

İÇİNDEKİLER

Giriş	6
Container Nedir:	7
Dockerfile Nedir? Dockerfile Komutları Nelerdir?	11
Docker Komutları ve Kullanımı	13
İmaj (Image) İşlemleri	13
Container İşlemleri	13
Container Bağlantı ve Kontrol	13
Ağ (Network) İşlemleri	14
Veritabanı ve Web Servisleri	14
MariaDB'ye Erişim:	14
HTTP ve SSH İşlemleri	14
Docker Daemon Nedir?	15
Docker Run vs Docker Start	16
1. docker run vs. docker start	16
2. Neden docker run ubuntu Hemen Sonlanır?	16
3. docker start Neden Arkada Çalıştırır?	16
Docker Image Katman Yapısı	17
1. Docker Image'leri Katmanlıdır	17
2. Katmanlar Nasıl Oluşur?	17
3. Katmanların Avantajları	17
4. Özeti: Pull İşleminin Mantığı	17
Docker Compose Nedir?	18
Temel Özellikleri	18
Temel Komutlar	18
Anahtar Kavramlar	18
Port Nedir ?	19
Port Mapping Nedir?	19
Adım Adım Port Mapping Nasıl Çalışır?	21
Neden Port Mapping Kullanız?	21
Port Mapping Türleri	21
Önemli Noktalar	22
Port Mapping'i Görselleştirme	22
Docker Volume'lar ve Türleri	22
Docker Volume Türleri	22
1. Named Volumes (İsimlendirilmiş Volume'lar)	22
2. Bind Mounts (Bağlanmış)	24
3. tmpfs Mounts (Geçici Bellekte Montajlar)	25
NGNIX	25
Ngnix configasyonu çalışma prensibi işleyişi:	25
Nginx Webserver Nedir ve Ayarları - Reverse Proxy - Özeti	25
Proxy Sunucusu Ve Çalışma Prensibi	25

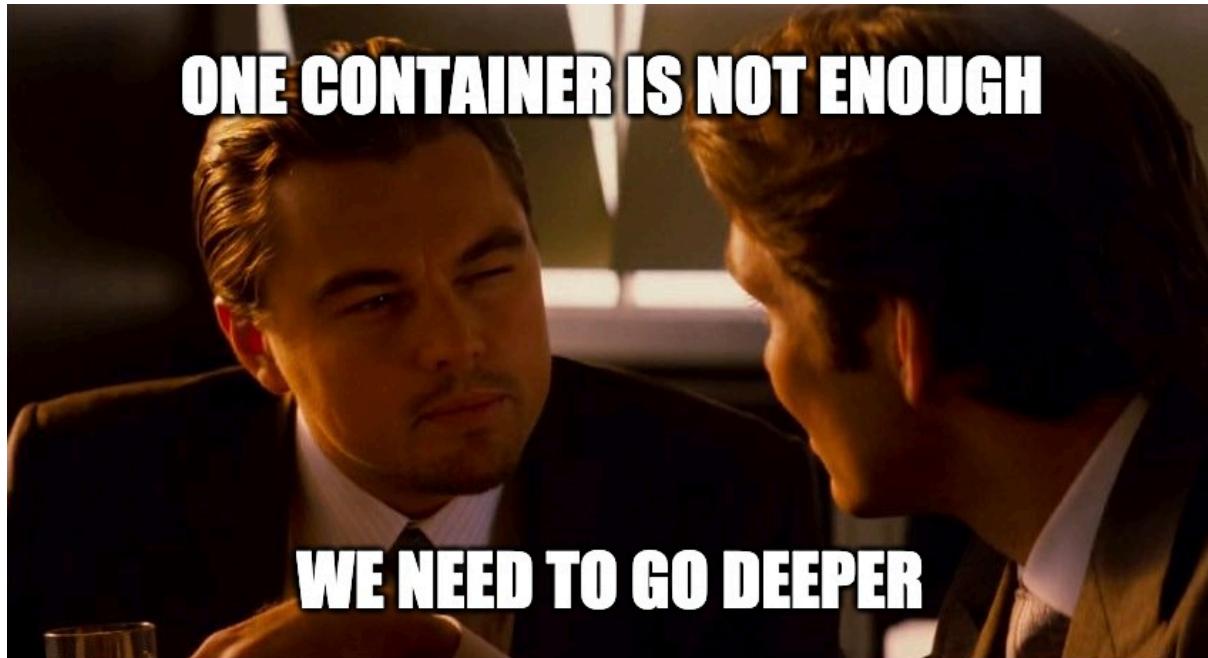
<i>Proxy Sunucusunun Yararları</i>	26
<i>Proxy Türleri</i>	27
<i>Forward Proxy: Herkesin erişebileceği ve kullanabilecegi proxy sunucularıdır. Genellikle anonimlik sağlamak için kullanılır.</i>	27
<i>Private Proxy: Belirli bir kullanıcı grubu için tasarlanmış ve genellikle ücretli olan proxy sunucularıdır. Daha yüksek güvenlik ve performans sunar.</i>	27
<i>Reverse Proxy: Sunucu tarafında çalışan proxy'dir. Kullanıcılarından gelen istekleri alır ve arka plandaki sunuculara yönlendirir. Genellikle yük dengelemesi ve güvenlik amaçlı kullanılır.</i>	28
<i>Nginx'in Event-Driven Mimarisi</i>	29
<i>Nginx Mimarisinin Diğer Web Sunucularıyla Farkları</i>	29
<i>Olay Tabanlı (event-driven) Yapı</i>	30
1. Geleneksel Sunucular Nasıl Çalışır?	31
2. NGINX'in Olay Tabanlı Mimarisi	31
3. NGINX vs. Çoklu İş Parçacıklı Sistemler	31
4. Tek Thread Nasıl Bu Kadar Hızlı Oluyor?	32
5. Peki Çoklu CPU Çekirdeklerinden Nasıl Yararlanıyor?	32
6. Örnek Olarak:	32
7. Özeti: Neden Tek Thread Yeterli?	32
8. "Tek Thread Yavaş Kalır" Yanılığısı	32
<i>MariaDB</i>	33
<i>TLS sertifikası nedir?</i>	33
<i>Neden SSL Artık Güvenli Değil?</i>	33
<i>SSL'in Güvensiz Olmasının Nedenleri:</i>	34
<i>Dockerfile oluşturma:</i>	35
<i>NGINX</i>	36
<i>NGINX DOCKERFILE</i>	36
1. Temel İmaj Seçimi	36
2. Paket Güncellemeye ve Yükleme	36
3. SSL Sertifikaları İçin Dizin Oluşturma	36
4. Özel NGINX Konfigürasyonunu Kopyalama	36
5. Kurulum Scripti Kopyalama	36
6. Kurulum Scriptine Çalıştırılabilir Izin Verme	37
7. NGINX İçin Geçici Dizin Oluşturma	37
8. Container Başlatıldığında Çalışacak Script (ENTRYPOINT Komutu)	37
9. Port Açma (EXPOSE Komutu)	37
10. NGINX'i Çalıştırma (CMD Komutu)	37
<i>ENTRYPOINT ve CMD Arasındaki Fark</i>	38
<i>Dockerfile çalışma yapısı</i>	38
<i>Nginx-installer.sh</i>	40
<i>OpenSSL ile Sertifika Oluşturma (Neden Gereklidir?)</i>	40
<i>SSL Sertifikası ve HTTPS</i>	40
<i>OpenSSL ile Sertifika Oluşturma</i>	41
<i>NGINX KONFIGÜRASYONU</i>	42
<i>NGINX Konfigürasyon Dosyası</i>	42

1. Genel Sunucu Ayarları	42
2. SSL Sertifikası ve Protokol Ayarları	42
3. Web Sayfalarının Konumunu Belirleme	43
4. Ana Sayfa Yönlendirme Ayarları	43
5. NGINX'te FastCGI ile PHP İşleme	44
2. PHP Dosya Yolunu Ayırma (fastcgisplitpath_info)	44
3. PHP-FPM'ye Yönlendirme	45
4. Varsayılan PHP Dosyası (index.php)	45
5. FastCGI Parametrelerini Dahil Etme	45
6. Çalıştırılacak PHP Dosyasını Belirleme	45
7. PHP Betik Adını Tanımlama	46
PHP (Personal Home Page)	46
PHP-FPM (FastCGI (Fast Common Gateway Interface) Process Manager)	46
FastCGI Nedir ve Neden Kullanılır?	46
CGI ile FastCGI Arasındaki Farklar:	46
FastCGI Nasıl Çalışır?	47
WordPress	47
WordPress Dockerfile	48
Gerekli Paketlerin Kurulumu	48
WP-CLI Kurulumu	49
PHP-FPM Yapılandırması	49
Socket Dizini Oluşturma	49
Kurulum Script'ini Hazırlama	49
Container Başlangıç Noktası	50
Çalışma Dizini	50
Port Açma	50
Container Ana Süreci	50
WordPress Kurulum Script'i (WordPressInstaller.sh)	50
Başlangıç Ayarları	50
Veritabanı Bilgilerinin Alınması	51
WordPress Kurulumunun Kontrolü	51
WordPress Dosyalarının Temizlenmesi	51
WordPress'in İndirilmesi ve Çıkarılması	52
WP-CLI Kontrolü ve Kurulumu	52
WordPress Yapılandırması	52
WordPress Kurulumu	53
Site Ayarları	53
Script'in Sonlandırılması	54
WordPress PHP-FPM www.conf Dosyası	54
Kullanıcı ve Grup Ayarları	54
Dinleme (Listen) Ayarı	55
Process Manager Ayarları	55
Process Sayısı Ayarları	55
Socket Kavramı ve İşleyişi	57

<i>Socket nedir UNIX socket dosyası:</i>	57
<i>Socket nedir ? Ne İşe Yarar ?</i>	57
<i>Socketlerin Temel Mantığı</i>	57
<i>Socket İletişiminin Adımları</i>	57
<i>Socket Adresleme</i>	57
<i>Detaylı TCP Socket İletişim Örneği</i>	58
UNIX Domain Socketleri	58
<i>UNIX Socketlerinin Çalışma Prensibi</i>	59
<i>UNIX Socketlerinin Kullanım Alanları</i>	59
<i>UNIX Socketlerinin Avantajları</i>	59
<i>TCP/IP Socketleri ve UNIX Socketleri Arasındaki Farklar</i>	60
<i>UNIX socketleri kopyala-yaz ekanızmasıyla çalışır. Peki nedir bu mekanizma ?</i>	60
Kopyala-Yaz Mekanizması ve UNIX Socketlerindeki Rolü	60
<i>Bellek Yönetimi ve Veri Paylaşımı</i>	61
<i>Kopyala-Yaz Mekanizmasının Çalışma Prensibi</i>	61
<i>UNIX Socketlerinde Kopyala-Yaz Mekanizması</i>	61
<i>Performans ve Verimlilik Avantajları</i>	61
MariaDB Veritabanı	62
<i>Dockerfile</i>	62
<i>MariaDB Kurulumu</i>	62
<i>Port Açma</i>	62
<i>Yapıllandırma Dosyasının Kopyalanması</i>	63
<i>Başlangıç Script'inin Kopyalanması ve Yetkilendirilmesi</i>	63
<i>Container Başlatıcısı</i>	63
<i>MariaDB Script</i>	64
<i>Güvenli Şekilde Şifrelerin Alınması</i>	64
<i>MariaDB Servisinin Başlatılması</i>	64
<i>Servisin Hazır Olmasının Beklenmesi</i>	65
<i>Veritabanı ve Kullanıcı Oluşturma</i>	65
<i>Geçici Servisin Durdurulması</i>	67
<i>Ana MariaDB Sürecinin Başlatılması</i>	67
<i>MariaDB 50-server.cnf Dosyası</i>	68
<i>Kod Kısmı</i>	68
<i>Bölüm Tanımlayıcısı</i>	68
<i>Kullanıcı ve Sistem Dosyaları</i>	68
<i>Dil ve Karakter Seti Ayarları</i>	69
<i>Güvenlik Ayarları</i>	70
<i>Günlük Dosyaları ve Kayıt Tutma</i>	70
<i>Karakter Seti Ayarları</i>	71
Docker Compose Dosyası	71
<i>Versiyon Tanımı</i>	71
<i>NGINX Servisi</i>	72
<i>MariaDB Servisi</i>	74

WordPress Servisi	75
Secrets Tanımları	76
Networks Tanımı	77
Volumes Tanımı	77
Docker Volume Yapılandırması	79
<i>Sadece Bind Mount Kullanılsaydı Nasıl Olurdu?</i>	80
<i>driver: local Kullanılmadığında Ne Olur?</i>	80
<i>Eğer Hiç Volume Tanımı Yapmasaydık?</i>	81
<i>Hibrit Yaklaşımımızın Detaylı Faydaları</i>	81
Sistemin Genel İşleyişi	82
Sistemin Genel Mimarisi	82
1. Kullanıcı İsteği ve Servisler Arası İletişim Akışı	82
2. Network İzolasyonu ve Güvenlik	83
3. Veri Kalıcılığı Mekanizması	83
4. Servisler Arası Bağımlılık Yönetimi	84
5. Hata Toleransı ve Servis Süreklliliği	84
Her Servisin Detaylı Analizi	84
1. NGINX Konteyneri	84
2. MariaDB Konteyneri	85
3. WordPress Konteyneri	85
Konfigurasyon Yönetimi ve Esneklik	86
1. Docker Secrets Kullanımı	86
2. Çevre Değişkenleri ile Parametreleştirmeye	87
3. Volume Yapılandırması ve Hibrit Yaklaşım	87

Giriş



Bu projeyi geliştirirken edindiğim tüm bilgileri bu dokümantasyonda toplamaya çalıştım. Her bölüm için ilgili kaynakları ayrı ayrı belirttim; yapılandırmaları nasıl gerçekleştirdiğimi, kullandığım komutların ve kodların ne işe yaradığını detaylı bir şekilde açıklamaya özen gösterdim.

Aynı zamanda proje sürecinde öğrendiğim diğer önemli bilgileri de paylaşarak bu içeriği daha kapsamlı hale getirmeye çalıştım. Docker, NGINX, MariaDB ve WordPress gibi teknolojilerin birbirleriyle nasıl entegre çalıştığını ve sistem mimarisinin detaylarını adım adım ele aldım.

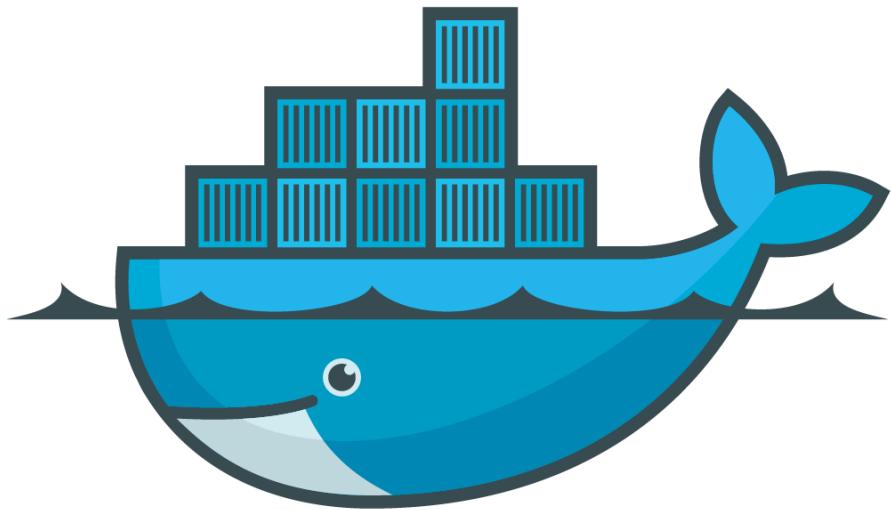
Subject kısmında da belirtildiği gibi: "**Tek bir container yetmez, we need to go deeper.**" Ben de bu felsefeyle hareket edip, sadece çalışan bir sistem kurmakla kalmayıp, nasıl ve neden çalıştığının derinliklerine inmeye çalıştım. Eğer bu seviye yeterli gelmezse, siz daha da derinlere inmeye devam edebilirsiniz. :)

Umarım bu dokümantasyon, sizin için de faydalı olur ve projeyi geliştirirken size katkı sağlar.

- 🔗 GitHub: github.com/menasy
- 📌 Proje Linki: [Inception Docker](https://github.com/menasy/Inception_Docker)
- 🌐 LinkedIn: linkedin.com/in/menasy

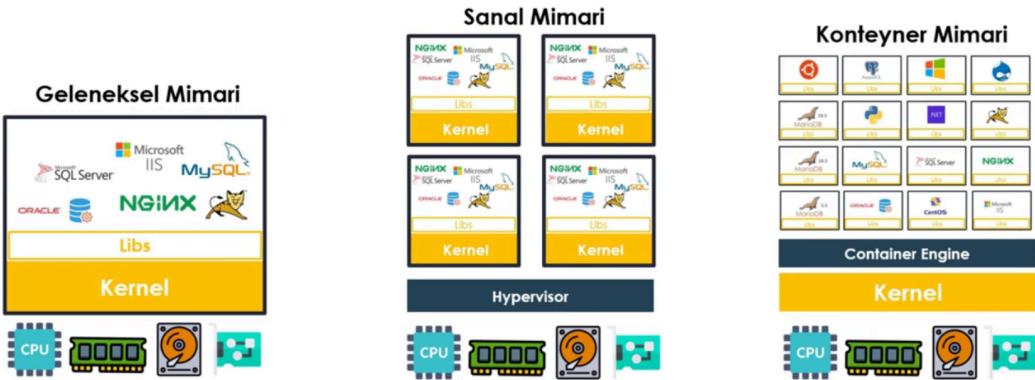
⌚ **Tavsiye:** Bu dokümantasyonu yazarken koyu tema (dark mode) kullandım. En iyi görünüm ve renk uyumu için siz de koyu temada okursanız daha sağlıklı bir deneyim yaşarsınız. Açık modda bazı uyumsuzluklar olabilir.

DOCKER

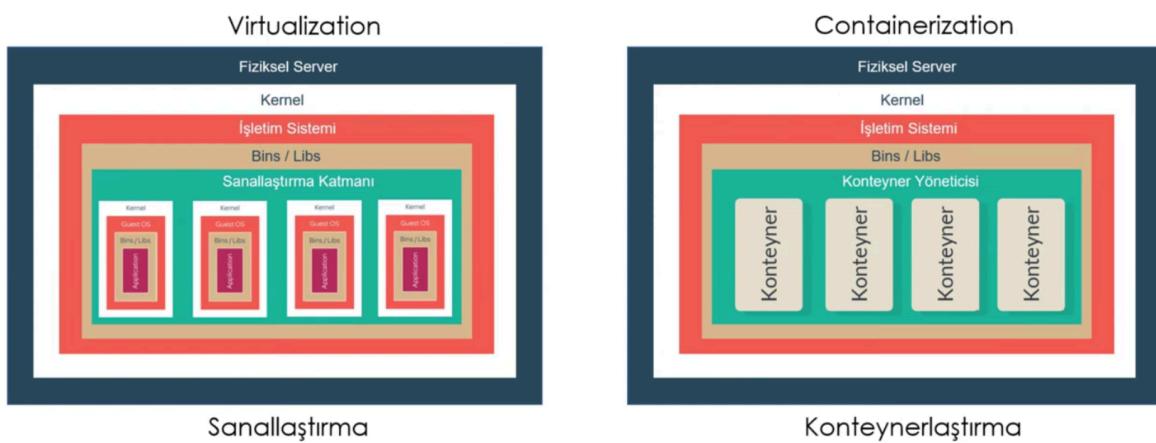


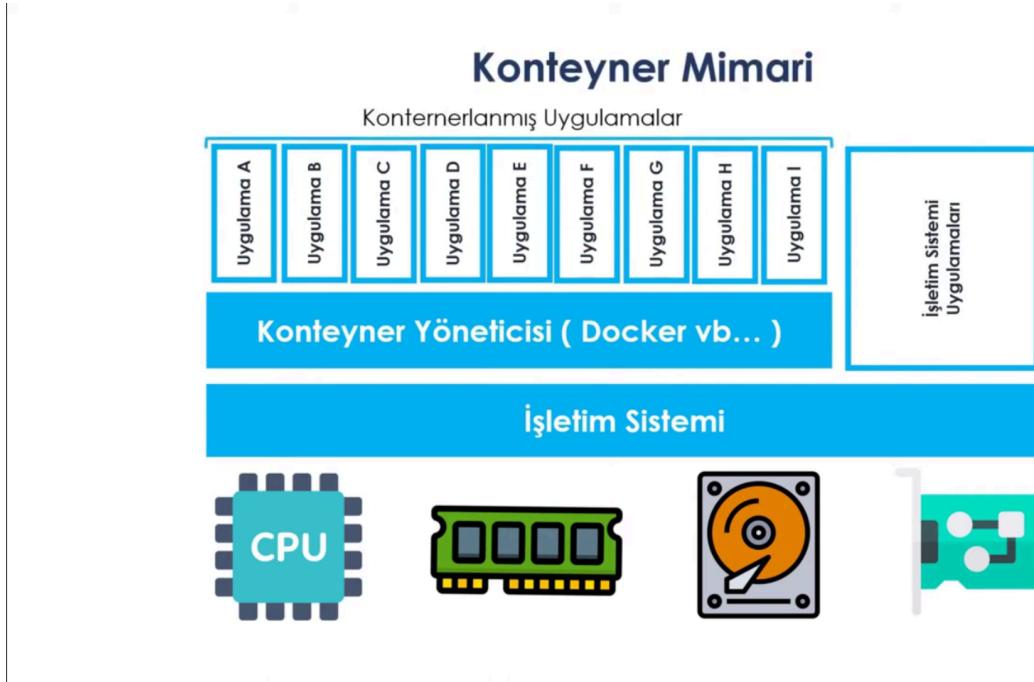
docker

Container Nedir:



Konteyner(Container) Nedir ? Konteyner Mimarisinde Neler oluyor





Sanallaştırma Öncesi ve Sonrası(DOCKER EĞİTİMİ) - - 1:

<https://www.youtube.com/watch?v=1O19rLq1jeY>

<https://www.youtube.com/watch?v=0lgXWnifEpo>

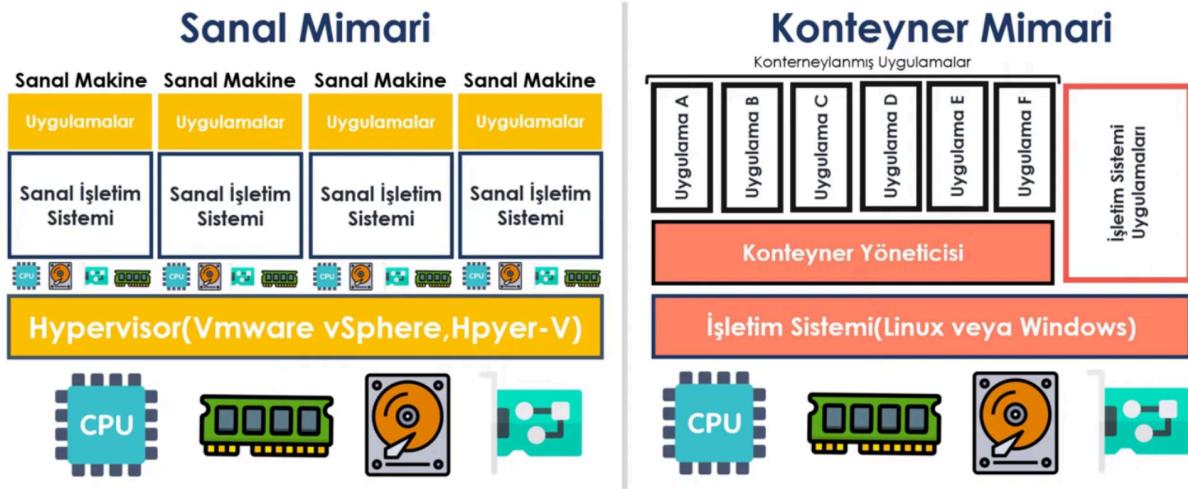
Container nedir:

<https://www.youtube.com/watch?v=mvSJdnvVvRU>

VmWare ve Docker Farkları

<https://www.youtube.com/watch?v=GRkvlHdTwlw&t=1s>

Sanal Mimari Mi Konteyner Mimarisi mi Arasında ki Farklar Nelerdir



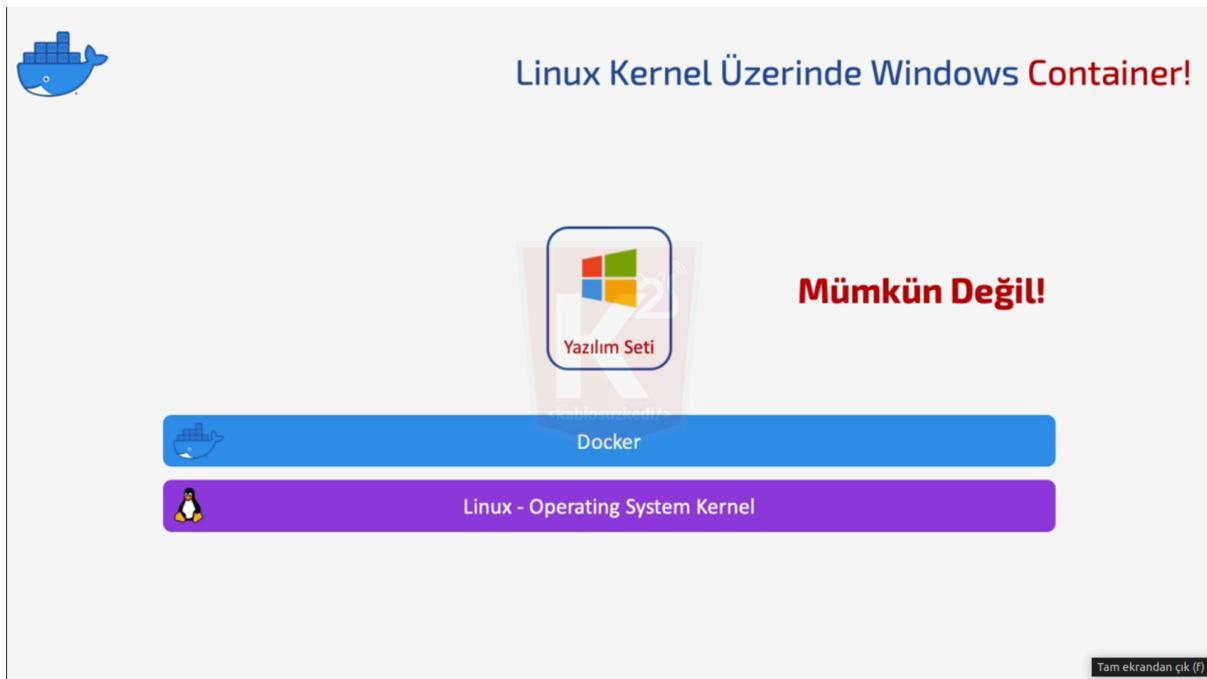
Sanallaştırmada fiziksel donanım birebir taklit edilerek kullanılıyor. Yani her kurulan virtual machine kurulduğu cihazın fiziksel donanımını taklit ederek kullanır. Sanal makine hyperVisor varlığından haberdar değildir sanki fiziksel bir makinedeymiş gibi çalışır.

