Docker Networking

Varsayılan bridge network'ün container'lar arasında isim çözümlemesini neden desteklemediği, user-defined network'lerle docker network create, container'dan container'a isimle iletişim, ve localhost'un bir container içinde neden farklı bir şey ifade ettiği.

Orta 30 dk

Son güncelleme: 6 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/docker-networking

---

Bu, Docker kursundaki ikinci kategori "Pratikte Docker"ın ilk dersi. "Docker CLI Temelleri", bir container'a host makineden ulaşmak için zaten `-p 5432:5432` kullanmıştı, ve "Bir Spring Boot Uygulamasını Docker'a Taşımak", bir container içinden host'taki bir servise `host.docker.internal` üzerinden ulaşmıştı -- ikisi de çalıştı, ama hiçbiri bir container'ın ağının gerçekte *nasıl* çalıştığını, ya da iki container'ın birbirine nasıl ulaşması gerektiğini açıklamadı. Bu ders bu boşluğu doğrudan dolduruyor, ve `host.docker.internal` geçici çözümünü gerçek mekanizmayla değiştiriyor: bu projenin kendi uygulamasını ve veritabanını, birbirine isim üzerinden ulaşan iki ayrı container olarak çalıştırarak.

Varsayılan Bridge Network'ü

Docker'ın başlattığı her container bir ağa bağlanır -- aksi söylenmediği sürece, bu Docker'ın kendi **varsayılan bridge network'ü**dür, açıkça bir tane belirtmeyen her container'ın paylaştığı özel, izole bir ağ. Üzerindeki container'lar dış internete ulaşabilir, ve host makine onlara yayınlanmış portlar üzerinden ulaşabilir (`-p` , "Bir Container Çalıştırmak: `docker run` "da zaten işlenen) -- ama varsayılan bridge network'ünün bilinçli olarak sağlaMADIĞI bir şey var:

> Docker'ın varsayılan bridge network'ündeki container'lar birbirine isimle ulaşa**maz** -- yalnızca her container yeniden başladığında değişen IP adresiyle. Bu, `host.docker.internal` ile etrafından dolaşılacak bir gözden kaçırma değil, gerçek, belgelenmiş bir Docker kısıtıdır -- gerçek çözüm, sonra işlenen bir **user-defined** (kullanıcı tanımlı) network'tür, ki tam olarak bu nedenle her gerçek çok-container kurulumu (bu kategoride ileride "Docker Compose" dahil) varsayılan yerine bir tane kullanır.

Portları Yayınlamak: `-p`'yi Yeniden Ziyaret Etmek

`-p 8080:8080` ("Bir Container Çalıştırmak: `docker run`"dan), host makinedeki bir portu bir container'ın kendi izole ağ namespace'i içindeki bir porta eşler -- olmadan, bir container'ın portları yalnızca Docker'ın dahili ağının içinde var olur ve Dockerfile'ındaki `EXPOSE`'un ne belgelediğine bakılmaksızın host'tan ya da dış dünyadan tamamen erişilemezdir. `-p <host-portu>:<container-portu>`, özellikle bir **host-to-container** köprüsüdür -- iki container'ın birbirine nasıl ulaştığıyla hiçbir ilgisi yoktur, ki tam olarak bu dersin geri kalanının işlediği boşluk budur.

Bir User-Defined Network Oluşturmak: `docker network create`

Bir **user-defined bridge network**, bir geliştiricinin açıkça oluşturduğu adlandırılmış bir ağdır, ve -- varsayılandan farklı olarak -- üzerine bağlanan container'lar arasında otomatik DNS-tabanlı isim çözümlemesiyle gelir.

```
docker network create learning-platform-net
```

```
4a3b2c1d0e9f8a7b6c5d4e3f2a1b0c9d8e7f6a5b4c3d2e1f0a9b8c7d6e5f4a3b
```

Bir container ona `docker run`'da `--network` ile katılır:

```
docker run --name learning-platform-db --network learning-platform-net -e
POSTGRES_PASSWORD=secret -d postgres:16
```

İsimle Container'dan Container'a İletişim

Gerçek kazanç bu: bir user-defined network üzerinde, bir container diğerine **diğer container'ın `--name`'ini** hostname olarak kullanarak ulaşabilir -- Docker, tam olarak bu amaç için gömülü bir DNS sunucusu çalıştırır.

```
docker run --name learning-platform-app \
  --network learning-platform-net \
  -e SPRING_DATASOURCE_URL=jdbc:postgresql://learning-platform-db:5432/postgres \
  -d learning-platform:0.1.0
```

O JDBC URL'sindeki `learning-platform-db`, internette hiçbir yerde gerçek bir DNS hostname'i değildir -- *yalnızca* `learning-platform-net` içinde, o ağda o anda o isimle çalışan hangi container varsa ona çözümlenir. Bu, "Bir Spring Boot Uygulamasını Docker'a Taşımak"ın kullandığı `host.docker.internal` geçici çözümünün kesinlikle daha iyi bir yerine geçenidir: o

hile *host makinedeki* bir servise ulaşır, birlikte çalışması amaçlanan iki container'dan temelde farklı (ve daha az taşınabilir) bir durum -- ki bu projenin gerçek bir deployment'ının gerçekte göründüğü şey tam olarak bu.

