Building Containers for Python Applications

Moshe Zadka – https://cobordism.com

2021

Python is a popular language for many applications. Those that run in backend services, now in the 2020s, are usually run inside containers. Building containers for Python applications is common.

Often, with microservice architectures, it makes sense to build a "root" base image which all of the services will build off of. Most of the following will focus on the base image, since this is where it is easiest to make mistakes.

However, the applications themselves will also be covered: what is a good base if not something to build on top of?

Before continuing, I want to make an acknowledgement of country. I come from the city of Belmont, in the San Francisco Bay Area peninsula.

It was built on the ancestral homeland of the Ramaytush Ohlone people. You can learn more about them on their website,

Acknowledgement of Country

Belmont (in San Francisco Bay Area Peninsula)

Ancestral homeland of the Ramaytush Ohlone people

0.1 Good and Bad

Before talking about *how* to build good containers, there needs to be understanding of what *are* good containers? What distinguishes good containers from bad ones?

What is good

- To crush your enemies
- To see them driven before you
- Um, wrong slides

Ah, woops, these slides are from a different talk, about what is good *in life*, not what is good *in containers*. It's time to focus on the topic at hand. What kind of criteria distinguish good containers from bad?

What is good

- Fast
- Small
- Secure
- Usable

This is pretty high-level. What does "fast" mean? Fast at what? How small is "small"? What does it mean to be "secure"?

These guidelines are rough. Time to focus on concrete, measurable, criteria.

Specifying the requirements

Let's be more concrete

- Keep up to date
- Reproducible builds
- No compilers in prod
- Keep size (reasonably) small

OK, a bit better. But still not specific enough. How exactly is "keep up to date" a criterion? What size is "reasonably" small?

Start with "up to date". The most important part is that security updates from the upstream distribution will be installed on a regular cadence.

Up to date

Install security updates

But when? Depends!

This directly conflicts with the next goal: "reproducible builds". The abstract theory of reproducible builds says that giving the same source must result in bit-for-bit identical results. This has many advantages, but is non-trivial to achieve.

Lowering the bar a bit, the same source must lead to equivalent results. While this removes some advantages, it maintains the most important one. *Changing* the source by some amount only results in *commensurate* changes.

This is the main benefit of reproducible builds. It allows pushing small fixes with confidence that there are no unrelated changes. This allows less testing for small fixes, and faster delivery of hot patches.

Reproducible builds

Same code gives same results

...mostly

The next criterion sounds almost trivial: "no compilers in prod". Compile ahead of time, and store results in the image.

This criterion is here because without careful thinking and implementation, it is surprisingly easy to get wrong. Many containers have been shippped with gcc included because someone did not write their Dockerfile carefully enough.

No compilers in prod

A common anti-pattern

...surprisingly easy to get wrong!

On size, however, it is possible to spend an infinite amount of time. Every byte can be debated if it is worth it.

In practice, after getting into the low hundreds of megabytes, this quickly becomes a game of diminishing returns. Hours of work can go into carefully trimming a few hundred extra kilobytes.

The point at which to stop depends on the cost structure. Do you pay per GB? How much? How many different images use the base image? Is there something more valuable to do?

In practice, getting images down to low hundreds of megabytes (200 or 300) is fairly easy. Getting them below 200 is possible with a little more work. This is usually a good stopping point.

Size

- Diminishing returns
- Cost savings

One way to make the process of building a container image faster and more reliable is to use *binary wheels* for packages with native code. Whether it is in getting the wheels from PyPI, building wheels into an internal package index, or even building the wheels as part of a multistage container build, binary wheels are a useful tool.

Support binary wheels

Installing and building

Faster

Simplifies images

It is important to add a dedicated user for the container to run applications as. This is important for several reasons, but the overarching themes of all of them is that it is an important intervention to reduce risk.

In most setups, root inside the container is the same as root outside the container. This makes it much more likely that root can find a "container escape".

While it is not impossible for a regular user to find a privilege escalation bug and then escape as root, this increases the complexity of such an attack. Forcing attackers to use complex attacks is important, by both frustrating less dedicated ones and increasing the chances that a persistent attacker will trip an auditing alarm.

The other big reason is more mundane: a root user can do anything *inside* the container. Limiting those abilities is both a smart bug avoiding strategy and reduces the attack surface.

Not run as root

General hygiene

Running as root is also a necessary component for the next good idea: running with minimal privileges. Most importantly, it is a good idea to avoid write permissions as much as possible. The most important thing to avoid write permisions for is the virtual environment from which the application is running.

Avoiding such write permissions again lowers the attack surface by preventing code modifications at runtime.

