Docker Nedir?

Docker'ın ne olduğu, container'ların sanal makinelerden nasıl farklı olduğu, image ile container arasındaki fark, Docker Engine, ve Docker Hub -- Docker kursunun ve Docker Fundamentals kategorisinin 1.'si.

Başlangıç 15 dk

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

---

Bu, yeni Docker kursunun ilk dersi, ve "Docker Temelleri" kategorisinin ilk dersi. Java'yı ve Spring Boot'u zaten biliyorsun, ve bu platformun kendi PostgreSQL kursu, ilişkisel bir veritabanının veriyi nasıl sakladığını ve sunduğunu zaten işledi. Hiçbiri henüz çalışan uygulamanın kendisinin -- kendi bağımlılıkları, kendi JDK sürümü, belirli bir ağ üzerinden belirli bir veritabanıyla konuşan gerçek JVM sürecinin -- "bir geliştiricinin makinesindeki kod" dan "başka her yerde aynı şekilde çalışan"a nasıl gittiğini açıklamadı. Docker'ın doldurduğu boşluk tam olarak bu, ve bu ders, bu kursun geri kalanının dayandığı zihinsel modeli inşa ediyor: bir container gerçekte nedir, bir image gerçekte nedir, ve bu problem neden en baştan çözülmesi gereken bir problemdi.

Docker Nedir?

Docker, bir uygulamayı çalışması için ihtiyaç duyduğu her şeyle -- bağımlılıkları, runtime'ı, konfigürasyonu -- birlikte, **container** adı verilen tek, taşınabilir bir birime paketleyen, ve sonra bu birimi Docker'ın kurulu olduğu herhangi bir makinede tutarlı bir şekilde çalıştıran bir platform. Bir geliştiricinin dizüstü bilgisayarında derlenmiş bir Spring Boot uygulaması ile tam olarak aynı uygulamanın bir production sunucusunda çalışan kopyası, Docker olmadan, yalnızca birbirine BENZEMEYE çalışan iki farklı ortamdır: farklı JDK patch sürümleri, farklı kurulu kütüphaneler, farklı işletim sistemleri. Docker ile, bunlar tam olarak *aynı* ortam olabilir -- bir kez paketlenip her yerde değişmeden çalışan.

Neden Var?

"Benim makinemde çalışıyor," Docker'ın çözmek için var olduğu belirli, tekrar eden problemdir. Container'lar yaygınlaşmadan önce, bir Java uygulamasını deploy etmek, hedef makinenin zaten doğru JDK sürümüne sahip olmasına, doğru ortam değişkenlerinin ayarlanmış olmasına, ve zaten başka bir bağımlılığın çakışan bir sürümünün mevcut olmamasına güvenmek anlamına geliyordu -- ve bir geliştiricinin makinesi, bir test sunucusu, ve production arasındaki herhangi bir uyumsuzluk, yalnızca bu üç yerden birinde

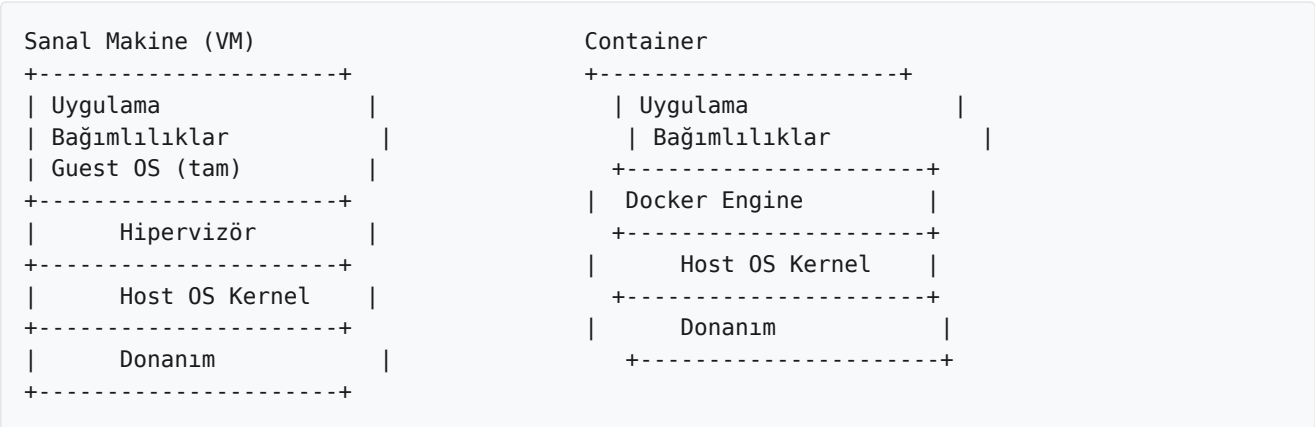
tekrarlanan bir hataya dönüşebiliyordu. Docker'ın cevabı, hedef makinenin kendi kurulu yazılımına hiç güvenmemek: uygulama, tüm çalışma zamanı ortamıyla birlikte, tek bir birim olarak gönderilir, böylece her makinenin ortak olması gereken tek şey Docker'ın kendisi olur.