Docker ise taklit yoktur kernel üzerinde yapılan ozaellestirmeler ve docker kurulumundan sonra direkt uzerine çalıştığı sistemin fiziksel donanımını kullanır. sistem kaynaklarına gore uygulamaları paketleyip izle hale getirir.

Docker Nedir Nasıl Kullanılır kablosuzKedi videosu:

https://www.youtube.com/watch?v=4XVfmGE1F_w&t=1447s

Docker çalışma yapısı ve Bağımlılıklar



örneğin bir containerında bir program ve bu programın bağımlılıkları var ama bu bağımlılıklar windows işletim sisteme ihtiyaç duyuyor. Ama bizim kullandığımız ve dockerin kurulu olduğu sistem linux. Bu durumda o container çalışmaz çünkü containerler VM ler gibi ayrı işletim sistemleri klonlamaz bulunduğu ssitemin işletm sistemini kullanır.



Windows Üzerinde Windows Container!



Mümkün Değil!



Peki bu windows tabanlı containeri windows işletim sistemi üzerine kurulan bir dockerler çalıştırabilir miyiz ?

Bu da olmaz ! çünkü docker containerleri yönetirken sanal bir linux karneli kullanır ve bu durumda yine windows ile çakışma olur.

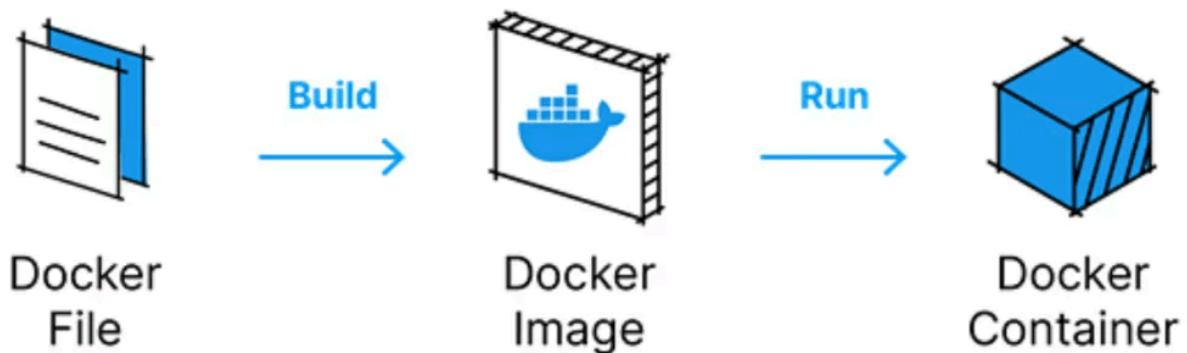
- Yani Bütün containerler linux üzerinden yönetiliyor. Ve linux tabanlı olmayan hiçbir programı containerlerde çalıştırılamayız.

Dockerfile Nedir? Dockerfile Komutları Nelerdir?

<https://kerteriz.net/dockerfile-nedir-dockerfile-komutlari/>

Docker Image Nedir ?

<https://kerteriz.net/docker-image-ve-container/>

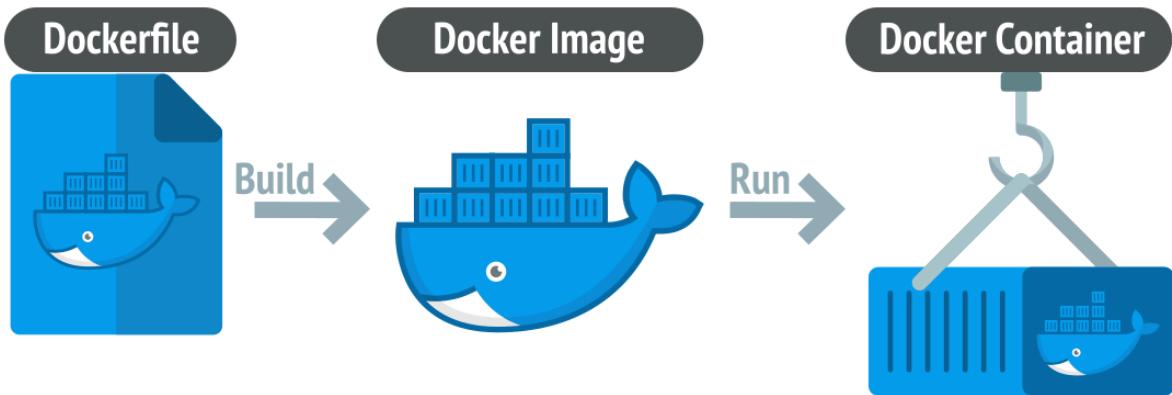


Docker image'ları, bir uygulamanın çalışması için gereken tüm dosyaları, kütüphaneleri ve ayarları içerir.

C programlama dili üzerinden örnek verecek olursak, Dockerfile bir `.C` dosyasına benzetilebilir; içinde program kodları, kütüphaneler ve programın sağlıklı çalışması için gerekli bağımlılıklar bulunur. Docker image ise, C'de `make` komutu ile `.o` dosyalarının derlenerek çalıştırılabilir hale getirilmesine benzer şekilde, `docker build` komutu ile oluşturulur. Docker container ise, bu oluşturulan image'in çalıştırılmış hali olup, C'de `./program`, Docker'da ise `docker run` komutu ile çalıştırılır. Bu benzetme konuyu somutlaştırmak açısından faydalı olabilir, ancak Dockerı tam anlatmak için yetersiz kalabilir docker bundan daha derin. **"We need to go deeper!"**

3 aşamadan oluşur container oluşturulma evresi:

- **1. Dockerfile oluşturun:** Uygulamanızın nasıl çalıştırılacağını ve hangi bağımlılıklarının yükleneceğini belirten bir Dockerfile oluşturun.
- **2. Image oluşturun:** Dockerfile'i kullanarak bir image oluşturun. Bu işlemi `docker build` komutu ile yapabilirsiniz.
- **3. Container oluşturun:** Image'ı kullanarak bir container oluşturun. Bu işlemi `docker run` komutu ile yapabilirsiniz.



- Bir imageden çok sayıda container oluşturulabilir. C programlama üzerinden verdiğimiz örnek üzerinden söyleyecek olursak .c dosyaları derlendi arşivlendi ve çalıştırılmaya hazır myProgram adında program oluşturdu. Bundan sonra

Docker Komutları ve Kullanımı

İmaj (Image) İşlemleri

- `docker build .`: Bulunduğumuz dizindeki Dockerfile'ı okuyarak yeni bir imaj oluşturur.
- `docker build -t nginxdeneme .`: Dockerfile'ı "nginxdeneme" ismiyle etiketleyerek imaj oluşturur.
- `docker images`: Sistemde bulunan tüm imajları listeler. TAG sütunu imajın sürümünü gösterir.
- `docker image tag ubuntu cpy-ubuntu`: Ubuntu imajının bir kopyasını "cpy-ubuntu" adıyla oluşturur.
- `docker rmi id/name`: Belirtilen imajı sistemden tamamen siler.

Container İşlemleri

- `docker ps` veya `docker container ls`: Aktif olarak çalışan container'ları listeler.
- `docker ps -a`: Çalışan ve durmuş tüm container'ları listeler.
- `docker run -it ubuntu`: Ubuntu imajını interaktif terminal (-it) modunda başlatır.
- `docker run -it --name bash-ubuntu ubuntu`: Container'a "bash-ubuntu" adını vererek interaktif modda başlatır.
- `docker start bash-ubuntu`: İsmi belirtilen container'ı başlatır (terminal erişimi olmadan arkaplanda çalışır).

Container Bağlantı ve Kontrol

- `docker attach bash-ubuntu`: Çalışan bir container'ın terminaline erişim sağlar.
- `docker stop name/id`: Çalışan bir container'ı durdurur (ID'nin ilk 2 karakteri yeterlidir).

- `docker rm id/name`: Container'ı sistemden tamamen kaldırır (önce durdurulmalıdır).
- `docker run -d image`: Container'ı arka planda (detached mod) çalıştırır. Terminal çıktılarını göstermez.
- `docker exec -it name/id bash`: Arka planda çalışan bir container'da bash terminali açar.
- `docker logs -f name/id`: Container'ın log çıktılarını gerçek zamanlı olarak gösterir.
- `docker inspect id/name`: Container hakkında detaylı teknik bilgileri JSON formatında gösterir.

Ağ (Network) İşlemleri

- `docker network ls`: Docker'in oluşturduğu tüm ağları listeler.
- `docker network inspect bridge`: Varsayılan bridge ağının detaylı bilgilerini gösterir.

Veritabanı ve Web Servisleri

- `docker run -e MARIADB_ROOT_PASSWORD="1234" -p 27018:3306 mariadb:`
 - `-e`: Çevre değişkeni tanımlar (root şifresi)
 - `-p 27018:3306`: Host portundan (27018) container portuna (3306) yönlendirme yapar
 - MariaDB container'ını belirtilen ayarlarla başlatır

MariaDB'ye Erişim:

- Host makineden MariaDB sunucusuna bağlanma:
`mariadb -h localhost -P 27018 -u root -p`
- Container içinden MariaDB'ye bağlanma:
`docker exec -it mariadb sh
mariadb -u mehmyilm -p`
- Veritabanı komutları:
`USE wordpress;
SHOW TABLES;
SELECT * FROM wp_posts;
DESCRIBE wp_posts;`
- Kullanıcı bilgilerini sorgulama:
`SELECT User, Host, plugin, Password FROM mysql.user WHERE User = 'root';`

HTTP ve SSH İşlemleri

- `curl -IL http://mehmyilm.42.fr:`
- `-I`: Sadece HTTP başlıklarını gösterir `-L`: Yönlendirmeleri takip eder

- Web sitesinin HTTP durum kodlarını ve yönlendirmelerini gösterir (80 portunun çalışıp çalışmadığını test eder)
-

`ssh -A user@192.168.25.31:`

- `-A`: SSH anahtar iletmeyi (forwarding) etkinleştirir
- Host makinemizin SSH anahtarını kullanarak sanal makineye bağlanır
- Bu sayede VM üzerinde, host anahtarımla Git gibi işlemler yapabiliriz

Docker Daemon Nedir?

Docker Daemon, Docker'ın arka planda çalışan ana sürecidir. Kullanıcı terminalinden Docker CLI ile komut girdiğinde, bu komut Docker Daemon'a iletılır. Daemon, şu görevleri üstlenir:

İmaj Yönetimi: Gerekirse, istenen imajı Docker Hub'dan indirir.

Container Oluşturma ve Başlatma: İstenen imajdan container oluşturur ve çalıştırır.

Kaynak İzolasyonu: Container'lar arasında kaynakları (CPU, bellek, ağ) izole eder.

Örneğin, kullanıcı terminale `docker run nginx` yazdığında:

- Docker CLI komutu Docker Daemon'a gönderir.
- Daemon, eğer yerel depoda nginx imajı yoksa Docker Hub'dan indirir.
- İmajdan yeni bir container oluşturur.
- Container'ı başlatır ve gerekli izolasyonu sağlar.

Kısaca, Docker Daemon, Docker container ve imaj işlemlerini yöneten arka plan yöneticisidir.

Docker Run vs Docker Start

1. docker run vs. docker start

docker run:

- Yeni bir container oluşturur ve başlatır.
- Örneğin, `docker run -it --name bash-ubuntu ubuntu` komutu:
 - ubuntu imajından yeni bir container oluşturur.
 - -it ile container'ı interactive terminal modunda başlatır (/bin/bash gibi bir shell açar).
 - Container'ı `bash-ubuntu` adıyla kaydeder.
 - Container'ın ana süreci (örneğin, /bin/bash) sonlanırsa (kullanıcı exit yazarsa), container da durur

docker start:

- Daha önce durdurulmuş bir container'ı yeniden başlatır.
- Örneğin, docker start bash-ubuntu komutu:
 - bash-ubuntu adlı container'ı yeniden başlatır.
 - Ancak terminal oturumunu otomatik olarak ekrana bağlamaz (yani -it olmadan başlatır).
 - Container arka planda (detached modda) çalışır, bu yüzden exit yazılmadığı sürece süreç sonlanmaz.

2. Neden docker run ubuntu Hemen Sonlanır?

ubuntu imajının varsayılan davranışı:

- Ubuntu imajı, varsayılan olarak /bin/bash komutunu çalıştırır. Ancak:
- -it parametresi olmadan çalıştırılırsa, Bash interactive modda açılmaz ve hemen sonlanır.

Örneğin:

- `docker run ubuntu #` Bash hemen sonlanır, container durur.

-it parametresinin önemi:

- -it ile container'ı başlatırsanız (`docker run -it ubuntu`), Bash interactive terminal modunda açılır ve kullanıcı girdisi bekler. Bu yüzden container çalışmaya devam eder.

3. docker start Neden Arkada Çalıştırır?

- docker start komutu, container'ı daha önceki konfigürasyonuyla (örneğin, -it ile oluşturulmuşsa) başlatır. Ancak:
- Terminal oturumunu otomatik olarak ekrana bağlamaz (attach yapmaz).
- Bu yüzden container arka planda çalışır, ancak Bash süreci hala aktif kalır (çünkü daha önce -it ile oluşturulmuştur).
- Container'ın arka planda çalıştığını görmek için docker ps komutunu kullanabilirsiniz.

Docker Image Katman Yapısı

1. Docker Image'leri Katmanlıdır

Her Docker imajı, birbiri üzerine eklenen **katmanlardan (layers)** oluşur. Her katman:

- Dockerfile'daki bir komuta (örneğin, `RUN`, `COPY`, `FROM`) karşılık gelir.
 - Salt okunur (read-only) olarak saklanır.
 - Benzersiz bir **kimlik (hash)** ile tanımlanır (örneğin, `5a7813e071bf`).
-

2. Katmanlar Nasıl Oluşur?

Bir Dockerfile örneği üzerinden inceleyelim:

<code>FROM ubuntu:22.04</code>	--> Temel katman (ubuntu:22.04 imajını çeker)
<code>RUN apt-get update</code>	--> Yeni bir katman oluşturur
<code>COPY app.py /app</code>	--> Başka bir katman
<code>CMD ["python3", "/app"]</code>	--> Son katman

- Her adım yeni bir katman ekler

3. Katmanların Avantajları

- **Depolama Tasarrufu:** Aynı katmanlar birden fazla imajda paylaşılır.
 - **Hız:** Değişmeyen katmanlar önbellekten kullanılır (`docker build` hızlanır).
 - **Güvenlik:** Her katman hash ile doğrulanır (değiştirilemez).
 - Katmanlar, Docker'in depolama alanında (`/var/lib/docker`) saklanır.
 - Kullanılmayan katmanları temizlemek için: `docker system prune`
-

4. Özet: Pull İşleminin Mantığı

1. Docker, imajın hangi katmanlardan olduğunu belirler (Docker Hub'dan manifest dosyasını okur).
2. Yerel depoda olmayan katmanları indirir (`Pull complete`).
3. Yerelde mevcut olanları atlar (`Already exists`).
4. Tüm katmanlar indirildikten sonra imaj oluşturulur ve kullanıma hazır hale gelir.

Docker Compose Nedir?

Docker Compose, birden fazla Docker container'ını tanımlamak, yapılandırmak ve yönetmek için kullanılan bir araçtır. Tek bir komutla tüm servisleri başlatıp durdurmanızı sağlayarak, mikroservis mimarileri veya birbiriyle bağlantılı servislerin yönetimini kolaylaştırır. Özellikle geliştirme, test ve demo ortamlarında hız ve tutarlılık sağlar.

Temel Özellikleri

- Çoklu Container Yönetimi:** İlgili tüm servisleri tek bir komutla ayağa kaldırır.
- Yapılurma Dosyası (`docker-compose.yml`):** Servisler, ağlar, volume'ler ve bağımlılıklar tek bir YAML dosyasında tanımlanır.
- Otomatik Ağ ve Volume Oluşturma:** Servisler otomatik olarak aynı ağa bağlanır; volume'ler veri kalıcılığını sağlar.
- Bağımlilik Yönetimi:** Servislerin başlama sırası ve birbirine bağımlılıkları kontrol edilir.

Temel Komutlar

`docker-compose up -d` Servisleri arka planda başlatır.

`docker-compose down` Servisleri durdurur ve kaynakları (ağ, volume) temizler.

`docker-compose build` Dockerfile'lardan imajları yeniden oluşturur.

`docker-compose logs -f [servis]` Belirli bir servisin loglarını gerçek zamanlı görüntüler.

`docker-compose exec [servis] [cmd]` Çalışan bir serviste komut çalıştırır (ör: exec web bash).

Anahtar Kavramlar

Servisler (services):

- Her servis bir container'a karşılık gelir.
- `build, image, ports, environment, volumes` gibi direktiflerle yapılandırılır.

Volume'ler (volumes):

- Veri kalıcılığını sağlar (ör: veritabanı verileri).
- Host dizini veya Docker tarafından yönetilen volume'ler kullanılabilir.

Ağlar (networks):

- Servislerin birbiriyle iletişim kurmasını sağlar.
- Varsayılan olarak Compose otomatik bir ağ oluşturur.

Bağımlılıklar (depends_on):

- Servislerin başlama sırasını belirler (ör: web, db'den sonra başlar).

Avantajlar

- **Hızlı Kurulum:** Tek komutla tüm ortamı ayağa kaldırma.
- **Tutarlı Yapılandırma:** Tüm ekip üyeleri aynı ortamı kullanır.
- **Esneklik:** Geliştirme, test ve demo ortamları için ideal.

Port Nedir ?

Port, bir IP adresi üzerinde belirli hizmetlerin çalışmasını sağlayan sanal kapılardır. Sunucular ve uygulamalar, farklı portlar üzerinden veri alışverişi yapar. Portlar, ağ trafiğini yönlendirmek ve belirli hizmetleri ayırmak için kullanılır.
Örneğin:

1. Evin Ana Kapısı = IP Adresi

Evinin adresi (IP adresi) herkese açıktır, ancak içeri girmek için hangi kapıyı (portu) kullanacaklarını belirtmeleri gereklidir.

Örneğin:

192.168.1.10:80 → "80 numaralı kapıdan gir, web sunucusu burada!"

2. Odalar = Portlar

Evin bahçesi(Port 80): Web trafiği buradan geçer. Herkese açıktır.

<http://example.com:80> → Tarayıcı otomatik olarak bu portu kullanır.

Salon (Port 22): SSH bağlantıları için ayrılmıştır. Sadece yetkili kişiler girebilir.
`ssh user@example.com -p 22`

Yatak Odası (Port 3306): Veritabanı hizmeti burada çalışır. Sadece güvenilir uygulamalar erişebilir.

bu şekilde örneklendirebiliriz. Kimin nereye erişmesini yönetebiliyor gerektiğinde kapatabiliyor. Kısacası trafiği yönetiyor.

Port Mapping Nedir?

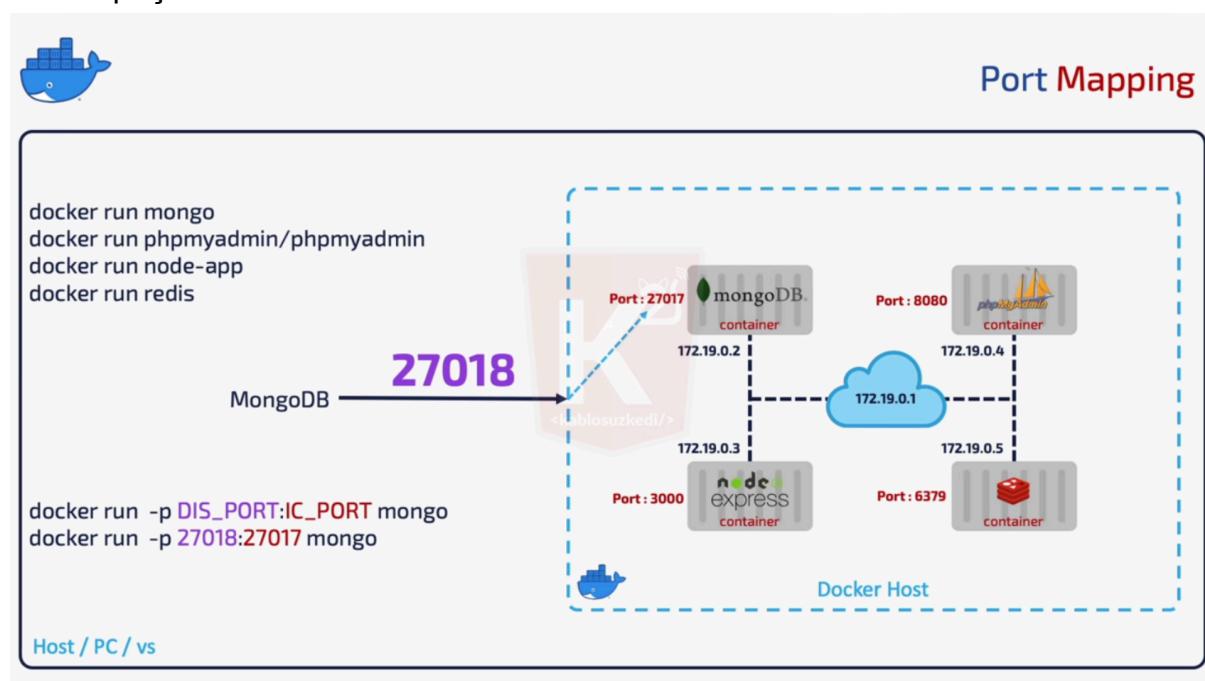
Bir servis; örneğin mariadb ya da mangodb bunlar başlatıldığında default olarak kendilerinin bağlı olduğu bir port vardır bu portlar docker host üzerinde erişilebilirdir.. Mangodb 27017 ve mariadb 3306 ama bu portlara direkt erişilemez. Kendi localhost umuzdan erişmek için port mapping yapılarak erişim sağlanabilir.

`docker run -p 27018:3306 mariadb` diyerek mariadb çalışıktan sonra web uzerindnen localhost:27018 diyerek kullanıma açık olduğunu gorebiliriz ama erişemeyiz burdan. erişim için: `mariadb -h localhost -P 27018 -u root -p` kullanabiliriz.

- h Bağlanılacak sunucunun adresi (hostname veya IP) -> localhost
- P MariaDB sunucusunun dinlediği port numarası. Mariadb normalde 3306 portunda çalışıyor ama kendi localimizden bağlanabilmek için port mapping yaptık (27018:3306). bu yuzden erişirken de `-P 27018` kullanıyoruz
- u servere hangi kullanıcıyla bağlanılacağını belirtir -> root
- p en sondaki bu flag de kullanıcı şifresini terminal üzerinden girmemizi sağlar. Şifreyi komut satırında belirtmek istersek: `mariadb -h localhost -P 27018 -u root -p` kullanabiliriz. Ama bu önem tavsiye edilmez. Güvenlik sebebiyle.

Özetle:

- h: Sunucu adresi
- P: Port numarası
- u: Kullanıcı adı
- p: Şifre istemi



Docker container'ları varsayılan olarak **izole bir ağ ortamında** çalışır. Container içindeki bir uygulama belirli bir portta dinleme yapsa bile, bu porta dış dünyadan (host makineden veya diğer cihazlardan) doğrudan erişemezsınız. Port mapping, **host makinenin belirli bir portunu**, container'ın dinlediği porta eşleyerek dış erişime izin verir.

Görselde 4 farklı container ve port eşlemeleri gösterilmiştir:

1. **MongoDB:** Container içinde **27017** portunda çalışıyor → Host'ta **27018** portuna eşlenmiş.

Adım Adım Port Mapping Nasıl Çalışır?

1. Container'ı Başlatma ve Port Eşleme

Örneğin, MongoDB container'ını başlatmak için:

```
docker run -d --name mongo-container -p 27018:27017 mongo
```

- `-p 27018:27017`: Host'taki **27018** portunu, container'ın içindeki **27017** portuna eşler.
- Artık host makineden `localhost:27018` adresini kullanarak MongoDB'ye erişebilirsiniz.

2. Host Makineden Erişim

- Host makinenizde (örneğin, fiziksel bilgisayarınızda) tarayıcı veya istemci yazılım kullanarak:
- MongoDB için: `localhost:27018`

3. Container'lar Arası Haberleşme

Container'lar Docker'ın varsayılan olarak oluşturduğu **bridge ağları** üzerinden birbirleriyle iletişim kurabilir. Örneğin:

- **Node.js Express** uygulaması, **MongoDB**'ye bağlanmak için Docker'ın atadığı dahili IP'yi kullanır (örneğin, `172.19.0.2:27017`).
- Container'lar birbirlerine **container ismi** veya **IP adresi** üzerinden erişebilir:

Neden Port Mapping Kullanırız?

1. **İzolasyonu Korumak**: Container'ların iç portları dış dünyaya direkt açılmaz, sadece belirlediğiniz portlar erişilebilir olur.
2. **Çakışmaları Önlemek**: Aynı host portunu farklı container'lara eşleyebilirsiniz (örneğin, iki web sunucusunu `8080` ve `8081` portlarına eşlemek).
3. **Güvenlik**: Sadece gerekli portları açarak saldırı yüzeyini küçültürsünüz.

Port Mapping Türleri

a) Basit Port Eşleme

```
docker run -p [HOST_PORT]:[CONTAINER_PORT] [IMAGE]
```

- Örnek: `docker run -p 3000:3000 node-app`

b) Belirli Bir IP Üzerinden Eşleme

```
docker run -p [HOST_IP]:[HOST_PORT]:[CONTAINER_PORT] [IMAGE]
```

- Örnek: `docker run -p 127.0.0.1:3306:3306 mysql`

c) Dinamik Port Atama (Host Portu Belirtmeden)

```
docker run -P [IMAGE]
```

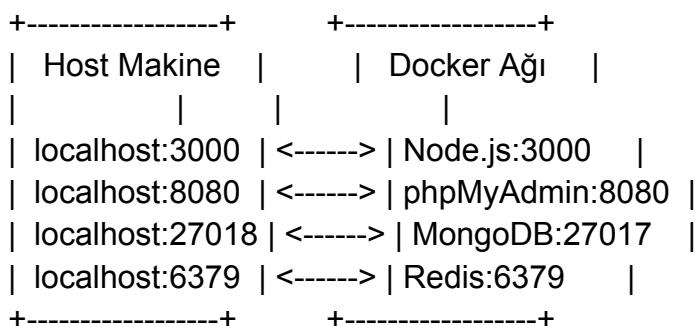
- Docker, rastgele bir host portu seçer (genellikle `32768-60999` arası).
- Hangi portun atandığını görmek için `docker ps` komutunu kullanın.

Önemli Noktalar

- **Aynı Host Portunu Kullanma:** Aynı host portunu iki container'a eşlerseniz, Docker hata verir:
Error: Cannot start container: port is already allocated.
- **Docker Ağ Yapısı:** Container'lar aynı Docker ağında olduğu sürece birbirlerine IP veya isimle erişebilir. Farklı ağlardaysa port eşleme dışında ek ayarlar gerekir.

