Docker İmajları ve Dockerfile

Bir Dockerfile nedir, FROM, WORKDIR, COPY, RUN, EXPOSE talimatları, CMD ile ENTRYPOINT arasındaki fark, ve docker build ile bir image inşa etmek -- jenerik, minimal bir web sunucusu örneği üzerinden.

Başlangıç 30 dk

Son güncelleme: 31 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/docker-images-and-dockerfiles

---

"Docker CLI Temelleri", başkasının zaten inşa ettiği bir image'ı çekti ve çalıştırdı -- `postgres:16`, Docker Hub'dan hazır geldi. Bu ders kendi image'ını inşa etmekle ilgili: Docker'ın yeni bir image üretmek için okuduğu düz metin tarifi olan bir `Dockerfile` yazmak, ve o tarifi gerçekten bir image'a dönüştürmek için `docker build` çalıştırmak. Buradaki örnek bilinçli olarak jenerik -- henüz Java'ya özgü hiçbir şey içermeyen, küçük bir web sunucusu -- böylece odakta kalan şey Dockerfile talimatlarının kendisi olsun; bu kategorideki bir sonraki ders "Bir Spring Boot Uygulamasını Docker'a Taşımak", tam olarak aynı talimat setini gerçek bir Spring Boot JAR'ına uyguluyor.

Dockerfile Nedir?

Bir `Dockerfile`, geleneksel olarak uzantısız tam olarak `Dockerfile` adını taşıyan, Docker'ın bir image üretmek için sırayla çalıştırdığı bir talimat dizisi içeren düz metin bir dosyadır. Aşağıda işlenen `FROM`, `COPY`, `RUN`, ve gerisi -- her talimat, bir öncekinin üzerine önbelleğe alınmış, yeni bir katman ekler, ki bu bir `Dockerfile`'dan inşa edilen bir image'ın "Image'lar vs. Container'lar"da tarif edilen dondurulmuş şablon gibi davranmasının nedenidir: bir kez inşa edildikten sonra, `docker build`'in değişmemiş girdilere karşı her yeniden çalıştırılışında farklı şekilde yeniden türettiği bir şey değil, sabit, tekrarlanabilir bir artifact'tir.

`FROM` — Bir Base Image Seçmek

Her `Dockerfile`, geri kalan her şeyin üzerine inşa edildiği image'ı adlandıran `FROM` ile başlar -- "Bir Image Çekmek: `docker pull`"in zaten işlediği tam olarak aynı `<repository>:<tag>` sözdizimi.

```
FROM alpine:3.20
```

Alpine Linux tam olarak bu nedenle yaygın bir base'dir: yalnızca birkaç megabayt büyüklüğünde, gerçek, minimal bir Linux dağıtımı, ihtiyaç duyulan daha fazlası için bir paket

yöneticisi (`apk`) mevcut -- bu belirli image'ın asla kullanmayacağı araçlarla dolu genel amaçlı bir OS image'ı yerine, bilinçli olarak küçük bir başlangıç noktası.

`WORKDIR` — Çalışma Dizinini Ayarlamak

`WORKDIR`, sonraki her talimatın göreli olarak çalıştığı dizini ayarlar, henüz yoksa image içinde onu oluşturur.

```
WORKDIR /app
```

Açık bir `WORKDIR` olmadan, `COPY` ve `RUN` gibi sonraki talimatlar image'ın dosya sistemi kökü göreli olarak çalışır -- teknik olarak geçerli, ama bir uygulamanın dosyalarını base image'ın kendi sistem dizinleriyle karıştırır. `WORKDIR /app`'i en başta bir kez ayarlamak, sonrasında gelen her şeyi belirsizlikten uzak ve düzenli tutar.

`COPY` — Dosyaları Image'ın İçine Getirmek

`COPY`, **build context**'ten -- `docker build`'in çalıştırıldığı klasör -- inşa edilen image'ın içine bir dosya (ya da dizin) getirir.

```
COPY index.html .
```

Buradaki `.`, "şu anki `WORKDIR`" için bir kısaltmadır -- `WORKDIR /app` zaten ayarlıyken, bu dosyayı image içinde `/app/index.html`'e yerleştirir. `COPY` yalnızca host makinedeki build context'ten okur; onun dışına hiç ulaşamaz -- `docker build`'in her zaman image'ın gerçekten ihtiyaç duyduğu her şeyi içeren klasörden çalıştırılmasının tam olarak nedeni budur.