Tarihçe

Genel bir fikir olarak container'lar Docker'dan onlarca yıl önceye uzanır -- Unix `chroot` (1979) ve daha sonra 2000'lerin ortasına kadar Linux çekirdeğine eklenen Linux namespace'leri ve cgroup'lar, bir sürecin sistemin geri kalanından izole edilmesine zaten izin veriyordu. Docker, Inc. (başlangıçta dotCloud adlı bir şirket), Docker Engine'i 2013'te açık kaynak bir proje olarak yayınladı, ve gerçek katkısı container izolasyonunu sıfırdan icat etmek değildi -- zaten var olan bu Linux çekirdek yeteneğini basit bir komut satırı aracı, standart bir image formatı, ve bu image'ları paylaşmak için halka açık bir yer olan Docker Hub aracılığıyla kullanılabilir kılmaktı. Container'ları uzman bir Linux tekniğinden, Java ve Spring Boot uygulamaları dahil, yazılımın nasıl inşa edilip gönderildiğinin varsayılan bir parçasına taşıyan şey bu kombinasyondu.

Container'lar vs. Sanal Makineler

Bir sanal makine (virtual machine) ve bir container, kulağa benzer gelen bir problemi -- bir uygulamanın ortamını başka bir uygulamanınkinden izole etmeyi -- temelde farklı şekillerde çözer, ve bu fark yan yana görülmeye değer.



Bir sanal makine, bir hipervizör -- donanımı emüle eden yazılımın kendisi -- üzerinde tam bir guest işletim sistemi çalıştırır: kendi çekirdeği, her şeyin kendi kopyası. Bir container, "Tarihçe"de bahsedilen aynı Linux namespace'leri ve cgroup'larla diğer container'lardan izole edilerek, host makinenin var olan çekirdeği üzerinde doğrudan çalışır -- altında ikinci bir işletim sistemi yoktur. Pratik sonuç boyut ve hızdır: bir VM image'ı tipik olarak gigabaytlarca büyüklüktedir ve tam bir OS'u boot etmek dakikalar mertebesinde sürer; bir container image'ı onlarca megabayt olabilir ve boot edilecek bir OS olmadığı için, kabaca uygulama sürecinin kendisinin başlaması kadar sürede başlar -- yalnızca uygulama süreci, doğrudan başlayarak.

Image'lar vs. Container'lar

Bu iki kelime Docker'da kesin olarak kullanılır, ve ikisini karıştırmak en yaygın erken hatalardan biridir (bkz. "Yaygın Hatalar"). Bir **image**, salt okunur, paketlenmiş bir şablondur -- uygulama kodu, bir runtime, kurulu bağımlılıklar, ve konfigürasyon, tek bir artifact'e dondurulmuş. Bir **container**, bir image'DAN oluşturulan, çalışan (ya da durmuş) bir örnektir -- bir Java sınıfının kendi nesneleriyle olan ilişkisiyle tam olarak aynı: bir `Book` sınıfı, çok sayıda `Book` nesnesi, her biri kendi durumuna sahip, hepsi aynı şablondan inşa edilmiş. Bir Spring Boot image'ı aynı anda birden fazla çalışan container'ın kaynağı olabilir -- aynı image'dan başlatılmış üç container, her biri bağımsız, izole bir süreç, hiçbiri diğerleriyle çalışma zamanı durumunu paylaşmıyor.

```
image: my-app:1.0            (şablon -- bir kez inşa edilir)
     |
     +--> container A  (çalışıyor)
     +--> container B  (çalışıyor)
     +--> container C  (durmuş)
```