Port Mapping'i Görselleştirme

Görseldeki gibi bir yapıyı aklinizda şöyle canlandırabilirsiniz:



Docker Volume'lar ve Türleri

Docker containerları doğası gereği geçici yapılardır; container silindiğinde içindeki tüm veriler de kaybolur. Bu sorunu çözmek için Docker, verileri kalıcı hale getirmenin ve containerlar arasında paylaşmanın yolu olarak volume kavramını geliştirmiştir.

Docker volume'lar, containerların dosya sisteminden bağımsız olarak çalışan, verilerin kalıcı olarak saklanmasıyı sağlayan yapılardır. Docker engine tarafından yönetilen volume'lar, containerlar oluşturulurken, çalışırken veya kaldırılırken veri bütünlüğünü korur.

Volume'lar, container'in yazılabilir katmanında (writable layer) değil, host makinede ayrı bir konumda saklanır. Bu sayede:

1. Container silinse bile veriler kaybolmaz
2. Veriler container'lar arasında paylaşılabilir
3. Container'ın performansı etkilenmez
4. Veriler daha güvenli bir şekilde yedeklenebilir

Docker volume'lar, container başlatıldığında `-v` veya `--volume` parametresi ile veya Docker Compose dosyasında `volumes:` bölümünde tanımlanır. Volume tanımlandığında, Docker belirtilen dizini container içindeki belirtilen yolunla eşleştirir.

Docker Volume Türleri

1. Named Volumes (İsimlendirilmiş Volume'lar)

Named volume'lar, Docker tarafından tamamen yönetilen ve belirli bir isimle anılan volume türüdür. Docker Daemon bu volume'ları `/var/lib/docker/volumes/` dizininde oluşturur ve yönetir. Named volume'lar, Docker volume'ların en gelişmiş ve önerilen kullanım şeklidir.

Çalışma Prensibi: Docker Engine, named volume oluşturulduğunda host sistem üzerinde kendisine ayrılmış bir alanda (`/var/lib/docker/volumes/`) içerik için bir klasör oluşturur. Bu klasörün adı, volume'a verdığınız isim ve bazı ek tanımlayıcılarından oluşur. Container bu volume'u kullandığında, veriler bu klasöre yazılır ve buradan okunur. Container'ın yaşam döngüsünden bağımsız olarak bu veriler korunur.

Avantajları:

- Tamamen Docker tarafından yönetilir, kullanıcının dosya sistemi yapısıyla ilgilenmesine gerek yoktur
- Volume adı ile kolayca referans verilebilir
- Containerlar arasında veri paylaşımını kolaylaştırır
- Docker komutlarıyla yönetimi oldukça basittir (`create, inspect, prune, rm` vb.)
- Docker'ın native yedekleme mekanizmalarıyla uyumludur
- Host makineden bağımsızdır, farklı ortamlara taşınabilir

Örnek (Docker Compose):

`volumes:`

`db_data:`

`driver: local`

Bu tanımlamayla Docker, "db_data" adında bir volume oluşturur. Bu volume, container'a bağlılığında içindeki veriler container yaşam döngüsünden bağımsız olarak saklanır. Named volume'lar container silinip yeniden oluşturulduğunda bile verilerini korur.

Named Volume'un CLI ile Kullanımı:

`# Volume oluşturma`

`docker volume create my_data`

`# Container'ı volume ile başlatma`

`docker run -d --name my_container -v my_data:/data nginx`

`# Volume hakkında bilgi alma`

`docker volume inspect my_data`

`# Volume'u silme (kullanılmıyorsa)`

`docker volume rm my_data`

2. Bind Mounts (Bağlanmış)

Bind mount'lar, host makinedeki belirli bir klasörü veya dosyayı container içindeki bir yola doğrudan bağlayan volume türüdür. Named volume'lardan farklı olarak, bind mount'larda host sistemindeki kesin dosya yolu belirtilir.

Çalışma Prensibi: Bind mount kullandığınızda, Docker container içindeki belirtilen yolu host makinenizdeki belirtilen yol ile doğrudan eşleştirir. Container, host sistemindeki bu klasöre doğrudan erişebilir ve değişiklikler anında her iki tarafta da görünür hale gelir. Bind mount'lar, Docker'in özel bir alanında değil, host sisteminin herhangi bir yerindeki dizinlerle çalışır.

Avantajları:

- Host sistemindeki dosya ve klasörlerle doğrudan erişim sağlar
- Geliştirme sürecinde yapılan değişiklikler anında container içinde görünür
- Host üzerindeki araçlarla doğrudan dosyaları düzenleme imkanı sunar
- Host sistemindeki dosya izinleri ve sahipliği korunur

Dezavantajları:

- Host makine yapısına bağımlıdır, farklı ortamlara taşınması zordur
- Host dosya sistemi yapısı değişirse containerlar etkilenebilir
- Güvenlik açısından named volume'lara göre daha riskli olabilir (container, host dosya sistemine erişebilir)

Örnek (Docker Compose):

```
volumes:  
  db_data:  
    driver: local  
    driver_opts:  
      type: none  
      o: bind  
      device: /home/user/data/mariadb
```

Bu yapılandırma ile `/home/user/data/mariadb` klasörü, container içindeki belirtilen yola bağlanır. Container bu yola bir dosya yazdığında, dosya aslında host makinedeki bu klasöre yazılır ve tersine host üzerinde yapılan değişiklikler anında container içinde görünür.

Bind Mount'un CLI ile Kullanımı:

```
# Container'ı bind mount ile başlatma  
docker run -d --name web_server -v /home/user/web:/usr/share/nginx/html nginx  
# Geçerli dizini container içindeki bir yola bağlama  
docker run -d --name app -v $(pwd):/app node
```

3. tmpfs Mounts (Geçici Bellekte Montajlar)

tmpfs mount'lar, verileri yalnızca host sistemin RAM'inde (belleğinde) tutan geçici depolama alanlarıdır. Container durduğunda veya silindiğinde içindeki tüm veriler tamamen silinir.

Çalışma Prensibi: Docker, tmpfs mount oluşturduğunda host sistemin RAM'inde belirtilen boyutta bir alan ayırır. Container bu alana yazma/okuma yapabilir, ancak bu veriler disk yerine bellekte tutulur. Container durduğunda veya yeniden başlatıldığında bu veriler tamamen silinir.

NGNIX

Ngnix configrasyonu çalışma prensibi işleyişi:

<https://www.youtube.com/watch?v=U7PrQklb3Ew>

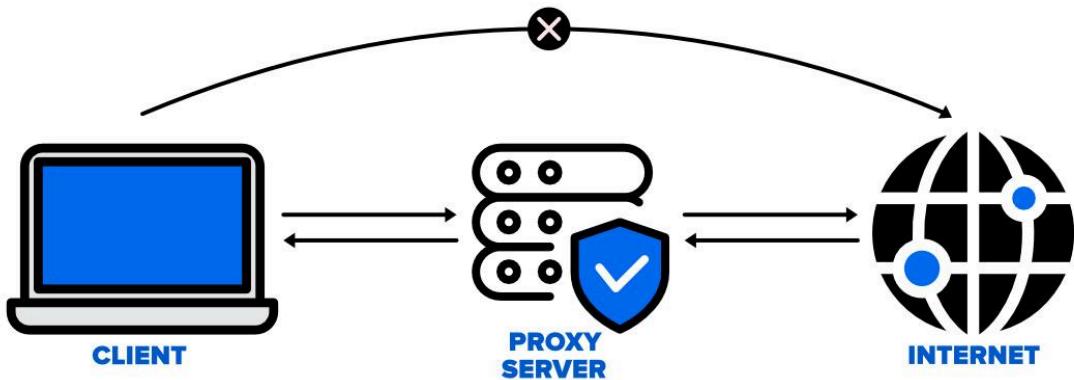
Nginx Webserver Nedir ve Ayarları - Reverse Proxy - Özeti

<https://devnot.com/2023/reverse-proxy-nedir/>

Proxy Sunucusu Ve Çalışma Prensibi

- **İstemci İsteği:** Kullanıcı, bir web sayfasına erişmek istediğiinde, isteği proxy sunucusuna gönderir.
- **Proxy İşlemi:** Proxy sunucusu, bu isteği alır, gerekli değişiklikleri yapabilir (örneğin, IP adresini gizleyebilir) ve isteği hedef sunucuya iletir.
- **Hedef Sunucu Yanıtı:** Hedef sunucu isteği alır ve yanıtı proxy sunucusuna gönderir.
- **Yanıt İletimi:** Proxy sunucusu, hedef sunucudan aldığı yanıtı kullanıcıya iletir.

Proxy Sunucusunun Yararları



Gizlilik ve Anonimlik:

Proxy sunucusu, kullanıcının gerçek IP adresini gizler. Bu, kullanıcıların internet üzerindeki aktivitelerinin takip edilmesini zorlaştırmır.

Örneğin, bir kullanıcı bir proxy sunucusu aracılığıyla bağlandığında, hedef sunucu proxy'nin IP adresini görür, kullanıcının değil.

Erişim Kontrolü ve Filtreleme:

Kurumsal veya okul ağlarında, belirli web sitelerine erişimi kısıtlamak için kullanılır.

Örneğin, sosyal medya veya oyun siteleri engellenebilir.

İçerik filtreleme ile zararlı veya istenmeyen içeriklerin engellenmesi sağlanabilir.

Hız ve Performans Artışı:

Proxy sunucuları, sıkça ziyaret edilen web sayfalarının verilerini önbelleğe alabilir. Bu, kullanıcıların bu sayfalara erişim hızını artırır.

Ön belleğe alınan içerikler, doğrudan hedef sunucudan almak yerine proxy sunucusundan hızlı bir şekilde yüklenebilir.

Güvenlik:

Proxy sunucuları, kötü niyetli yazılımlardan koruma sağlayabilir.

Örneğin, zararlı içeriklerin engellenmesi veya kullanıcıların güvenli bir şekilde internete bağlanması için kullanılabilir.

Ayrıca, proxy sunucuları, kullanıcının gerçek IP adresini gizleyerek daha fazla güvenlik sağlar.

Coğrafi Erişim:

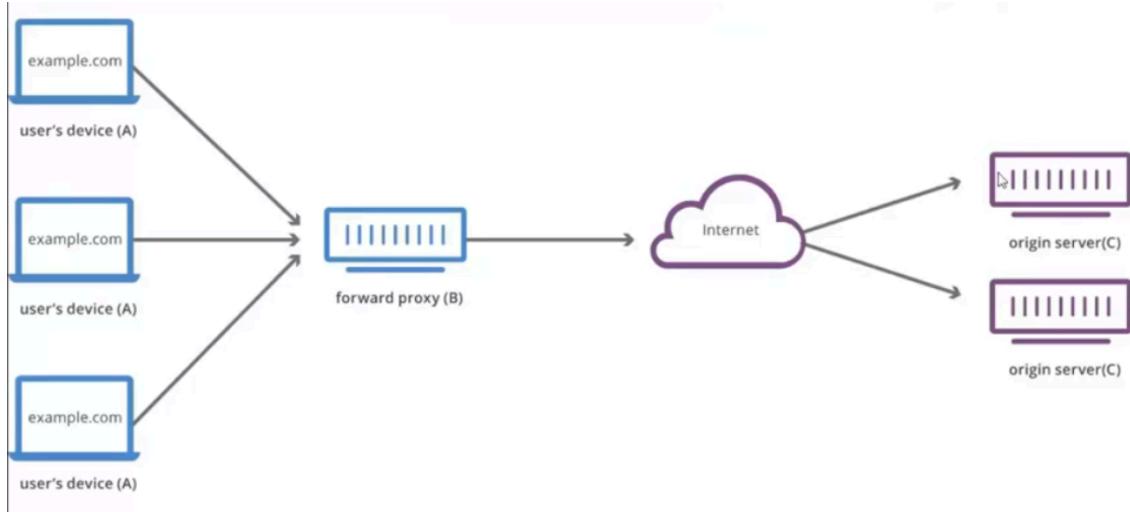
Bazı içerikler, belirli coğrafi bölgelerde erişime kapalı olabilir. Proxy sunucuları, farklı bir coğrafi konumdan erişim sağlamak için kullanılabilir.

Örneğin, bir kullanıcı, başka bir ülkedeki bir proxy sunucusunu kullanarak o ülkede erişilebilen içeriklere ulaşabilir.

Proxy Türleri

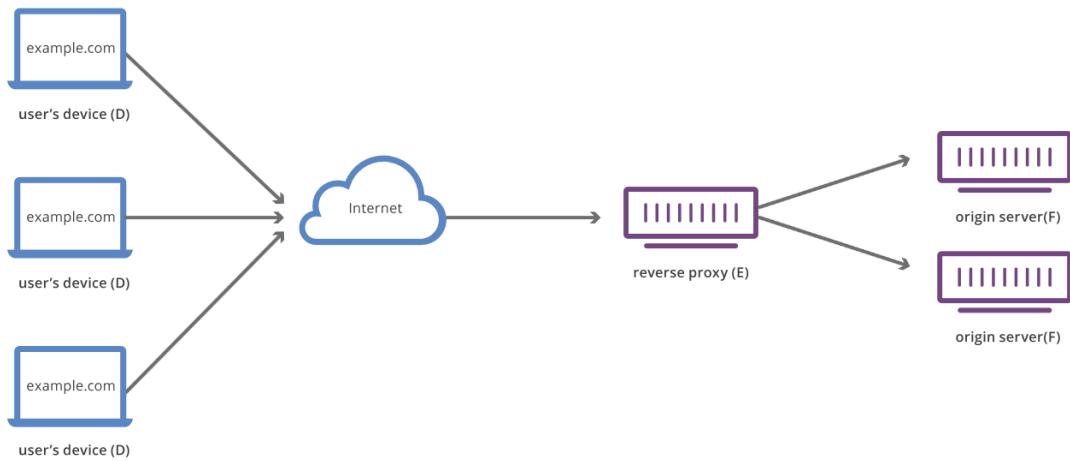
Forward Proxy: Herkesin erişebileceği ve kullanabileceği proxy sunucularıdır. Genellikle anonimlik sağlamak için kullanılır.

Forward Proxy



Private Proxy: Belirli bir kullanıcı grubu için tasarlanmış ve genellikle ücretli olan proxy sunucularıdır. Daha yüksek güvenlik ve performans sunar.

Reverse Proxy Flow



Reverse Proxy: Sunucu tarafında çalışan proxy'dir. Kullanıcılarından gelen istekleri alır ve arka plandaki sunuculara yönlendirir. Genellikle yük dengelemesi ve güvenlik amaçlı kullanılır.

Reverse proxy normal bir proxy sunucusunun tam tersidir. Reverse proxy sunucusu, kullanıcıların sunucuya yaptığı talepleri alır ve bu talepleri sunucunun arkasındaki farklı sunuculara yönlendirir. Reverse Proxy, sunucuya gelen istekleri hedef sunuculara yönlendirirken istemcilerle sunucular arasında bir ara yüz görevi görür. İstemciler, bu arayüz ile iletişim kurar. Reverse Proxy, isteği alırken bu isteği hedef sunucuya ileterek yanıtını alır ve istemciye ileterek gönderir. Yani, "Reverse Proxy" kullanmak, talepleri doğrudan kendi sunucunuza yönlendirmek yerine araya başka bir sunucu koymak anlamına gelir. Bu sunucu, istekleri alır, hedef sunucuya yönlendirir ve yanıtını geri göndererek sunucunuzun yükünü azaltır ve performansınızı artırır.

Reverse Proxy Neden Kullanılır?

- Bir web sitesini korumak için reverse proxy kullanılabilir. Sunucuların IP adresi bu sayede gizli kalır.
- Reverse proxy kullanan bir web sitesini bir DDoS (distributed denial-of-service) saldırısıyla hedef almak daha zordur.
- Load balancing (yük dengeleme) yaparak yoğun trafiği geniş bir sunucu havuzuna dağıtmak amacıyla kullanılabilir.
- Statik içeriği önbellekte tutarak aynı içeriğe istek gelmesi durumunda önbellekteki versiyonu ile çok daha hızlı cevap verir.
- SSL şifrelemesini bizim için halledebilir. Ek bir işlem masraflı yaratılan SSL handshake mekanizmasını tüm istemciler için yönetmek yerine, yalnızca az sayıda reverse proxy ile bunu yaparak verimliliği artırır.

Nginx'in Event-Driven Mimarisi

Asenkron İşlem:

Nginx, istemci bağlantılarını asenkron olarak işler. Bu, Nginx'in birden fazla istemci isteğini aynı anda yönetebilmesini sağlar. bir iş yapılmayıken başka bir istek/iş geldiğinde diğer işin bitirilmesi beklenmez. İş direkt olarak uygulamaya konur ve işlenir. Böylelikle aynı anda birden fazla iş yapılmış olur. **Olay tabanlı (event-driven) yapı** sayesinde

Tek İş Parçacığı:

Nginx, genellikle tek bir iş parçacığı içinde çalışır ve bu iş parçacığı, birden fazla bağlantıyı yönetir. Yani, her bağlantı için yeni bir iş parçacığı oluşturmak yerine, mevcut iş parçacığını kullanarak istekleri işler.

Olay Döngüsü:

Nginx, bir olay döngüsü mekanizması kullanır. Bu mekanizma, bağlantılardan gelen olayları dinler ve bu oylara yanıt verir. Örneğin, bir istemci bir istek gönderdiğinde, Nginx bu isteği işlemek için uygun bir yanıt oluşturur.

Nginx Mimarisinin Diğer Web Sunucularıyla Farkları

Apache:

Apache genellikle her bağlantı için yeni bir iş parçacığı veya işlem oluşturur (MPM - Multi-Processing Module). Bu, yüksek eşzamanlı bağlantınlarda daha fazla kaynak tüketimine yol açabilir. Nginx ise asenkron yapısıyla daha az kaynak kullanır.

Kaynak Kullanımı:

Nginx, daha az bellek ve işlemci gücü kullanarak daha fazla eşzamanlı bağlantı yönetebilir. Bu, özellikle yoğun trafik alan web siteleri için büyük bir avantajdır.

Performans:

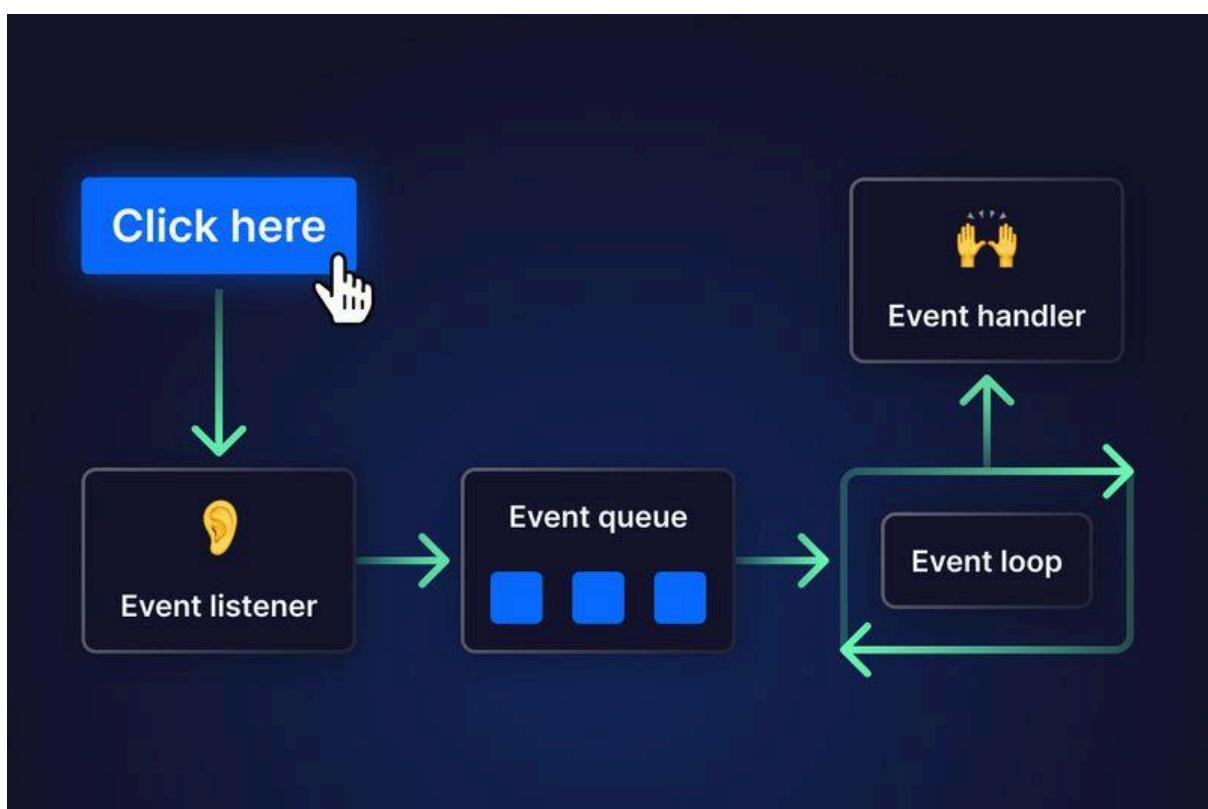
Nginx'in event-driven yaklaşımı, daha hızlı yanıt süreleri ve daha yüksek işlem hacmi sağlar. Apache gibi diğer sunucular, yoğun yük altında performans kaybı yaşayabilir.

Nginx'in event-driven mimarisi, yüksek performans ve verimlilik sağlarken, diğer geleneksel web sunucularının kaynak tüketimini minimize eder. Bu, Nginx'i özellikle yüksek trafiğe sahip web uygulamaları için ideal bir seçim haline getirir.

Olay Tabanlı (event-driven) Yapı

NGINX, olay tabanlı (event-driven) ve non-blocking I/O mimarisi sayesinde yüksek performans sağlar. Bir istek geldiğinde, bu isteği hemen işleme almak yerine bir olay kuyruğuna ekler ve diğer istekleri işlemeye devam eder. Örneğin, bir dosya okuma veya veritabanı sorgusu gibi uzun süren işlemler sırasında NGINX beklemek yerine, bu işlemi arka planda başlatır ve sonucun hazır olmasını beklemeden yeni gelen istekleri kuyruğa alır.

İşletim sistemi, arka plandaki işlem tamamlandığında NGINX'e bir bildirim (event) gönderir. NGINX de bu bildirimi alınca, tamamlanan işlemi ele alıp istemciye yanıtı döndürür. Bu yaklaşım, tek bir iş parçacığıyla binlerce bağlantıyı aynı anda yönetebilmesini sağlarken, CPU ve bellek kullanımını minimize eder. Ayrıca, worker prosesler ile çoklu CPU çekirdeklerinden yararlanarak paralel işlem gücünü optimize eder. Böylece, geleneksel çoklu iş parçacıklı sistemlerin aksine, kaynak tüketmeden yüksek eşzamanlılık sunar.



1. Geleneksel Sunucular Nasıl Çalışır?

Apache gibi geleneksel web sunucuları, genellikle "**thread-per-connection**" modelini kullanır:

- Her yeni bağlantı için yeni bir iş parçacığı (thread) veya proses oluşturur.
- Thread'ler aynı anda **bloklanır** (örneğin, veritabanı sorgusu beklerken).
- **Problem:** 10.000 bağlantı → 10.000 thread → Yüksek bellek ve CPU tüketimi.

2. NGINX'in Olay Tabanlı Mimarisi

NGINX, "**single-threaded event loop**" (tek iş parçacıklı olay döngüsü) kullanır.

Olay Döngüsü (Event Loop)

- Tek bir iş parçacığı, tüm bağlantıları sırayla dinler.
- Her bağlantı için **non-blocking I/O** kullanır (yani, bir işlem beklerken diğerlerini işlemeye devam eder).

Non-Blocking I/O

- Örnek: Bir dosya okuma isteği geldiğinde:
 1. NGINX, işletim sistemine "bu dosyayı oku" komutu verir.
 2. Dosya okunurken **thread bloklanmaz**; hemen başka bir bağlantıya geçer.
 3. Dosya okuma işlemi tamamlandığında, NGINX'e bir **olay (event)** bildirilir.
 4. NGINX, olayı işler ve istemciye yanıt gönderir.

3. NGINX vs. Çoklu İş Parçacıklı Sistemler

3. NGINX vs. Çoklu İş Parçacıklı Sistemler

Özellik	NGINX (Event-Driven)	Geleneksel (Thread-Per-Connection)
Kaynak Tüketimi	Düşük (1 thread)	Yüksek (N thread)
Bağlantı Yönetimi	Tüm bağlantılar tek thread'de	Her bağlantı için ayrı thread
CPU/Memory	Optimize	Yüksek yük altında şiser
Gerçek Kullanım	1 thread ile 10.000 bağlantı	10.000 bağlantı → 10.000 thread

4. Tek Thread Nasıl Bu Kadar Hızlı Oluyor?

- **Bloklanmayan İşlemler:** NGINX, bir isteği işlerken asla **beklemez**. Örneğin:
- Veritabanı sorgusu → İşletim sistemi tamamlayınca NGINX'e haber verir.
- Dosya okuma → Aynı şekilde non-blocking olarak yapılır.
- **Olayların Sıralı İşlenmesi:** NGINX, olayları bir kuyrukta (event queue) toplar ve sırayla işler.

5. Peki Çoklu CPU Çekirdeklerinden Nasıl Yararlanıyor?

NGINX, varsayılan olarak **CPU çekirdek sayısı kadar worker proses** oluşturur. Her worker:

- Kendi **tek iş parçacıklı event loop**'unu çalıştırır.
- Böylece çoklu çekirdekler paralel olarak kullanılır.

Örnek: 4 çekirdek → 4 worker proses → Her biri 10.000 bağlantıyı yönetebilir.

6. Örnek Olarak:

- **Senaryo:** 10.000 eşzamanlı kullanıcı bir web sitesine bağlanıyor.
- **Apache (Thread-Per-Connection):** 10.000 thread oluşturur → Yüksek bellek tüketimi ve CPU context switching maliyeti.
- **NGINX:** 4 worker proses (4 çekirdek) → Her worker 2.500 bağlantıyı **non-blocking** olarak yönetir → Düşük kaynak tüketimi.

7. Özeti: Neden Tek Thread Yeterli?

1. **Non-Blocking I/O:** İşlemler bloklanmaz, thread sürekli aktif kalır.
2. **Olay Tabanlı Mimari:** Tüm bağlantılar tek bir thread'de sıralı olarak işlenir.
3. **Worker Prosesler:** Çoklu çekirdeklerden yararlanılır.

8. "Tek Thread Yavaş Kalır" Yanılığı

- Bir işi birden çok thread yapmak daha hızlı değildir
- Thread'ler arası geçiş (context switching) ve senkronizasyon maliyetlidir. NGINX, bu maliyeti ortadan kaldırarak daha verimli çalışır.

Örnek:

- 1 thread, 10.000 bağlantıyı non-blocking ile yönetir.
- 10.000 thread, sürekli context switching yapar → Performans düşer.

