udemy-docker-notes:

işletim sistemi: en genel anlamıyla kullanıcının bir bilgisayardaki diğer uygulamaları çalıştırmasın izin veren ana yazılımdır.

yani işletim sistemleri bilgisayar donanımıyla uygulamalar arasında köprü görevi görür

işletim sistemi bileşenleri: 1- kernel 2-arayüz uygulaması 3-uygulamaların paketlenmiş hali

kernel(çekirdek) : işletim sisteminin kendisidir diyebiliriz, sistemin fiziksel altyapısıyla üzerine çalışan uygulamalar arasında bir katmandır ve ana görevi bu donanımı yönetmektir.
kernel işletim sisteminin kalbidir ve ana görevi işletim siteminin kaynaklarını yönetmek, kaynaklarla uygulamalar arasında köprü görevi görmektir.

=>Sanallaştırma:
Sunucu-Server -> hizmeti sunar
-> güçlü ve yüksek kapasiteli işlem gücü
-> uzun süreli, kesintisiz ve çoklu isteklere cevap vermek üzere tasarlanmıştır.

İstemci-Client -> hizmeti kullanır
-> görece düşük kapasite işlem gücü
-> tek bir kullanıcıya hizmet vermek için tasarlanmıştır.
-> uzun süreli ve kesintisiz çalışma öncelikli değildir

****Temelde bir fiziksel makinanın üzerinde birden fazla sanal makina kurmamız izin veren ve kaynak dağıtımını ve ortak kaynak kullanımını sağlayan sisteme verilen isimdir.

->namespace: kernel'a eklenen namespace özelliği sayesinde iki ayrı process'i sanki iki ayrı makine üzerine çalışıyormuş gibi birbirlerinden habersiz hale getirebildik.
->control groups: kernel'a eklenen ve processlerin kaynak kullanımını sınırlayan/izole eden özellik

**** namespace ve control groups özellikleri kullanılarak 2010 yıllarında linux container ortaya çıkarıldı.**

** böylece iki app arasında ayrı bir işletim sistemi kurmadan containerlar vasıtasıyla izola**syon sağlayan container teknolojisi docker ortaya çıktı(2013).**

Docker nedir?
Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

Docker Engine: docker platformunun kalbidir, server-client mimarisindedir. 3 componenti vardır
-Docker Daemon(server) ->docker engine'nın temelidir. image, container, volume, network gibi docker objelerinin oluşturulmasını ve yönetilmesini sağlar.
-Docker RestAPI: docker daemon'ın dış dünyayla konuşabilmesini sağlar
-Docker CLI(client): komut satırıyla docker daemon'a ulaşılabilmesini ve yönetilmesini sağlar

**Docker engine piyasada iki ürün olarak bulunur.
1-ce ->docker engine community : ücretsiz, açık kaynak,
2-ee ->docker engine enterprise: ücretli, temelde aynı, ek güvenlik ve docker firmasının desteği vardır(mirantıs firmasına satılmış)

=> IMAGE:

- bir uygulamanın ve onun içinde çalışması için gereken tüm ek kütüphane ve diğer öğelerin paketlenmiş haline image denir. İçinde çekirdek/kernel yoktur çünkü containerlar içinde koştukları işletim sisteminin kernel'ını kullanırlar.
- docker imajını erişebileceğim herhangi bir yerde depolarsan içinde docker engine yüklü herhangi bir bilgisayara bu imajı indirir ve bu imajdan istediğim kadar container oluşturabilirim.
- imaj bir şablondur, bu şablondan istediğim kadar container yaratabilirim.

=> bare-metal -> VM (hypervisor)-> Container(linux-docker)

*Virtual Machine(VM) : tam bir işletim sistemi barındırır ve fiziksel makineyi sanallaştırır
*Container: İçinde işletim sistemi barındırmaz, uygulamayı sanallaştırır, uygulama izolasyonu sağlar, image halinde her yere taşınabilir, çok daha hızlı başlatılır,

$ docker —version
$ docker info
$ docker **kullanılabilecek komutları ve opsiyonları listeler
$ docker image —help **komutların nasıl kullanılacağına bakmak için
$ docker image rm —help **help'i kullanmayı öğren

=> container oluşturma komutları
$ docker container run —name ilkcontainer ozgurozturknet/app1
**(run=create+start)

=> her container imajında, o imajdan bir container yarattığımız zaman varsayılan olarak çalışması için ayarlanmış bir uygulama vardır.
-bu uygulama çalıştığı sürece container ayakta kalır, uygulama çalışmayı bıraktığında container da kapatılır.