`RUN` — İnşa Sırasında Komut Çalıştırmak

`RUN`, *image inşa edilirken* bir komut çalıştırır, ve o komutun disk üzerinde değiştirdiği her şey, ortaya çıkan image'ın kalıcı bir parçası olur.

```
RUN apk add --no-cache python3
```

Bu, Alpine'in paket yöneticisini kullanarak Python 3'ü image'a kurar -- `--no-cache`, paket yöneticisinin kendi indirme önbelleğini geride bırakmaktan kaçınır, ortaya çıkan image'ı daha küçük tutar (bkz. "Best Practices"). Bu, `alpine:3.20` gibi bir base image'ı bitmiş, amaca özel bir image'dan ayıran mekanizmadır: `RUN`, o belirli image'ın ihtiyaç duyduğu her şeyin, bir kez, inşa zamanında kurulduğu yerdir -- ondan bir container her başlatıldığında tekrarlanan bir şey değil.

`EXPOSE` — Bir Container'ın Portunu Belgelemek

`EXPOSE`, container içindeki uygulamanın hangi portu dinlediğini bildirir.

```
EXPOSE 8080
```

> `EXPOSE` belgeleme amaçlıdır, konfigürasyon değil -- portu kendi başına host makineye gerçekten yayınlamaz. Container'a alınmış bir uygulamaya dışarıdan ulaşmak hâlâ `docker run`'da `-p` gerektirir (`-p 8080:8080`, "Bir Container Çalıştırmak: `docker run`"da zaten işlenen), ki bu kursta ileride "Docker Networking" bunu tam olarak açıklıyor. `EXPOSE` esas olarak `Dockerfile`'ı okuyan bir insanın (ya da başka bir aracın) image'ın neyi beklediğini bilmesine yardımcı olur.

`CMD` vs `ENTRYPOINT`

Hem `CMD` hem `ENTRYPOINT`, bir container başladığında neyin çalışacağını belirtir -- fark, `docker run`'a ekstra argüman verildiğinde ne olduğudur.

```
FROM alpine:3.20

CMD ["echo", "Hello from CMD"]
```

```
docker run my-image
# Çıktı: Hello from CMD

docker run my-image echo "Overridden"
# Çıktı: Overridden
```

Yalnızca `CMD` varken, `docker run`'a verilen herhangi bir komut onu tamamen **değiştirir**. `ENTRYPOINT` farklı davranır -- o, her zaman çalışan sabit kısımdır, ve `CMD` (ikisi de mevcutken) ona *varsayılan argümanlarını* sağlar, ki `docker run` entrypoint'in kendisine dokunmadan bunu hâlâ geçersiz kılabilir:

```
FROM alpine:3.20

ENTRYPOINT ["echo"]
CMD ["Hello from CMD"]
```

```
docker run my-image
# Çıktı: Hello from CMD

docker run my-image "Overridden"
# Çıktı: Overridden
```

İkinci versiyonda, `echo` her zaman çalışır -- `docker run my-image "Overridden"` yalnızca varsayılan argümanı değiştirir, tamamen farklı bir komut değil, `echo "Overridden"` üretir. Bu `ENTRYPOINT` + varsayılan-`CMD` eşleşmesi, gerçek dünyadaki çoğu image'ın kullandığı desendir, "Docker CLI Temelleri"nde zaten çalıştırılmış resmi `postgres` image'ı dahil -- onun entrypoint'i her zaman PostgreSQL'i başlatır; sonrasındaki argümanlar, bunu kaybetmeden geçersiz kılınabilir.