MariaDB

<https://wmaraci.com/nedir/mariadb>

Farklılıkların özeti: MySQL ve MariaDB

	MySQL	MariaDB
JSON	MySQL, JSON raporlarını ikili nesneler olarak depolar.	MariaDB, JSON raporlarını dizelerde saklar. MariaDB'nin JSON veri türü, <i>LONGTEXT</i> için bir takma addır.
Oracle veritabanı uyumluluğu	MySQL, yüksek düzeyde uyumluluğa sahiptir ancak PL/SQL'i desteklemez.	MariaDB, yüksek düzeyde uyumluluğa sahiptir ve 10.3 sürümünden bu yana PL/SQL'i destekler.
Hız ve performans	MySQL, çoğaltma ve sorgulama işlemlerinde MariaDB'den biraz daha yavaşır.	MariaDB, çoğaltma ve sorgulama işlemlerinde MySQL'den biraz daha hızlıdır.
İşlevsellik	MySQL, süper salt okunur işlevi, dinamik sütunları ve veri masklemeyi destekler.	MariaDB, görünmez sütunları ve geçici tablo alanını destekler.
Kimlik doğrulaması	MySQL, <i>validate_password</i> bileşenine sahiptir.	MariaDB'nin üç adet parola doğrulayıcı eklentisi vardır.
Şifreleme	MySQL veritabanları, bekleyen verileri şifrelemek MariaDB, geçici günlük şifrelemesini ve ikili günlük şifrelemesini destekler.	
Depolama altyapıları	MySQL, MariaDB'den daha az depolama altyapısına sahiptir.	MariaDB, MySQL'den daha fazla depolama altyapısına sahiptir ve tek bir tabloda birden fazla altyapı kullanabilir.
Lisans	MySQL'in iki sürümü vardır: MySQL Enterprise Edition ve bir GPL sürümü.	MariaDB tamamen GPL altındadır.
İş parçası havuzu	MySQL, Enterprise Edition'da iş parçası havuzuna sahiptir.	MariaDB aynı anda 200.000'den fazla bağlantıyı yönetebilir ve bu da MySQL'den daha fazladır.

TLS sertifikası nedir?

<https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/>

Neden SSL Artık Güvenli Değil?

SSL (Secure Sockets Layer), HTTPS ile güvenli bağlantılar sağlamak için kullanılan eski bir şifreleme protokolüdür. Ancak **SSL 2.0 ve SSL 3.0** artık güvensiz kabul ediliyor ve modern tarayıcılar ile sunucular tarafından desteklenmiyor.

SSL'in Güvensiz Olmasının Nedenleri:

1. Eski ve Güçsüz Şifreleme

- SSL 2.0 (1995) ve SSL 3.0 (1996), eski ve kolayca kırılabilen şifreleme yöntemleri kullanıyor.
- Modern işlemcilerle brute-force saldırıyla bu şifrelemeleri kırmak artık çok kolay.

2. Forward Secrecy Desteği Yok

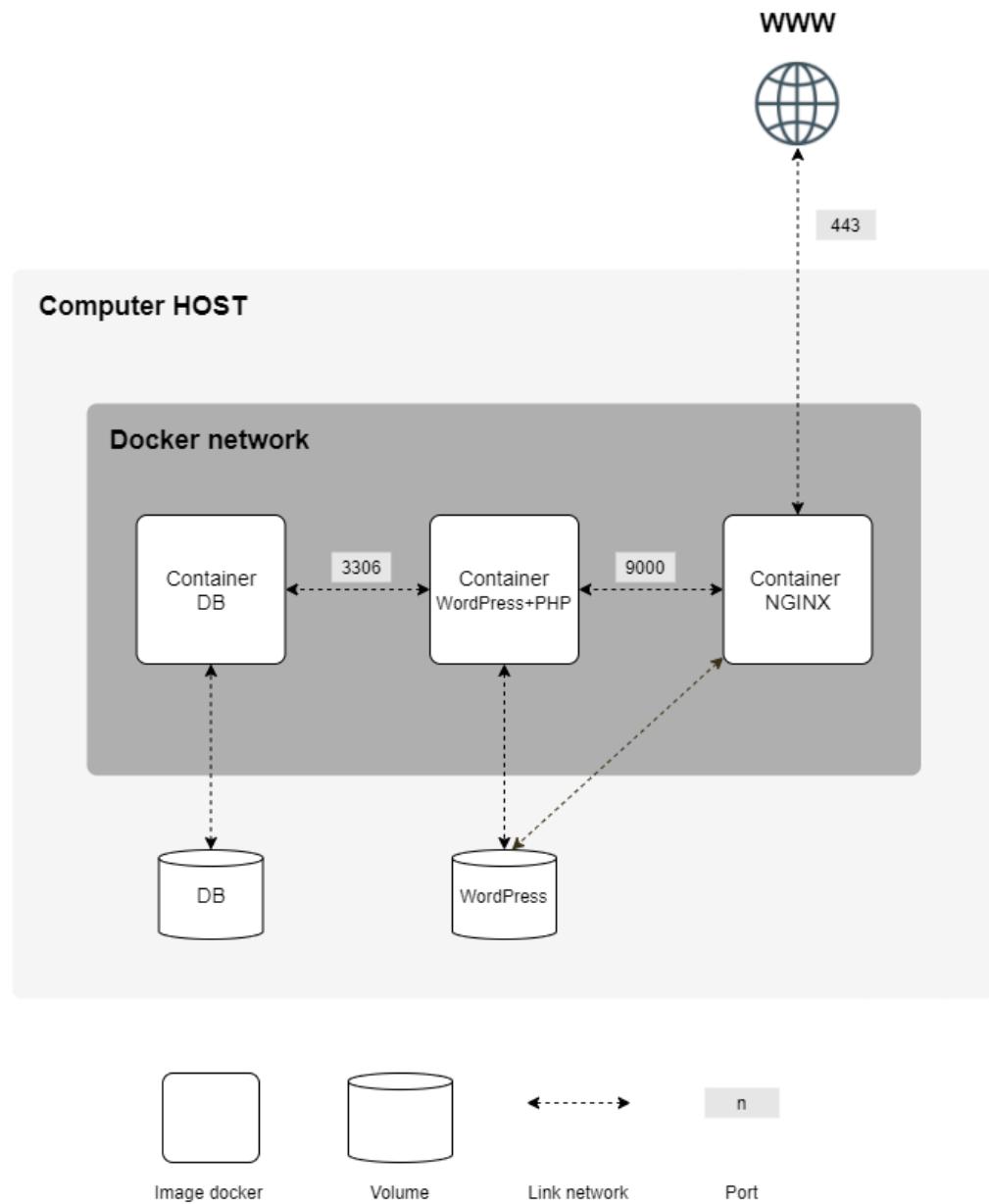
Bilgileri şifrelemek ve şifresini çözmek için kullanılan anahtarları sık sık ve otomatik olarak değiştiren bir şifreleme sistemini ifade eder.

- SSL, eski şifreleme anahtarlarını saklayarak geçmiş verileri koruyamıyor.
- **TLS 1.2 ve 1.3**, forward secrecy desteği sunarak daha güvenli hale geldi.

Gerekli bilgileri öğrendiğimize göre Dockerfile oluşturarak INCEPTION projesine başlayabiliriz.



INCEPTION GİRİŞ



Dockerfile oluşturma:

<https://acokgungordu.medium.com/docker-serisi-dockerfile-olu%C5%9Fturma-a21dfcfdb2bc>

NGINX

NGINX DOCKERFILE

```
FROM debian:bullseye
RUN apt-get update && apt-get install -y nginx openssl
RUN mkdir /etc/nginx/ssl
COPY ./conf/nginx.conf /etc/nginx/sites-enabled/default
COPY ./tools/installer.sh /var/www
RUN chmod +x /var/www/installer.sh
RUN mkdir -p /run/nginx
ENTRYPOINT ["var/www/installer.sh"]
EXPOSE 443
CMD ["nginx", "-g", "daemon off;"]
```

1. Temel İmaj Seçimi

FROM debian:bullseye

Bu komutla, **Debian Bullseye** sürümünü temel alarak bir Docker imajı başlatıyorum. Debian, güvenilir ve stabil bir Linux dağıtımıdır ve genellikle sunucular için tercih edilir. Bu imaj üzerinde bütün yapılandırmalar yapılacak.

2. Paket Güncelleme ve Yükleme

```
RUN apt-get update && apt-get install -y nginx openssl
```

Burada, **apt-get update** ile sistemdeki paketler güncelleniyor. Bu işlem, sistemin en güncel paket bilgilerini almasını sağlar. Sonrasında **nginx** (web sunucusu) ve **openssl** (SSL sertifikaları için gerekli araç) yükleniyor. **-y** parametresi ile yükleme sırasında herhangi bir onay istenmeden işlemler otomatik gerçekleşiyor.

3. SSL Sertifikaları İçin Dizin Oluşturma

```
RUN mkdir /etc/nginx/ssl
```

Bu komutla, **SSL sertifikaları** için bir dizin oluşturuyorum. HTTPS bağlantıları kurabilmek için bu sertifikaları burada saklayacağım.

4. Özel NGINX Konfigürasyonunu Kopyalama

```
COPY ./conf/nginx.conf /etc/nginx/sites-enabled/default
```

Bu komutla, projede bulunan özel **nginx.conf** dosyasını **/etc/nginx/sites-enabled/default** dizinine kopyalıyorum. Bu dosya, NGINX'in nasıl çalışacağına dair ayarları içeriyor. Burada, hangi portların dinleneceği, hangi dosyaların sunulacağı ve SSL yapılandırması gibi şeyler yer alıyor.

5. Kurulum Scripti Kopyalama

```
COPY ./tools/installer.sh /var/www
```

Burada, **installer.sh** adında bir bash scriptini container'a kopyalıyorum. Bu scriptin amacı, container başlatıldığında gerekli olan bazı kurulum ve yapılandırma işlemlerini yapmak olacaktır.

6. Kurulum Scriptine Çalıştırılabilir İzin Verme

RUN chmod +x /var/www/installer.sh

Bu komutla, **installer.sh** scriptine çalıştırılabilir izin veriyorum. Böylece script, container başlatıldığında doğru şekilde çalıştırılabilecek bir program haline geliyor.

7. NGINX İçin Geçici Dizin Oluşturma

RUN mkdir -p /run/nginx

Bu komut, NGINX'in **geçici dosyalarını** saklayacağı dizini oluşturuyor. Bu dizin, NGINX'in pid dosyaları gibi geçici dosyalarını tutacak. Eğer bu dizin yoksa, NGINX düzgün çalışmamayabilir.

8. Container Başlatıldığında Çalışacak Script (ENTRYPOINT Komutu)

ENTRYPOINT ["var/www/installer.sh"]

ENTRYPOINT komutuyla, container başlatıldığında mutlaka çalıştırılacak komutu belirliyorum. Burada **installer.sh** scripti belirtilmiş. Yani, container başlatıldığında bu script otomatik olarak çalıştırılacak. SSL sertifikasını selfsigned olarak burada oluşturuyoruz.

9. Port Açma (EXPOSE Komutu)

EXPOSE 443

EXPOSE komutuyla, container'da **443 numaralı portu** dış dünyaya açıyorum. Bu port, HTTPS bağlantıları için kullanılır. Eğer container'ı çalıştırırken port yönlendirmesi yapılmazsa, bu portla dışarıdan bağlantı kurulamaz. Bu portu açarak, güvenli bir bağlantı sağlanması mümkün olur. Aslında expose komutu semboliktir temel olarak bir işlevi olmaz port işlemlerini compose.yml dosyası ve conf dosyasında ele alıyoruz. [Bu konuya yine degeneceğiz.](#)

10. NGINX'i Çalıştırma (CMD Komutu)

CMD ["nginx", "-g", "daemon off;"]

CMD komutuyla, container çalıştırıldığında **NGINX** sunucusunu başlatıyorum. [-g "daemon off;"](#) parametresiyle de NGINX'in **foreground'da** çalışmasını sağlıyorum. Eğer bu parametreyi eklemeseydim, NGINX arka planda çalışacak ve container hemen kapanabilecekti. Bu yüzden, NGINX'in sürekli çalışmasını sağlamak için foreground'da başlatıyorum. Subjectte de bizde bu isteniyor nginx in foregroundda çalışması lazım.

ENTRYPOINT ve CMD Arasındaki Fark

- **ENTRYPOINT**: Container başlatıldığından **kesinlikle çalıştırılması gereken komuttur**. Yani, container'ın ana işlevini sağlayacak komut burada belirtilir. **ENTRYPOINT** komutunu verdiğinde, container her zaman bu komutu çalıştırır.
- **CMD**: **CMD, ENTRYPOINT** komutuna ek olarak çalıştırılacak **varsayılan komuttur**. Eğer **docker run** komutunda başka bir şey belirtilmese, **CMD komutundaki komut çalıştırılır**. **CMD** komutu bir **varsayılan argümandır**, yani **ENTRYPOINT** komutuyla birlikte çalışacak şekilde eklenebilir.

Özetle, **ENTRYPOINT** container'ın ana komutudur ve container'ın düzgün çalışabilmesi için mutlaka çalıştırılması gereken bir komut olur. **CMD** ise bu komutun varsayılan parametrelerini belirler ya da bir yedek komut sağlar.

Nginx dockerfile ni oluşturduk ama nginx başlatmak ve çalıştmak için **ssl setfikasını ayarlayıp configrasyonlarını yapmamız** lazım. Bundan once de dockerfile çalışma yapısını detaylıca ele alalım.

Dockerfile çalışma yapısı

Entrypoint ile başlangıç notası belirlenir ve bu durumda bir betik verilmiş. cmd ile de verilen komutlar **exec "\$@" ile işlenir**

Bu komut, **Docker konteyneri içinde betik çalıştırılırken verilen komutları yürütmek için kullanılır**.

1. exec Komutu Ne Yapar?

exec komutu, **çalışan betiği tamamen değiştirerek yeni bir işlem başlatır**.

Normalde bir betik içerisinde başka bir komut çalıştırıldığımızda, **betik çalışmaya devam eder**.

Ama **exec** komutunu kullanırsak, **mevcut betik süreci sona erer ve yerine yeni komut süreci başlar**.

Örnek:

```
#!/bin/bash  
echo "Fenerbahce"  
exec ls -l  
echo "1907"
```

Sadece Fenerbahce yazar ardından kod exece geçer ve ls -l çalıştırılır. 1907 yazdırılmaz.

2. "\$@" Ne Anlama Geliyor?

"\$@", betiğe **verilen tüm parametreleri** temsil eder.

Bu, Docker konteyneri çalıştırılırken verilen komutların burada yürütülmesini sağlar.

Ayrıca betiğe dışardan bir arguman da girilirse artık onun çalıştırılmasını sağlar.

Örnek:

Eğer bunu şöyle çalıştırırsam:

`./betik.sh ls -l`

betiğin içindeki exec @ artık exec ls -l olur.

3. Docker İçindeki Kullanımı

Docker'da, **ENTRYPOINT** veya **CMD** ile verilen komutlar **bu betik sayesinde çalıştırılabilir**.

Örneğin, Dockerfile şu şekilde olsun:

```
ENTRYPOINT ["/bin/bash", "/var/www/installer.sh"]
```

```
CMD["ls"]
```

Bu durumda **Docker konteyneri başlatıldığındá**:

- Önce **installer.sh** çalışır.
- **exec "\$@"** sayesinde, **Docker konteynerine verilen komut buraya iletilir ve çalıştırılır**.
- Cmd deki komutta en son olarak **instaler.sh** iletilir ve çalıştırılır.

Eğer konteyneri şu şekilde başlatırsak:

```
docker run my_container nginx -g "daemon off;"
```

Bunun sonucu olarak, betik **şu hale gelir**:

```
exec nginx -g "daemon off;"
```

ve konteyner içinde sadece **NGINX süreci çalışmaya başlar**.

Özetle

- **exec**, betiği **verilen komut ile değiştirir** ve yeni bir süreç başlatır.
- **"\$@", betiğe girilen tüm argümanları alıp çalıştırır**.
- Docker içinde, konteyner çalıştırıldığında verilen komutun **doğrudan yürütülmesini sağlar**.
- Bu sayede **installer.sh** sadece ön hazırlıkları yapar, asıl konteyner sürecini **kesintisiz şekilde başlatır**.

Nginx-Instaler.sh

`nginx-selfsigned.key` ve `nginx-selfsigned.crt` nedir, ne işe yarar?

Bu iki dosya **NGINX için kullanılan SSL/TLS sertifikalarıdır**. Bu sertifikalara yukarıda detaylıca değindik.

- **nginx-selfsigned.key (Özel Anahtar)**: Web sunucusunun **şifreleme ve kimlik doğrulama işlemleri** için kullandığı **gizli anahtardır**. Bu anahtarın **gizli tutulması** gereklidir.
- **nginx-selfsigned.crt (Sertifika)**
 - **Web sunucusunun kimliğini doğrular**. Tarayıcılar ve istemciler, bu sertifika aracılığıyla **sunucunun gerçekten güvenilir olup olmadığını** kontrol eder.
 - Burada kullanılan sertifika **kendinden imzalıdır (self-signed)**, yani **herhangi bir resmi sertifika otoritesi tarafından doğrulanmamıştır**.

Bu sertifikalar ne işe yarar?

- HTTPS bağlantısını etkinleştirerek, **veri iletiminin şifrelenmesini sağlar**. Tarayıcı ve istemcilerin, sunucunun **gerçekten doğru yer olup olmadığını** anlamasına yardımcı olur.
- Kendinden imzalı olduğu için, **üretim ortamında değil, test ve geliştirme aşamalarında kullanılır**.

OpenSSL ile Sertifika Oluşturma (Neden Gereklidir?)

SSL sertifikası, web sunucularının **güvenli bağlantı (HTTPS)** kurmasını sağlayan kritik bir bileşendir. **OpenSSL**, bu sertifikaları oluşturmak için yaygın olarak kullanılan bir araçtır.

SSL Sertifikası ve HTTPS

Web sunucuları arasındaki **veri iletimi**, HTTPS (HTTP Secure) protokolü üzerinden güvence altına alınır. **HTTPS, SSL/TLS sertifikaları** sayesinde tarayıcılar ile sunucular arasındaki iletişimi **şifreler** ve bu sayede verilerin üçüncü şahıslar tarafından dinlenmesini engeller.

Web siteniz için **güvenli bir bağlantı** sağlamak istiyorsanız, **SSL/TLS sertifikasına** ihtiyacınız vardır. **Kendinden imzalı sertifikalar (selfsigned)** genellikle geliştirme ve test aşamalarında kullanılır çünkü resmi sertifika otoriteleri tarafından onaylanmış değildir.

OpenSSL ile Sertifika Oluşturma

```
openssl req -x509 -nodes -days 365 -newkey rsa:4096 \
-keyout /etc/nginx/ssl/nginx.key \
-out /etc/nginx/ssl/nginx.crt \
-subj
"/C=TR/ST=ISTANBUL/L=SARIYER/O=42ISTANBUL/CN=$DOMAIN_NAME"
```

- **openssl req:** Bu komut, SSL/TLS sertifikası oluşturmak için bir **sertifika isteği (CSR)** yaratır.
- **req** komutunu kullanarak **istek** oluşturmak, **sertifikayı almak için gereken bilgiyi** sağlamaya yardımcı olur.
- **-x509:** Bu bayrak, **X.509 sertifika standardını** belirtir. SSL/TLS sertifikaları **X.509 formatında** olur ve tarayıcılar bu formatı kabul eder. Bu sertifika formatı, **kimlik doğrulama** ve **şifreleme** için kullanılır.
- **-nodes:** Sertifika anahtarının **şifresiz (unencrypted) olarak** oluşturulmasını sağlar. Eğer bu bayrak kullanılmazsa, özel anahtar şifrelenir ve bir parola gerektirir. **-nodes** kullanarak, anahtar **şifrelenmez** ve herhangi bir parola sorulmaz.
- **-days 365:** Sertifikanın geçerlilik süresi **365 gün (1 yıl)** olarak ayarlanır. Sertifikanın geçerlilik süresi dolduğunda, **yeniden oluşturulması gerekecektir**.
- **-newkey rsa:4096:** Burada **RSA algoritması ile yeni bir anahtar çifti** oluşturuluyor. **4096 bitlik RSA anahtarları**, daha **güvenli** ve **şifreleme** için güçlündür. Bu uzunluk, sertifikanın **güvenliğini** artırır. Daha kısa anahtarlar daha hızlı olsa da **daha az güvenli** olabilir.
- **-keyout /etc/nginx/ssl/nginx.key:** Bu parametre, oluşturulacak olan **özel anahtar dosyasının yolunu belirler**. Bu anahtar, **sunucunun kimliğini doğrulamak ve bağlantıları şifrelemek için kullanılır**.
- **-out /etc/nginx/ssl/nginx.crt:** Bu parametre, oluşturulacak **sertifika dosyasının yolunu belirler**. Sertifika, tarayıcıların sunucunun **kimliğini doğrulamak için kullandığı dosyadır**.
- **-subj**
"/C=TR/ST=ISTANBUL/L=SARIYER/O=42ISTANBUL/CN=\$DOMAIN_NAME": Sertifika için gerekli olan **kimlik bilgilerini** tanımlar. Bu bilgiler genellikle:
 - **C:** Ülke kodu (örneğin, **TR** Türkiye için).
 - **ST:** Eyalet veya şehir.
 - **L:** İlçe.
 - **O:** Kuruluş adı (örneğin, "42ISTANBUL").

- **CN:** Ortak Ad (Common Name), yani alan adı (örneğin, mehmyilm.42.fr). Bu bilgiler **sertifikada yer alır ve sunucunun kimliğini doğrulamak** için kullanılır.

NGINX KONFIGÜRASYONU

Konfigürasyon Dosyaları:

Konfigürasyon dosyaları, bir sistemin veya yazılımın nasıl çalışacağını belirleyen ayarları içerir. Bu dosyalar sayesinde kullanıcılar **servisleri özelleştirebilir, performansı artırabilir ve güvenliği sağlayabilir**. Örneğin, ağ ayarları, erişim izinleri ve kaynak yönetimi genellikle bu dosyalar üzerinden kontrol edilir.

NGINX Konfigürasyon Dosyası

NGINX'in konfigürasyon dosyası, web sunucusunun nasıl çalışacağını belirleyen temel bileşendir. Bu dosyada hangi portların dinleneceği, hangi alan adının kullanılacağı, SSL sertifikalarının nerede olduğu ve PHP gibi dinamik içeriklerin nasıl işlendiği gibi kritik ayarlar bulunur. **Yanlış yapılandırma sunucunun çalışmamasına, güvenlik açıklarına veya performans sorunlarına neden olabilir.**

1. Genel Sunucu Ayarları

```
server {  
    listen 443 ssl;  
    listen [::]:443 ssl;  
    server_name mehmyilm.42.fr
```

- **server {}:** Bu blok, bir NGINX sunucu tanımını ifade eder. İçindeki ayarlar, bu sunucunun nasıl çalışacağını belirler.
- **listen 443 ssl;:** Sunucunun **443 numaralı port üzerinden SSL ile çalışacağını** belirtir. 443, HTTPS trafiği için kullanılan standart porttir.
- **listen [::]:443 ssl;:** IPv6 desteği eklenerek yine 443 numaralı porttan SSL bağlantılarını dinlemesini sağlar.
- **server_name mehmyilm.42.fr;:** Bu sunucunun **hangi alan adı (domain) üzerinden hizmet vereceğini** belirtir. Yani "mehmyilm.42.fr" adresine gelen HTTPS istekleri bu sunucu tarafından işlenecek.

2. SSL Sertifikası ve Protokol Ayarları

```
ssl_certificate /etc/nginx/ssl/nginx.crt;
ssl_certificate_key /etc/nginx/ssl/nginx.key;
ssl_protocols TLSv1.2 TLSv1.3;
```

- **ssl_certificate /etc/nginx/ssl/nginx.crt;**: SSL/TLS şifrelemesi için kullanılacak sertifikayı tanımlar. Bu sertifika, tarayıcıların sunucuya güvenli şekilde bağlanmasını sağlar.
- **ssl_certificate_key /etc/nginx/ssl/nginx.key;**: Sertifikaya ait özel anahtarın konumunu belirtir. Bu anahtar, istemcilerin (tarayıcıların) sunucu ile güvenli bir bağlantı kurmasını sağlar.
- **ssl_protocols TLSv1.2 TLSv1.3;**: Sadece güvenli TLS protokollerini kullanacak şekilde yapılandırma yapılır. TLSv1.0 ve TLSv1.1 eski ve güvensiz olduğu için desteklenmez.

“Bu ayarlar sayesinde web sitesi **HTTPS ile güvenli bir şekilde çalışır**. Eğer bir kullanıcı <https://mehmyilm.42.fr> adresine bağlanırsa, sunucu ona bu sertifikayı sunar ve güvenli bağlantı başlatılır.

3. Web Sayfalarının Konumunu Belirleme

```
root /var/www/html;
index index.php index.nginx-debian.html;
```

- **root /var/www/html;**: Web sunucusunun dosyaları nereden sunacağını belirler. Yani, istemciler (ziyaretçiler) bu sunucuya bağlandığında `/var/www/html` dizini içindeki dosyalar gösterilecek.
- **index index.php index.nginx-debian.html;**: Varsayılan açılacak dosyaları belirler. Eğer bir kullanıcı doğrudan <https://mehmyilm.42.fr> adresine girerse:
 - İlk olarak `index.php` dosyasına bakılır.
 - Eğer `index.php` yoksa `index.nginx-debian.html` dosyası gösterilir.

Bu ayarlar, web sitesinin **hangi dosyaları varsayılan olarak sunacağını belirlemek için önemlidir**.

4. Ana Sayfa Yönlendirme Ayarları

```
location / {
    try_files $uri $uri/ /index.php$is_args$args;
}
```

- `location / {}`: Tüm gelen istekleri kapsayan ana kural. Yani <https://mehmyilm.42.fr/> adresine gelen herhangi bir isteği bu blok yönetecek.
- `try_files $uri $uri/ /index.php$is_args$args;;`
- İlk olarak kullanıcının istediği dosya veya klasör var mı kontrol edilir (`$uri`).
- Eğer yoksa aynı isimde bir dizin var mı kontrol edilir (`$uri/`).
- Bunların hiçbirini bulunamazsa, istek `index.php` dosyasına yönlendirilir.

Bu kural özellikle WordPress gibi PHP tabanlı sistemlerde önemlidir çünkü URL yapıları dinamik olabilir. Eğer bir kullanıcı <https://mehmyilm.42.fr/blog> adresine giderse ve `/var/www/html/blog` klasörü yoksa, bu istek **otomatik olarak index.php dosyasına yönlendirilir**.

5. NGINX'te FastCGI ile PHP İşleme

Şimdi NGINX konfigürasyonundaki FastCGI yapılandırması

```
location ~ \.php$ {  
    fastcgi_split_path_info ^(.+\.php)(/.+)$;  
    fastcgi_pass wordpress:9000;  
    fastcgi_index index.php;  
    include fastcgi_params;  
    fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;  
    fastcgi_param SCRIPT_NAME $fastcgi_script_name;  
}
```

Bu Blok Ne İşe Yarıyor?

Bu blok, **PHP uzantılı dosyalara yönelik özel bir yönlendirme tanımlar**. NGINX, doğrudan PHP dosyalarını çalıştırılamadığı için, bu dosyaları FastCGI (PHP-FPM) üzerinden işlemeye yönlendirir.

1. PHP Dosyalarını Hedefleme

```
location ~ \.php$ {
```

- `location ~ \.php$`:
- Sunucuya gelen isteklerde **.php ile biten dosyaları** yakalar.
- Örneğin, <https://mehmyilm.42.fr/index.php> veya <https://mehmyilm.42.fr/contact.php> gibi istekler bu blok tarafından yönetilir.

