os.path.listdir('dist'))[-1])

    # Uplaod the archive and make some temporary space in /tmp
    put(filename, '/tmp/yourapp.tar.gz')
    run('rm -rf /tmp/yourapp && mkdir -p /tmp/yourapp')

    # Now enter the temporary folder
    with cd('/tmp/yourapp'):
        # Extract the archive, throwing away the toplevel folder
        run('tar --strip-components=1 -xzf /tmp/yourapp.tar.gz')

        # Ask for the version
        version = str(run('cat VERSION'))
        # Install into a version specific directory
        run('./install.sh /srv/yourapp/versions/%s' % version)

        # Create a symlink for the current version
        run('ln -sf %s /srv/yourapp/versions/current' % version)

    # Clean up the mess
    run('rm -rf /tmp/yourapp /tmp/yourapp.tar.gz')
```

You can then deploy an archive trivially:

```
$ fab -H myserver deploy
```

1.5 Frequently Asked Questions

These are some questions that came up about the library.

1.5.1 Can I use it on a Plane?

Platter will automatically use two levels of caching. The first level of caching is done by pip which will automatically cache downloaded packages in its local download cache. This allows pip to not download source archives if it has already downloaded them. This still needs internet access however. The second level of caching is the caching of entire pre-compiled wheels. Platter by default will place wheels in a wheel cache. It will still contact the internet to check for updates according to the version specification but you can disable this behavior by passing `--no-download` to the build command.

1.5.2 Where are Wheels Cached?

This depends on your operating system:

Operating System	Path
Linux	`~/.cache/platter`
OS X	`~/Library/Caches/platter`
Windows	`%LOCALAPPDATA%/platter/Cache`

1.5.3 How Can I Clean the Cache?

Either delete that folder yourself or run `platter clean-cache`.

1.5.4 Is the Cache Safe?

The cache is not particularly safe if you use multiple different Python versions next to each other in some circumstances. Normally you should not run into any issues except if you run different Pythons compiled against different libc's or unicode versions.

In that case it's recommended to use different cache paths for different incompatible interpreters. You can override the cache path by passing `--wheel-cache=/path/to/the/cache` to the build command.

CHAPTER 2

Miscellaneous Pages

2.1 Platter Changelog

This contains all major version changes between Platter releases.

2.1.1 Version 1.0

(no codename yet, release date to be decided)

- Initial release

2.2 License

Platter is licensed under a three-clause BSD License. It basically means: do whatever you want with it as long as the copyright in Platter sticks around, the conditions are not modified and the disclaimer is present. Furthermore, you must not use the names of the authors to promote derivatives of the software without written consent.

2.2.1 License Text

Copyright (c) 2015 by Armin Ronacher.

Some rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The names of the contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED