Multi-stage builds are a new feature requiring Docker 17.05 or higher on the daemon and client. Multistage builds are useful to anyone who has struggled to optimize Dockerfiles while keeping them easy to read and maintain.

> **Acknowledgment**: Special thanks to Alex Ellis for granting permission to use his blog post Builder pattern vs. Multi-stage builds in Docker as the basis of the examples below.

Before multi-stage builds

One of the most challenging things about building images is keeping the image size down. Each instruction in the Dockerfile adds a layer to the image, and you need to remember to clean up any artifacts you don't need before moving on to the next layer. To write a really efficient Dockerfile, you have traditionally needed to employ shell tricks and other logic to keep the layers as small as possible and to ensure that each layer has the artifacts it needs from the previous layer and nothing else.

It was actually very common to have one Dockerfile to use for development (which contained everything needed to build your application), and a slimmed-down one to use for production, which only contained your application and exactly what was needed to run it. This has been referred to as the "builder pattern". Maintaining two Dockerfiles is not ideal.

Here's an example of a `Dockerfile.build` and `Dockerfile` which adhere to the builder pattern above:

`Dockerfile.build`:

```
FROM golang:1.7.3
WORKDIR /go/src/github.com/alexellis/href-counter/
COPY app.go .
RUN go get -d -v golang.org/x/net/html \
  && CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .
```

Notice that this example also artificially compresses two `RUN` commands together using the Bash `&&` operator, to avoid creating an additional layer in the image. This is failure-prone and hard to maintain. It's easy to insert another command and forget to continue the line using the `\` character, for example.

`Dockerfile`:

```
FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY app .
CMD ["./app"]
```

`build.sh`:

```
#!/bin/sh
echo Building alexellis2/href-counter:build
```

```
docker build --build-arg https_proxy=$https_proxy --build-arg
http_proxy=$http_proxy \
    -t alexellis2/href-counter:build . -f Dockerfile.build

docker container create --name extract alexellis2/href-counter:build
docker container cp extract:/go/src/github.com/alexellis/href-counter/app ./app
docker container rm -f extract

echo Building alexellis2/href-counter:latest

docker build --no-cache -t alexellis2/href-counter:latest .
rm ./app
```

When you run the `build.sh` script, it needs to build the first image, create a container from it to copy the artifact out, then build the second image. Both images take up room on your system and you still have the `app` artifact on your local disk as well.

Multi-stage builds vastly simplify this situation!

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. To show how this works, let's adapt the Dockerfile from the previous section to use multi-stage builds.

`Dockerfile`:

```
FROM golang:1.7.3
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=0 /go/src/github.com/alexellis/href-counter/app .
CMD ["./app"]
```

You only need the single Dockerfile. You don't need a separate build script, either. Just run `docker build`.

```
$ docker build -t alexellis2/href-counter:latest .
```

The end result is the same tiny production image as before, with a significant reduction in complexity. You don't need to create any intermediate images and you don't need to extract any artifacts to your local system

at all.

How does it work? The second `FROM` instruction starts a new build stage with the `alpine:latest` 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 are not 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.

```
FROM golang:1.7.3 AS builder
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go    .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /go/src/github.com/alexellis/href-counter/app .
CMD ["./app"]
```

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 `builder`:

```
$ docker build --target builder -t alexellis2/href-counter:latest .
```

A few scenarios where this might be very powerful 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 are not 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:

```
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
```