Bir image'dan bir container başlatmak o image'ı tüketmez ya da değiştirmez -- aynı image, bağımsız olarak herhangi sayıda container başlatmak için kullanılabilir, ve image'ın kendisi tam olarak inşa edildiği gibi kalır. Bu kategorideki sonraki dersler, bir image'ın tam olarak nasıl inşa edildiğini ("Docker İmajları ve Dockerfile") ve bir container'ın komut satırından tam olarak nasıl başlatıldığını, incelendiğini, ve durdurulduğunu ("Docker CLI Temelleri") işliyor.

Docker Engine

Docker Engine, gerçek işi yapan arka plan servisidir (geleneksel olarak `dockerd` adı verilen bir daemon): image'lar inşa etmek, container'ları başlatıp durdurmak, her container'ın aldığı izole ağı ve depolamayı yönetmek. Bir geliştiricinin komut satırında yazdığı her şey -- `docker run`, `docker build`, ve gerisi, "Docker CLI Temelleri"nde tam olarak işleniyor -- bu daemon'la konuşan ve ondan bir şey yapmasını isteyen bir client komutudur; CLI'ın kendisi container'ları doğrudan çalıştırmaz, işi gerçekten yapan engine'e ince bir client'tır.

Docker Hub ve Image Registry'leri

Bir image, bir kez inşa edildikten sonra, farklı bir makinede indirilip çalıştırılabilmesi için bir yerde yaşamalıdır -- bir **image registry**'nin ne olduğu tam olarak budur, ve **Docker Hub**, Docker'ın kendisinin kutudan çıktığı gibi işaret ettiği varsayılan, halka açık olanıdır. Bir registry, image'ları bir isim ve tag altında saklar (`postgres:16`, `eclipse-temurin:21-jre`) -- Maven Central gibi bir paket deposunun JAR artifact'lerini bir group, artifact, ve versiyon altında sakladığı aynı şekilde -- ve bu projenin kendi `pom.xml`'inin bağımlılıkları elle göndermek yerine koordinat üzerinden Maven Central'dan çektiği gibi, bir `docker pull` da her seferinde sıfırdan yerel olarak inşa etmek yerine bir image'ı isim ve tag üzerinden bir registry'den getirir. Bir takım, Docker Hub yerine ya da onunla birlikte kendi özel registry'sini de çalıştırabilir -- mekanizma her iki durumda da aynıdır, yalnızca konum değişir.

Docker Bu Proje Etrafında Nereye Oturuyor

Şu anda, bu projeyi yerel olarak çalıştırmak, kurulu bir JDK'ya sahip olmak, Maven'in bağımlılıkları çözmesini sağlamak, ve zaten çalışan ve erişilebilir bir PostgreSQL sunucusuna sahip olmak anlamına geliyor -- yeni bir geliştiricinin, `mvn spring-boot:run` (ya da bu sandbox'ın kendisinin ihtiyaç duyduğu geçici çözümler, bkz. `docs/known-constraints.md`) başarıyla başlamadan önce, kendi makinesinde bağımsız olarak doğru yapması gereken üç ayrı kurulum parçası. Docker yüzünden bu projenin kendi kodunda hiçbir şey değişmiyor -- Maven'in zaten ürettiği JAR, hâlâ çalışan JAR'dır. Docker'ın eklediği şey, o JAR'ı uyumlu bir JDK ile birlikte tek bir image'a paketlemenin bir yolu ("Bir Spring Boot Uygulamasını Docker'a Taşımak", bu kategoride ileride), ve "bu uygulama artı bir PostgreSQL örneği, ağ üzerinden birbirine bağlı" durumunu tek, tekrarlanabilir bir birim olarak tarif etmenin bir yolu ("Docker Compose", bir sonraki kategoride) -- böylece bu projeyi çalıştırmak, üzerinde ne kurulu olduğuna bağlı olmak yerine, Docker'ın kurulu olduğu herhangi bir makinede aynı görünür.

