# Docker - Einführung in die Welt der Container - mfgDocker Dökümantasyonu

# Docker Nedir?

Docker en net tanımlamayla open source bir ‘container’ teknolojisidir. Docker, aynı işletim sistemi üzerinde, yüzlerce hatta binlerce birbirinden izole ve bağımsız containerlar sayesinde sanallaştırma sağlayan bir teknolojidir. Web uygulamalarımızın kolayca kurulumunu, testini, çalışmasını ve deploymentını sağlar. Bunun yanında sunucu maliyetlerini önemli ölçüde azaltır.

## Container Nedir:

Docker Daemon tarafından Linux çekirdeği içerisinde birbirinden izole olarak çalıştırılan process’lerin her birine verilen isimdir. Virtual Machine (Sanal Makina) analojisinde Docker’ı Hypervisor’e benzetirsek fiziksel sunucu üzerinde halihazırda koşturulmakta olan her bir işletim sisteminin (sanal sunucunun) Docker’daki karşılığı Container’dır.

## Image Nedir:

Containerlar layer halindeki Image’lardan oluşur. Docker Image ise containerlara kurulacak ve run edilecek olan uygulamaların veya OS’lerin image dosyalarıdır. Örnek verecek olursak mysql, mongodb, redis, ubuntu, mariadb.. Yüzlercesi mevcut. Buyrun: [Docker Images List](https://hub.docker.com/search?q=&type=image" \t "_blank).

Fakat burada şöyle bir ayrıntı var. Aslında container içerisindeki imagelerin içeriklerinin ve tanımlamalarının gerekli tutulduğu bir dosya var. Bu dosyamızın adı: **Dockerfile**. Tamamen bu isimde, ne büyük ne de küçük harflerden oluşan, bir bakıma containerlar içindeki imagelerın registration’ını yapan bir dosya. Burda önemli nokta şu: Her bir image Dockerfile dediğimiz bu dosyanın altında tanımlanması zorunlu.

## ****Docker Registry Nedir:****

Gelelim bence işin en zevkli kısmına. Tıpkı github gibi geliştiriciler açık kaynaklı olarak docker imagelerini yükleyerek ve [DockerHub](https://hub.docker.com/" \t "_blank)’ta paylaşarak imagelerin bizim de indirip kullanmamıza olanak sağlıyorlar. Kısaca imagelar Docker Registrylerde tutuluyor. Örneğin siz postgres image’ını kullanmak istiyorsunuz; [postgres image](https://hub.docker.com/_/postgres" \t "_blank) linkinden

*docker pull postgres*

komutu ile indirip artık bu image ile container oluşturabiliyorsunuz. Aslında github’a çok benziyor. Siz de kendiniz imageleri oluşturup yükleyip başkalarının da sizin yarattığınız imageları kullanmalarını sağlayabiliyorsunuz. İsterseniz Private olarak ta registerynizi tutabilirsiniz. Docker bu hizmeti de size sunuyor.

## Docker CLI Nedir:

Kullanıcının Docker Daemon ile konuşmasını sağlayan, docker komutlarının çalıştırıldığı CLI ekranıdır.

## Integrated Terminal for Running Containers, Extended Integration with Containerd, and More in Docker Desktop 4.12 | Docker

docker cli screen

## Docker Compose Nedir:

Compose, birden fazla containere sahip docker uygulamalarını tanımlamak ve çalıştırmak için kullanılır. Compose ile uygulamanızın servislerini configure etmek için bir YAML dosyası kullanılır. Ardından, tek bir komutla configure ettiğiniz ayarlar ile tüm servislerinizi oluşturup ve başlatabilirsiniz.

Compose tüm ortamlarda çalışır: production, staging, development, testing ve diğer CI iş akışları da dahil olmak üzere.

Compose kullanmak temel olarak üç adımlı bir işlemdir:

1. Uygulamanızın ortamını Dockerfile dosyası ile herhangi bir yerde yeniden üretilebilecek şekilde tanımlayın .
2. docker-compose.yml uygulamalarınızı izole ortamda beraber çalışacak şekilde yaml dosyası içinde set edebilirsiniz.
3. docker-compose up –build ya da docker-compose build && docker-compose up  komutuyla birlikte tüm uygulamanızı başlatır ve çalıştırabilirsiniz.

