Getting Started
Frequently Asked Questions
Part 1
Part 2
Part 3
Introduction to Part 3
Official Images and trust
Deployment pipelines
Using a non-root user
Optimizing the image size
Multi-host environments
End

Part 3 › Optimizing the image size

Optimizing the image size

A small image size has many advantages, firstly, it takes much less time to pull a small image from the registry. Another thing is the security: the bigger your image is the larger the surface area for an attack it has.

The following tutorial on "Building Small Containers" from Google is an excellent video to showcase the importance of optimizing your Dockerfiles:

Before going on to the tricks that were shown in the video, let us start by reducing the number of layers of a image. What actually is a layer? According to the documentation:

To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization.

So each command that is executed to the base image, forms a layer. The resulting image is the final layer, which combines the changes that all the intermediate layers contain. Each layer potentially adds something extra to the resulting image so it might be a good idea to minimize the number of layers.

To keep track of the improvements, we keep on note of the image size after each new Dockerfile. The starting point is

```
FROM ubuntu:22.04

WORKDIR /mydir

RUN apt-get update && apt-get install -y curl python3
RUN curl -L https://github.com/yt-dlp/yt-dlp/releases/latest/download/yt-dlp -o /usr/local/bin/yt-dlp
RUN chmod a+x /usr/local/bin/yt-dlp

RUN useradd -m appuser

RUN chown appuser .

USER appuser

ENTRYPOINT ["/usr/local/bin/yt-dlp"]
```

The built image has size **155MB**

As was said each command that is executed to the base image, forms a layer. The command here refers to one Dockerfile directive such as `RUN`. We could now glue all `RUN` commands together to reduce the number of layers that are created when building the image:

```
FROM ubuntu:22.04

WORKDIR /mydir

RUN apt-get update && apt-get install -y curl python3 && \
    curl -L https://github.com/yt-dlp/yt-dlp/releases/latest/download/yt-dlp -o /usr/local/bin/yt-dlp &
    chmod a+x /usr/local/bin/yt-dlp && \
    useradd -m appuser && \
    chown appuser .

USER appuser

ENTRYPOINT ["/usr/local/bin/yt-dlp"]
```

Image size is **153MB**.

There is not that much difference, the image with fewer layers is only 2 MB smaller.

As a sidenote not directly related to Docker: remember that if needed, it is possible to bind packages to versions with `curl=1.2.3` - this will ensure that if the image is built at a later date the image is more likely to work as the versions are exact. On the other hand, the packages will be old and have security issues.

With `docker image history` we can see that our single `RUN` layer adds 83.8 megabytes to the image:

```
$ docker image history yt-dlp

IMAGE          CREATED         CREATED BY                                      SIZE      COMMENT
a3f296f27a17   3 minutes ago   ENTRYPOINT ["/usr/local/bin/yt-dlp"]            0B        buildkit.docke
<missing>      3 minutes ago   USER appuser                                    0B        buildkit.docke
<missing>      3 minutes ago   RUN /bin/sh -c apt-get update && apt-get ins…   83.8MB    buildkit.docke
  ...
```

The next step is to remove everything that is not needed in the final image. We don't need the apt source lists anymore, so we can glue the next line to our single `RUN`

```
.. && \
rm -rf /var/lib/apt/lists/*
```

Now, after we build, the size of the layer is **108 megabytes**. We can optimize even further by removing the `curl` all the dependencies it installed. This is done by extending the command as follows:

```
.. && \
apt-get purge -y --auto-remove curl && \
rm -rf /var/lib/apt/lists/*
```

This brings us down to **104 MB**.

Exercise 3.6

EXERCISE 3.6

Return now back to our frontend and backend Dockerfile.

Document both image sizes at this point, as was done in the material. Optimize the Dockerfiles of both app frontend and backend, by joining the RUN commands and removing useless parts.

After your improvements document the image sizes again.