Network'leri İncelemek: `docker network ls` / `docker network inspect`

```
docker network ls
```

```
NETWORK ID     NAME                     DRIVER    SCOPE
a1b2c3d4e5f6   bridge                   bridge    local
b2c3d4e5f6a7   learning-platform-net    bridge    local
```

`docker network inspect <isim>`, o anda belirli bir ağa hangi container'ların bağlı olduğunu tam olarak gösterir -- birbirine ulaşması gereken iki container'ın gerçekten aynı ağda olduğunu doğrulamanın en hızlı yolu, ki bu "İsimle Container'dan Container'a İletişim"in beklendiği gibi çalışmamasının en yaygın tek nedenidir (bkz. "Yaygın Hatalar").

`localhost` Bir Container İçinde Neden Farklı Bir Şey İfade Eder

Her container kendi izole ağ namespace'ini alır -- bu, bir container *içindeki* `localhost`'un (ya da `127.0.0.1`'in) host makinenin değil, başka herhangi bir container'ın da değil, o container'ın kendi loopback arayüzüne başvurduğu anlamına gelir. Bir geliştiricinin kendi makinesinde doğrudan çalışan, yerel olarak kurulu bir PostgreSQL'e `jdbc:postgresql://localhost:5432/...` üzerinden bağlanan bir Spring Boot uygulaması, hem uygulama hem PostgreSQL ayrı container'lara taşındığında o aynı URL'yi basitçe yeniden kullanamaz -- uygulama container'ının içinden `localhost`, hiç PostgreSQL çalıştırmayan uygulama container'ının kendisi anlamına gelirdi. Tam olarak bu nedenle "İsimle Container'dan Container'a İletişim", `localhost` yerine *diğer container'ın adını*, `learning-platform-db`'yi kullanır.

Hepsini Bir Araya Getirmek

Tam akış -- bir network oluşturmak, bu projenin kendi veritabanı ve uygulama container'larını üzerinde çalıştırmak, ve uygulamanın veritabanına gerçekten isim üzerinden ulaştığını doğrulamak:

```
#!/bin/sh
# Running this project's own app and its PostgreSQL database as two separate
# containers that reach each other by name, instead of the host.docker.internal
# workaround "Dockerizing a Spring Boot Application" used as a stand-in.

docker network create learning-platform-net
# Output:
# 4a3b2c1d0e9f8a7b6c5d4e3f2a1b0c9d8e7f6a5b4c3d2e1f0a9b8c7d6e5f4a3b

docker run --name learning-platform-db \
  --network learning-platform-net \
  -e POSTGRES_PASSWORD=secret \
  -d postgres:16
# Output:
# 7f8e9a0b1c2d3e4f5a6b7c8d9e0f1a2b3c4d5e6f7a8b9c0d1e2f3a4b5c6d7e8f

docker run --name learning-platform-app \
  --network learning-platform-net \
  -p 8080:8080 \
  -e SPRING_DATASOURCE_URL=jdbc:postgresql://learning-platform-db:5432/postgres \
  -e SPRING_DATASOURCE_PASSWORD=secret \
  -d learning-platform:0.1.0
# Output:
# 9b8c7d6e5f4a3b2c1d0e9f8a7b6c5d4e3f2a1b0c9d8e7f6a5b4c3d2e1f0a9b8c

# Note the datasource URL: "learning-platform-db" -- the OTHER container's
# --name -- resolves automatically over this user-defined network, no IP
# address and no host.docker.internal needed. Confirm it actually worked:

docker logs learning-platform-app
# Output (trimmed):
# Started LearningPlatformApplication in 3.4 seconds

curl http://localhost:8080/en
# Output (trimmed):
# <!doctype html> ... the homepage renders, meaning the app reached the database ...

docker network inspect learning-platform-net --format '{{range .Containers}}{{.Name}}
{{end}}'
# Output:
# learning-platform-db learning-platform-app

# Clean up.
docker stop learning-platform-app learning-platform-db
docker rm learning-platform-app learning-platform-db
docker network rm learning-platform-net
```

Buradaki her şey bu dersin işlediklerinin doğrudan bir uygulaması: `docker network create` ağı oluşturur, `--network` her iki container'ı da ona bağlar, ve JDBC URL'si veritabanı container'ına IP ya da `localhost` yerine `--name`'i üzerinden ulaşır -- bu kategorideki bir sonraki ders "Docker Compose"un, bu ağ-oluşturma adımının hiç elle yazılmasına gerek kalmasın diye otomatikleştirdiği tam olarak aynı mekanizma.