## Docker Daemon Nedir:

## Docker Daemon, Docker'ın arka planında çalışan bir servistir. Docker CLI gibi client'ların gönderdiği istekleri alır ve bunları işleyerek Docker container'larını, image'larını, ağlarını ve diğer bileşenlerini yönetir. Özetle, Docker Daemon şunlardan sorumludur:

## Container'ların oluşturulması, çalıştırılması ve durdurulması.

## Image'ların oluşturulması, saklanması ve transfer edilmesi.

• Ağ yapılandırmaları ve volume yönetimi.

## Docker Volume'ların Özellikleri ve İşlevleri:

## Kalıcılık (Persistence):

## Docker volume'lar, verilerin container'lar arası ve container yeniden başlatmalarında bile korunmasını sağlar.

## Bir volume, bir veya birden fazla container tarafından kullanılabilir ve container'lar silinse bile volume üzerindeki veriler korunur.

## Veri Paylaşımı ve Yedekleme:

## Volume'lar, birden fazla container arasında veri paylaşımını kolaylaştırır.

## Verileri yedeklemek ve gerektiğinde geri yüklemek için kullanılabilirler.

## Bağımsızlık (Independence):

## Volume'lar, container'ların yaşam döngüsünden bağımsızdır. Bir container silindiğinde, volume üzerindeki veriler etkilenmez.

## Bu, verilerin container'lardan bağımsız olarak yönetilmesine olanak tanır.

## Performans ve Güvenlik:

## Bazı durumlarda, volume'lar daha iyi disk performansı ve güvenliği sağlayabilir.

## Özellikle büyük veri setleri veya hassas veriler için tercih edilebilir.

Docker Volume Kullanımı:

Volume Oluşturma:



Container'a Volume Bağlama:



Bu, my\_volume adındaki Docker volume'unu container içindeki /path/in/container yoluna bağlar.

Veri Saklama ve Erişim:

Container içinde /path/in/container yoluna yazılan tüm veriler, Docker volume'da saklanır ve **container silinse bile korunur**.

Docker volume'lar, uygulamaların veri kalıcılığı ihtiyaçlarını karşılamak için etkili bir çözüm sunar. Özellikle **veritabanları**, dosya depolama sistemleri ve uygulama durumları gibi durumlu (stateful) verilerin yönetimi için kullanılır. Docker volume'lar, Docker'ın izole ve taşınabilir container yapılarını korurken, **uygulama verilerini güvenli ve kalıcı bir şekilde saklamayı sağlar**.

## Containerler Arası Haberleşme

Docker container'ları arasında veri akışı ve haberleşme, Docker'ın sunduğu ağ özellikleri ile mümkündür. Bu bölümde, farklı Docker container'larının birbiriyle nasıl haberleşebileceği ve ağ ayarlarının nasıl yapılandırılacağı üzerine detaylı bilgiler sunulacaktır.

a. Bridge Driver

Kullanımı: Docker'ın varsayılan ağ sürücüsüdür.

Amaç: Aynı Docker host üzerindeki container'ların birbirleriyle iletişim kurmasını sağlar.

Özellikleri: Her container, bir bridge ağa bağlanır ve bu ağ üzerinde kendi iç IP adresine sahiptir. Container'lar, bu ağ üzerinden birbirleriyle iletişim kurabilirler. Dış dünyaya bağlantı, NAT (Network Address Translation) üzerinden yapılır.

Senaryolar: Aynı host üzerinde çalışan, birbirleriyle iletişim kurması gereken container'lar için uygundur.

b. Host Driver

Kullanımı: Host ağ sürücüsü.

Amaç: Container'ın ağını direkt olarak host makinenin ağına bağlar.

Özellikleri: Bu modda çalışan container'lar, host makinenin ağ arabirimini ve IP adresini kullanır. Bu, NAT işlemine gerek kalmadan container'ların dış ağa doğrudan erişmesini sağlar.