Alpine Linux variant

Our Ubuntu base image adds the most megabytes to our image. Alpine Linux provides a popular alternative base in https://hub.docker.com/_/alpine/ that is around 8 megabytes. It's based on alternative glibc implementation musl and busybox binaries, so not all software runs well (or at all) with it, but our container should run just fine. We'll create the following `Dockerfile.alpine` file:

```
FROM alpine:3.19

WORKDIR /mydir
```

```
RUN apk add --no-cache curl python3 ca-certificates && \
    curl -L https://github.com/yt-dlp/yt-dlp/releases/latest/download/yt-dlp -o /usr/local/bin/yt-dlp &
    chmod a+x /usr/local/bin/yt-dlp && \
    adduser -D appuser && \
    chown appuser . && \
    apk del curl

USER appuser

ENTRYPOINT ["/usr/local/bin/yt-dlp"]
```

Size of the resulting image is **57.6MB**

Notes:

- The package manager is `apk` and it can work without downloading sources (caches) first with `--no-cache`.
- For creating user the command `useradd` is missing, but `adduser` can be used instead.
- Most of the package names are the same - there's a good package browser at https://pkgs.alpinelinux.org/packages.

We build this file with `:alpine-3.19` as the tag:

```
$ docker build -t yt-dlp:alpine-3.19 -f Dockerfile.alpine .
```

It seems to run fine:

```
$ docker run -v "$(pwd):/mydir" yt-dlp:alpine-3.19 https://www.youtube.com/watch\?v\=bNw2i-mRT4I
```

From the history, we can see that our single `RUN` layer size is 49.8MB

```
$ docker image history yt-dlp:alpine-3.19

  ...
<missing>      6 minutes ago   RUN /bin/sh -c apk add --no-cache curl pytho…   49.8MB    buildkit.docke
  ...
<missing>      7 weeks ago     /bin/sh -c #(nop) ADD file:d0764a717d1e9d0af…   7.73MB
```

So in total, our Alpine variant is about 57.6 megabytes, significantly less than our Ubuntu-based image.

Image with preinstalled environment

As seen, yt-dlp requires Python to function. Installing Python to Ubuntu- or Alpine-based image is very easy, it can be done with a single command. In general, installing the environment that is required to build and run a program inside a container can be quite a burden.

Luckily, there are preinstalled images for many programming languages readily available on DockerHub, and instead of relying upon "manual" installation steps in a Dockerfile, it's quite often a good idea to use a pre-installed image.

Let us use the one made for Python to run the yt-dpl:

```
# we are using a new base image
FROM python:3.12-alpine

WORKDIR /mydir

# no need to install python3 anymore
RUN apk add --no-cache curl ca-certificates && \
    curl -L https://github.com/yt-dlp/yt-dlp/releases/latest/download/yt-dlp -o /usr/local/bin/yt-dlp &
    chmod a+x /usr/local/bin/yt-dlp && \
    adduser -D appuser && \
    chown appuser . && \
    apk del curl

USER appuser

ENTRYPOINT ["/usr/local/bin/yt-dlp"]
```

There are many variants for the Python images, we have selected *python:3.12-alpine* which has Python version 3.12 and is based on Alpine Linux.

The resulting image size is **59.5MB** so it is slightly larger than the previous one where we installed Python by ourselves.

Back in part 1, we published the Ubuntu version of yl-dlp with the tag *latest*.

We can publish whatever variants we want without overriding the others by publishing them with a describing tag:

```
$ docker image tag yt-dlp:alpine-3.19 <username>/yt-dlp:alpine-3.19
$ docker image push <username>/yt-dlp:alpine-3.19
$ docker image tag yt-dlp:python-alpine <username>/yt-dlp:python-alpine
$ docker image push <username>/yt-dlp:python-alpine
```

Or if we don't want to keep the Ubuntu version anymore we can replace that pushing an Alpine-based image as the latest. Someone might depend on the image being Ubuntu though.

```
$ docker image tag yt-dlp:python-alpine <username>/yt-dlp
$ docker image push <username>/yt-dlp
```

It's important to keep in mind that if not specified, the tag `:latest` simply refers to the most recent image that has been built and pushed, which can potentially contain any updates or changes.

