Deciphering

Glyph

Archives ⁜ Twitter ⁜ GitHub ⁜ Twisted

Deploying Python Applications with Docker - A Suggestion

A template for deploying Python applications into Docker containers.

Friday March 06, 2015

python deployment docker ops

Deploying python applications is much trickier than it should be.

Docker can simplify this, but even with Docker, there are a lot of nuances around how you package your python application, how you build it, how you pull in your python and non-python dependencies, and how you structure your images.

I would like to share with you a strategy that I have developed for deploying Python apps that deals with a number of these issues. I don't want to claim that this is the *only* way to deploy Python apps, or even a particularly *right* way; in the rapidly evolving containerization ecosystem, new techniques pop up every day, and everyone's application is different. However, I humbly submit that this process is a good *default*.

Rather than equivocate further about its abstract goodness, here are some properties of the following container construction idiom:

1. It reduces build times from a naive "`sudo setup.py install`" by using Python wheels to cache repeatably built binary artifacts.
2. It reduces container size by separating *build* containers from *run* containers.
3. It is independent of other tooling, and should work fine with whatever configuration management or container orchestration system you want to use.
4. It uses *existing* Python tooling of `pip` and `virtualenv`, and therefore doesn't depend heavily on Docker. A lot of the same concepts apply if you have to build or deploy the same Python code into a non-containerized environment. You can also incrementally migrate towards containerization: if your deploy environment is not containerized, you can still *build* and *test* your wheels within a container and get the advantages of containerization there, as long as your base image matches the non-containerized environment you're deploying to. This means you can quickly upgrade your build and test environments without having to upgrade the host environment on finicky continuous integration hosts, such as Jenkins or Buildbot.

To test these instructions, I used Docker 1.5.0 (via boot2docker, but hopefully that is an irrelevant detail). I also used an Ubuntu 14.04 base image (as you can see in the docker files) but hopefully the concepts should translate to other base images as well.

In order to show how to deploy a sample application, we'll need a sample application to deploy; to keep it simple, here's some "hello world” sample code using Klein:

```
# deployme/__init__.py
from klein import run, route

@route('/')
def home(request):
    request.setHeader("content-type", "text/plain")
    return 'Hello, world!'

def main():
    run("", 8081)
```

And an accompanying `setup.py`:

```
from setuptools import setup, find_packages

setup (
    name             = "DeployMe",
    version          = "0.1",
    description      = "Example application to be deployed.",
    packages         = find_packages(),
    install_requires = ["twisted>=15.0.0",
                        "klein>=15.0.0",
                        "treq>=15.0.0",
                        "service_identity>=14.0.0"],
    entry_points     = {'console_scripts':
                        ['run-the-app = deployme:main']}
)
```

Generating certificates is a bit tedious for a simple example like this one, but in a real-life application we are likely to face the deployment issue of native dependencies, so to demonstrate how to deal with that issue, that this `setup.py` depends on the `service_identity` module, which pulls in `cryptography` (which depends on OpenSSL) and its dependency `cffi` (which depends on `libffi`).

To get started telling Docker what to do, we'll need a base image that we can use for both build and run images, to ensure that certain things match up; particularly the native libraries that are used to build against. This also speeds up subsquent builds, by giving a nice common point for caching.

In this base image, we'll set up:

1. a Python runtime (PyPy)
2. the C libraries we need (the `libffi6` and `openssl` ubuntu packages)
3. a virtual environment in which to do our building and packaging

```
# base.docker
FROM ubuntu:trusty

RUN echo "deb http://ppa.launchpad.net/pypy/ppa/ubuntu trusty main" > \
    /etc/apt/sources.list.d/pypy-ppa.list

RUN apt-key adv --keyserver keyserver.ubuntu.com \
                --recv-keys 2862D0785AFACD8C65B23DB0251104D968854915
RUN apt-get update

RUN apt-get install -qyy \
    -o APT::Install-Recommends=false -o APT::Install-Suggests=false \
    python-virtualenv pypy libffi6 openssl

RUN virtualenv -p /usr/bin/pypy /appenv
RUN . /appenv/bin/activate; pip install pip==6.0.8
```

The apt options `APT::Install-Recommends` and `APT::Install-Suggests` are just there to prevent `python-virtualenv` from pulling in a whole C development toolchain with it; we'll get to that stuff in the build container. In the run container, which is also based on this base container, we will just use virtualenv and pip for putting the already-built artifacts into the right place. Ubuntu expects that these are purely development tools, which is why it recommends installation of python development tools as well.

You might wonder “why bother with a virtualenv if I’m already in a container”? This is belt-and-suspenders isolation, but you can never have too much isolation.

It’s true that in many cases, perhaps even most, simply installing stuff into the system Python with Pip works fine; however, for more elaborate applications, you may end up wanting to invoke a tool provided by your base container that is implemented in Python, but which requires dependencies managed by the host. By putting things into a virtualenv regardless, we keep the things set up by the base image’s package system tidily separated from the things our application is building, which means that there should be no unforseen interactions, regardless of how complex the application’s usage of Python might be.