2. PHP Dosya Yolunu Ayırma (fastcgisplitpath_info)

```
fastcgi_split_path_info ^(.+\.php)(/.+)$;
```

- PHP dosya yolunu iki parçaya ayırır:
 - **Gerçek PHP dosyası** (örneğin: `/var/www/html/index.php`).
 - **Eğer varsa, PHP betiğine geçirilen ek path bilgisi** (örneğin: `/var/www/html/index.php/foo`).
- Bu, WordPress gibi framework'lerin düzgün çalışmasını sağlar.

3. PHP-FPM'ye Yönlendirme

```
fastcgi_pass wordpress:9000;
```

- **fastcgi_pass** direktifi, PHP dosyalarının FastCGI sunucusuna (PHP-FPM) yönlendirilmesini sağlar.
- **wordpress:9000**, Docker kullanıldığındá **wordpress adlı bir konteyner** içinde çalışan PHP-FPM servisine yönlendirme yapıldığını gösterir.

4. Varsayılan PHP Dosyası (index.php)

```
fastcgi_index index.php;
```

- Eğer bir dizin çağrıldığında doğrudan bir PHP dosyası belirtilmemişse, **varsayılan olarak index.php çalıştırılır**.
- Örneğin, kullanıcı `https://mehmyilm.42.fr/blog/` adresine gittiğinde, `/blog/index.php` dosyası çalıştırılır.

5. FastCGI Parametrelerini Dahil Etme

```
include fastcgi_params;
```

- **fastcgi_params** dosyası, FastCGI'ye iletilecek temel HTTP başlıklarını ve ortam değişkenlerini içerir.
- Örneğin:
- **REMOTE_ADDR**: Kullanıcının IP adresi.
- **REQUEST_METHOD**: GET veya POST isteği olup olmadığı.
- **CONTENT_TYPE**: Gonderilen içeriğin türü (JSON, XML, vb.).

6. Çalıştırılacak PHP Dosyasını Belirleme

```
fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
```

- Bu parametre, PHP-FPM'nin **hangi dosyanın çalıştırılacağını** belirler.
- **\$document_root** → `/var/www/html`
- **\$fastcgi_script_name** → `index.php`

Sonuç:

```
/var/www/html/index.php
```

PHP-FPM, bu dosyayı çalıştırarak istemciye dönen çıktıyı oluşturur.

7. PHP Betik Adını Tanımlama

```
fastcgi_param SCRIPT_NAME $fastcgi_script_name;
```

- Bu değişken, PHP betiğiinin URL'de nasıl görüneceğini belirler.
- Örneğin, bir kullanıcı <https://mehmyilm.42.fr/contact.php> adresine giderse, `SCRIPT_NAME` değeri `/contact.php` olacaktır.

PHP (Personal Home Page)

<https://www.natro.com/blog/php-nedir-ne-icin-kullanilir-nasil-calisir/>

PHP (Hypertext Preprocessor), dinamik web siteleri ve uygulamalar geliştirmek için kullanılan, sunucu tarafında çalışan bir programlama dilidir. HTML ile birlikte kullanılarak sayfaların içeriğini dinamik olarak oluşturabilir ve güncelleyebilir. Veritabanlarıyla (MySQL, PostgreSQL vb.) kolayca iletişim kurarak veri ekleme, silme, güncelleme gibi işlemleri yapabilir. WordPress, Laravel gibi popüler sistemler PHP ile geliştirilmiştir. Esnek yapısı, geniş kütüphane desteği ve hızlı çalışması sayesinde, kullanıcı giriş sistemlerinden e-ticaret sitelerine kadar birçok alanda yaygın olarak tercih edilir.

PHP-FPM (FastCGI (Fast Common Gateway Interface) Process Manager)

Bu medium yazısında PHP-FPM ve CGI güzel açıklanmış
<https://medium.com/@cemaytan/daha-yak%C4%B1ndan-php-fpm-5ae7afd88e2c>

FastCGI Nedir ve Neden Kullanılır?

FastCGI, web sunucuları ile uygulama sunucuları arasındaki iletişimini hızlandıran ve daha verimli hale getiren bir protokoldür. Geleneksel CGI (Common Gateway Interface) ile karşılaşıldığında, **daha az sistem kaynağı tüketir ve performansı önemli ölçüde artırır.**

CGI ile FastCGI Arasındaki Farklar:

CGI (Common Gateway Interface):

- Her HTTP isteği için **yeni bir işlem (process) başlatır** ve tamamlandığında süreci sonlandırır.
- Bu, fazla yük altında **sunucunun yavaşlamasına** neden olabilir.

FastCGI:

- Tek seferde başlatılan **uzun ömürlü bir işlem havuzu kullanır.**
- Sunucu **sürekli yeni işlemler başlatıp kapatmaz**, bu da CPU ve bellek kullanımını azaltır.

- **Aynı anda birden fazla isteği** verimli şekilde işleyebilir.

PHP çalıştırılan web sunucularında **PHP-FPM (PHP FastCGI Process Manager)** kullanılır. PHP-FPM, PHP betiklerini FastCGI üzerinden çalıştırarak **yüksek trafikli sitelerde performansı artırır**.

FastCGI Nasıl Çalışır?

FastCGI, web sunucuları ile uygulama sunucuları arasında hızlı bir iletişim sağlamak için kullanılan bir protokoldür. İşleyişi şu adımlarla özetlenebilir:

İstek Alma:

- Web sunucusu (örneğin NGINX), istemciden (tarayıcı) gelen HTTP isteklerini alır.
- Eğer istek bir PHP dosyasına yönlendirilmişse, web sunucusu bu isteği FastCGI uygulamasına (PHP-FPM) yönlendirir.

Yol Ayrımı:

- `fastcgi_split_path_info` direktifi, istek yolunu iki parçaaya ayırır:
 - Gerçek PHP dosyası.
 - Ek yol bilgisi (varsayılan).

FastCGI Sunucusuna İletim:

- Web sunucusu, `fastcgi_pass` direktifi ile istekleri PHP-FPM sunucusuna iletir. Bu, ya bir Docker konteyneri ya da yerel bir soket üzerinden yapılabilir.

Parametrelerin Gönderilmesi:

- `include fastcgi_params` ile temel HTTP başlıklarını ve ortam değişkenlerini FastCGI uygulamasına gönderilir.

PHP Betığının Çalıştırılması:

- PHP-FPM, `SCRIPT_FILENAME` ve `SCRIPT_NAME` gibi parametreleri kullanarak doğru PHP betığını çalıştırır.

Yanıtın İletilmesi:

- PHP-FPM, işlenen isteğe göre bir yanıt oluşturur ve bu yanıtı web sunucusuna geri gönderir.
- Web sunucusu, PHP-FPM'den aldığı yanıt istemciye (tarayıcıya) iletir.

WordPress

wordpress normal kurulumu :

<https://www.youtube.com/watch?v=7yU7jRTUPS4&t=319s>



Below you should enter your database connection details. If you're not sure about these, contact your host.

Database Name	<input type="text" value="wordpress"/>	The name of the database you want to use with WordPress.
Username	<input type="text" value="username"/>	Your database username.
Password	<input type="text" value="password"/>	Your database password.
Database Host	<input type="text" value="localhost"/>	You should be able to get this info from your web host, if localhost doesn't work.
Table Prefix	<input type="text" value="wp_"/>	If you want to run multiple WordPress installations in a single database, change this.

normalde bu şekilde kurulur ama biz docker ile yapacağımız için make attığımız andan itibaren her şeyin otomatik olarak yapılması gerek bu yuzden burda olan işlemleri WordPress CLI üzerinden komutlarla yapacağız. Container ayağa kalktığında da WordPress kurulmuş ve kullanılmaya hazır olacak.

WordPress Dockerfile

Bu dosya, projemizin WordPress servisini yapılandırıyor ve otomatik kurulum sürecini yönetiyor.

Gerekli Paketlerin Kurulumu

```
RUN apt-get update && apt-get -y install \
wget \
curl \
bash \
php \
php-cgi \
php-mysql \
php-fpm \
php-pdo \
php-gd php-cli \
php-mbstring \
```

```
&& rm -rf /var/lib/apt/lists/*
```

Bu bölümde WordPress'in çalışması için gereken tüm paketleri yükliyorum:

- **wget** ve **curl**: WordPress dosyalarını indirmek için kullanıyorum aynı zamanda indirme ve kurulumu WordPress CLI üzerinde yapmak için Cli indirmesi için de kullanacağım.
- **bash**: Script'lerimi çalıştırırmak için gerekli instaler.sh
- **php** ve modülleri: WordPress PHP tabanlı olduğu için bunlar olmazsa olmaz:
- **php-fpm**: FastCGI Process Manager, web sunucusuyla PHP arasındaki bağlantıyı sağlıyor
- **php-mysql**: MySQL veritabanına bağlanmak için
- **php-mbstring** ve diğerleri: WordPress'in çeşitli özellikleri için gerekli

Son kısımda apt önbelleğini temizliyorum ki container boyutu daha küçük olsun. Kullanılmayan ve önbelleğe alınan gereksiz dosyalar.

WP-CLI Kurulumu

RUN curl -O

```
https://raw.githubusercontent.com/wp-cli/builds/gh-pages/phar/wp-cli.phar \
```

```
&& chmod +x wp-cli.phar \
```

```
&& mv wp-cli.phar /usr/local/bin/wp
```

WP-CLI, WordPress'i komut satırından yönetmemi sağlayan bir araç. Bu komutlarla önce indiriyorum, çalıştırılabilir yapıyorum ve sistem genelinde erişilebilir olması için **/usr/local/bin/wp** konumuna taşıyorum. Bu sayede WordPress kurulumunu otomatikleştiriyorum. Yani manuel oalrak yapaacağımız her şeyi instaler.sh dosyasında komutlarla yapıp wordpresi hazır hale getireceğiz.

PHP-FPM Yapılandırması

```
COPY ./conf/www.conf /etc/php/7.4/fpm/pool.d/
```

PHP-FPM için özel yapılandırma dosyamı container'a kopyalıyorum. Bu dosyada PHP-FPM'in nasıl çalışacağını, kaç işlem başlatacağını, hangi kullanıcı izinleriyle çalışacağını ve performans ayarlarını belirliyorum. Buna sonraki yazında değineceğiz detaylı olarak.

Socket Dizini Oluşturma

RUN mkdir /run/php

PHP-FPM'in socket dosyasını oluşturacağı dizini hazırlıyorum. Bu socket dosyası, web sunucusu (Nginx) ile PHP-FPM arasındaki iletişimini sağlayacak.Peki soket ne ? Buna da aşağıda detaylı olarak değineceğiz. **Wordpress bolumunden sonra socketleri ele alacağız.**

Kurulum Script'ini Hazırlama

```
COPY ./tools/WordPressInstaller.sh /usr/local/bin/
```

RUN chmod +x /usr/local/bin/WordPressInstaller.sh

WordPress kurulumunu otomatikleştiren script'imi container'a kopyalıyorum ve çalıştırılabilir yapıyorum. Bu script, container her başladığında WordPress'in doğru şekilde kurulu olup olmadığını kontrol edecek.

Container Başlangıç Noktası

ENTRYPOINT ["/usr/local/bin/WordPressInstaller.sh"]

Container başladığında ilk olarak bu script'in çalışmasını sağlıyorum. Böylece WordPress otomatik olarak kurulacak veya var olan kurulum kontrol edilecek.

Çalışma Dizini

WORKDIR /var/www/html/

Container içindeki varsayılan çalışma dizinini WordPress dosyalarının bulunacağı klasör olarak ayarlıyorum.

Port Açıma

EXPOSE 9000

PHP-FPM'in dinleyeceği 9000 numaralı portu dışarıya açıyorum. Web sunucusu (Nginx) bu port üzerinden PHP-FPM ile iletişim kuracak. Daha önce de soylediğim gibi expose tamamıyla sembolik port işlemlerini conf dosyalarında ve .yml dosyalarında yapacağız.

Container Ana Süreci

CMD ["/usr/sbin/php-fpm7.4", "-F"]

Container'ın ana görevi olarak PHP-FPM'i başlatıyorum. **-F** parametresi sayesinde PHP-FPM ön planda çalışıyor, böylece container beklenmedik şekilde kapanmıyor ve logları doğrudan görebiliyorum.

WordPress Kurulum Script'i (WordPressInstaller.sh)

WordPressInstaller.sh script'ımız, Docker container'ımız başlatıldığında otomatik olarak çalışan ve WordPress'in kurulumunu ve yapılandırmasını gerçekleştiren önemli bir bileşendir. Bu script sayesinde WordPress'i her seferinde manuel olarak kurmak yerine, container başlatıldığında otomatik olarak hazır hale gelir.

Başlangıç Ayarları

#!/bin/sh

set -e

İlk satır (`#!/bin/sh`), bu dosyanın `/bin/sh` kabuğu ile çalıştırılması gerektiğini sisteme bildiriyor. `/bin/sh`, Debian sistemlerinde genellikle `bash` kabuğu na sembolik link olduğundan, `bash`'e göre daha hızlı ve hafif bir kabuk kullanmış oluyoruz.

İkinci satırdaki `set -e` komutu, script içindeki herhangi bir komut sıfırdan farklı bir çıkış kodu (yani hata) döndürdüğünde, script'in anında durmasını sağlıyor. Bu, olası sorunları erken tespit etmemize yardımcı oluyor. Örneğin, veritabanı kullanıcısı oluşturma aşamasında bir hata olursa, script ilerlemeyecek ve böylece hatalı bir container oluşumunu engellemiş olacağız.

Bunlara bir önceki yazılar da değinmemiştir burda belirtiyim dedim.

Veritabanı Bilgilerinin Alınması

```
MYSQL_PASSWORD=$(cat /run/secrets/db_password)
MYSQL_ROOT_PASSWORD=$(cat /run/secrets/db_root_password)
WORDPRESS_DB_USER=$(cat /run/secrets/db_user)
```

Burada Docker Swarm veya Kubernetes ortamlarında kullanılan "secrets" mekanizmasını kullanıyoruz. Veritabanı şifresi, root şifresi ve kullanıcı adı gibi hassas bilgileri container dosya sisteminde saklanmış secret'lardan okuyoruz. Bu, güvenlik açısından önemli bir yaklaşım:

- Şifreler kod içinde hardcoded olarak tutulmuyor
- Şifreler environment variable olarak saklanmıyor (daha az güvenli olurdu)
- Sadece yetkili kullanıcılar bu secret'lara erişebiliyor

WordPress Kurulumunun Kontrolü

```
echo "WordPress Downloading"
if [ -f ./wp-config.php ]; then
    echo "WordPress already downloaded"
    exec "$@"
fi
```

Script önce WordPress'in zaten kurulu olup olmadığını kontrol ediyor. Eğer `wp-config.php` dosyası mevcutsa, WordPress zaten kurulu demektir. Bu durumda sadece bir bilgi mesajı yazdırıp `exec "$@"` komutu ile container'ın ana komutunu (PHP-FPM'i) çalıştırıyoruz. Bu kontrol, container'ı yeniden başlattığımızda gereksiz yere WordPress'i tekrar kurmamak için önemli bir optimizasyon.

WordPress Dosyalarının Temizlenmesi

```
if [ -d ./wp-admin ] || [ -d ./wp-content ] || [ -d ./wp-includes ]; then
    echo "WordPress files already exist. Skipping extraction."
    rm -rf ./wp-admin ./wp-content ./wp-includes
fi
```

Bu bölümde, WordPress'in ana dizinlerinden herhangi bir şey varsa temizliyoruz. Bu kontrol, önceki kurulumun yarı kalmış olabileceği durumlar için bir güvenlik önlemi. Dosyaları silip temiz bir kurulum sağlıyoruz.

WordPress'in İndirilmesi ve Çıkarılması

```
echo "Downloading WordPress..."  
wget -q http://wordpress.org/latest.tar.gz  
echo "Extracting WordPress..."  
tar xfz latest.tar.gz  
mv wordpress/* .  
rm -rf latest.tar.gz  
rm -rf wordpress
```

WordPress'in en son sürümünü resmi siteden indiriyoruz. `-q` parametresi ile wget'in çıktılarını susturuyoruz (quiet mode). İndirilen tar.gz dosyasını açıp, içindeki tüm dosyaları mevcut dizine taşıyoruz. İşlem tamamlandıktan sonra artık ihtiyacımız olmayan arşiv dosyasını ve geçici "wordpress" dizinini siliyoruz. Böylece disk alanını optimize etmiş oluyoruz.

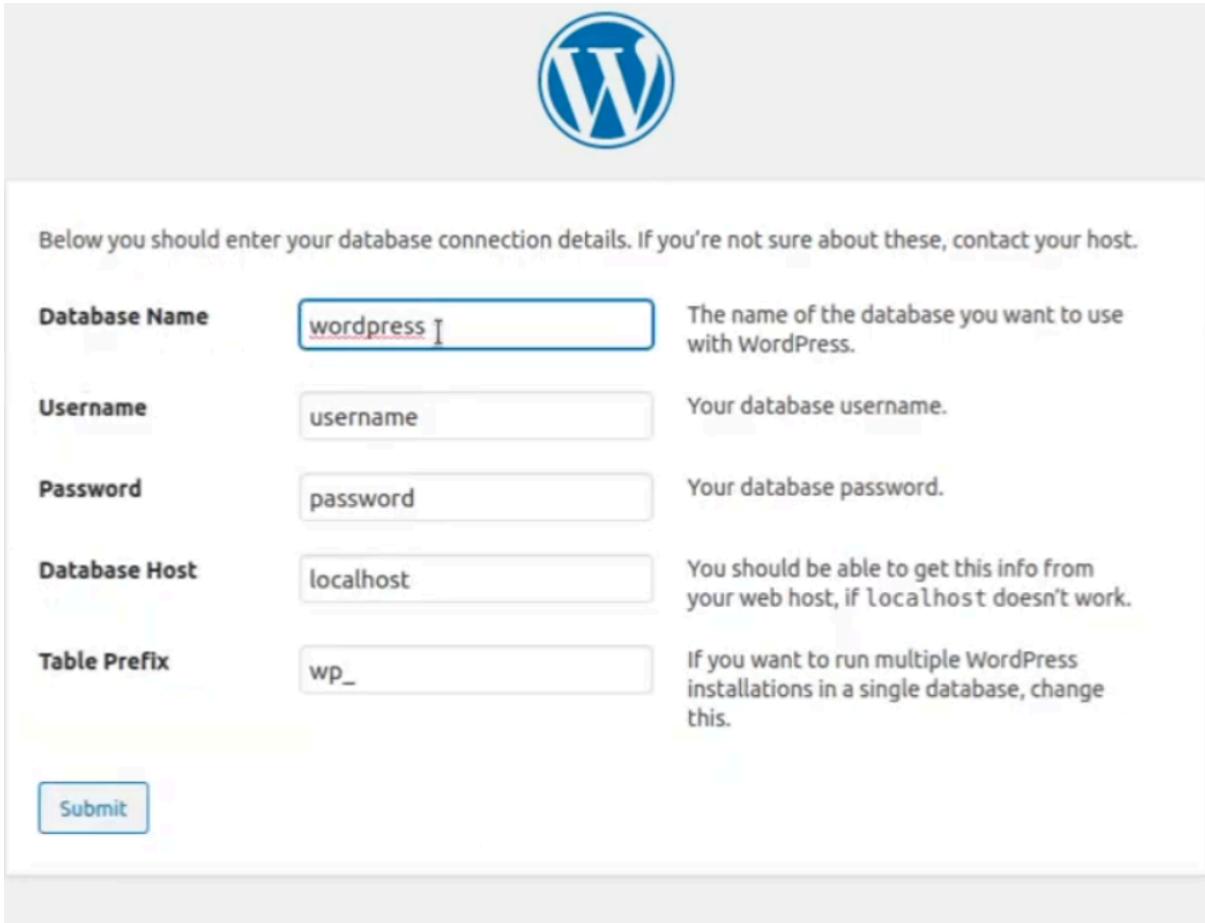
WP-CLI Kontrolü ve Kurulumu

```
if ! command -v wp &> /dev/null; then  
    echo "WP-CLI not found. Installing..."  
    curl -O https://raw.githubusercontent.com/wp-cli/builds/gh-pages/phar/wp-cli.phar  
    chmod +x wp-cli.phar  
    mv wp-cli.phar /usr/local/bin/wp  
    php /usr/local/bin/wp --info  
    if [ $? -ne 0 ]; then  
        echo "Error: WP-CLI installation failed."  
        exit 1  
    fi  
fi
```

Bu bölümde, WP-CLI'ın kurulu olup olmadığını kontrol ediyoruz. Eğer kurulu değilse, GitHub'dan indirip çalıştırılabilir hale getiriyoruz. Kurulum başarılı olup olmadığını test etmek için `wp --info` komutunu çalıştırıyoruz. Eğer komut başarısız olursa (`-ne = "not equal"` yani "eşit değilse"), hata mesajı gösterip script'i sonlandırıyor. Bu kontrol, WP-CLI'ın düzgün çalıştığından emin olmamızı sağlıyor.

WordPress Yapılandırması

```
wp config create --dbname=$WORDPRESS_DB_NAME  
--dbuser=$WORDPRESS_DB_USER --dbpass=$MYSQL_PASSWORD  
--dbhost=$WORDPRESS_DB_HOST --skip-check --allow-root
```



The image shows a screenshot of the WordPress database connection setup page. At the top is the classic blue 'W' WordPress logo. Below it, a message reads: "Below you should enter your database connection details. If you're not sure about these, contact your host." There are five input fields with their respective descriptions:

Setting	Value	Description
Database Name	wordpress	The name of the database you want to use with WordPress.
Username	username	Your database username.
Password	password	Your database password.
Database Host	localhost	You should be able to get this info from your web host, if localhost doesn't work.
Table Prefix	wp_	If you want to run multiple WordPress installations in a single database, change this.

At the bottom left is a blue "Submit" button.

WP-CLI kullanarak WordPress'in veritabanı yapılandırmasını oluşturuyoruz. Bu komut `wp-config.php` dosyasını otomatik olarak yaratıyor ve veritabanı bağlantı ayarlarını içine yazıyor. `--skip-check` parametresi ile veritabanı bağlantısının kontrolünü atlıyoruz (veritabanı servisi henüz hazır olmayı bekliyor). `--allow-root` parametresi ise root kullanıcısı olarak çalıştırabilmemizi sağlıyor.

WordPress Kurulumu

```
wp core install --url="https://mehmyilm.42.fr" --title="My WordPress Site"  
--admin_user="menasy" --admin_password=$MYSQL_ROOT_PASSWORD  
--admin_email="mehmyilm@example.com" --allow-root
```

Bu komut, WordPress'in ilk kurulumunu gerçekleştiriyor:

- Site URL'ini ayarlıyoruz (<https://mehmyilm.42.fr>)
- Site başlığını belirliyoruz ("My WordPress Site")
- Admin kullanıcısını oluşturuyoruz (kullanıcı adı: menasy)
- Admin şifresini ayarlıyoruz (MYSQLROOTPASSWORD değişkeninden alıyoruz)
- Admin e-postasını belirtiyoruz

Site Ayarları

```
wp option update blogname "My WordPress Site" --allow-root  
wp option update blogdescription "Just another WordPress site" --allow-root
```

WordPress site başlığını ve açıklamasını güncelliyoruz. Bu ayarlar WordPress admin panelinden de değiştirilebilir, ancak script ile otomatik olarak yapılandırmak başlangıç için tutarlı bir durum sağlıyor.

Script'in Sonlandırılması

```
exec "$@"
```

Tüm kurulum işlemleri tamamlandıktan sonra, container'ın asıl çalıştırması gereken komutu (CMD direktifinde belirtilen komutu) çalıştırıyoruz. Bu komut, container'ın Dockerfile'ında belirtilen PHP-FPM'i başlatacak. exec kullanarak script'in process'ini PHP-FPM process'i ile değiştiriyoruz, böylece container'ın ana süreci PHP-FPM oluyor.

Bu script, WordPress'in otomatik kurulumunu ve yapılandımasını sağlayarak, container'ın her başlatılışında tutarlı bir çalışma ortamı oluşturuyor. Docker ortamında servis keşfi ve container'ların sırayla başlatılması gibi konuları da dikkate alarak tasarlanmıştır. Veritabanı bilgilerini güvenli bir şekilde alıyor, mevcut kurulumları kontrol ediyor ve WordPress'i tam olarak istediğimiz şekilde yapılandırıyor. Bu tür otomatikleştirme script'leri, mikroservis mimarisinin temel taşılarından biridir ve container'ların "stateless" (durumsuz) olmasını, böylece herhangi bir zamanda güvenle yeniden oluşturulabilmesini sağlar.

WordPress PHP-FPM www.conf Dosyası

www.conf dosyamız, PHP-FPM servisimizin nasıl çalışacağını belirleyen temel yapılandırma dosyasıdır. Bu dosya sayesinde PHP scriptlerimizin web server tarafından nasıl işleneceğini, kaç process kullanılacağını ve sistem kaynaklarımızın nasıl yönetileceğini kontrol edebiliyoruz.

Kullanıcı ve Grup Ayarları

```
user = www-data  
group = www-data
```

PHP-FPM process'lerimizin hangi sistem kullanıcısı ve grubu altında çalışacağını belirliyoruz. www-data kullanıcısı ve grubunu tercih etmemizin nedeni, bu kullanıcının web servislerinin standart kullanıcı olması ve güvenlik açısından sınırlı yetkilere sahip olmasıdır. Böylece PHP kodlarımız, root yetkisi olmadan çalışarak olası güvenlik risklerini azaltıyoruz.

Dinleme (Listen) Ayarı

`listen = wordpress:9000`

Burada PHP-FPM'in hangi adres ve portu dinleyeceğini ayarlıyoruz. Docker ortamımızda `wordpress` servis adını ve `9000` portunu belirtiyoruz. Docker Compose veya Swarm kullanarak, servisler arası iletişim kolaylaştırıyoruz. NGINX veya Apache gibi web sunucularımız, PHP dosyalarını çalıştırmak için bu adres ve porta bağlanacaklar.

Process Manager Ayarları

`pm = dynamic`

Process Manager modunu "dynamic" olarak ayarlıyoruz. Bu, sistemimizin gelen isteklere göre process sayısını otomatik olarak ayarlamasını sağlıyor. Böylece düşük trafikte kaynak tasarrufu yaparken, yoğun trafikte de performansımızı koruyoruz.

Process Sayısı Ayarları

`pm.start_servers = 6`
`pm.max_children = 25`
`pm.min_spare_servers = 2`
`pm.max_spare_servers = 10`

Bu bölüm, PHP-FPM'in process yönetiminin kalbini oluşturuyor ve özel bir açıklama hak ediyor:

pm.start_servers = 6

PHP-FPM servisimiz başlatıldığında hemen 6 adet worker process oluşturuyoruz. Bu değer, ilk başlatma sırasında anında hizmet verebilmek için önemli. Eğer sitemiz genellikle orta düzey bir trafik alıyorsa, 6 process başlangıç için ideal. Sistem başladığında 6 process hazır bekleyeceği için, ilk kullanıcılar herhangi bir gecikme yaşamadan hizmet alabilir.

pm.max_children = 25

Bu parametre belki de en kritik olanı - aynı anda çalışabilecek maksimum PHP-FPM worker process sayısını 25 ile sınırlıyoruz. Bu sınır sayesinde:

- Sunucumuzun RAM kaynaklarını koruyoruz (her PHP process belirli miktarda RAM kullanır)
- CPU kullanımını kontrol altında tutuyoruz
- Sistemin aşırı yüklenmesini ve çökmesini engelliyoruz

25 değeri orta ölçekli bir WordPress sitesi için makul bir değer. Örneğin, her process 50MB RAM kullanıyorsa, maksimum 1.25GB RAM tüketilecektir.