Minimal privileges

Especially avoid permissions to pip install

Leaving security behind, the next thing to optimize for is performance. The most important speed-up criterion here is *rebuild* time.

Modern buildkit-based builds try to be smart about which steps prevent which cache invalidations. In a mutlistage build, they also try to run steps which provably are independent of each other in prallel.

Writing the Dockerfile to take advantage of this techniques is a non-trivial skill to master, but worthwhile. Especially useful is to think about which files change less than others.

One example trick: first copying requirements.txt and using it as an argument to pip install −r, before copying the source code and installing it.

This means that downloading and installing (and sometimes even compiling) the dependencies will only be cache-invalidated by the requirements.txt file. This allows faster rebuilds for the more common use-case the local source code changes.

Fast rebuilds

Responsiveness!

0.2 Bases

To make an apple pie from scratch, first create the universe. Creating the universe is a lot of thankless work, and there are probably more valuable ways to spend work time.

All this is to say you will probably start with FROM <some distro>. But which distro? Are we really going back to the 20th century to relight the distro wars? Hopefully not!

Instead, this will cover what kind of issues apply, specifically, to base container OSs for Python applications. Some distributions end up being better for this use-case, in some ways, than others.

Make your own choices!

Base OS

The distro wars are back?

One thing that is more important for containers than traditional uses of operating systems is that they are more sensitive to size overhead. This is because containers images tend to be in 1:1 correspondence with applications.

If an application builds a test build on every PR, and stores it in a registry for a while so that tests can be run on different environments on this PR, this stores a lot of versions of the OS in the registry.

Some of this is alleviated by containers sharing base layers, but in practice, less than naively assumed. This is because images will be built to take in security and critical bug patches. This tends to preturb the base OS often enough that caching, while useful, is no substitute for a smaller size.

Base - size

Most modern distros have a decent minimal server

...but Debian is easiest to get smallest.

Since applications are built on top of the base, it is useful if bumping the base version does not need to happen too often. Time application teams spend moving to a new base is time they are not spending developing useful customer-facing features.

This means it is good to find a base that has a long-term support version. Having a base with around 5 years worth of LTS allows reasonable planning for upgrades without making it a frequent exercise.

Base - LTS/support

Usually around 5 years

Gives you time to upgrade!

Together with LTS, it matters what is the policy of the base about updates. Does it update for general bugs? Only critical bugs? Security fixes? Does it do backports or tries to upgrade to new upstream versions?

Base - Volatility

How much change?

Security? Backports? Fixes?

Getting concrete, one popular choice is Debian. It is a conservative police on updates, and a 5 year LTS.

Debian

LTS: 5 years

Conservative

Another popular choice is Ubuntu. It has slightly more liberal polices (for example, it will allow backports for sufficiently good reasons). Those policies also depend on the subtle differences between universe and multiverse, that are beyond the scope of this talk.

Ubuntu

LTS: 5 years

(Universe, Multiverse, etc...)

Fairly conservative

Alpine is not a good choice for Python-based applications. Since it uses musl, and not glibc, it is not manylinux compatible. This makes a lot of binary wheel issues unnecessarily more complicated. This might change in the future with musllinux potential support, but for now, this is not the best choice.

Alpine (probably not)

Uses musl, not manylinux compatible

Some distributions have so-called "rolling releases". Instead of having a scheduled release updating to new upstream versions of all packages, new upstream versions added as they are released and integrated.

This works well for desktop, where using up to date versions is fun. It can even work well for non-ephemeral servers, where being able to do in-place upgrades long term allowes minimization of the need to do complete machine rebuilds.

For containers, rolling releases are a poor match. The main benefit of updating incrementally is completely lost, as each image is built from scratch. Container are built to be replaced wholesale.

The biggest downside of rolling releases is there is no way to get security updates without, potentially, getting new versions of upstream software. This can mean an expensive, immediate, need to support a new version of an upstream dependency to push out a security fix.

Rolling releases (probably not)

Up to date, but...

updates can change major versions!

Starting with CentOS 8, CentOS is more akin to a rolling release. It receives new versions of the upstream software and suffers from the same downsides as a base OS for containers.

CentOS

Rolling release!

0.3 Installing Python

Now that there is an operating system installed in the container, it is time for the piece de resistance: a Python interpreter. Running Python applications requires the interpeter and the standard library.

Somehow, the container needs to include them.

How to get Python?

So many options...

The most obvious choice, using `apt install pythonX.Y` or `dnf install pythonX.Y` is also, unfortunately, the worst one. The OS Python is primarily designed for other packages needing Python, not for arbitrary applications.