Yaygın Hatalar

- İki container'ın Docker'ın **varsayılan** bridge network'ünde birbirine isimle ulaşmasını beklemek -- ulaşamazlar; bu yalnızca bir user-defined network'te çalışır (bkz. "Varsayılan Bridge Network'ü").
- *Farklı* bir container'a ulaşmak için bir container'ın kendi konfigürasyonunda `localhost` kullanmak -- bir container içinde `localhost` her zaman o container'ın kendisi anlamına gelir, hiçbir zaman başka birinin değil (bkz. "`localhost` Bir Container İçinde Neden Farklı Bir Şey İfade Eder").
- Birbirine konuşması amaçlanan iki container'dan birinde `--network`'ü unutmak, ve isim çözümlemesi başarısız olduğunda kafası karışmak -- `docker network inspect` (yukarıda), ikisinin de gerçekten aynı ağa bağlı olduğunu doğrulamanın en hızlı yoludur.
- `host.docker.internal`'i (bir container içinden *host makineye* ulaşmak) container'dan container'a isim çözümlemesiyle (*başka bir container'a* ulaşmak) karıştırmak -- farklı problemleri çözerler ve birbirinin yerine geçmezler.

Best Practices

- Birbiriyle konuşması gereken birden fazla container içeren her kurulum için her zaman bir user-defined network oluştur ve kullan -- bunun için asla varsayılan bridge network'e güvenme.
- Bağlantı string'lerinde ve konfigürasyonda, her yeniden başlatmada değişen sabit kodlanmış bir IP adresi yerine diğer container'ın `--name`'ini hostname olarak kullan.
- Birbiriyle iletişim kurması gereken container'lar birbirine ulaşamadığında ilk sorun giderme adımı olarak `docker network inspect`'i kullan.
- `-p`'yi (host-to-container) ve `--network`'ü (container'dan container'a) zihinsel olarak ayrı tut -- farklı problemleri çözerler ve hiçbiri diğerinin yerine geçmez.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Her container bir ağa bağlanır -- varsayılan olarak, başka container'lara isimle ulaşmayı **desteklemeyen** Docker'ın kendi varsayılan bridge network'ü.
- `-p <host>:<container>`, host makineyi bir container'ın portuna köprüler; iki container'ın birbirine nasıl ulaştığıyla hiçbir ilgisi yoktur.
- Varsayılandan farklı olarak, bir user-defined network (`docker network create`), bağlı her container'a diğer her bağlı container'ın `--name`'inin otomatik DNS çözümlemesini verir.
- Bir container içinde, `localhost` her zaman o container'ın kendi loopback arayüzüne başvurur -- hiçbir zaman host makinenin, ve hiçbir zaman başka bir container'ın değil.

- `host.docker.internal` (bir container içinden host'a ulaşmak) ve bir user-defined network'te bir container'ın `--name`'i (başka bir container'a ulaşmak) iki farklı problemi çözer.

Cheat Sheet

```
docker network create <isim>                 # bir user-defined bridge network oluştur
docker run --network <isim> ...              # bir container'ı ona bağla
docker network ls                            # tüm ağları listele
docker network inspect <isim>                # hangi container'ların bağlı olduğunu göster
```

```
jdbc:postgresql://<diğer-container-adı>:5432/...   # başka bir container'a isimle ulaş
                                                       # (yalnızca paylaşılan bir user-
defined network'te çalışır)
```

Terimler Sözlüğü

- **Varsayılan bridge network**: aksi söylenmediği sürece her container'ın katıldığı ağ; başka container'lara isimle ulaşmayı desteklemez.
- **User-defined network**: `docker network create` ile oluşturulan, bağlı container'lar arasında otomatik DNS-tabanlı isim çözümlemesi sağlayan adlandırılmış bir ağ.
- **Port yayınlama (`-p`)**: bir host makine portunu bir container'ın portuna eşlemek -- container'dan container'a iletişimle ilgisi olmayan bir host-to-container köprüsü.
- **`host.docker.internal`**: bir container'ın host makinede çalışan bir servise ulaşmak için kullanabileceği özel bir hostname -- başka bir container'a ulaşmak için değil.