Bir Image İnşa Etmek: `docker build`

`docker build`, bir `Dockerfile`'ı okur ve ondan bir image üretir.

```
docker build -t minimal-web-server:1.0 .
```

```
[+] Building 4.2s (9/9) FINISHED
 => [1/4] FROM docker.io/library/alpine:3.20
 => [2/4] RUN apk add --no-cache python3
 => [3/4] WORKDIR /app
 => [4/4] COPY index.html .
 => exporting to image
 => naming to docker.io/library/minimal-web-server:1.0
```

`-t minimal-web-server:1.0`, ortaya çıkan image'ı bir repository adı ve versiyonla etiketler -- `postgres:16` ile tam olarak aynı `<repository>:<tag>` şekli, şimdi Docker Hub'dan çekilen değil yerel olarak inşa edilen bir image'a uygulanıyor. Sondaki `.`, **build context**'tir: `docker build`'in `Dockerfile`'ı ve herhangi bir `COPY`'nin referans verdiği dosyaları okuduğu dizin -- yukarıdaki `Dockerfile`'daki `COPY index.html .`'in çalışmasının tam olarak nedeni bu, çünkü `index.html`, o dizinde `Dockerfile`'ın tam yanında oturuyor.

Hepsini Bir Araya Getirmek: Minimal Bir Dockerfile

Bu dersin inşa ettiği tam, çalışan `Dockerfile`:

```
FROM alpine:3.20

RUN apk add --no-cache python3

WORKDIR /app

COPY index.html .

EXPOSE 8080

CMD ["python3", "-m", "http.server", "8080"]
```

```
<!doctype html>
<html>
  <head>
    <title>Minimal Web Server</title>
  </head>
  <body>
    <h1>Served from inside a container.</h1>
  </body>
</html>
```

Ve tam inşa-çalıştır-doğrula-temizle döngüsü, "Docker CLI Temelleri"nin zaten işlediği tam olarak aynı CLI komutlarını kullanarak:

```
#!/bin/sh
# Building the minimal web server image from its Dockerfile, running it,
# confirming it actually serves the page, and cleaning up afterward.

docker build -t minimal-web-server:1.0 .
# Output (trimmed):
# [+] Building 4.2s (9/9) FINISHED
# => [1/4] FROM docker.io/library/alpine:3.20
# => [2/4] RUN apk add --no-cache python3
# => [3/4] WORKDIR /app
# => [4/4] COPY index.html .
# => exporting to image
# => naming to docker.io/library/minimal-web-server:1.0

docker run -d --name minimal-web-server -p 8080:8080 minimal-web-server:1.0
# Output:
# 3c4d5e6f7a8b9c0d1e2f3a4b5c6d7e8f9a0b1c2d3e4f5a6b7c8d9e0f1a2b3c4d

curl http://localhost:8080
# Output:
# <!doctype html>
# <html>
#   <head>
#     <title>Minimal Web Server</title>
#   </head>
#   <body>
#     <h1>Served from inside a container.</h1>
#   </body>
# </html>

# Confirmed -- the container is actually serving the file COPY placed inside
# the image. Clean up.
docker stop minimal-web-server
docker rm minimal-web-server
```

Bu `Dockerfile`'daki her talimat yukarıda tek tek tanıtıldı -- `FROM` base'i seçer, `RUN` Python'ı kurar, `WORKDIR` ve `COPY` sunulan dosyayı yerleştirir, `EXPOSE` portu belgeler, ve `CMD` sunucuyu başlatır. Burada henüz Java'ya özgü hiçbir şey yok, ki tam olarak mesele bu: "Bir

Spring Boot Uygulamasını Docker'a Taşımak", bu aynı talimat setini, değiştirmeden, yalnızca Python ve statik bir dosya yerine bir Spring Boot JAR'ına ve bir JRE base image'ına yönelterek yeniden kullanıyor.

Yaygın Hatalar

- `EXPOSE`'u sanki kendi başına bir portu yayınlıyormuş gibi ele almak -- yayınlamaz; container'a alınmış bir servisi dışarıdan gerçekten erişilebilir kılan şey `docker run`'daki `-p`'dir (yukarıdaki "`EXPOSE`"a bkz.).
- Niyet gerçekten "bu container her zaman şu tek şeyi yapar" olduğunda `CMD` kullanmak -- varsayılan bir `CMD`'li `ENTRYPOINT`, bu niyeti iletir ve `docker run`'a ekstra argüman verildiğinde doğru davranır.
- `docker build`'i yanlış dizinden çalıştırıp bir `COPY` talimatının beklediği dosyayı bulamaması -- build context her zaman `docker build`'e geçirilen dizindir (genelde `.`), farklıysa `Dockerfile`'ın kendi konumu değil.
- `WORKDIR`'i atlayıp dosyaların image'ın dosya sistemi kökünde, base image'ın kendi sistem dizinleriyle karışmış şekilde bitmesine izin vermek.