Senaryolar: Yüksek performans gerektiren ve host makine ağ kaynaklarına doğrudan erişmesi gereken uygulamalar için uygundur.

c. Overlay Driver

Kullanımı: Çoklu host ağ sürücüsü.

Amaç: Farklı Docker hostları üzerinde çalışan container'ların birbirleriyle iletişim kurmasını sağlar.

Özellikleri: Overlay ağlar, farklı hostlardaki container'ların sanki aynı ağ üzerindeymiş gibi iletişim kurmalarını sağlar. Bu, Docker Swarm gibi cluster yönetim araçları ile kullanılır.

Senaryolar: Çoklu host ortamlarında, container'ların birbirleriyle iletişim kurması gereken durumlar için uygundur.

d. None Driver

Kullanımı: İzole ağ sürücüsü.

Amaç: Herhangi bir ağa bağlanmayan izole container'lar oluşturur.

Özellikleri: none ağ sürücüsü kullanıldığında, container hiçbir ağa bağlı değildir ve dış dünya ile iletişim kuramaz.

Senaryolar: Ağ izolasyonu gerektiren güvenlik odaklı uygulamalar için uygundur.

e. Macvlan Driver

Kullanımı: Macvlan ağ sürücüsü.

Amaç: Container'lara fiziksel ağa bağlıymış gibi görünecek şekilde bağımsız MAC adresleri atar.

Özellikleri: Container'lar, ağ üzerindeki diğer cihazlar gibi doğrudan erişilebilir olur ve kendi benzersiz MAC adreslerine sahip olurlar.

Senaryolar: Legacy uygulamaların entegrasyonu veya ağ yönetimi gerektiren özel durumlar için uygundur.

Her ağ sürücüsünün kullanımı, uygulamanızın gereksinimlerine ve çalışma ortamınıza bağlı olarak değişiklik gösterir. Özellikle üretim ortamlarında, ağ güvenliği ve performansı göz önünde bulundurularak doğru ağ sürücüsünün seçilmesi önemlidir.

**Bridge(default)** ile birbirine bağlanmış bir Docker-compose örnek dosyası aşağıdaki gibidir :

|  |
| --- |
| version: '3.8'  services:  app1:  build: ./app1  ports:  - "5000:5000"  networks:  - my\_network  app2:  build: ./app2  ports:  - "5001:5001"  networks:  - my\_network  networks:  my\_network:  driver: bridge |

Docker’ın Avantajları Nelerdir?

Docker saniyeler içerisinde başlar, çünkü içerisinde barındırdığı her bir container sadece birer processtir. Böylece karşımıza lightweight bir yapı karşımıza çıkar. Bu da bizi sanal makinelerin hantallığından kurtarmış oluyor aslında :)

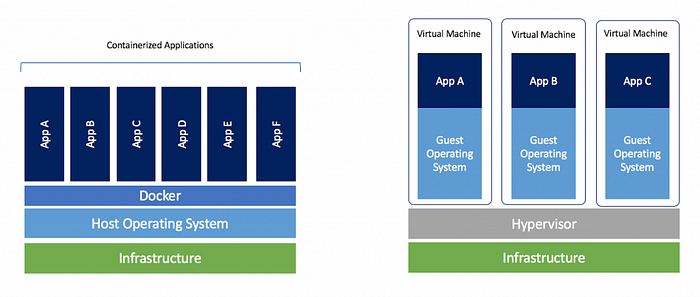
Daha hızlı deployment süreci: İşte bence en önemli avantajı diyebilirim docker için. Dockerı kullanmak için yeni bir environment kurmaya gerek yoktur. Farklı sunucularda çalışmak isteyen developerlar sadece docker imageleri indirip o sunucuda imageleri çalıştırmaları yeterlidir. Böylece ‘benim makinemde çalışıyordu, sunucuda neden çalışmıyor!’ gibi sorunlardan da kurtulmuş oluruz :)