It often does not include venv or pip out of the box, and is sometimes even missing standard library modules such as `sqlite3`. This makes perfect sense for the OS Python interpreter. Packages which need Python are installed in their own area, and do not need pip or venv. If they need `sqlite3`, for example, they can explicitly note a dependency on the relevant OS package.

For applications this makes it a poor choice.

Not system Python

Distros aim Python at distro packages

not user programs.

There are some 3rd party repositories packaging Python for use in distributions as an OS package. The most famous one is deadsnakes for Ubuntu, which precompiles Python packages.

This is a popular choice. It does mean waiting for the right version to appear in the repository, but usually this happens with little delay.

Appropriate repositories

Famous examples: deadsnakes PPA for Ubuntu

Another option is to use pyenv. This is particularly useful if a single dev Python container image needs to have multiple versions of Python. The runtime versions can be built from it via careful copying, and allows some flows which require multiple versions of Python at buildtime to work.

Even without the need for multiple versions of Python, pyenv can be a popular choice. It is a well-trusted tool that can build Python inside a container.

pyenv

Builds and installs Python

One way to get the biggest benefit of pyenv without needing some of the overhead that is less useful in containers (like shims and the ability to switch versions) is to use python−build. This is the engine, inside pyenv, which builds Python.

Using it directly not only allows skipping redundancies but also configuring build details on a more granular basis. These are possible in pyenv, but the need to do a pass-through to python−build makes them more awkward, especially when there are a lot.

python-build

Builds and installs Python

Finally, or maybe initially, it is possible to do it like the people in the beforetimes. The configure/make/make install flow works, and removes any barriers between the developer and the build.

Any build parameters can be set, and tweaked. The main downside is the need to securely grab a tarball of the source code and avoiding supply-chain attacks.

Source

```
RUN configure [...]
RUN make
RUN make install
```

Build from source + Debian?python: images are basically that!

There are inherent trade-offs when choosing. The trade-offs are three-ways:

- How much control the local build has over the result
- How much work it is to implement
- The potential for issues

Ultimately, each team must decide for itself what trade-offs are right for it.

Trade-offs

Control vs. Work vs. Problems

It is usually a good idea to build several versions of the "base level" Python containers. This allows dependent containers to move to a new version at different times.

The minimum needed for this to work is 2. While more than 3 are possible, in practice, this is usually unnecessary. Python releases yearly, so three versions give two years to upgrade to a new, mostly-backwards compatible, version of Python.

If a team does not have slack over the course of two years, the problem is not one of Python versions. In practice, this means the choice is between supporting two or three versions of Python.

Versions

Support multiple for upgrade path

2-3

0.4 Thinking in Stages

Docker can build in *multiple stages*. Only one stage is output: by default, it is the last one. A different stage can be output by selecting it on the command-line.

The other stages can *help* that stage get build in two different ways.

Container multistage build (quick recap)

Only one stage output

other stages help

One way is to use a previous stage in a FROM command from in a new stage. This is the same as FROM an external image: it will start from the previous image, and then run the next steps as additional layers.

FROM

Use previous stage as starting image

Another way to use a non-output stage is to COPY files from it. This is similar to COPY from the Docker build context, but instead of using the build context, it uses a previous stage. The semantics of COPY as far as recursion, files, and directories, remain the same.

COPY –from

Copy files from previous stage

The FROM <stage> technique allows stages to be used as "common modules" in a Docker build file. If two images need several initial steps in common, those can be added to an internal "base" stage, and then both images use it as their starting point.

The downside is that this means that common modules, and all their dependents, must be in the same file. In general, as unpleasant as it is, projects should keep their Docker logic in one file, and not split it into several.

Stages a as modules

```
FROM ubuntu as security-updates
RUN add-apt-repository ppa:deadsnakes/ppa
RUN apt-get update
RUN apt-get upgrade

FROM security-updates as with-38
RUN apt-get install python3.8

FROM security-updates as with-39
RUN apt-get install python3.9
```

One of the biggest benefits of stages is that they allow separating build and runtime dependencies. The build time dependencies are installed in one stage, the build logic is executed, and the build artifacts are copied to the next stage that starts from a pristine image, without any of the build dependencies.

Separate build and runtime

Especially when building from source!

```
FROM ubuntu as builder
# install build dependencies
# build Python into /opt/myorg/python

FROM ubuntu as as runtime
COPY --from=builder \
      /opt/myorg/python \
      /opt/myorg/python
```

Especially for runtime images, there is a benefit in reducing the number of layers. One way to accomplish that is to have a stage of "preparing" a directory like /opt/myorg using several commands and file manipulations.

The next stage can be done in only one additional layer on the base: one COPY --from=prep-stage /opt/myorg/ /opt/myorg.