Eğer sunucumuzda daha fazla RAM varsa ve yoğun trafik alıyorsak, bu değeri 50 veya 100'e çıkarabiliriz. Ancak bu değeri artırırken, her process'in kullandığı RAM miktarını hesaba katmalıyız. Örneğin:

- 2GB RAM'li sistem: 25-30 process makul olabilir

- 4GB RAM'li sistem: 50-60 process düşünülebilir
- 8GB RAM'li sistem: 100 veya daha fazla process uygundur

pm.min_spare_servers = 2

Boşta bekleyen (idle) minimum process sayısını 2 olarak belirliyoruz. Bu, trafik düşük olsa bile sistemin her zaman en az 2 process'i hazır tutacağı anlamına gelir. Böylece ani trafik artışlarında ilk istekleri hemen karşılayabiliyoruz. Bu değeri çok yüksek tutmak gereksiz RAM kullanımına, çok düşük tutmak ise ani trafik artışlarında gecikmelere neden olabilir.

pm.max_spare_servers = 10

Boşta bekleyebilecek maksimum process sayısını 10 ile sınırlıyoruz. Yani, trafik düştüğünde sistem en fazla 10 process'i boşta tutacak, gerisi kapatılacak. Bu sayede trafik azaldığında sistem kaynaklarını geri kazanıyor, ancak makul sayıda process'i de hazır tutarak olası trafik artışlarına hazırlıklı oluyoruz.

Bu değerlerin birlikte nasıl çalıştığını dair bir senaryo düşünelim:

1. Sistem başlangıçta 6 process ile başlar ([pm.start_servers](#))
2. Yoğun bir dönemde, process sayısı 25'e kadar çıkabilir ([pm.max_children](#))
3. Trafik azaldığında, sistem boşta kalan process'leri kapatmaya başlar, ancak minimum 2 process'i her zaman hazır tutar ([pm.min_spare_servers](#))
4. Trafik tamamen düştüğünde bile, sistem en fazla 10 process'i boşta tutar ([pm.max_spare_servers](#))

Bu yapılandırma sayesinde sistemimiz dinamik olarak ölçeklenebilir, hem düşük hem de yüksek trafikte verimli çalışabilir.

Ozetle:

www.conf dosyamızdaki bu ayarlar, WordPress uygulamamızın performansını doğrudan etkiliyor. Dynamic process yönetimi modeli ve belirlediğimiz değerler, kaynakları verimli kullanırken iyi bir performans sunuyor. Sunucumuzun özelliklerine ve uygulamamızın trafik modellerine göre bu değerleri zaman içinde optimize edebiliriz. Özellikle pm.max_children değerini belirlerken sunucu RAM'ini ve her PHP process'in ne kadar bellek kullanacağını hesaba katmalıyız. WordPress eklentilerimizin ve temanın karmaşıklığına göre bu değer 50MB ile 150MB arasında değişebilir. Sonuç olarak, bu ayarlar ile WordPress sitemizin PHP tarafının verimli çalışmasını sağlıyoruz, böylece kullanıcılarımıza hızlı ve kesintisiz bir deneyim sunabiliyoruz.

Mariadb ve diğer kısımlaraa geçmeden yukarıda da bahsettiğimiz pid dosyası ve socket kavramlarını da anlayalım.

Socket Kavramı ve İşleyişi

Socket nedir UNIX socket dosyası:

<https://www.howtogeek.com/devops/what-are-unix-sockets-and-how-do-they-work/>

Socket nedir ? Ne İşe Yarar ?

<https://medium.com/@gokhansengun/unix-domain-socket-nedir-ve-ne-i%C5%9Fe-yarar-c72fe8decb30>

Socketler, bilgisayar dünyasında programların birbirile haberleşmesini sağlayan sanal iletişim kanallarıdır. İnsanların telefon kullanarak birbirile konuşması gibi, bilgisayar programları da socketler aracılığıyla veri alışverişi yaparlar.

Socketlerin Temel Mantığı

Socket, aslında bir veri kanalının iki ucunu temsil eder. Bir ucta veri gönderen program, diğer ucta ise bu veriyi alan program bulunur. Veri aktarımının gerçekleşmesi için önce bu kanalın kurulması gereklidir.

Socketler, ağ programlamasında soyutlama sağlar. Yani programcı, alttaki karmaşık ağ detaylarıyla uğraşmadan, basit bir arayüz kullanarak programlar arası iletişim kurabilir. Bir socket oluşturduğunuzda, işletim sistemi size bir (file descriptor) verir ve tüm iletişimiminizi bu tanımlayıcı üzerinden yürütürsünüz.

Socket İletişiminin Adımları

Socket iletişim genellikle şu adımlarda gerçekleşir:

- Socket Oluşturma:** Hem istemci hem de sunucu bir socket nesnesi oluşturur.
- Adres Bağlama (Binding):** Sunucu, socketini belirli bir adrese ve port numarasına bağlar.
- Dinleme (Listening):** Sunucu, bağlantı taleplerini dinlemeye başlar.
- Bağlantı İsteği (Connecting):** İstemci, sunucuya bağlantı isteği gönderir.
- Bağlantı Kabulü (Accepting):** Sunucu, gelen bağlantı isteğini kabul eder.
- Veri Transferi:** İki taraf arasında veri alışverişi başlar.
- Bağlantı Kapanışı:** İşlem bitince socket bağlantısı kapatılır.

Socket Adresleme

Her socket, bir adres bilgisi ile tanımlanır. Bu adres, kullanılan socket türüne göre değişiklik gösterir:

- **TCP/IP Socketleri:** IP adresi ve port numarası çifti ile tanımlanır (örn. 192.168.1.100:8080).
- **UNIX Domain Socketleri:** Dosya sistemi yolu ile tanımlanır (örn. /tmp/my_socket)

Detaylı TCP Socket İletişim Örneği

Bir web tarayıcısı ve web sunucusu arasındaki iletişimini adım adım inceleyelim:

1. Web Sunucusu Socket Hazırlığı:

- Sunucu bir socket oluşturur (`socket()` çağrıları).
- Bu socketi 80 numaralı porta bağlar (`bind()` çağrıları).
- Socket'i dinleme moduna alır (`listen()` çağrıları).
- Bağlantı isteklerini bekler (`accept()` çağrıları).

1. Web Tarayıcısı Bağlantı İsteği:

- Tarayıcı bir socket oluşturur.
- Sunucunun IP adresi ve 80 portu ile bağlantı kurar (`connect()` çağrıları).

1. Bağlantı Kurulumu:

- Sunucunun `accept()` çağrıları, yeni bir socket döndürür. Bu socket, tarayıcı ile iletişim için kullanılır.
- Artık iki program arasında bir iletişim kanalı kurulmuştur.

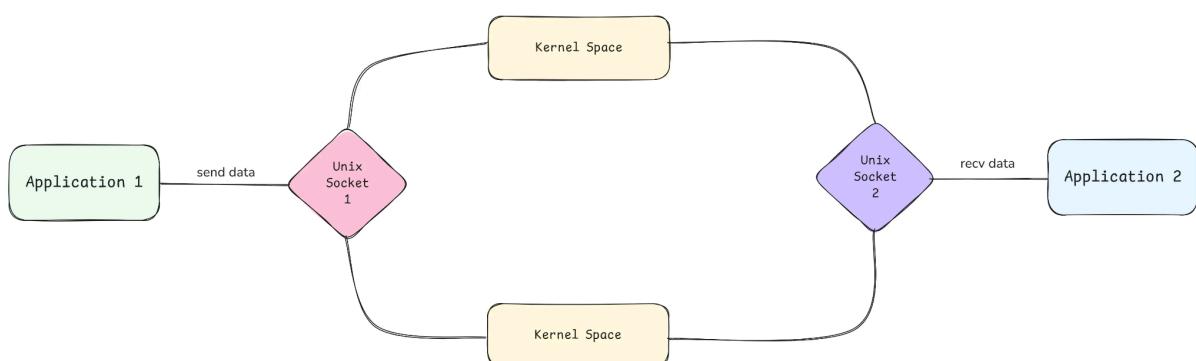
1. Veri Transferi:

- Tarayıcı, HTTP isteğini sunucuya gönderir (`send()` veya `write()` çağrıları).
- Sunucu, isteği okur (`recv()` veya `read()` çağrıları).
- Sunucu, istenen web sayfasını hazırlar ve tarayıcıya gönderir.
- Tarayıcı, sunucudan gelen yanıt okur ve ekranda görüntüler.

1. Bağlantı Sonlandırma:

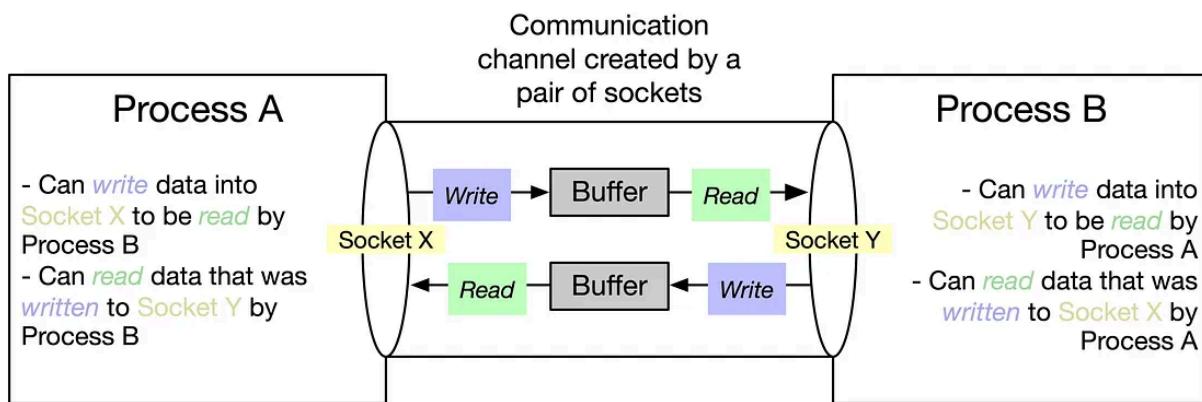
- İşlem tamamlandığında her iki taraf da socketlerini kapatır (`close()` çağrıları).

UNIX Domain Socketleri



UNIX Domain Socketleri (UNIX socketleri olarak da bilinir), aynı bilgisayar üzerindeki süreçler arasında iletişim için kullanılan özel bir socket türüdür. Ağ protokollerini kullanmadan doğrudan işletim sistemi çekirdeği üzerinden iletişim sağlarlar.

UNIX Socketlerinin Çalışma Prensibi



UNIX socketleri, IP adresi ve port numarası yerine dosya sistemi yolu kullanır. Bir UNIX socket oluşturulduğunda, bu socket dosya sisteminde bir dosya olarak görünür (örneğin: `/tmp/my_socket`). Bu dosya özel bir dosya türüdür ve normal bir metin dosyası gibi okunup yazılamaz.

UNIX socketleri, veri transferi için ağ protokol yığını kullanmaz. Bunun yerine, çekirdek içinde doğrudan bellek kopyalaması yoluyla veri aktarımı gerçekleştirir. Bu nedenle, aynı makinedeki süreçler arasında iletişim için TCP/IP socketlerine göre çok daha hızlıdır.

UNIX Socketlerinin Kullanım Alanları

UNIX socketleri yaygın olarak şu alanlarda kullanılır:

- Veritabanı Sunucuları:** MySQL, PostgreSQL gibi veritabanları, yerel istemcilerle iletişim için UNIX socketlerini kullanır.
- Web Sunucuları ve Uygulama Sunucuları:** Nginx veya Apache gibi web sunucuları, PHP-FPM veya uWSGI gibi uygulama sunucularıyla UNIX socketleri üzerinden haberleşebilir.
- Sistem Servisleri:** Systemd gibi sistem yöneticileri, kontrol ettikleri servislerle iletişim için UNIX socketlerini kullanır.
- Docker ve Container Teknolojileri:** Docker daemon, containerlar ile iletişim için UNIX socketlerini kullanır (`/var/run/docker.sock`).

UNIX Socketlerinin Avantajları

- Yüksek Performans:** TCP/IP socketlerine göre 2-3 kat daha hızlı olabilirler çünkü ağ protokol yığını atlanan adımları yoktur.

- Gelişmiş Güvenlik:** UNIX socketleri, dosya sistemi izinleri ile korunur. Bu, hangi kullanıcıların veya grupların sockete erişebileceğini kontrol etmenizi sağlar.
- Kimlik Doğrulama:** UNIX socketleri, bağlanan istemcinin kullanıcı kimliğini (UID) ve grup kimliğini (GID) elde etmenize olanak tanır. Bu özellik, kimlik doğrulama için ek protokollere gerek kalmadan güvenlik sağlar.
- Ek Veri İletimi:** UNIX socketleri, dosya tanımlayıcılarının (file descriptors) ve erişim haklarının süreçler arasında aktarılmasına olanak tanır.

TCP/IP Socketleri ve UNIX Socketleri Arasındaki Farklar

TCP/IP socketleri ile UNIX socketleri arasındaki temel farklar şunlardır:

- Adres Yapısı:**
 - TCP/IP socketleri: IP adresi ve port numarası ile tanımlanır.
 - UNIX socketleri: Dosya sistemi yolu ile tanımlanır.
- İletişim Kapsamı:**
 - TCP/IP socketleri: Farklı bilgisayarlar arasında iletişim sağlar.
 - UNIX socketleri: Sadece aynı bilgisayar üzerindeki süreçler arasında iletişim sağlar.
- Performans:**
 - UNIX socketleri, ağ protokol yığını kullanmadığı için genellikle daha hızlıdır.
 - UNIX socketleri ile veri transferi, TCP/IP socketlerine göre 2-3 kat daha hızlı olabilir.
- Güvenlik:**
 - UNIX socketleri, dosya sistemi izinleri ile korunur.
 - UNIX socketleri, bağlanan istemcinin UID/GID bilgisini sağlayarak kimlik doğrulamayı kolaylaştırır.
- Protokol Desteği:**
 - TCP/IP socketleri: TCP veya UDP protokollerini kullanır.
 - UNIX socketleri: İşletim sistemi çekirdeği üzerinden doğrudan iletişim sağlar.

UNIX socketleri kopyala-yaz ekanızmasıyla çalışır. Peki nedir bu mekanizma ?

Kopyala-Yaz Mekanizması ve UNIX Socketlerindeki Rolü

İşletim sistemleri, süreçler arasında veri paylaşımını yönetirken bellek verimliliğini sağlamak için çeşitli mekanizmalar kullanır. Bunlardan biri de "kopyala-yaz" (copy-on-write) mekanizmasıdır. Bu mekanizma, özellikle UNIX socketleri aracılığıyla gerçekleşen iletişimde önemli bir performans avantajı sağlar.

Bellek Yönetimi ve Veri Paylaşımı

İşletim sistemleri, süreçlere kendi izole sanal bellek alanlarını tahsis eder. Bu sayede her süreç kendi bellek alanına sahip olur ve diğer süreçlerin bellek alanlarına doğrudan erişemez. Bu izolasyon, sistemin güvenliği ve kararlılığı için gereklidir. Ancak süreçler arası iletişim gerektiğinde, bu izolasyon veri paylaşımını zorlaştırbılır. Geleneksel yaklaşımda, bir süreçten diğerine veri gönderildiğinde, verinin tam bir kopyası oluşturulur. Bu yöntem, özellikle büyük miktarda veri söz konusu olduğunda bellek ve işlemci kaynaklarını önemli ölçüde tüketebilir. Kopyala-yaz mekanizması, bu sorunu çözmek için tasarlanmıştır.

Kopyala-Yaz Mekanizmasının Çalışma Prensibi

Kopyala-yaz mekanizması, başlangıçta verilerin fiziki olarak kopyalanması yerine, birden fazla sürecin aynı fiziksel bellek sayfalarını paylaşmasına izin verir. Bu paylaşım "sadece okuma" modunda gerçekleşir. Yani süreçler, veriler üzerinde herhangi bir değişiklik yapmadıkları sürece aynı bellek sayfalarına erişebilirler. Süreçlerden biri paylaşılan veri üzerinde değişiklik yapmaya çalıştığı anda, işletim sistemi otomatik olarak o bellek sayfasının bir kopyasını oluşturur ve değişiklik yapan süreç için bu yeni kopyayı kullanır. Bu sayede diğer süreçler hala orijinal veriyi görmeye devam ederken, değişiklik yapan süreç kendi kopyası üzerinde serbestçe çalışabilir.

UNIX Socketlerinde Kopyala-Yaz Mekanizması

UNIX socketleri, aynı bilgisayar üzerindeki süreçler arasında veri paylaşımı için tasarlanmış özel bir iletişim kanalıdır. Bu socketler, veri transferi sırasında kopyala-yaz mekanizmasından yararlanarak verileri çok daha verimli bir şekilde ileter. Bir süreç, socket aracılığıyla veri gönderdiğinde, bu veriler öncelikle çekirdek alanında bulunan bir tampon belleğe (buffer) yazılır. Bu buffer, hem gönderen hem de alıcı süreç tarafından erişilebilir durumdadır. Alıcı süreç veriyi okumaya başladığında, doğrudan bu buffer üzerinden okuma yapar ve eğer veri üzerinde değişiklik yapmayacaksa, herhangi bir bellek kopyalama işlemi gerçekleşmez. Ancak alıcı süreç okuduğu veri üzerinde değişiklik yapmak isterse, kopyala-yaz mekanizması devreye girer ve ilgili bellek sayfasının bir kopyası oluşturulur. Bu kopya, alıcı sürecin kullanımına sunulur ve değişiklikler bu kopya üzerinde gerçekleştirilir.

Performans ve Verimlilik Avantajları

Kopyala-yaz mekanizması, UNIX socketlerinin veri transferinde önemli performans avantajları sağlar:

- Bellek Verimliliği:** Veriler yalnızca gerektiğinde (yazma işlemi sırasında) kopyalanır, bu da bellek kullanımını optimize eder.
- Hızlı Veri Transferi:** İki süreç arasında veri transferi, fiziksel kopyalama olmadan gerçekleştirilebilir, bu da işlem hızını artırır.

3. **CPU Yükünün Azaltılması:** Gereksiz bellek kopyalama işlemleri olmadığı için, CPU üzerindeki yük azalır.
4. **Büyük Veri Transferlerinde Etkinlik:** Özellikle büyük veri bloklarının transferinde, bu mekanizma sayesinde önemli performans kazanımları elde edilir.

Bu avantajlar, UNIX socketlerini aynı bilgisayar üzerindeki süreçler arasında iletişim için ideal bir seçenek haline getirir. Web sunucuları, veritabanı sistemleri, uygulama sunucuları ve diğer birçok sistem bileşeni, yüksek performanslı yerel iletişim için UNIX socketlerini tercih eder.

Sonuç olarak, kopyala-yaz mekanizması, modern işletim sistemlerinde bellek yönetiminin ve süreçler arası iletişim verimliliğini artıran önemli bir teknolojidir. UNIX socketleri, bu mekanizmayı kullanarak aynı sistemdeki süreçler arasında son derece hızlı ve verimli veri transferi sağlar, bu da sistem performansını ve kaynak kullanımını optimize eder.

MariaDB Veritabanı

Dockerfile

MariaDB Dockerfile'ımız, veritabanı servisimizi oluşturmak için kullanılan temel yapılandırma dosyasıdır. Bu Dockerfile sayesinde MariaDB veritabanı sunucumuzu standart bir şekilde kuruyor, özelleştiriyor ve WordPress uygulamamız için hazır hale getiriyoruz.

MariaDB Kurulumu

```
RUN apt-get update && apt-get install -y mariadb-client mariadb-server \
&& apt-get clean \
&& rm -rf /var/lib/apt/lists/*
```

Bu bölümde MariaDB veritabanı sunucusu ve istemcisini kuruyoruz. Kurulum işlemi tek bir RUN komutu içinde gerçekleştiriliyor, bu Docker'da "layer" sayısını azaltmak için önemli bir best practice. Kullandığımız yöntemler:

- `mariadb-client mariadb-server`: Hem sunucu hem de istemci paketlerini kuruyoruz
- `apt-get clean`: Paket önbelleklerini temizliyoruz
- `rm -rf /var/lib/apt/lists/*`: Paket listelerini siliyoruz

Kurulum sonrası temizlik adımları, Docker imajının boyutunu küçültmek için kritik önem taşıyor. Bu sayede imajımız daha az disk alanı kaplıyor ve daha hızlı dağıtılabiliyor.

Port Açma

```
EXPOSE 3306
```

- Bu, **container içinde MariaDB'nin 3306 portunun kullanılacağını Docker'a bildirir**. Ama dış dünyadan erişim sağlamak için **ekstra bir bağlantı kurulması gereklidir**.
- Eğer MariaDB'yi **docker run komutuyla başlatıyorsak**, şu şekilde port yönlendirmesi yapmamız gereklidir: `docker run -d -p 3306:3306 --name mariadb_container mariadb`
- Biz docker-compose.yml üzerinden yapacağımız için EXPOSE haricinde bu dosyada port kısmında yönlendirmeleri yapacağız.
- Yani EXPOSE sadece bir **bilgilendirme** satırıdır. Docker run -p 3306:3306 komutu kullanıldığında, Dockerfile içinde EXPOSE 3306 yazmaya bile port yönlendirme yapılabilir.

MariaDB'nin standart portu olan 3306'yi dışarıya açıyoruz. Bu satır, containerımızın hangi porttan hizmet vereceğini Docker'a bildiriyor. EXPOSE komutu dokümantasyon amaçlıdır; gerçek port yönlendirmesi Docker Compose veya `docker run` komutunda `-p` parametresi ile yapılır. WordPress container'ımız ve diğer servisler bu port üzerinden veritabanına bağlanacak.

Yapılardırma Dosyasının Kopyalanması

`COPY ./conf/50-server.cnf /etc/mysql/mariadb.conf.d/`

MariaDB için özel yapılandırma dosyamızı (`50-server.cnf`) container içine kopyalıyoruz. Bu dosya, MariaDB'nin nasıl çalışacağını belirleyen ayarları içeriyor: Dosyayı `/etc/mysql/mariadb.conf.d/` dizinine kopyalayarak, MariaDB'nin başlangıçta bu ayarları okumasını sağlıyoruz. Debian/Ubuntu sistemlerinde MariaDB, bu dizindeki tüm `.cnf` dosyalarını otomatik olarak yükler.

Başlangıç Script'inin Kopyalanması ve Yetkilendirilmesi

`COPY ./tools/script.sh /script.sh`

`RUN chmod +x /script.sh`

`script.sh` dosyamızı container'a kopyalıyoruz ve çalıştırılabilir hale getiriyoruz:

1. `COPY` komutu ile host makinemizdeki `./tools/script.sh` dosyasını container içindeki `/script.sh` konumuna kopyalıyoruz
2. `chmod +x` komutu ile dosyaya çalışma yetkisi veriyoruz

Container Başlatıcısı

`ENTRYPOINT ["/script.sh"]`

Container başlatıldığından otomatik olarak çalıştırılacak komutu belirliyoruz. Burada `/script.sh` dosyamızı container'ın ana giriş noktası olarak belirtiyoruz.

MariaDB Script

`script.sh` dosyamız, MariaDB container'ımızın başlangıç noktasını oluşturuyor. Dockerfile'da ENTRYPPOINT olarak belirttiğimiz bu script, container her başlatıldığından otomatik olarak çalışıyor. script'in temel amacı MySQL veritabanımızı güvenli bir şekilde yapılandırmak, kullanıcılar oluşturmak ve WordPress için gerekli veritabanını hazırlamaktır.

```
#!/bin/sh  
set -e
```

güvenli şekilde başlatır diğer scriptlerde olduğu gibi.

Güvenli Şekilde Şifrelerin Alınması

```
DB_ROOT_PASSWORD=$(cat /run/secrets/db_root_password)  
DB_USER=$(cat /run/secrets/db_user)  
DB_PASSWORD=$(cat /run/secrets/db_password)
```

Burada `cat` komutu ile dosya içeriğini okuyup değişkenlere atıyoruz.

`/run/secrets/` dizini, Docker Swarm veya Kubernetes tarafından container içine mount edilen özel bir dizindir. Bu dizindeki dosyalar, orchestration sisteminin yönettiği şifre ve hassas bilgileri içerir. Her bir dosya, bir secret'i temsil eder:

- `/run/secrets/db_root_password` - MySQL root kullanıcısının şifresi
- `/run/secrets/db_user` - WordPress için oluşturacağımız veritabanı kullanıcı adı
- `/run/secrets/db_password` - WordPress kullanıcısının şifresi

MariaDB Servisinin Başlatılması

```
service mariadb start
```

Bu komut, Debian'in servis yönetim sistemi aracılığıyla MariaDB'yi başlatıyor.

`service` komutu, sistem servislerini başlatmak, durdurmak veya yeniden başlatmak için kullanılan standart bir araçtır. Bu komut çalıştığında şunlar gerçekleşir:

1. `/etc/init.d/mariadb` betiği çağrırlır
2. MariaDB'nin konfigürasyon dosyalarından okuma yapılır
3. MySQL daemon'u (`mysqld`) başlatılır
4. Servis PID dosyası oluşturulur
5. Servis başarıyla başlayıp başlamadığını kontrol eden bir durum testi yapılır

Burada MariaDB'yi bir sistem servisi olarak başlatıyoruz çünkü veritabanı ve kullanıcı oluşturma işlemlerini gerçekleştirmek için aktif bir veritabanı bağlantısına ihtiyacımız var. Bu, geçici bir başlatmadır.

Servisin Hazır Olmasının Beklenmesi

```
until mysqladmin ping -h "localhost" --silent; do
    echo "MariaDB is starting..."
    echo "Waiting"
done
```

Bu blok, MariaDB servisinin tamamen başlamasını beklemek için bir döngü oluşturuyor. `until` komutu, verilen koşul doğru olana kadar içindeki komutları tekrar tekrar çalıştırır.

`mysqladmin ping -h "localhost" --silent` komutu MySQL'e bir ping gönderir ve yanıt alabilirse 0 (başarılı) değerini döndürür. `-h "localhost"` parametresi localhost üzerinden bağlanmak istediğimizi belirtir. `--silent` parametresi gereksiz çıktıları engeller.

Bu döngü şöyle çalışır:

1. MariaDB'ye ping gönderilir
2. Eğer ping başarılı olursa (MariaDB hazırlırsa), döngüden çıkarılır
3. Eğer başarısız olursa, bilgilendirme mesajları gösterilir ve döngü tekrarlanır

Container'larda servis başlatma işlemi değişken sürelerde tamamlanabilir. Bazı durumlarda, özellikle yüksek yük altındaki sistemlerde, MariaDB'nin tam olarak başlaması birkaç saniye sürebilir. Bu döngü, script'in MariaDB tam olarak hazır olmadan SQL komutlarına geçmesini engelleyerek olası hataları önler.