Daha Kolay Yönetim ve Ölçeklendirme: Bir sanal makineye göre docker üzerindeki containerleri çok daha kolay bir şekilde çalıştırabiliriz veya istediğimiz zaman yok edebiliriz. Containerleri manage etmek için farklı toollar mevcut. En çok ta Orchestrator diye nitelendirdiğimiz Kubernetes teknolojisi daha popüler olarak kullanılıyor. Kubernetes, kısaca container kullanan uygulamaların dağıtımını, ölçeklendirmesini ve yönetilmesini otomatik hale getiren açık kaynak kodlu bir sistem. Kubernetes mimarisini ve çalışma sistemini detaylı olarak öğrenmek isterseniz sizin için çok faydalı olacağını düşündüğüm bir linki bırakıyorum: https://www.mshowto.org/kubernetes-nedir.html

Daha İyi Kaynak Kullanımı Sanal makinelere göre tek bir sunucu üzerindeki kaynak tüketimi dockerda çok daha verimlidir. Daha az kaynak tüketimi ile daha fazla containeri çalıştırabiliriz.Deployment Verimliliği: Dockerın en güzel yanlarında birisi de şu: Siz localinizde test ettiniz, uygulamanızı Test veya Live ortamınıza attınız. (veya daha çeşitli ortamlar (dev, staging, pre-prod vs..) Localde çalıştırdığınız her şey burada da aynı şekilde çalışacak. Container ve Imagelerin tutulduğu dockerfile’ları da Git üzerinde tutmanız işinizi daha da kolaylaştıracağından eminim.

Farklı İşletim Sistemlerine Destek Vermesi Docker Windows, Linux, MacOs gibi farklı işletim sistemlerine destek verir.

Popüler Cloud Servislerle Entegre Edilebilir. Docker , AWS, Microsoft Azure, Ansible, Kubernetes, Istio ve daha fazla tool ve cloud hizmetlerle entegre şekilde çalışabilir.

Docker ve Sanal Makine Arasındaki Farklar Nedir?

Docker open source bir sanallaştırma teknolojisidir. Ama bildiğimiz klasik sanal makinelerden (Hypervisor, VMware) biraz farklı olarak sanallaştırma yapar. Bunun nedeni ise sanal makinelerde bulunan hypervisor katmanının bulunmaması ve container dediğimiz birbirinden tamamen bağımsız ve izole processlerden oluşmasıdır. Docker üzerinde host edildiği tek bir OS(İşletim sistemi) üzerinde yüzlerce ve binlerce docker conteiner çalışabilir ve bu conteinerlar sistem dosyalarını paylaşımlı olarak kullandıkları için kaynak tüketimleri oldukça düşük olduğu için maliyetleri düşüktür.

Sanal makine sistemlerinde, her bir sanal makine kendi işletim sistemini kullanılır ve kendi kütüphaneleri vardır. Aslında az çok hepimiz VMware kullanmışızdır. Örneğin Windows makinemizde MacOs işletim sistemini koşturmak istersek VMware ile bunu sağlayabiliyoruz. Ama farkettiyseniz bu sizin için maliyetli olur çünkü kaynak tüketimi fazladır, çünkü tamamen farklı bir işletim sistemini ayağa kaldırırsınız ve de açılış hızı yavaştır; bir süre beklemek zorunda kalırsınız. Fakat Docker teknolojisi, contenierları çalıştımak için üzerinde host edildiği tek bir işletim sistemine bağlıdır, kaynak tüketimi azdır. Ayrıca conteinerlar saniyeler içinde kullanıma hazır hale geliyor, istediğiniz zaten duraklatabiliyor, durdurabiliyor veya yeniden başlatabiliyorsunuz.

|  |
| --- |
| VM (Virtual Machine): |
|  |
| OS : Tam işletim sistemi |
| İzolasyon : Yüksek |
| Çalışır hale gelmesi : Dakikalar |
| Versiyonlama : Yok |
| Kolay paylaşılabilirlik : Düşük |

|  |  |
| --- | --- |
| Docker: |  |
|  |  |
| OS : Küçültülmüş işletim sistemi imajı |  |
| İzolasyon : Daha düşük |  |
| Çalışır hale gelmesi : Saniyeler |  |
| Versiyonlama : Yüksek |  |
| Kolay paylaşılabilirlik : Yüksek |  |

Microservice Mimarisinde Docker Neden Önemli?