Best Practices

- "Image" ve "container"ı en baştan kesin olarak ayrı kavramlar olarak tut (bkz. "Image'lar vs. Container'lar") -- bu kurstaki neredeyse her sonraki konu, bu ayrımın zaten sağlam olduğunu varsayıyor.
- Docker'ı bir sanal makineyle karşılaştırırken, önce paylaşılan-çekirdek gerçeğine başvur (bkz. "Container'lar vs. Sanal Makineler") -- hem boyut hem başlangıç süresi farkını açıklayan tek mimari fark bu.
- Docker Hub'ı, bu projenin kendi `pom.xml`'inin Maven Central'ı zaten ele aldığı gibi düşün -- yerel olarak yeniden icat edilecek bir şey değil, paylaşılan, isimlendirilmiş, önceden inşa edilmiş artifact'ler kaynağı.
- Container'ları gerçekte çalıştıran şeyin `docker` komutunun kendisi değil, Docker Engine olduğunu hatırla -- CLI, bir arka plan servisiyle konuşan bir client'tır.

Yaygın Hatalar

- "Image" ve "container"ı birbirinin yerine kullanmak -- bir image donmuş şablondur; bir container ondan oluşturulan, çalışan (ya da durmuş) tek bir örnektir, ve tek bir image aynı anda birçok container'ın kaynağı olabilir.
- Bir container'ın "hafif bir sanal makine" olduğunu varsaymak -- hiç VM değildir; bir hipervizör altında kendi guest OS'unu çalıştırmak yerine, host'un çekirdeğini doğrudan paylaşır.
- Docker'ın bu projenin Java kodunun ne yaptığını değiştirmesini beklemek -- değiştirmez; aynı JAR her iki durumda da çalışır, Docker yalnızca etrafındaki ortamın ne kadar tutarlı paketlenip gönderildiğini değiştirir.

- Docker Hub'ın tek olası registry olduğunu varsaymak -- o varsayılandır, ama özel ve kendi barındırılan registry'ler de aynı pull/push mekanizmasıyla çalışır.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Docker, bir uygulamayı tüm çalışma zamanı ortamıyla birlikte bir container'a paketler, böylece Docker'ın kurulu olduğu herhangi bir makinede aynı şekilde çalışır -- "benim makinemde çalışıyor" problemini doğrudan çözer.
- Bir container, tam bir guest işletim sistemi çalıştırmak yerine host makinenin çekirdeğini paylaşır -- container'ların sanal makinelerden çarpıcı biçimde daha küçük ve daha hızlı başlamasının nedeni budur.
- Bir image, salt okunur, bir kez inşa edilmiş bir şablondur; bir container, o image'dan oluşturulan, çalışan (ya da durmuş) bir örnektir -- tek bir image birçok bağımsız container'ın kaynağı olabilir.
- Docker Engine (`dockerd`), image'ları gerçekten inşa eden ve container'ları çalıştıran arka plan servisidir; `docker` komut satırı onunla konuşan bir client'tır.
- Docker Hub halka açık bir image registry'sidir -- Maven Central'ın bu projenin kendi JAR bağımlılıkları için oynadığı aynı rol, ama container image'ları için.

Cheat Sheet

```
image        salt okunur şablon, bir kez inşa edilir   (örn. my-app:1.0)
container    bir image'ın çalışan/durmuş bir örneği
dockerd      gerçek işi yapan Docker Engine daemon'ı
docker CLI   komutları yazdığın client; dockerd ile konuşur
registry     image'ların yaşadığı ve çekildiği yer (Docker Hub = varsayılan)
```

Terimler Sözlüğü

- **Container**: bir image'dan oluşturulan, host makinenin çekirdeğini doğrudan paylaşan, izole, çalışan (ya da durmuş) bir örnek.
- **Image**: container'ların oluşturulduğu, salt okunur, paketlenmiş bir şablon -- kod, runtime, bağımlılıklar, ve konfigürasyon.
- **Docker Engine (`dockerd`)**: image'ları inşa eden ve container'ları çalıştıran arka plan daemon'ı; `docker` komutunun arkasındaki gerçek işçi.
- **Image registry**: image'ları isim ve tag üzerinden saklayan ve sunan bir servis; Docker Hub varsayılan, halka açık olanıdır.
- **Docker Hub**: Docker'ın kendi halka açık image registry'si, `docker pull`'un varsayılan olarak ulaştığı kaynak.