Exercise 3.7

> **EXERCISE 3.7**
>
> As you may have guessed, you shall now return to the frontend and backend from the previous exercise.
>
> Change the base image in FROM to something more suitable. To avoid the extra hassle, it is a good idea to use a pre-installed image for both Node.js and Goland. Both should have at least Alpine variants ready in DockerHub.
>
> Note that the frontend requires Node.js version 16 to work, so you must search for a bit older image.
>
> Make sure the application still works after the changes.
>
> Document the size before and after your changes.

Multi-stage builds

Multi-stage builds are useful when you need some tools just for the build but not for the execution of the image (that is for CMD or ENTRYPOINT). This is an easy way to reduce size in some cases.

Let's create a website with Jekyll, build the site for production and serve the static files with Nginx. Start by creating the recipe for Jekyll to build the site.

```
FROM ruby:3

WORKDIR /usr/app

RUN gem install jekyll
RUN jekyll new .
RUN jekyll build
```

This creates a new Jekyll application and builds it. We are going to use Nginx to serve the site page but you can test how the site works if you add the following directive:

```
CMD bundle exec jekyll serve --host 0.0.0.0
```

We could start thinking about optimizations at this point but instead, we're going to add a new FROM for Nginx, this is what the resulting image will be. Then we will copy the built static files from the Ruby image to our Nginx image:

```
# the  first stage needs to be given a name
# the "first" stage needs to be given a name
FROM ruby:3 as build-stage
WORKDIR /usr/app

RUN gem install jekyll
RUN jekyll new .
RUN jekyll build

# we will now add a new stage
FROM nginx:1.19-alpine

COPY --from=build-stage /usr/app/_site/ /usr/share/nginx/html
```

Now Docker copies contents from the first image `/usr/app/_site/` to `/usr/share/nginx/html` Note the naming from Ruby to *build-stage*. We could also use an external image as a stage, `--from=python:3.12` for example.

Let's build and check the size difference:

```
$ docker build . -t jekyll
$ docker image ls
  REPOSITORY    TAG     IMAGE ID        CREATED          SIZE
  jekyll        nginx   9e2f597ad99e    8 seconds ago    21.3MB
  jekyll        ruby    5dae3d9f8dfb    26 minutes ago   1.05GB
```

As you can see, even though our Jekyll image needed Ruby during the build stage, it is considerably smaller since it only has Nginx and the static files in the resulting image. `docker run -it -p 8080:80 jekyll:nginx` also works as expected.

Often the best choice is to use a FROM **scratch** image as it doesn't have anything we don't explicitly add there, making it the most secure option over time.

Exercises 3.8 - 3.10

EXERCISE 3.8: MULTI-STAGE FRONTEND

Do now a multi-stage build for the example frontend.

Even though multi-stage builds are designed mostly for binaries in mind, we can leverage the benefits with our frontend project as having original source code with the final assets makes little sense. Build it with the instructions in README and the built assets should be in `build` folder.

You can still use the `serve` to serve the static files or try out something else.

EXERCISE 3.9: MULTI-STAGE BACKEND

Let us do a multi-stage build for the backend project since we've come so far with the application.

The project is in Golang and building a binary that runs in a container, while straightforward, isn't exactly trivial. Use resources that you have available (Google, example projects) to build the binary and run it inside a container that uses `FROM scratch`.

To successfully complete the exercise the image must be smaller than **25MB**.

EXERCISE 3.10

Do all or most of the optimizations from security to size for **one** other Dockerfile you have access to, in your own project or for example the ones used in previous "standalone" exercises.

Please document Dockerfiles both before and after.

Edit this page

Previous
« Using a non-root user

Next
Multi-host environments »