Multi-stage builds

Table of contents

Multi-stage builds are useful to anyone who has struggled to optimize Dockerfiles while keeping them easy to read and maintain.

Use multi-stage builds

With multi-stage builds, you use multiple `FROM` statements in your Dockerfile. Each `FROM` instruction can use a different base, and each of them begins a new stage of the build. You can selectively copy artifacts from one stage to another, leaving behind everything you don't want in the final image.

The following Dockerfile has two separate stages: one for building a binary, and another where the binary gets copied from the first stage into the next stage.

```
# syntax=docker/dockerfile:1
FROM golang:1.23
WORKDIR /src
```

```
COPY <<EOF ./main.go
package main

import "fmt"

func main() {
  fmt.Println("hello, world")
}
EOF
RUN go build -o /bin/hello ./main.go

FROM scratch
COPY --from=0 /bin/hello /bin/hello
CMD ["/bin/hello"]
```

You only need the single Dockerfile. No need for a separate build script. Just run `docker build`.

```
$ docker build -t hello .
```

The end result is a tiny production image with nothing but the binary inside. None of the build tools required to build the application are included in the resulting image.

How does it work? The second `FROM` instruction starts a new build stage with the `scratch` image as its base. The `COPY --from=0` line copies just the built artifact from the previous stage into this new stage. The Go SDK and any intermediate artifacts are left behind, and not saved in the final image.

Name your build stages

By default, the stages aren't named, and you refer to them by their integer number, starting with 0 for the first `FROM` instruction. However, you can name your stages, by adding an `AS <NAME>` to the `FROM` instruction. This example improves the previous one by naming the stages and using the name in the `COPY` instruction. This means that even if the instructions in your Dockerfile are re-ordered later, the `COPY` doesn't break.

```
# syntax=docker/dockerfile:1
FROM golang:1.23 AS build
WORKDIR /src
COPY <<EOF /src/main.go
package main

import "fmt"

func main() {
  fmt.Println("hello, world")
}
EOF
RUN go build -o /bin/hello ./main.go

FROM scratch
COPY --from=build /bin/hello /bin/hello
CMD ["/bin/hello"]
```

Stop at a specific build stage

When you build your image, you don't necessarily need to build the entire Dockerfile including every stage. You can specify a target build stage. The following command assumes you are using the previous `Dockerfile` but stops at the stage named `build`:

```
$ docker build --target build -t hello .
```

A few scenarios where this might be useful are:

- Debugging a specific build stage
- Using a `debug` stage with all debugging symbols or tools enabled, and a lean `production` stage
- Using a `testing` stage in which your app gets populated with test data, but building for production using a different stage which uses real data

Use an external image as a stage

When using multi-stage builds, you aren't limited to copying from stages you created earlier in your Dockerfile. You can use the `COPY --from` instruction to copy from a separate image, either using the local image name, a tag available locally or on a Docker registry, or a tag ID. The Docker client pulls the image if necessary and copies the artifact from there. The syntax is:

```
COPY --from=nginx:latest /etc/nginx/nginx.conf /nginx.conf
```

Use a previous stage as a new stage

You can pick up where a previous stage left off by referring to it when using the `FROM` directive. For example:

```
# syntax=docker/dockerfile:1

FROM alpine:latest AS builder
RUN apk --no-cache add build-base

FROM builder AS build1
COPY source1.cpp source.cpp
RUN g++ -o /binary source.cpp

FROM builder AS build2
COPY source2.cpp source.cpp
RUN g++ -o /binary source.cpp
```

Differences between legacy builder and BuildKit

The legacy Docker Engine builder processes all stages of a Dockerfile leading up to the selected `--target`. It will build a stage even if the selected target doesn't depend on that stage.

BuildKit only builds the stages that the target stage depends on.

For example, given the following Dockerfile:

```
# syntax=docker/dockerfile:1
```

```
FROM ubuntu AS base
RUN echo "base"

FROM base AS stage1
RUN echo "stage1"

FROM base AS stage2
RUN echo "stage2"
```

With BuildKit enabled, building the `stage2` target in this Dockerfile means only `base` and `stage2` are processed. There is no dependency on `stage1`, so it's skipped.

```
$ DOCKER_BUILDKIT=1 docker build --no-cache -f Dockerfile --target stage2 .
[+] Building 0.4s (7/7) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 36B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/ubuntu:latest
 => CACHED [base 1/2] FROM docker.io/library/ubuntu
 => [base 2/2] RUN echo "base"
 => [stage2 1/1] RUN echo "stage2"
 => exporting to image
 => => exporting layers
 => => writing image sha256:f55003b607cef37614f607f0728e6fd4d113a4bf7ef12210da3
```

On the other hand, building the same target without BuildKit results in all stages being processed:

```
$ DOCKER_BUILDKIT=0 docker build --no-cache -f Dockerfile --target stage2 .
Sending build context to Docker daemon  219.1kB
Step 1/6 : FROM ubuntu AS base
 ---> a7870fd478f4
Step 2/6 : RUN echo "base"
 ---> Running in e850d0e42eca
base
Removing intermediate container e850d0e42eca
```

```
 ---> d9f69f23cac8
Step 3/6 : FROM base AS stage1
 ---> d9f69f23cac8
Step 4/6 : RUN echo "stage1"
 ---> Running in 758ba6c1a9a3
stage1
Removing intermediate container 758ba6c1a9a3
 ---> 396baa55b8c3
Step 5/6 : FROM base AS stage2
 ---> d9f69f23cac8
Step 6/6 : RUN echo "stage2"
 ---> Running in bbc025b93175
stage2
Removing intermediate container bbc025b93175
 ---> 09fc3770a9c4
Successfully built 09fc3770a9c4
```

The legacy builder processes `stage1`, even if `stage2` doesn't depend on it.