**Docker container başlatıldığı zaman otomatik çalışması için tek bir uygulamanın ayarlanmasına izin verir. Bu container içerisinde sadece tek bir uygulama çalıştırılabileceği anlamına gelmez, docker container başlatıldığı zaman otomatik çalışması için tek bir uygulamanın ayarlanmasına izin verir fakat container içerisinde daha sonra bu uygulamanın yanında başka uygulamalar da çalışabilir.

**Container yaratılırken hangi uygulamayı çalıştırmasını istediğimizi belirtebiliriz. Bu varsayılan/default uygulamadan farklı olabilir.

$ docker container run -p 80:80 ozgurozturknet/adanzyedocker

**ID -> docker ile yaratılan her obje eşsiz/uniq bir ID ile yaratılır.

$ docker container logs <containerID> **LOG'ları görebilme

=> container'a isim vermek için —> - - name

$ docker container run -d -p 80:80 ozgurozturknet/adanzyedocker

**çalışan bir container silinemez, önce stop etmek gerekir, ya da -f opsiyonu kullanılır.
$ docker container rm -f d3

**ideal olarak container tek bir uygulamanın çalışması için ayarlanır ve container çalıştırıldığı zaman da uygulama çalıştığı müddetçe container ayakta kalır. Uygulama ile her türlü ayarın imaj oluşturma aşamasında yapılması ideal olandır, container bağlanıp değişiklik yapmak gerekmez, fakat, istenirse bağlanılabilir.

$ docker container run - -name websunucu -p 80:80 -d ozgurozturknet/adanzyedocker

**container'a bağlanmak için;
$ docker container exec -it websunucu sh

**container'ın içindeyken "ps" komutunu girersek çalışan uygulamaları görebiliriz.
-PID :1 olan uygulama ilk çalışan uygulamadır. docker her zaman PID 1 olan uygulamayı gözler ve bu çalıştığı müddetçe çalışmaya devam eder. bu uygulamayı container içerisinden kapatırsak(kill 1) container stop eder, docker container start <name> ile yeniden çalıştırabiliriz.

=> Docker Katmanları:
- docker depolama altyapısında union file system(birleşim dosya sistemi) kullanır, image'lar mevcut bir base image üzerine inşa edilir. bu base imaj üzerinde yapılan her değişiklik katman katman (layer) inşa edilir. her bir katman ayrı bir dosya olarak muhafaza edilir. Fakat tek bir dosyaymış gibi çalışır.

-Docker image read only olarak kalır.

**Containerlar tek bir uygulama çalıştırmak için oluşturulurlar
**Containerlar bu tek uygulamanın çalışması için gerekli tüm gereksinimlerin önceden hazırlandığı image denilen objelerden oluşturulurlar.
**Container içerisinde çalışan ana uygulama durduğu zaman container da durur. Bu durumu algılayan ayarlar kurularak hemen yeni bir container'ın ayağa kalması sağlanır.
**Container içinde sıkıntı varsa düzeltmekle uğraşılmaz, hemen yenisi oluşturulur. Containerlar tek kullanımlıktır.

**Sorun imajdaysa yeni bir imaj ayarlanır

==>Docker Volume
*bir docker objesidir,
-default olarak tutuldukları host makine(docker daemon'ın kurulu olduğu makine) üzerinde oluşturulur.
-istenirse nfs veya cloud üzerinde de oluşturulabilir.
$ docker volume create ilkvolume
$ docker volume inspect ilkvolume ->detayları incelemek için


```
[
    {
        "CreatedAt": "2022-12-25T12:11:52Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/ilkvolume/_data",
        "Name": "ilkvolume",
        "Options": {},
        "Scope": "local"
    }
]
```