Gelelim dockerın microservice mimarisinde kullanım avantajlarına. Dockerın microservicelerde kullanımının en büyük avantajı sunuculardaki maliyetlerin düşürülmesi ve sunucu kaynaklarnın tüketiminin en aza indirilmesi olarak açıklayabiliriz.

Mikroserviceleri küçük uygulamalara benzediğinden, farklı ortamlar sağlamak için mikroserviceleri kendi VM instancelarına dağıtmamız gerekir. Tahmin edebileceğiniz gibi, bir sanal makinenin tamamını uygulamanın yalnızca küçük bir bölümünü dağıtmaya ayırmak en etkili yöntem değildir. Bununla birlikte, Docker ile performans ek yükünü azaltmak ve aynı sunucuya binlerce mikroservice dağıtmak mümkündür.

Çünkü Docker conteinerları sanal makinelere göre çok daha az sistem kaynağı gerektirir..

Docker, uygulamaları geliştirmek ve dağıtmak için diğer tüm yazılım araçlarından daha fazla özellik sunar.

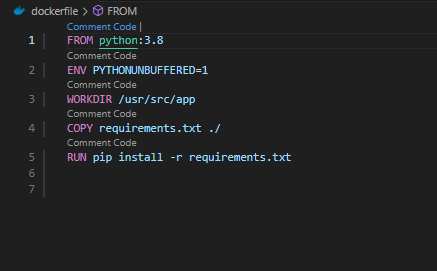
Yazılımın geliştirilmesinde ve deployment esnasında bize büyük kolaylık sağlar.

Projemiz için Dockerfile Oluşturma

Dockerfile Hazırlama:

Projenizin ana dizininde, Django uygulamanız için bir Dockerfile oluşturun. Bu Dockerfile, Django uygulamanızı Docker container'ında nasıl çalıştıracağınızı tanımlar.

Bu Dockerfile, Python 3.8 image'ından başlar, bağımlılıkları yükler ve Django uygulamanızı çalıştırır.

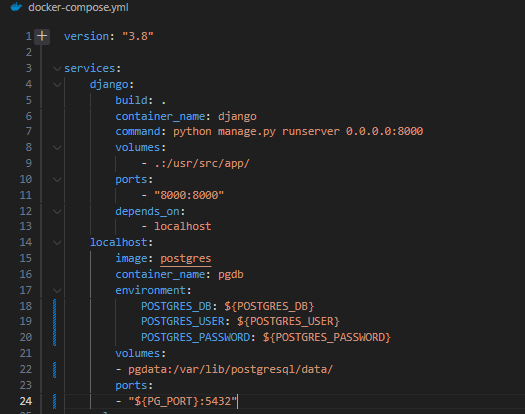


Dockerfile

Docker Compose ile Django Uygulamasını Yapılandırma

docker-compose.yml Dosyası Oluşturma:

Aynı dizinde docker-compose.yml dosyasını oluşturun. Bu dosya, Docker container'ınızı ve gerekirse diğer hizmetleri (veritabanı gibi) nasıl yapılandıracağınızı tanımlar.



Bu yapılandırmada, django ve localhost(postgres) adında servisler tanımlanmıştır. Bu servisler , yukarıda oluşturduğunuz Dockerfile'ı kullanarak build edilir ve 8000 portunu kullanarak dış dünyaya açılır.

Uygulamayı Başlatma

Docker Compose ile Uygulamayı Başlatma:

Docker ve Docker Compose yüklü bir sistemde, aşağıdaki komutu çalıştırarak Django uygulamanızı başlatabilirsiniz:

>> docker-compose up –build ya da

>>docker-compose build && docker-compose up

Bu komut, Docker Compose'a docker-compose.yml dosyasındaki talimatları takip ederek Django uygulamanızın container'ını oluşturur ve çalıştırır.

Uygulamaya Erişim:

Uygulamanız artık http://localhost:8000 adresinde çalışıyor olmalıdır. Tarayıcınızda bu adrese giderek Django uygulamanızı görebilirsiniz.