Best Practices

- Büyük, genel amaçlı bir base image yerine küçük, amaca özel bir base image tercih et (burada `alpine:3.20`) -- base image'daki her gereksiz araç, son image'ın ihtiyaç duymadığı boyut ve saldırı yüzeyidir.
- `RUN` kurulum komutlarında Alpine için `--no-cache`'i ya da paket yöneticinin eşdeğerini kullan -- geride kalan paket önbellekleri, hiçbir fayda sağlamadan ortaya çıkan image'ı yalnızca büyütür.
- `WORKDIR`'i, ona bağlı herhangi bir `COPY` ya da `RUN`'dan önce, `Dockerfile`'ın en başına yakın açıkça ayarla, dosya sistemi köküne güvenmek yerine.
- Her zaman tek bir belirli programı çalıştırması gereken image'lar için varsayılan bir `CMD`'li `ENTRYPOINT`'e başvur -- niyeti tek başına `CMD`'den daha kesin şekilde belgeler.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Bir `Dockerfile`, Docker'ın bir image üretmek için sırayla çalıştırdığı, düz metin bir talimat tarifidir -- her talimat bir katman ekler.
- `FROM` base image'ı seçer; `WORKDIR` sonraki talimatların göreli olarak çalıştığı dizini ayarlar; `COPY` build context'ten dosyaları image'a getirir; `RUN` inşa zamanında bir komut çalıştırır ve dosya sistemi değişikliklerini tutar.
- `EXPOSE`, container'a alınmış uygulamanın hangi portu dinlediğini belgeler -- o portu kendi başına yayınlamaz; `docker run`'daki `-p` yayınlar.

- Yalnızca `CMD`, `docker run`'a verilen herhangi bir komut tarafından tamamen değiştirilir; `ENTRYPOINT` sabit kalır ve ikisi de mevcutken `CMD` onun geçersiz kılınabilir varsayılan argümanı olur.
- `docker build -t <isim>:<tag> <context>`, bir `Dockerfile`'ı okur ve verilen dizini build context olarak kullanarak ondan etiketli bir image üretir.

Cheat Sheet

```
FROM <base-image>:<tag>      # geri kalan her şeyin üzerine inşa edildiği image
WORKDIR <yol>                 # sonraki talimatların göreli olarak çalıştığı dizin
COPY <kaynak> <hedef>         # build context'ten image'a bir dosya getir
RUN <komut>                   # inşa zamanında bir komut çalıştır, dosya sistemi
değişikliklerini tut
EXPOSE <port>                 # uygulamanın hangi portu dinlediğini belgele
CMD ["executable", "arg"]     # varsayılan komut; docker run argümanlarıyla tamamen
değiştirilir
ENTRYPOINT ["executable"]     # sabit komut; CMD onun varsayılan (geçersiz kılınabilir)
argümanı olur
```

```
docker build -t <isim>:<tag> .   # şu anki dizindeki Dockerfile'dan bir image inşa et
```

Terimler Sözlüğü

- **Dockerfile**: Docker'ın bir image inşa etmek için çalıştırdığı, sıralı talimatlardan oluşan düz metin bir dosya.
- **Katman (layer)**: bir Dockerfile talimatının ürettiği, önbelleğe alınmış ve bir öncekinin üzerine yığılmış dosya sistemi değişikliği.
- **Build context**: `docker build`'e geçirilen dizin -- `COPY`'nin (ve benzer talimatların) dosya okuyabileceği tek yer.
- **Entrypoint**: `ENTRYPOINT` ile yapılandırılmış bir container'ın, `docker run`'a hangi argümanlar verilirse verilsin her zaman çalıştırdığı sabit komut.