Next we need to *build* the base image, which is accomplished easily enough with a docker command like:

```
$ docker build -t deployme-base -f base.docker .;
```

Next, we need a container for building our application and its Python dependencies. The dockerfile for that is as follows:

```
# build.docker
FROM deployme-base

RUN apt-get install -qy libffi-dev libssl-dev pypy-dev
RUN . /appenv/bin/activate; \
    pip install wheel

ENV WHEELHOUSE=/wheelhouse
ENV PIP_WHEEL_DIR=/wheelhouse
ENV PIP_FIND_LINKS=/wheelhouse

VOLUME /wheelhouse
VOLUME /application

ENTRYPOINT . /appenv/bin/activate; \
           cd /application; \
           pip wheel .
```

Breaking this down, we first have it pulling from the base image we just built. Then, we install the development libraries and headers for each of the C-level dependencies we have to work with, as well as PyPy's development toolchain itself. Then, to get ready to build some wheels, we install the `wheel` package into the virtualenv we set up in the base image. Note that the `wheel` package is only necessary for *building* wheels; the functionality to install them is built in to pip.

Note that we then have two volumes: `/wheelhouse`, where the wheel output should go, and `/application`, where the application's distribution (i.e. the directory containing `setup.py`) should go.

The entrypoint for this image is simply running "`pip wheel`" with the appropriate virtualenv activated. It runs against whatever is in the `/application` volume, so we could potentially build wheels for multiple different applications. In this example, I'm using `pip wheel .` which builds the current directory, but you may have a `requirements.txt` which pins all your dependencies, in which case you might want to use `pip wheel -r requirements.txt` instead.

At this point, we need to build the builder image, which can be accomplished with:

```
$ docker build -t deployme-builder -f build.docker .;
```

This builds a `deployme-builder` that we can use to build the wheels for the application. Since this is a prerequisite step for building the application container itself, you can go ahead and do that now. In order to do so, we must tell the builder to use the current directory as the application being built (the volume at `/application`) and to put the wheels into a wheelhouse directory (one called `wheelhouse` will do):

```
$ mkdir -p wheelhouse;
$ docker run --rm \
         -v "$(pwd)":/application \
         -v "$(pwd)"/wheelhouse:/wheelhouse \
         deployme-builder;
```

After running this, if you look in the `wheelhouse` directory, you should see a bunch of wheels built there, including one for the application being built:

```
$ ls wheelhouse
DeployMe-0.1-py2-none-any.whl
Twisted-15.0.0-pp27-none-linux_x86_64.whl
Werkzeug-0.10.1-py2-none-any.whl
cffi-0.9.0-py2-none-any.whl
# ...
```

At last, time to build the application container itself. The setup for that is very short, since most of the work has already been done for us in the production of the wheels:

```
# run.docker
FROM deployme-base

ADD wheelhouse /wheelhouse
RUN . /appenv/bin/activate; \
    pip install --no-index -f wheelhouse DeployMe

EXPOSE 8081

ENTRYPOINT . /appenv/bin/activate; \
           run-the-app
```

During build, this dockerfile pulls from our shared base image, then adds the wheelhouse we just produced as a directory at `/wheelhouse`. The only shell command that needs to run in order to get the wheels installed is `pip install TheApplicationYouJustBuilt`, with two options: `--no-index` to tell pip “don’t bother downloading anything from PyPI, everything you need should be right here”, and, `-f wheelhouse` which tells it where “here” is.

The entrypoint for this one activates the virtualenv and invokes `run-the-app`, the setuptools entrypoint defined above in `setup.py`, which should be on the `$PATH` once that virtualenv is activated.

The application build is very simple, just

```
$ docker build -t deployme-run -f run.docker .;
```

to build the docker file.

Similarly, running the application is just like any other docker container:

```
$ docker run --rm -it -p 8081:8081 deployme-run
```

You can then hit port 8081 on your docker host to load the application.

The command-line for `docker run` here is just an example; for example, I'm passing `--rm` so that if you run this example just so that it won't clutter up your container list. Your environment will have its own way to call `docker run`, how to get your `VOLUME`s and `EXPOSE`d ports mapped, and discussing how to orchestrate your containers is out of scope for this post; you can pretty much run it however you like. Everything the image needs is built in at this point.

To review:

1. have a common base container that contains all your non-Python (C libraries and utilities) dependencies. Avoid installing development tools here.
2. use a virtualenv even though you're in a container to avoid any surprises from the host Python environment
3. have a "build" container that just makes the virtualenv and puts wheel and pip into it, and runs `pip wheel`
4. run the build container with your application code in a volume as input and a wheelhouse volume as output
5. create an application container by starting from the same base image and, once again not installing any dev tools, `pip install` all the wheels that you just built, turning off access to PyPI for that installation so it goes quickly and deterministically based on the wheels you've built.

While this sample application uses Twisted, it's quite possible to apply this same process to just about any Python application you want to run inside Docker.

I've put a sample project up on Github which contain all the files referenced here, as well as "build" and "run" shell scripts that combine the necessary docker commandlines to go through the full process to build and run this sample app. While it defaults to the PyPy runtime (as most networked Python apps generally should these days, since performance is so much better than CPython), if you have an application with a hard CPython dependency, I've also made a branch and pull request on that project for CPython, and you can look at the relatively minor patch required to get it working for CPython as well.

Now that you have a container with an application in it that you might want to deploy, my previous write-up on a quick way to securely push stuff to a production service might be of interest.

(*Once again, thanks to my employer, Rackspace, for sponsoring the time for me to write this post. Thanks also to Shawn Ashlee and Jesse Spears for helping me refine these ideas and listening to me rant about them. However, that expression of gratitude should not be taken as any kind of endorsement from any of these parties as to my technical suggestions or opinions here, as they are entirely my own.*)