Production-ready Docker packaging for Python

Itamar Turner-Trauring

https://pythonspeed.com

Why Docker packaging is complicated

- 1970s: Unix
- 1980s: TCP/IP networking
- 1990s: Python
- 2000s: Linux cgroups
- 2010s: Docker, modern Python packaging
- 2020s: 

The problem: too much to cover

The problem: too much to cover

- We only have 30 minutes.

The problem: too much to cover

- We only have 30 minutes.
- Over 60 packaging best practices.

The problem: too much to cover

- We only have 30 minutes.
- Over 60 packaging best practices.
- My training class takes 1.5 days.

Today: learn a process

- You have limited time at work, can get interrupted at any moment.
- Thus:
  - Iterative development.
  - Most important parts first.
  - Each step builds on previous steps.
- Will give some examples best practices, and link to resources at end of talk with far more details.

An iterative process

1. Get something working.
2. Security.
3. Running in CI.
4. Make images easy to identify and debug.
5. Improved operational correctness.
6. Reproducible builds.
7. Faster builds.
8. Smaller images.

1. Get something working

```
FROM python:3.8-slim-buster
COPY . .
RUN pip install .
ENTRYPOINT ["./run-server.sh"]
```

2. Security

- Before you can deploy *anything* publicly, it needs to be secure.
- So we do that next.

2. Security: Don't run as root

```
FROM python:3.8-slim-buster
RUN useradd --create-home appuser
USER appuser

WORKDIR /home/appuser
COPY . .
RUN pip install .
ENTRYPOINT ["./run-server.sh"]
```

2. Security: Other best practices

- Run with reduced capabilities.
- Make sure to install system package updates.
- Organizational processes to update dependencies when security fixes come out.
- And more!

3. CI

- You don't want to manually hand-build each image.
- You want teammates to be able to build images.
- So next step: integrate image building to your build/CI system.

```
#!/bin/bash
set -euo pipefail

py.test
docker build -t yourimage:latest .
docker push yourimage:latest
```

3. CI: Tag based on branch

- You want to build image for feature branch `123-more-cowbell` automatically.
- You want production not to be impacted.

```
#!/bin/bash
set -euo pipefail

GIT_BRANCH=$(git rev-parse --abbrev-ref HEAD)

docker build -t "yourimage:$GIT_BRANCH" .
docker push "yourimage:$GIT_BRANCH"
```

3. CI: Other best practices

- Once a week, rebuild without caching (`--pull --no-cache`) and redeploy.
- Run security scanners.
- Warm up the build cache with `docker pull` to get faster builds.
- And more!

4. Make it debuggable

- You've started automatically building and (probably) deploying.
- More likely to see errors.
- Lots of images all over the place.
- Next step: make images identifiable and easier to debug.

4. Debuggable: Tracebacks on crashes in C code

- If you have a bug in Python code, you get a traceback.
- If you have a bug in C code, you get a silent crash...
- ...unless you enable Python's built-in `faulthandler`.

```
ENV PYTHONFAULTHANDLER=1
ENTRYPOINT ["python", "yourprogram.py"]
```

4. Debuggable: Other best practices

- Record build metadata in the image using Docker labels.
- Write a smoke test for the build.
- Pre-install useful debugging tools.

5. Improve operational correctness

- Running in production, so you want to prevent operational problems.
- Correct and fast startup.
- Fast shutdown.
- Help the runtime environment correctly detect frozen processes.

5. Operational correctness: Pre-compile bytecode

- Python compiles source code `.pyc` for faster startup.
- If your image doesn't have `.pyc`, startup will be slower.

```
# Compile installed code:
RUN python -c "import compileall; \
  compileall.compile_path(maxlevels=10)"

# Compile code in a directory:
RUN python -m compileall yourpackage/
```

5. Operational correctness: Other best practices

- Correct signal handling for shutdowns.
- Handle zombie processes with `init`.
- Health checks.
- And more!

6. Reproducible builds

- Over the course of two weeks, your major dependencies won't change dramatically.
- Over six months, some of them will.
- Over two years, most of them will.
- So next, you want reproducible builds so you can update in a controlled manner.

6. Reproducible builds: Choose a good base image

- You'll want a Linux OS which does security updates while still guaranteeing backwards compatibility, for example Ubuntu LTS, Debian Stable, or CentOS.
- The official `python` images are based on Debian Stable, but give access to newer (or older) Python.
- `python:3.8-slim-buster` means "Python 3.8, on Debian Buster, the smaller version".

6. Reproducible builds: More best practices

- Pin Python package dependencies.
- Set up an organizational process to update Python dependencies.
- Optionally, pin system package dependencies.
- And more!

7. Faster builds

- Your images are now packaged *correctly*, so now you can focus on optimizations.
- Starting point: your time is expensive, you don't want to wait for builds.

7. Faster builds: Don't use Alpine Linux

- Alpine Linux is a small base image—but it can't use precompiled wheels from PyPI.
- As a result, you need to compile everything.
- Example: install `pandas` and `matplotlib`.
  - **`python:3.8-slim-buster`**: 30 seconds.
  - **`python:3.8-alpine`**: 1500 seconds, 50× slower!

7. Faster builds: More best practices

- `COPY` in files only when needed
- Like `COPY`, use `ARG` as late as possible.
- Install dependencies separately from your code.

8. Smaller images

- Final step is to make smaller images.
- It's nice to be more efficient, it can speed up test runs and production startup, but usually not the first thing to do.

8. Smaller images: Disable `pip`'s caching

- By default `pip` keeps copies of the downloaded package, in case you reinstall later.
- This wastes space, and you won't need it.

```
RUN pip install --no-cache-dir -r requirements.txt
```

8. Smaller images: Other best practices

- Add files to `.dockerignore`.
- Avoid extra `chown`.
- Minimize system package installation.
- And more!

Recap

1. Get something working.
2. Security.
3. Running in CI.
4. Make images easy to identify and debug.
5. Improved operational correctness.
6. Reproducible builds.
7. Faster builds.
8. Smaller images.

Thank you!

- Many of these best practices are covered in detail on a free guide on my website.
- Get the slides, and links to the free guide and other Python on Docker resources: https://pythonspeed.com/europython2020/
- Email: itamar@pythonspeed.com
- Twitter: @itamarst