Veritabanı ve Kullanıcı Oluşturma

```
mysql -uroot -p"${DB_ROOT_PASSWORD}" <<-EOSQL
CREATE USER IF NOT EXISTS '${DB_USER}'@'%' IDENTIFIED BY
'${DB_PASSWORD}';
GRANT ALL PRIVILEGES ON *.* TO '${DB_USER}'@'%' IDENTIFIED BY
'${DB_PASSWORD}';
GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY
'${DB_ROOT_PASSWORD}';
FLUSH PRIVILEGES;
CREATE DATABASE IF NOT EXISTS ${MYSQL_DATABASE};
EOSQL
```

Bu bölüm, MySQL istemcisi aracılığıyla veritabanı sunucusuna bağlanıp bir dizi SQL komutu çalıştırıyor.

- `mysql -uroot -p"${DB_ROOT_PASSWORD}"`: MySQL komut satırı istemcisini başlatır
 - `-uroot`: Root kullanıcısı olarak bağlan
 - `-p"${DB_ROOT_PASSWORD}"`: Şifreyi komut satırında doğrudan ver (container içinde olduğumuz için güvenlik riski minimum)
- `<<-EOSQL ... EOSQL`: Burada "here-document" (heredoc) kullanıyoruz. Bu yapı, birden çok satırlık metni veya komutları bir programa standart giriş olarak iletmemizi sağlar. `-` işaretini, içerisindeki satırların başındaki tab karakterlerinin kaldırılmasını sağlar (girintileme için önemli).

SQL komutları sunları yapıyor:

```
CREATE USER IF NOT EXISTS '${DB_USER}'@'%' IDENTIFIED BY
'${DB_PASSWORD}';
```

- WordPress için belirtilen kullanıcıyı oluşturur
- `IF NOT EXISTS` ile kullanıcı zaten varsa hata vermez
- `@'%'` ifadesi, bu kullanıcının herhangi bir host'tan bağlanabilmesini sağlar (Docker ağında farklı container'lardan bağlantıya izin vermek için)
- `IDENTIFIED BY` ile kullanıcı şifresini belirler

```
GRANT ALL PRIVILEGES ON *.* TO '${DB_USER}'@'%'
IDENTIFIED BY '${DB_PASSWORD}';
```

- WordPress kullanıcısına tüm veritabanları (ilk `*`) ve tüm tablolar (ikinci `*`) üzerinde tam yetki verir
- Bu, WordPress'in veritabanı oluşturma, tablo oluşturma, veri ekleme/silme/güncelleme gibi tüm işlemleri yapabilmesini sağlar

```
GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY
'${DB_ROOT_PASSWORD}';
```

- Root kullanıcısının da tüm hostlardan erişimine izin verir
- Normalde bu, güvenlik açısından riskli olabilir ancak Docker container'ları içinde kaldığı ve şifre güvenli olduğu sürece kabul edilebilir
- Root kullanıcı varsayılan olarak sadece localhost'tan erişilebilir olduğundan, dışarıdan root bağlantısı için bu adım gereklidir

FLUSH PRIVILEGES;

- Yetki tablolarını yeniler ve değişiklikleri arasında etkili hale getirir
- Bu komut olmadan, bazı durumlarda yeni yetkiler veya kullanıcılar MySQL tarafından tanınmayabilir

```
CREATE DATABASE IF NOT EXISTS ${MYSQL_DATABASE};
```

- WordPress için kullanılacak veritabanını oluşturur

- `IF NOT EXISTS` ile veritabanı zaten varsa hata vermez
- `${MYSQL_DATABASE}` değişkeni environment variable olarak geliyor

Geçici Servisin Durdurulması

`service mariadb stop`

Bu komut, daha önce geçici olarak başlattığımız MariaDB servisini düzgün bir şekilde durduruyor. Service komutu, sistemdeki init scriptlerini kullanarak MariaDB'nin tüm süreçlerini düzgün şekilde sonlandırır:

1. MySQL'e güvenli bir şekilde kapanması için SIGTERM sinyali gönderilir
2. Açık bağlantılar kapatılır
3. İşlemekte olan sorgular tamamlanır
4. Veri dosyaları senkronize edilir
5. PID dosyası kaldırılır

Bu adım çok önemlidir, çünkü bir sonraki adımda MariaDB'yi farklı bir modda başlatacağız. Servisi önceden düzgün bir şekilde kapatmadan yeniden başlatmak, veri bozulmasına veya kilitleme sorunlarına neden olabilir.

Ana MariaDB Sürecinin Başlatılması

`exec mysqld`

Son olarak, `exec` komutu ile mevcut bash sürecini `mysqld` (MySQL daemon) ile değiştiriyoruz. Bu, Docker container'larında kritik bir tekniktir ve şunları sağlar:

- `exec` komutu, mevcut süreci belirtilen programla değiştirir, yeni bir süreç oluşturmaz
- Bu sayede `mysqld`, container'ın PID 1 süreci haline gelir
- PID 1 olması, Docker'ın SIGTERM gibi sinyalleri doğrudan MySQL'e iletmesini sağlar
- Container durdurulduğunda, sinyal doğrudan MySQL'e gider ve düzgün kapanma sağlanır
- İkincil bir süreç olarak çalıştırılmak yerine, doğrudan mysqld çalıştırarak gereksiz süreç katmanlarını ortadan kaldırırız

mysqld komutu, MySQL'in daemon sürecini başlatır ve şunları yapar:

1. Yapılandırma dosyalarını okur (/etc/mysql/mariadb.conf.d/ altındaki dosyalar dahil)
2. Veritabanı dosyalarını açar (/var/lib/mysql altında)
3. Ağ soketlerini oluşturur ve bağlantıları dinlemeye başlar
4. InnoDB ve diğer storage engine'leri başlatır
5. Ön planda çalışmaya devam eder

MariaDB 50-server.cnf Dosyası

`50-server.cnf` (dosyanın 50 şeklinde isimlendirilmesinin sebebi mariadb config dosyaslarını okurken belirli bir sırayla okumasıdır.) bu yapılandırma dosyamız, MariaDB veritabanı sunucumuzun çalışma davranışını belirleyen temel ayarları içeriyor.

/etc/mysql/mariadb.conf.d/ dizini altında bulunan dosyalar alfabetik sıraya göre okunur.

`/etc/mysql/mariadb.conf.d/`

```
| — 10-database.cnf  
| — 50-server.cnf  
| — 90-custom.cnf
```

Debian/Ubuntu sistemlerinde genellikle `/etc/mysql/mariadb.conf.d/` dizini altında bulunur ve MySQL daemon'u başlatıldığında otomatik olarak yüklenir. Bu dosya sayesinde veritabanı sunucumuzun performansını, güvenliğini ve davranışını ihtiyaçlarımıza göre özelleştirebiliyoruz.

Kod Kısmı

Bölüm Tanımlayıcısı

`[mysqld]`

Bu satır, dosyanın `mysqld` daemon'u (MySQL sunucu süreci) için ayarlar içerdigini belirtir. MySQL/MariaDB yapılandırma dosyaları böülümlere ayrılmıştır ve her bölüm köşeli parantezlerle belirtilir. Diğer olası bölümler `[client]`, `[mysql]` veya `[mysqldump]` olabilir. Buradaki ayarların tümü, sadece MySQL sunucu sürecini etkiler, istemcileri etkilemez.

Kullanıcı ve Sistem Dosyaları

<code>user</code>	<code>= mysql</code>
<code>pid-file</code>	<code>= /run/mysqld/mysqld.pid</code>
<code>socket</code>	<code>= /run/mysqld/mysqld.sock</code>
<code>port</code>	<code>= 3306</code>
<code>basedir</code>	<code>= /usr</code>
<code>datadir</code>	<code>= /var/lib/mysql</code>
<code>tmpdir</code>	<code>= /tmp</code>

Bu bölümdeki ayarlar, MySQL sunucusunun çalışması için temel sistem parametrelerini belirler:

- **user = mysql**: MySQL daemon'unun çalışacağı sistem kullanıcısını belirtir. Güvenlik açısından, MySQL'in **root** yerine daha sınırlı yetkilere sahip özel bir kullanıcı (**mysql**) ile çalışmasını sağlarız. Bu kullanıcı, genellikle MySQL kurulumu sırasında otomatik olarak oluşturulur.
- **pid-file = /run/mysqld/mysqld.pid**: MySQL'in process ID'sini (PID) sakladığı dosyanın konumunu belirtir. Bu dosya, sistemin MySQL sürecini tanımı ve yönetmesi için kullanılır. Örneğin, **service mysql stop** komutu bu dosyadan PID'yi okuyarak hangi süreci sonlandıracağını bilir.
 - **Docker ortamında genellikle tek instance çalıştırıldığı için PID dosyaları çoğu zaman gereksizdir. Ancak MySQL gibi bazı uygulamalar, süreç yönetimini daha iyi kontrol etmek için PID dosyası kullanır.**
 - **Nginx conf dosyasında da bunu belirtebilirdik ama zaten kendisi eğer pid dosyası belirtilmezse oluşturuyor. NGINX zaten bir master-worker yapısında çalışır ve kendi süreçlerini içsel olarak yönetir.**
 - **WordPress, kendi başına bir servis değil, PHP üzerinden çalışan bir uygulama olduğu için PID dosyasına ihtiyaç duymaz.**
- **socket = /run/mysqld/mysqld.sock**: Unix soket dosyasının konumunu belirtir. Bu soket, aynı makinedeki MySQL istemcilerinin sunucuya bağlanmak için kullandığı yerel bir iletişim kanalıdır. TCP/IP bağlantısına göre daha hızlıdır ve güvenlidir, çünkü ağ üzerinden geçmez. Bu conf dosyasında iki socket turunu de kullanıyoruz hem port ve ip belirterek network socketini hem de burda verildiği gibi UNIX socketini kullanıyoruz. Bu iki socket turune de [Detaylı olarak yukarıda dejindim.](#)
- **port = 3306**: MySQL'in TCP/IP bağlantıları için dinleyeceği port numarasıdır. 3306, MySQL'in standart portudur. Docker içinde çalıştığımızda, bu port container dışına açılacak ve diğer servisler bu port üzerinden veritabanına bağlanabilecek.
- **basedir = /usr**: MySQL kurulumunun kök dizinini belirtir. Buradan program dosyaları, kütüphaneler ve komut dosyaları yüklenir.
- **datadir = /var/lib/mysql**: Veritabanı dosyalarının saklandığı dizini belirtir. Tüm veritabanları, tablolar ve indeksler bu dizin altında depolanır. Bu dizinin yedeklenmesi, veritabanı yedeklemesi için kritik öneme sahiptir.
- **tmpdir = /tmp**: Geçici dosyaların (sıralama, geçici tablolar vb.) oluşturulacağı dizini belirtir. Büyük sorgular çalıştırıldığında, MySQL bu dizini kullanarak disk üzerinde geçici çalışma alanları oluşturur.

Dil ve Karakter Seti Ayarları

lc-messages-dir = /usr/share/mysql
lc-messages = en_US

Bu ayarlar, MySQL'in hata mesajları ve sistem mesajları için kullanacağı dil ayarlarını belirler:

- `lc-messages-dir = /usr/share/mysql`: Dil dosyalarının bulunduğu dizin.
- `lc-messages = en_US`: Kullanılacak dil. Burada İngilizce (ABD) seçilmiş.

Güvenlik Ayarları

```
skip-external-locking  
bind-address      = 0.0.0.0
```

- **skip-external-locking**: Bu ayar, MySQL'in dosya bazlı kilitleme mekanizmasını devre dışı bırakır. Modern sistemlerde genellikle gerekli olmadığı için performansı artırmak amacıyla kullanılır. MariaDB tek sunuculu çalıştığı için harici kilitlemeye ihtiyaç duymaz ve bu ayar güvenle açılabilir.
- Tek sunuculu veritabanları, tüm veritabanı işlemlerinin tek bir sunucu tarafından yürütüldüğü sistemlerdir. Tüm veriler tek bir makinede tutulur ve erişim sadece buradan sağlanır.
- Çok sunuculu sistemlerde ise veritabanı birden fazla sunucuya dağıtılr. Bu yapı, **yük paylaşımı, yedeklilik ve yüksek erişilebilirlik** sağlar.
- Projemiz tek sunuculu olduğu için **skip-external-locking** ayarının açılması, gereksiz dosya kilitlemelerini önleyerek veritabanının daha verimli çalışmasını sağlar.

`bind-address = 0.0.0.0`: MySQL'in hangi ağ arabirimlerinden bağlantıları dinleyeceğini belirtir. `0.0.0.0` tüm ağ arabirimlerini dinlemek anlamına gelir. Bu, Docker container'da çalışan MariaDB için kritik bir ayardır - dışarıdan bağlantıları kabul etmesi için gereklidir. Güvenlik açısından daha kısıtlayıcı bir yaklaşım olarak `127.0.0.1` kullanılarak sadece yerel bağlantılar kabul edilebilir, ancak Docker ortamında container'lar arası iletişim için `0.0.0.0` gereklidir.

Günlük Dosyaları ve Kayıt Tutma

```
log_error        = /var/log/mysql/error.log  
expire_logs_days = 10
```

- `log_error = /var/log/mysql/error.log`: Hata mesajlarının kaydedileceği günlük dosyasının konumunu belirtir. Bu dosya, MySQL'in çalışması sırasında karşılaştığı hataları, uyarıları ve bilgilendirme mesajlarını içerir. Sorun giderme sırasında bu dosya incelenir.
- `expire_logs_days = 10`: Binary log dosyalarının kaç gün saklanacağını belirtir. Binary log, veritabanında yapılan değişiklikleri kaydeder ve replikasyon ile point-in-time recovery için kullanılır. 10 günden eski log dosyaları otomatik olarak silinir, böylece disk alanı tasarrufu sağlanır.

Karakter Seti Ayarları

```
character-set-server = utf8mb4  
collation-server    = utf8mb4_general_ci
```

Bu ayarlar, MySQL sunucusunun varsayılan karakter seti ve sıralama düzenini belirler:

- **character-set-server = utf8mb4**: Sunucunun varsayılan karakter seti olarak **utf8mb4** kullanılır. Bu, tam UTF-8 desteği sağlayan 4 byte'lık karakter kodlamasıdır ve emojiler, özel karakterler ve çeşitli dillerin karakterleri gibi geniş Unicode karakter setini destekler. MySQL'in eski **utf8** karakter seti aslında tam UTF-8 değildir ve bazı karakterleri desteklemez.
- **collation-server = utf8mb4_general_ci**: Karakter sıralaması ve karşılaştırması için kullanılacak düzeni belirtir. **_general_ci** son eki, "case insensitive" (büyük/küçük harf duyarsız) karşılaştırma yapılacağını gösterir. Bu, örneğin "a" ve "A" harflerinin sorgu sonuçlarında eşit olarak değerlendirileceği anlamına gelir.

skip-name-resolve

- **skip-name-resolve**: Bu ayar, MySQL'in bağlanan istemcilerin IP adreslerini DNS üzerinden çözümlemesini engeller. Bu, özellikle DNS çözümlemesinin yavaş olduğu veya sorunlu olduğu ortamlarda performansı artırabilir. Ancak, bunun bir sonucu olarak, MySQL'e bağlanmak için kullanıcı tanımlarında host adı yerine IP adresi kullanmak gereklidir (**user@hostname** yerine **user@ip_address**).

Configrasyonlar için kaynak:

<https://www.ibm.com/docs/en/ztpf/1.1.2023?topic=performance-mariadb-configuration-file-example>

Docker Compose Dosyası

Bu docker-compose.yml dosyası, bir WordPress uygulamasını üç farklı servis (NGINX, MariaDB ve WordPress) aracılığıyla containerize edip yönetmek için tasarlanmıştır. Şimdi bu dosyanın her bir bölümünü derinlemesine inceleyelim.

Versiyon Tanımı

Docker Compose dosyasının en başında yer alan **version: '3.1'** ifadesi, Docker Compose'un hangi sözdizimi ve özellik setini kullanacağını belirler. 3.1 sürümü, secrets yönetimi, gelişmiş ağ özellikleri ve volume driver options gibi özelliklerini

destekler. Docker Compose sürüm numaraları, Docker Engine ile uyumlu olarak geliştirilir ve her yeni sürüm ek özellikler getirir.

NGINX Servisi

NGINX servisi, WordPress uygulamanız için bir web sunucusu ve ters proxy olarak görev yapar. İstemcilerden gelen HTTP/HTTPS isteklerini alıp, gerekiğinde WordPress uygulamasına yönlendirir.

```
nginx:  
  build:  
    context: ./requirements/nginx  
  container_name: nginx  
  ports:  
    - "443:443"  
  volumes:  
    - wordpress_files:/var/www/html  
  environment:  
    DOMAIN_NAME: ${DOMAIN_NAME}  
  networks:  
    - my_network  
  depends_on:  
    - wordpress  
  restart: always
```

build kısmı, Docker'a bu servis için bir imaj oluşturması gerektiğini söyler. **context** parametresi, Docker'a Dockerfile'in bulunduğu dizini gösterir. Bu durumda, NGINX'in kurulum ve yapılandırma detaylarını içeren Dockerfile, [./requirements/nginx](#) dizininde bulunur. Docker bu dizine gidip buradaki Dockerfile'ı kullanarak NGINX imajını oluşturur.

container_name: nginx parametresi, oluşturulacak konteynere özel bir isim verir. Docker Compose normalde "projeadiservisad1" gibi otomatik isimler atar, ancak burada spesifik olarak "nginx" ismini kullanmak istiyoruz. Bu, konteynerinizi diğer Docker araçlarıyla daha kolay yönetmenizi sağlar.

ports parametresi, host makinenizdeki portları konteyner içindeki portlara bağlar. "443:443" formatı, host makinenin 443 portuna gelen trafiklerin konteyner içindeki 443 portuna yönlendirileceğini belirtir. 443 portu HTTPS trafiği için standart porttur, yani web sitenize güvenli bağlantı sağlar. Bu yapılandırma sayesinde, dış dünyadaki kullanıcılar host makinenizin IP'sine veya domain adına HTTPS üzerinden bağlanabilir ve trafik otomatik olarak NGINX konteyneri içindeki web sunucusuna yönlendirilir.

`volumes` parametresi, Docker host makinesi ile konteyner arasında veri paylaşımını sağlar. Bu durumda, `wordpress_files` adlı bir Docker volume, konteyner içindeki `/var/www/html` dizinine bağlanıyor. Bu volume, WordPress'in tüm dosyalarını (WordPress core, temalar, eklentiler, yüklenen medya vb.) içerir. Bu volume, verilerinizin kalıcı olmasını sağlar – yani konteyner silinse veya yeniden oluşturulsa bile WordPress dosyalarınız kaybolmaz. Ayrıca, bu volume WordPress ve NGINX arasında paylaşıldığı için, WordPress'in oluşturduğu dosyalara NGINX de erişebilir ve bu dosyaları ziyaretçilere sunabilir. **Volume ların daha detaylı açıklaması yukarıda mevcut.**

`environment` parametresi, konteynerin içinde kullanılacak çevre değişkenlerini tanımlar. Burada `DOMAIN_NAME` değişkeni, `.env` dosyasından alınan değerle ayarlanır (`${DOMAIN_NAME}` şeklindeki sözdizimi, Docker Compose'a bu değeri dış bir kaynaktan almasını söyler). Bu değişken NGINX yapılandırmasında, hangi domain adına gelen istekleri cevaplayacağını belirlemek için kullanılır.

`networks` parametresi, bu konteynerin hangi Docker ağına bağlanacağını belirtir. Bu örnekte `my_network` adlı özel bir ağ tanımlanmış. Farklı konteynerler aynı ağa bağlanarak birbirleriyle güvenli bir şekilde iletişim kurabilirler. Örneğin, NGINX konteynerinin WordPress konteyneri ile PHP-FPM üzerinden iletişim kurması gereklidir, ve bu iletişim tanımlanan ağ üzerinden sağlanır.

- Docker Networks için bu kaynakta daha detaylı açıklama mevcut:
<https://kerteriz.net/docker-network-driver-tipleri>

`depends_on` parametresi, konteynerler arasındaki başlatma sırasını belirler. Bu örnekte, NGINX konteyneri başlatılmadan önce WordPress konteynerinin başlatılması gerektiği belirtilmiş. Bu, NGINX'in çalışmaya başladığında WordPress servisinin hazır olmasını sağlar, böylece web istekleri doğru şekilde yönlendirilebilir.

Burada **NGINX'in, WordPress başlamadan çalıştırılmaması gerektiği** belirtilmiş. Bunun nedeni, NGINX'in web sunucusu olarak WordPress'e istekleri yönlendirmesi ve WordPress'in de düzgün çalışabilmesi için MariaDB'ye bağlı olmasıdır. Eğer NGINX başlatıldığında WordPress henüz hazır değilse, web istekleri başarısız olur veya hata verebilir. Bu yüzden:

- MariaDB önce başlatılır, böylece veritabanı hazır olur.
- WordPress başlatılır, MariaDB'ye bağlanarak veritabanı işlemlerini yapabilir.
- NGINX en son başlatılır, böylece WordPress'e gelen HTTP isteklerini düzgün şekilde yönlendirebilir.

`restart: always` parametresi, konteyner herhangi bir nedenle durduğunda (çökme, sistemi yeniden başlatma vb.) otomatik olarak yeniden başlatılmasını sağlar. Bu, servisinizin sürekli çalışır durumda kalmasını garantiler ve hizmet kesintilerini minimuma indirir.

MariaDB Servisi

MariaDB servisi, WordPress uygulamamız için veritabanı sunucusu olarak çalışır. Tüm WordPress içeriğimiz, kullanıcı bilgilerimiz, ayarlarımız burada saklanır.

`mariadb:`

`build:`

`context: ./requirements/mariadb`

`container_name: mariadb`

`expose:`

`- "3306"`

`volumes:`

`- db_data:/var/lib/mysql`

`environment:`

`MYSQL_DATABASE: ${DB_NAME}`

`MYSQL_USER: /run/secrets/db_user`

`MYSQL_ROOT_PASSWORD: run/secrets/db_root_password`

`MYSQL_PASSWORD: /run/secrets/db_password`

`secrets:`

`- db_root_password`

`- db_user`

`- db_password`

`networks:`

`- my_network`

`restart: always`

`build` ve `container_name` parametreleri NGINX'te olduğu gibi çalışır. Burada MariaDB imajı `./requirements/mariadb` dizinindeki Dockerfile'dan oluşturulur ve "mariadb" adını alır.

`expose` parametresi, konteynerin belirtilen portu diğer Docker konteynerlerine açmasını sağlar, ancak host makinenize açmaz. Bu, `ports` parametresinden farklıdır.

`expose: - "3306"` ifadesi, MariaDB'nin standart MySQL/MariaDB portu olan 3306'nın sadece aynı Docker ağındaki diğer konteynerlere (bu durumda WordPress) erişilebilir olmasını sağlar. Bu, veritabanınızın dış dünyadan izole edilmesine yardımcı olur ve güvenlik sağlar. WordPress konteyneriniz bu porta bağlanarak veritabanına erişebilirken, dış dünyadan gelen bağlantılar bu porta ulaşamaz.

`volumes` parametresi burada kritik bir role sahiptir. `db_data:/var/lib/mysql` ifadesi, veritabanı dosyalarının `db_data` adlı bir Docker volume'ünde saklanacağını belirtir. `/var/lib/mysql` MariaDB'nin tüm veritabanı dosyalarını sakladığı dizindir. Bu volume sayesinde, konteyner yeniden oluşturulsa bile veritabanı verileriniz kaybolmaz. Bu, üretim ortamlarında veri bütünlüğünü korumak için son derece önemlidir. Volume kullanmadan konteyner oluşturursanız, konteyner silindiğinde tüm veritabanı içeriği de silinir.

`environment` parametresi, MariaDB'nin ilk kurulumu ve yapılandırması için gerekli çevre değişkenlerini tanımlar:

- `MYSQL_DATABASE: ${DB_NAME}` - Oluşturulacak veritabanının adını belirler, değer .env dosyasından alınır.
- `MYSQL_USER, MYSQL_PASSWORD` ve `MYSQL_ROOT_PASSWORD` - Veritabanı kullanıcı bilgilerini ve root şifresini belirler. Burada güvenlik açısından önemli bir nokta var: Bu hassas bilgiler Docker Secrets olarak yönetiliyor. `/run/secrets/db_user` gibi yollar, bu gizli bilgilerin konteyner içinde nerede bulunacağını gösterir. (Not: `MYSQL_ROOT_PASSWORD` satırında bir yazım hatası var, doğrusu `/run/secrets/db_root_password` olmalı.)

`secrets` parametresi, bu konteynerin hangi Docker Secrets'a erişebileceğini belirtir. Docker Secrets, şifreler, API anahtarları gibi hassas verileri güvenli bir şekilde yönetmenizi sağlar. Bu bilgiler konteyner içinde özel bir dizinde (`/run/secrets/`) dosyalar olarak erişilebilir hale gelir, böylece çevre değişkenleri veya yapılandırma dosyalarında açık metin olarak saklanmak zorunda kalmazlar.

`networks` ve `restart` parametreleri NGINX ile aynı şekilde çalışır.

WordPress Servisi

WordPress servisi, PHP uygulamanızı çalıştıran konteynerdir. PHP-FPM kullanarak WordPress kodunu işler ve veritabanı ile etkileşime girer.

`wordpress:`

`build:`

`context: ./requirements/wordpress`

`container_name: wordpress`

`expose:`

`- "9000"`

`volumes:`

`- wordpress_files:/var/www/html`

`environment:`

`WORDPRESS_DB_HOST: ${DB_HOST}`

`WORDPRESS_DB_NAME: ${DB_NAME}`

`WORDPRESS_DB_USER: /run/secrets/db_user`

`WORDPRESS_DB_PASSWORD: /run/secrets/db_password`

`secrets:`

```
- db_user  
- db_password  
- db_root_password  
depends_on:  
- mariadb  
networks:  
- my_network  
restart: always
```

build ve **container_name** parametreleri önceki servislerle aynı şekilde çalışır.
expose: - "9000" ifadesi, PHP-FPM'nin çalıştığı 9000 portunu aynı ağdaki diğer konteynerlere (özellikle NGINX) açar. Bu port, NGINX'in PHP dosyalarını işlemek için PHP-FPM ile iletişim kurduğu porttur. FastCGI protokolü üzerinden çalışır ve NGINX ile WordPress arasındaki iletişimini sağlar.

volumes: - **wordpress_files:/var/www/html** ifadesi, WordPress dosyalarının saklandığı volume'ü konteyner içindeki **/var/www/html** dizinine bağlar. Bu, WordPress core dosyalarının, temaların, eklentilerin ve yüklenen medya dosyalarının bulunduğu dizindir. Aynı volume NGINX konteynerine de bağlanır, böylece NGINX WordPress dosyalarına erişebilir ve HTTP isteklerine cevap verebilir. Bu volume kullanımı, WordPress kurulumunuzun kalıcı olmasını sağlar ve konteynerler arasında dosya paylaşımını mümkün kılar.

environment parametresi, WordPress'in veritabanına bağlanabilmesi için gerekli bilgileri içerir:

- **WORDPRESS_DB_HOST**: \${DB_HOST} - Veritabanı sunucusunun adresini belirtir (.env dosyasından alınır)
 - **WORDPRESS_DB_NAME**: \${DB_NAME} - Bağlanılacak veritabanının adını belirtir (.env dosyasından alınır)
 - **WORDPRESS_DB_USER** ve **WORDPRESS_DB_PASSWORD** - Veritabanına bağlanmak için kullanılacak kullanıcı adı ve şifre, Docker Secrets'tan alınır
- secrets** parametresi, MariaDB'de olduğu gibi, hassas veritabanı bilgilerine güvenli erişim sağlar.

depends_on: - **mariadb** ifadesi, WordPress konteynerinin MariaDB konteynerinden sonra başlatılmasını sağlar. Bu, WordPress'in çalışmaya başladığında veritabanı servisinin hazır olmasını garantiler.

networks ve **restart** parametreleri önceki servislerle aynı şekilde çalışır.