** Mountpoint kısmına dikkat et —> volume'un fiziksel olarak yerini gösteriyor, bu volume bağlayacağımız yeni bir container oluşturmak için;

$ docker container run -it -v ilkvolume:/uygulama alpine sh

->volume isminden (ilkvolume:/) volume'ü container içinde hangi klasöre bağlamak istiyorsak onu yazıyoruz.
***volume bağlamak için illa varolan bir klasör kullanmak zorunda değiliz, klasör yoksa container ile birlikte volume'un bağlanacağı klasörde yaratılır.
-> böylece container silinse bile volume içinde kaydedilen bilgiler silinmez, yeni oluşturduğumuz container'a aynı volume'u bağlayarak volume'de kayıtlı verilere ulaşabiliriz.

**Bir volume birden fazla container'a bağlanabilir.

=>container'a read only yetkisi vererek mount etme
$ docker container run -it -v ilkvolume:/deneme:ro alpine sh

***volume imaj içerisinde bulunan mevcut bir klasöre mount edildiğinde, volume de daha önceden dosyalar varsa,
—> klasörde daha önceden başka dosya bulunsa da bulunmasa da mount edildikten sonra volume'de ne dosya varsa onu görürsünüz.
-volume boş - klasör boş -> klasör boş
-volume boş - klasör dolu -> klasör volume'e kopyalanır
-volume dolu- klasör boş -> volume klasöre kopyalanır
-volume dolu- klasör dolu -> volume de ne varsa klasörde o görünür.

=>Bind Mount:
*production ortamında kullanılmaz,
-test veye development ortamında kullanılabilir.

->host üstündeki bir klasör ya da dosyayı (volume oluşturmadan)container içerisine map etmeye bind mount denir. (developerlar yazdıkları kodları kontrol ederken bu yöntemi kullanırlar, böylece her seferinde yazdıkları kodları container içerisine atmak zorunda kalmazlar, mesela kendi web sitesi folderlarına kaydettikleri dosyalar bind mount sayesinde oluşturulan container ile local hostta görüntülenebilir.)

*volumeler docker'ın kendi objesiyken bind mountta kendi bilgisayarımızdaki bir klasörü bağlamış oluyoruz

=>container yaratırken varsayılandan başka bir uygulama ile çalışmasını sağlamak için;
$ docker container run alpine ls

"Batteries Included but Removable"

** Diğer driverlardan farklı olarak, her Network driver'ı, o driver ile oluşturulan ağ alt yapısının nasıl davranacağını ve o ağa bağlı container'ların ne şekilde haberleşeceğini de belirler.

=> NETWORK:
Docker Network Driverlar:
-Bridge: default,
-Host: network izolasyonu istenmediği durumlarda kullanılır. üstünde koştuğu hostun direk parçası olur. Arada herhangi bir katman olmaz. yani üzerinde bağlı olduğu sistemin network altyapısını kullanır, ifconfig yaparsak hostun ip numaraları olduğunu görürüz.

-Macvlan: containerlara mac adresi atanır. Böylece containerlar fiziksel cihaz gibi davranır. Docker network trafiğini container'a bu mac adresi üstünden yönlendirir. NAT veya Routing yapmadan container'ın direk ağ ile haberleşmesi sağlanır. özellikle eski uygulamalarda işe yarar
-None: container'a hiçbir şekilde ağ bağlantısı olmasın istendiğinde none driver'ı ile yaratılmış bir docker objesine bağlanmanız yeterlidir.
-Overlay: swarm ile ilgili

$ docker network ls

*bridge hem driver çeşidi hem de isim vermezsek driver'a default olarak bridge adı verilir.

$ docker network inspect bridge

CTRL + PQ —>> container' ı kapatmadan çıkmak için

**Aynı bridge network'e bağlı containerlar birbirleriyle direk haberleşebilirler.

$ docker container run -it —name deneme1 —net host

**containerın içinde ifconfig komutunu girerek ip no, driver gibi network ayarlarını/özelliklerini görebiliriz.

$ docker container run -d —publish 8080:80
                                -p host_port:container_port

-default olarak TCP port açılır, -9 53:53/udp şeklinde udp protocol'u kullanılabilir.