Optimizing layers

Put everything under /opt/myorg

Use one COPY −−from=...

If you build Python locally, remove (in the runtime image) the big things you want need: the static library, tests, various temporary build artifacts, etc. This is often done in a preparation stage, with the minimalistic Python build output copied to the next stage.

Optimizing size

After building Python, remove:

- Tests
- Builder dependencies (in runtime)
- ....and more

0.5 Use in Applications

Sometimes an application will have some parts written in native code. More often, the application will need third-party dependencies with native code. If those need to be built locally, they should be built in a separate stage from the runtime.

A popular technique is to first build all dependencies, then copying them to the runtime image to be installed in a virtual environment.

Binary wheels

- Build with builder
- Copy to runtime
- Install in virtual environment

Alternatively, the runtime image can be kept even smaller by installing in a virtual environment and then copying over the virtual environment as one big directory. This does need careful matching of the precise Python versions, and so depends on how the base system was created.

Binary wheels (alt)

- Build with builder
- Install in virtual environment
- Copy virtual environment to runtime

If building wheels is necessary, it is sometimes useful to make them self-contained. For that, a few dependencies are needed.

The most annoying one is patchelf. This is a tool to manipulate "ELF" files, especially shared libraries. The version in most distributions is broken, and the only way to get a working version is to compile it from recent sources.

Patchelf

Used to make wheels self-contained

Newest version needed

Patchelf is the low-level part. It is non-trivial to install, but does need a little wrapping. The tool that will make wheels self-contained is auditwheel. Luckily, once patchelf is properly installed, getting auditwheel can be done as long as Python and pip are properly configured.

Auditwheel

Use pip to install

Auditwheel can be used to create self-contained binary wheels. Such binary wheels will have any binary dependencies directly patched into them. This allows the need to install the "runtime" version of the library in the runtime image.

This reduces layers and complexity, but does require a high-degree of fidelity between the runtime and dev images.

Self-contained binary wheels

Run

```
auditwheel repair --platform linux_x86_64
```

No need for binary dependencies!

The need for this degree of fidelity can be an annoying requirement. Moreover, it could be nice to build the wheels once, not on evey docker build. If you have an internal package index (like devpi or any of the commercial alternatives) this can be arranged.

In order to build portable binary wheels, decide what is the oldest glibc you need to support. After you build a wheel on that platform, use auditwheel with portable tags to create an uploadable wheel.

This wheel will only be used by compatible systems, and more than one wheel can be uploaded.

Portable binary wheels

- Oldest supported?

Example:

auditwheel repair --platform manylinux_2_27_x86_64

Regardless of what the end-game of the binary wheel is, somehow it does need to get built. The actual build is simple: python -m build. The problem is what comes before. For some wheels, this is enough.

For other wheels, a few apt/dnf install of -dev libraries would do. For yet others, building them requires installing the Fortran or Rust toolchains.

Some require installing Java and then getting a custom build tool written in Java. Unfortunately, this is not a joke.

The instructions are, hopefully, in the package's documentation. At least encoding the instructions in a container build file is concrete, computer-readable, and repeatable, regardless of how long it takes to translate the documentation to these instructions.

Generating binary wheels

Build instructions in docs

Build dependencies

Now that Python *and* the PyPI packages are ready, they need to be copied to the runtime image. One way to reduce the layers is by reducing the copy instructions. Properly prepping directories in the dev image is better than copying bits and pieces to the runtime image.

Optimizing layers

Reduce copies

Prep

Carefully think about caching. Put time consuming steps as early as possible. Copy files from the context as late as possible. This means copying files *separately* if only some are needed.

The local Python sources change the fastest. Copy them last. If done right, the bottleneck is usually the final copy to the runtime image.

One way to speed up things is to have the dev image usable as a runtime image, so it can be used to debug locally.

Optimizing caching

Where to build wheel?

What invalidates caching?

0.6 Final Thoughts

There are many factors to consider when building a container for Python applications. While there are no objectively right answers, there are a lot of objectively wrong answers. There are more ways to be wrong than right, so doing things carelessly will lead to regrets.

It is worth it to think about these things and plan. The hours you spend in planning and thinking can repay themselves many times over by giving high quality images that are simpler to build, run, and audit.

Container build files are sometimes an afterthought, done haphazardly after "the code is done". This can hurt you. Spend time thinking before you implement container builds.

Conclusion

- Wrong easier than right
- But right is amazing
- Think before you docker

I have only touched on the surface of the things you need to know. Itamar has written a series of articles diving into many of these issues deeper.

Further Resources

Itamar's series – https://pythonspeed.com/docker/