Secrets Tanımları

```
secrets:  
db_user:  
file: ./secrets/credentials.txt
```

```
db_root_password:  
  file: ./secrets/db_root_password.txt  
db_password:  
  file: ./secrets/db_password.txt
```

Bu bölüm, Docker Compose'a secrets'ların nasıl tanımlanacağını ve nereden alınacağını söyler. Docker Secrets, hassas verileri (şifreler, API anahtarları, sertifikalar vb.) güvenli bir şekilde yönetmenizi sağlayan bir mekanizmadır.

Her secret için:

- Bir isim tanımlanır (örn. `db_user`)
- Bu secret'in değerinin alınacağı dosya belirtilir (örn. `file: ./secrets/credentials.txt`)

Bu yapılandırmada, secret'lar host makinenizdeki dosyalardan okunur. Örneğin, `db_user` secret'inin değeri `./secrets/credentials.txt` dosyasından alınır.

Docker Compose, bu dosyaların içeriğini okur ve ilgili konteynerlerin içinde `/run/secrets/` dizini altında erişilebilir hale getirir.

Bu yaklaşım, hassas bilgilerin Docker Compose dosyasında veya çevre değişkenlerinde açık metin olarak saklanması yerine, ayrı dosyalarda tutulmasını sağlar. Bu, güvenlik açısından önemli bir uygulamadır ve credentials'ların yanlışlıkla sızmasını engeller.

Networks Tanımı

```
networks:  
  my_network:  
    driver: bridge
```

Bu bölüm, Docker konteynerlerinizin iletişim kuracağı ağı tanımlar. Ağ adı `my_network` olarak belirlenmiş ve `bridge` driver'i kullanılıyor.

Bridge network driver, Docker'ın varsayılan ağ sürücüsüdür ve aynı host üzerinde çalışan konteynerler arasında iletişim kurmanın en yaygın yoludur. Bu driver, konteynerler için sanal bir ağ oluşturur ve her konteynere bu ağa bir IP adresi atar. Bu sayede konteynerler birbirleriyle servis adlarını kullanarak iletişim kurabilirler (örneğin WordPress konteynerinden `mariadb` adına erişilebilir).

Özel bir ağ tanımlamak, konteynerlerinizin izole bir ortamda iletişim kurmasını sağlar ve aynı host üzerinde çalışan diğer konteynerlerden ayırrı. Bu, güvenlik ve kaynak yönetimi açısından iyi bir uygulamadır.

docker network ls
docker inspect id
komutlarıyla docker ağlarını görüntüleyebilriiz.

Volumes Tanımı

```
volumes:
```

```
db_data:  
  driver: local  
  driver_opts:  
    type: none  
    o: bind  
    device: /home/user/data/mariadb  
wordpress_files:  
  driver: local  
  driver_opts:  
    type: none  
    o: bind  
    device: /home/user/data/wordpress
```

Bu bölüm, Docker Compose'a volume'lerin nasıl oluşturulacağını ve yapılandırılacağını söyler. Volume'ler, konteynerler arasında veri paylaşmak ve konteynerlerin yaşam döngüsünden bağımsız olarak veri saklamak için kullanılır. İki ayrı volume tanımlanmış:

1. `db_data`: MariaDB veritabanı dosyalarını saklar
2. `wordpress_files`: WordPress uygulama dosyalarını saklar

Her volume için:

- `driver: local`: Verilerin yerel dosya sisteminde saklanacağını belirtir
- `driver_opts`: Volume sürücüsü için spesifik yapılandırma seçenekleri sağlar
- `type: none`: Herhangi bir özel dosya sistemi türü kullanılmadığını belirtir
- `o: bind`: Bu bir bind mount olduğunu belirtir, yani host makinesindeki belirli bir dizin konteynere bağlanır
- `device`: Host makinesinde hangi dizinin bağlanacağını belirtir

Bu yapılandırmada, veritabanı dosyaları host makinesinin `/home/user/data/mariadb` dizininde, WordPress dosyaları ise `/home/user/data/wordpress` dizininde saklanır. Bu, verilerin konteynerler silinse veya yeniden oluşturulsa bile kalıcı olmasını sağlar.

Bind mount kullanmak, host makinesindeki belirli dizinleri konteynerlere bağlamanın en doğrudan yoludur. Bu, geliştirme ortamlarında dosyaları kolayca değiştirmenize olanak tanır ve verilerin nerede saklandığı üzerinde tam kontrol sağlar.

**Volume kısmında biz hem Named hem de Binding mount kullandık.
Peki bunu neden yaptık ya da farklı şekilde yapsak ne olurdu ?**

Docker Volume Yapılandırması

Docker Compose dosyasındaki volume yapılandırma bölümünü derinlemesine analiz edelim. Bu konfigürasyon, veri kalıcılığı ve erişimi için karmaşık bir yaklaşım kullanıyor ve her bir parametrenin önemli bir rolü var.

volumes:

```
db_data:  
  driver: local  
  driver_opts:  
    type: none  
    o: bind  
    device: /home/user/data/mariadb  
  
wordpress_files:  
  driver: local  
  driver_opts:  
    type: none  
    o: bind  
    device: /home/user/data/wordpress
```

Bu yapılandırma, geleneksel named volume ve bind mount özelliklerini birleştirerek hibrit bir çözüm oluşturuyor. Her bir parametrenin işlevi şöyle:

driver: local Parametresi

driver: local ifadesi, Docker'a bu volume'ün yerel dosya sistemi sürücüsünü kullanacağını belirtir. Bu parametre şu işlevlere sahiptir:

- Docker Engine'in volume yönetim mekanizmalarını etkinleştirir.
- Volume'ün `docker volume ls` komutıyla görünür olmasını sağlar.
- Docker Volume API üzerinden yönetilebilir hale getirir.
- Volume silme, yedekleme ve taşıma gibi operasyonlarda Docker'ın kendi süreçlerini kullanmasına olanak sağlar.

Eğer **driver: local** belirtmeseydik, Docker varsayılan olarak yine **local** sürücüsünü kullanırdı, ancak bunu açıkça belirtmek, volume yönetiminde ek seçenekler tanımlamamamıza izin verir.

driver_opts Altındaki Parametreler

type: none

Bu parametre, kullanılan dosya sistemi türünü belirtir. **none** değeri:

- Özel bir dosya sistemi tipi kullanılmayacağını belirtir
- Docker'ın dosya sistemini manipüle etmesini engeller
- Docker'a "bu dizini olduğu gibi kullan, dönüştürme yapma" talimatı verir
- Mountpoint üzerinde özel dosya sistemi operasyonları gerçekleştirmez

o: bind

o parametresi, mount seçeneklerini belirtir (**options** ifadesinin kısaltması). **bind** değeri:

- Volume'ün bir bind mount olarak davranışını sağlar
- Kaynak dizindeki içeriğin hedef dizine doğrudan bağlanması gerektiğini belirtir
- Verilerin doğrudan host dosya sisteminde depolanmasını sağlar
- Docker container silindikten sonra bile verilere host üzerinden erişimi mümkün kılar

device: /home/user/data/mariadb

device parametresi, host üzerindeki gerçek dizin yolunu belirtir:

- Volume verilerinin fiziksel olarak nerede saklanacağını tam olarak tanımlar
- Docker'a "volume içeriğini bu spesifik konumda oluştur ve sakla" talimatı verir
- Host üzerinde bu dizinin önceden var olması gerekir, yoksa Docker hata verir
- Volume verileri doğrudan bu dizinde saklanır ve erişilebilir

Sadece Bind Mount Kullanılsaydı Nasıl Olurdu?

Eğer hibrit yaklaşım yerine sadece geleneksel bind mount kullanmak isteseydik, konfigürasyon şöyle olurdu:

`services:`

`mariadb:`

`volumes:`

`- /home/user/data/mariadb:/var/lib/mysql`

`wordpress:`

`volumes:`

`- /home/user/data/wordpress:/var/www/html`

Bu yaklaşımın özellikleri:

1. Docker Volume Yönetimi Olmaz:

- `docker volume ls` komutunda bu volümler görünmez
- Docker'in volume yönetim araçlarıyla yönetilemezler
- Docker Volume API üzerinden erişilemezler

1. Manual Dizin Yönetimi:

- Dizinleri oluşturmak, yetkilendirmek ve yönetmek tamamen kullanıcının sorumluluğundadır
- Dizin izinleri manuel olarak ayarlanmalıdır

1. Silme/Taşıma İşlemleri:

- `docker volume rm` veya `docker volume prune` komutları bu bağlantıları etkilemez
- Silme/taşıma işlemleri doğrudan host dosya sistemi üzerinden yapılmalıdır

1. Container Yapılandırması:

- Her container için volume bağlantılarını ayrı ayrı belirtmek gerekir
- Volume tanımı container tanımına bağlıdır, merkezileştirilmiş değildir

driver: local Kullanılmadığında Ne Olur?

Eğer `driver: local` belirtmeseydik ancak hala volume tanımı yapsaydık:

`volumes:`

`db_data:`

`driver_opts:`

`type: none`

`o: bind`

`device: /home/user/data/mariadb`

Bu durumda:

1. Varsayılan Sürücü Kullanılır:

- Docker varsayılan olarak yine `local` sürücüsünü kullanır, dolayısıyla pratikte aynı etki oluşur

- Açıkça `driver: local` belirtmek, volume konfigürasyonunun daha açık ve anlaşılır olmasını sağlar

1. Volume Yönetimi Etkilenmez:

- Volume yine Docker tarafından yönetilir ve `docker volume ls` ile görünür
- Bu volume üzerinde Docker volume komutları kullanılabilir

`driver: local` belirtmesek bile Docker `/var/lib/docker/volumes/` dizini altında bir referans oluşturur, ancak veriler hala `device:` parametresinde belirtilen konumda saklanır.

Eğer Hiç Volume Tanımı Yapmasaydık?

Docker Compose dosyasında hiç `volumes:` bölümü tanımlamasaydık, yani nginx veya wp içindeki volumesi kastetmiyorum yml dosyasının en aşagısındaki volumes tanımlamasından bahsediyorum. Sadece container tanımlarında doğrudan volume bağlantıları yapsaydık:

`services:`

`mariadb:`

`volumes:`

 - `db_data:/var/lib/mysql`

`# diğer servisler...`

Bu durumda:

1. Anonim Volume Oluşturulur:

- Docker, `/var/lib/docker/volumes/` dizini altında rastgele bir isimle volume oluşturur
- Bu volume'ler Docker'in varsayılan volume davranışını takip eder
- Verilerin tam olarak nerede saklandığını kontrol edemezsiniz
- Veriler `/var/lib/docker/volumes/[random-id]/_data` gibi bir dizinde saklanır

1. Docker Yönetimli Olur:

- Volume tamamen Docker tarafından yönetilir
- Silmek için `docker volume rm` kullanmanız gereklidir
- Host üzerinde doğrudan erişim için Docker'in oluşturduğu dizin yapısını bilmeniz gereklidir

Hibrit Yaklaşımımızın Detaylı Faydaları

Şu an kullandığımız yaklaşım (named volume + bind mount), en iyi özelliklerden faydalananmamızı sağlıyor:

1. Docker Volume Yönetim Araçlarının Tam Entegrasyonu:

- Volümler `docker volume` komutlarıyla yönetilebilir
- Volüm bakımı ve izleme Docker ekosistemi içinde yapılabilir
- Docker'in volume hook ve driver özellikleri kullanılabilir
- Volume etiketleme, yedekleme ve restore özellikleri kullanılabilir

1. Host Üzerinde Doğrudan ve Öngörülebilir Veri Erişimi:

- Verilere host üzerinden doğrudan erişilebilir
- Veriler tam olarak nerede olduğunu bildiğimiz bir konumda saklanır
- Veri dizinleri manuel olarak yedeklenebilir veya taşınabilir
- Host üzerindeki başka araçlar bu verilere doğrudan erişebilir

Sistemin Genel İşleyışı

Bu Docker Compose yapılandırmasıyla oluşturulan sistem şu şekilde çalışır:

1. Kullanıcı, web tarayıcısından site adresine HTTPS üzerinden (443 portu) bağlanır.
2. İstek önce NGINX konteynerine gelir. NGINX, statik dosyaları (CSS, JavaScript, resimler vb.) doğrudan sunabilir.
3. PHP dosyaları için istekler (WordPress sayfaları, gönderiler vb.), NGINX tarafından WordPress konteynerine (PHP-FPM, 9000 portu üzerinden) yönlendirilir.
4. WordPress konteynerinde PHP kodu çalıştırılır, gerektiğinde MariaDB konteynerine (3306 portu üzerinden) bağlanarak veritabanı işlemleri gerçekleştirilir.
5. İşlenen veri NGINX üzerinden kullanıcıya geri döner.

Tüm bu servisler aynı `my_network` ağında bulunur ve birbirleriyle servis adları üzerinden iletişim kurabilirler. Veriler, `db_data` ve `wordpress_files` volume'lerinde saklanır ve konteynerler silinse veya yeniden başlatılsa bile korunur. Bu yapılandırma, WordPress sitenizi üç ayrı bileşene ayırarak, her birinin kendi rolünü en iyi şekilde yerine getirmesini sağlar: NGINX web sunucusu olarak, WordPress PHP uygulaması olarak ve MariaDB veritabanı sunucusu olarak. Bu ayrim, sistemin ölçeklenebilirliğini, güvenliğini ve bakım kolaylığını artırır.

Sistemin Genel Mimarisi

Bu Docker Compose dosyası, WordPress tabanlı bir web uygulamasını üç ana bileşene ayıran bir mikroservis mimarisi oluşturuyor. Her bir servis, konfigürasyonun farklı yönleriyle ilgileniyor ve birlikte entegre bir sistem olarak çalışıyor.

1. Kullanıcı İsteği ve Servisler Arası İletişim Akışı

Bir kullanıcı web sitemize bağlandığında aşağıdaki adımlar gerçekleşir:

1. İlk Temas Noktası: NGINX

- Kullanıcının web tarayıcısı, HTTPS protokolü üzerinden (443 portu) NGINX konteynerine bağlanır
- NGINX, tek açık dış port olan 443'ü dinleyen bir reverse proxy ve web sunucusu olarak görev yapar
- Gelen her HTTP isteği ilk olarak NGINX tarafından karşılanır ve yönlendirilir

1. İstek Yönlendirme Mekanizması

- NGINX, aldığı isteğin türüne göre farklı işlemler gerçekleştirir:
 - **Statik İçerik (.css, .js, resimler):** Doğrudan `/var/www/html` dizininden (`wordpress_files` volume'inden) sunulur

- **Dinamik İçerik (.php dosyaları):** WordPress konteynerine FastCGI protokolü (9000 portu) üzerinden iletilir

1. WordPress PHP İşleme Süreci

- WordPress konteyneri, PHP-FPM ile çalışan PHP uygulamasıdır
- PHP kodları interpret edilir ve çalıştırılır
- Bu süreçte veritabanına erişim gerektiğiinde MariaDB konteynerine bağlanır

1. Veritabanı Etkileşimi

- WordPress konteyneri, MariaDB konteynerine 3306 portu üzerinden bağlanır
- Veritabanı sorguları gerçekleştirilir (veri okuma, yazma, güncelleme işlemleri)
- MariaDB, sonuçları WordPress konteynerine geri döndürür

1. Yanıt Oluşturma ve İletim

- WordPress, aldığı veritabanı verilerini işler ve HTML çıktısı oluşturur
- Bu çıktı NGINX'e geri döndürülür
- NGINX, final HTML içeriğini kullanıcının tarayıcısına iletir

2. Network İzolasyonu ve Güvenlik

Bu mimaride network izolasyonu kritik bir güvenlik katmanı sağlıyor:

- **my_network Bridge Ağı:**
 - Tüm konteynerler `my_network` adlı özel bir bridge ağına bağlıdır
 - Bu ağ, konteynerler arasında izole iletişim sağlar
 - Dış dünyaya açık tek port NGINX'in 443 portudur
- **Port Açıklama Stratejisi:**
 - **NGINX:** 443 portu dış dünyaya açıktır (`ports` direktifi)
 - **WordPress ve MariaDB:** Sadece `expose` direktifi kullanılarak yalnızca iç ağda erişilebilir durumdadır
 - Bu, MariaDB ve WordPress konteynerlerinin doğrudan dış erişime kapalı olduğu anlamına gelir

3. Veri Kalıcılığı Mekanizması

Sistem, iki kritik volume kullanarak veri kalıcılığını sağlar:

- **db_data Volume:**
 - MariaDB'nin tüm veritabanı dosyalarını `/var/lib/mysql` dizininde saklar
 - Konteyner yeniden başlatılsa veya silinse bile veritabanı verileri korunur
 - Host makinede tam olarak `/home/user/data/mariadb` dizininde bulunur
- **wordpress_files Volume:**
 - WordPress dosyalarını, temalarını, eklentilerini ve yüklemelerini `/var/www/html` dizininde saklar
 - Bu volume hem WordPress hem de NGINX konteynerine bağlıdır
 - Host makinede tam olarak `/home/user/data/wordpress` dizininde bulunur

4. Servisler Arası Bağımlılık Yönetimi

Servisler arasındaki başlatma sırası ve bağımlılıklar `depends_on` direktifi ile yönetilir:

- **NGINX**, WordPress'e bağımlıdır - WordPress hazır olmadan NGINX tam işlevsel olamaz
- **WordPress**, MariaDB'ye bağımlıdır - Veritabanı olmadan WordPress çalışmaz
- Bu hiyerarşî, servislerin doğru sırayla başlamasını sağlar: önce MariaDB, sonra WordPress, en son NGINX

5. Hata Toleransı ve Servis Süreklliliği

Her servis için `restart: always` tanımlanmıştır. Bu:

- Herhangi bir çökme durumunda servislerin otomatik olarak yeniden başlatılmasını sağlar
- Host makinenin yeniden başlatılması durumunda tüm yapının otomatik olarak ayağa kalkmasını garanti eder
- Sistemin dayanıklılığını artırır ve minimum downtime sağlar

Her Servisin Detaylı Analizi

1. NGINX Konteyneri

```
nginx:  
  build:  
    context: ./requirements/nginx  
  container_name: nginx  
  ports:  
    - "443:443"  
  volumes:  
    - wordpress_files:/var/www/html  
  environment:  
    DOMAIN_NAME: ${DOMAIN_NAME}  
  networks:  
    - my_network  
  depends_on:  
    - wordpress  
  restart: always
```

Görevleri ve Özellikleri:

- **Web Sunucusu ve Reverse Proxy**: HTTP isteklerini karşılar, statik içeriği sunar ve PHP isteklerini WordPress'e yönlendirir
- **SSL Sonlandırma**: HTTPS bağlantılarını sonlandırır (443 portu)
- **İçerik Dağıtıımı**: WordPress dosyalarına erişir ve kullanıcıya sunar
- **Güvenlik Katmanı**: Doğrudan WordPress ve MariaDB'ye erişimi engeller, ön cephe görevi görür

- **Özelleştirme:** `DOMAIN_NAME` çevre değişkeni ile site alan adına göre yapılandırılabilir

2. MariaDB Konteyneri

```
mariadb:  
  build:  
    context: ./requirements/mariadb  
  container_name: mariadb  
  expose:  
    - "3306"  
  volumes:  
    - db_data:/var/lib/mysql  
  environment:  
    MYSQL_DATABASE: ${DB_NAME}  
    MYSQL_USER: /run/secrets/db_user  
    MYSQL_ROOT_PASSWORD: run/secrets/db_root_password  
    MYSQL_PASSWORD: /run/secrets/db_password  
  secrets:  
    - db_root_password  
    - db_user  
    - db_password  
  networks:  
    - my_network  
  restart: always
```

Görevleri ve Özellikleri:

- **Veritabanı Hizmeti:** WordPress için temel verileri saklar (kullanıcılar, gönderiler, ayarlar, vb.)
- **Veri Kalıcılığı:** `db_data` volume ile veritabanı dosyalarını kalıcı hale getirir
- **Güvenli Yapılandırma:** Veritabanı kimlik bilgileri Docker secrets kullanılarak korunur
- **Ağ İzolasyonu:** Sadece iç ağda erişilebilir, dış dünyaya kapalıdır
- **Özelleştirme:** Veritabanı adı `DB_NAME` çevre değişkeni ile yapılandırılabilir

3. WordPress Konteyneri

```
wordpress:  
  build:  
    context: ./requirements/wordpress  
  container_name: wordpress  
  expose:  
    - "9000"  
  volumes:  
    - wordpress_files:/var/www/html  
  environment:  
    WORDPRESS_DB_HOST: ${DB_HOST}
```

```
WORDPRESS_DB_NAME: ${DB_NAME}
WORDPRESS_DB_USER: /run/secrets/db_user
WORDPRESS_DB_PASSWORD: /run/secrets/db_password
secrets:
  - db_user
  - db_password
  - db_root_password
depends_on:
  - mariadb
networks:
  - my_network
restart: always
```

Görevleri ve Özellikleri:

- **PHP Uygulama Sunucusu:** WordPress PHP kodlarını çalıştırır (PHP-FPM kullanır)
- **Dinamik İçerik Oluşturma:** Kullanıcı isteklerine göre dinamik sayfalar oluşturur
- **Veritabanı Bağlantı Yönetimi:** MariaDB ile bağlantı kurar ve veritabanı işlemlerini gerçekleştirir
- **İçerik Saklama:** WordPress dosyaları, temalar ve eklentiler `wordpress_files` volume'ünde saklanır
- **Güvenlik:** Veritabanı bağlantı bilgileri Docker secrets ile korunur
- **Bağlantı Ayarları:** Veritabanı host (`DB_HOST`) ve adı (`DB_NAME`) çevre değişkenleri ile yapılandırılabilir

Konfigurasyon Yönetimi ve Esneklik

1. Docker Secrets Kullanımı

Hassas bilgiler (şifreler, kullanıcı adları) Docker secrets kullanılarak saklanır:

secrets:

```
db_user:
  file: ./secrets/credentials.txt
db_root_password:
  file: ./secrets/db_root_password.txt
db_password:
  file: ./secrets/db_password.txt
```

Bu yaklaşım:

- Hassas bilgileri Docker Compose dosyasından ve çevre değişkenlerinden ayırır
- Şifrelerin plain text olarak görülmemesini engeller
- Konteyner içinde özel bir dosya sistemi (`/run/secrets/`) üzerinden güvenli erişim sağlar

2. Çevre Değişkenleri ile Parametreleştirme

Sistem, dış kaynaklardan gelen değişkenlerle yapılandırılabilir:

- `${DOMAIN_NAME}`: Site alan adı
- `${DB_NAME}`: Veritabanı adı
- `${DB_HOST}`: Veritabanı sunucu adresi

Bu yaklaşım, farklı ortamlarda (geliştirme, test, produksiyon) aynı yapılandırma dosyasını kullanmayı mümkün kılar.

3. Volume Yapılandırması ve Hibrit Yaklaşım

`volumes:`

```
db_data:  
  driver: local  
  driver_opts:  
    type: none  
    o: bind  
    device: /home/user/data/mariadb  
  
wordpress_files:  
  driver: local  
  driver_opts:  
    type: none  
    o: bind  
    device: /home/user/data/wordpress
```

Bu yapılandırma:

- **Named Volume ile Bind Mount'un Birleşimi:** Docker'ın volume yönetimini kullanırken, verilerin belirli bir konumda saklanması sağlar
- **Kolay Yedekleme:** Veriler belirli bir konumda olduğu için yedekleme ve geri yükleme işlemleri kolaylaşır
- **Doğrudan Erişim:** Host makinesinde dosyalara doğrudan erişim sağlar
- **Docker Yönetimi:** Volume'ler Docker tarafından yönetilir, `docker volume` komutlarıyla kontrol edilebilir

VERİTABANI İŞLEMLERİ

`show databases;` komutu MariaDB'de tüm veritabanlarını listeliyor. Bunlar arasında bazıları sistem tarafından otomatik geliyor.

- `information_schema`: bu aslında fiziksel bir veritabanı değil, daha çok veritabanlarının yapısı hakkında bilgi tutuyor. Hangi tablolar var, hangi sütunlar var, indeksler vs.
- `mysql`: kullanıcı bilgileri, şifreler, izinler gibi sistemsel şeylerin tutıldığı yer. Root şifresi falan burada.
- `performance_schema`: veritabanı sorgularının performansını ölçmek için kullanılıyor. Ne kadar CPU kullanıldı, hangi sorgu ne kadar sürdü gibi şeyler burada.
- `wordpress`: bu benim oluşturduğum veritabanı, WordPress'in kendi verilerini (kullanıcılar, yazılar, ayarlar vs.) burada tutuyor.

Yani ilk üçü sistemin düzgün çalışması için lazım, sonucusu benim uygulamaya özel. Bunları silmemem lazım, özellikle `mysql` kritik.

`where ->` koşul belirtir

`SHOW TABLES;` -> tablolara bakar.

`SELECT * FROM wp_posts;` -> bir tabloyu okur

`SELECT * FROM wp_posts LIMIT 10;` ilk 10 satır

`DESCRIBE wp_posts;` tablo yapısını gorur daha okunaklı.

`SELECT User, Host, plugin, Password FROM mysql.user WHERE User = 'root';` -> mariadb sunucusundaki root userlerini ve yapılarını şifrelerini getrir

`mariadb -u root -p -h 127.0.0.1` -> başka bir kullanıcidan roota erişmek için şifre kontrolu

`SELECT post_title FROM wp_posts WHERE post_status = 'publish' ;`
-> yayılanmış yazı başlıklarını çeker.

`SELECT comment_content FROM wp_comments where comment_author = 'menasy';` -> dah okunaklı olarak yorumu gösterir

Bu dokümantasyonda, konteyner teknolojilerinin karmaşık yapısını elimden geldiğince sade ve anlaşılır bir şekilde aktarmaya çalıştım. Başta basit gibi görünen Docker'ın, içine girdikçe aslında ne kadar derin, güçlü ve öğretici bir sistem olduğunu gördüm. Bu süreç sadece bir proje değil, aynı zamanda benim için ciddi bir öğrenme yolculuğu oldu. Meğer konteynerin içinde koskoca bir dünya varmış... Docker'a saygım arttı. Dokümantasyonda elbette eksikler olacaktır, onları da size bırakıyorum :)