**Aynı container run komutu içinde birden fazla port publish yapabilirim

**Custom Bridge Network Yaratma:
-Containerlar arası network izolasyonu sağlamak istersek ayrı bridge networkler yaratarak bunu sağlayabiliriz
-Kullanıcı tanımlı bridge network'e bağlı containerlar birbirleriyle isimler üzerinden haberleşebilirler. Yani DNS çözümlemesi vardır.
- Kullanıcının tanımladığı bridge networkler basit bir DNS hizmeti de

sunduğundan bu ağa bağlı containerlar ip adreslerinin yanında isimleriyle de iletişime geçebilirler.
-Containerlar çalışır durumdayken de kullanıcı tanımlı bridge networke bağlanıp, bağlantıyı kesebilirler. Default bridge varsa container çalışırken bu bağlantıyı kesemezsiniz.

$ docker network create kopru1
$ docker container run -dit --name websunucu —net kopru1 ozgurozturknet/adanzyedocker sh

**container ‘ı oluştururken -dit opsiyonunu kullanırsak container oluştuktan sonra “docker attach websunucu” komutuyla container ‘a bağlanabiliriz.

**komut satırından subnet ve ip range tanımlı network oluşturma
$ docker network create --driver=bridge --subnet=10.10.0.0/16 --ip-range=10.10.10.0/24 --gateway=10.10.10.10 kopru2

$ docker network inspect kopru2

**çalışan container’a custom network bağlamak için;
$ docker network connect kopru2 database

*** bir container’ ı aynı anda birden fazla network’e bağlayabiliriz.

**network bağlantısını kesmek için;
$ docker network disconnect kopru2 database
$ docker network rm kopru2

=> linux de log kayıtları root /var/log klasöründe tutulur
=> linux de nginx log kayıtlarını cd var/log/nginx altında access.log ve error.log dosyalarında tutar

==>> docker log kayıtlarına ulaşabilmek için -> docker logs management

$ docker logs <containername>
-> bu komut ile container çalıştığı sürece biz bu komutu girene kadar oluşturduğu tüm logları, bütün stout ve stderr çıkışına gönderdiği bütün mesajları listeler.

$ docker run -d --name con1 -p 80:80 nginx
*local hosttan nginx test yayını var mı kontrol et
$ docker logs con1
*container loglarını yukarıdaki komutla görebilirsin(standarout akşını görürüz.)
*normalde linux de loglar, var/log/nginx dosyası altında bir klasöre(access.log, error.log) kaydedilir.(simlink)
$ docker exec -it con1 sh
# cd /var/log/nginx
# ls
access.log error.log
ls -l
total 0
lrwxrwxrwx 1 root root 11 Dec 21 14:10 access.log -> /dev/stdout
lrwxrwxrwx 1 root root 11 Dec 21 14:10 error.log -> /dev/stderr
*** log dosyalarını aynı zamanda stdout ve stderr dosyalarına da kaydedilecek şekilde ayar yapılmış olduğunu gördük, docker logs da bu iki akışı görecek şekilde tasarlandığı için bu kayıtları görebiliyoruz
*container silinene kadar bu logları görebiliriz

$ docker logs - -details con1

**logları zaman damgalı görebilmek için,
$ docker logs -t con1
$ docker logs - -until 2022-12-29T19:20:28.120458545Z con1
- -since
$ docker logs - -tail 3 con1 —> son 3 satırı görmek için
$ docker logs -f con1 => komutları canlı olarak takip etmek için** -f —>follow

**dockerın desteklediği loggin altyapıları
Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog —>docker info komutlu ile gördük, defaul json driver
$ docker container run --log-driver splunt nginx -> varsayılanı splunk olarak ayarladık.

=>> Docker Top
*linux de ps komutunu girdiğimizde hangi processlerin çalıştığını görürüz.
** Docker ile içinde girmeden bir containerda hangi processlerin çalıştığını görmek için docker top komutu kullanılır.

$ docker top <containername>

=**>> Docker Stats**
**containerların ne kadar ram, cpu, io kullandığını görürüz
*docker host üzerinde çalışan bütün containerların kullandıkları kaynakları görmek için
$ docker stats

=>> container cpu ve memory **LİMİT EKLEME:**

** containerlar default olarak üzerinde çalıştıkları host sisteminin cpu ve memory kaynaklarını herhangi bir kısıtlama olmadan kullanabilirler. bu nedenle container oluştururken kısıtlama yapmak gereklidir(- -memory opsiyonu kullanılır)
$ docker container run -d --memory=100m ozgurozturk/adanzyedocker
** aşarsa belirli bir swqp alanı vermek için --memory-swap kullanılır
$ docker container run -d --memory=100m --memory-swap=200m ozgurozturk/adanzyedocker
$ docker container run -d --cpus=“1.5” imagename
--cpus —>core sayısısını ne kadarını kullanabileceğin gösterir.
--cpuset-cpus= “core numarası” -> hangi core’ları kullanacağını söylemek için

*Değişkenler -> bizim değerler atayabildiğimiz adlandırılmış verilere denir.

==>>ENVIRONMENT VARIABLE:

**işletim sisteminin tamamında geçerli olan ver her yerden çağırılabilen değişkenlere verilen isimlere environment variable denir.
**linux de environment variable’ ları görebilmek için;
$ printenv

**ENVIRONMENT VARIABLE TANIMLAMAK İÇİN:
$ export test=“denemedir”
$ echo $test --> çağırmak için

—>container oluştururken env var atamak için;
$ docker container run -it --env VAR1=deneme1 --env VAR2=deneme2 ubuntu bash

**env variable lar case sensitive dir

**dockerın üzerinde çalıştığı hostun environment variable'larını container içine aktarmak için;
$ docker container run -it --env (TEMP-name of env var) ubuntu bash

**ENV VAR ları belirli bir dosyadan çağırma
$ docker container run -it --env-file ./env.list ubuntu bash
->bu örnekte host üzerinde değişkenlerin tanımlandığı dosyanın adı env.list

*ENV VAR nerede kullanılır?
-her containerde tek tek değişiklik yapmak yerine sadece değişkenin kayıtlı olduğu yer değiştirilir.

==>> DOCKER IMAGE

*Uygulamananın paketlenmiş haline verilen isime docker image denir.
*Docker da her şeyin bir id numarası vardır, isim yerine bu id numaraları da kullanılabilir.

**Docker image isimleri/tagleri sadece isim değildir, aynı zamanda image'ın nerede depolandığını da gösterir.
---> docker.io/ozgurozturknet/adanzyedocker:latest
registryURL - repository - tag

**aynı repository içinde birden fazla image version' u saklanabilmesi için tag' ler kullanılır.
** latest --> varsayılan tag
-> bir tag belirlenmezse latest çağırılır, tag'inin latest olması en güncel olduğu anlamına gelmez, varsayılan olarak sunulan anlamına gelir.

**official image başka bir tag atama;
$ docker image tag mysql:5 ozgurozturknet/adanzyedocker:latest

DOCKERFILE
FROM | Oluşturulacak imajın hangi imajdan oluşturulacağını belirten talimat. Dockerfile içerisinde geçmesi mecburi tek talimat budur. Mutlaka olmalıdır.
Ör: FROM ubuntu:18.04

LABEL | İmaj metadata'sına key=value şeklinde değer çiftleri eklemek için kullanılır. Örneğin team=development şeklinde bir etiket eklenerek bu imajın development ekibinin kullanması için yaratıldığı belirtilebilir.
Ör: LABEL version:1.0.8

RUN | İmaj oluşturulurken shell'de bir komut çalıştırmak istersek bu talimat kullanılır. Örneğin apt-get install xxx ile xxx isimli uygulamanın bu imaja yüklenmesi sağlanabilir.
Ör: RUN apt-get update

WORKDIR | cd xxx komutuyla ile istediğimiz klasöre geçmek yerine bu talimat kullanılarak istediğimiz klasöre geçer ve oradan çalışmaya devam ederiz. Boyle bir klasor yoksa oluşturur
Ör: WORKDIR /usr/src/app

USER | gireceğimiz komutları hangi kullanıcı ile çalıştırmasını istiyorsak bu talimat ile onu seçebiliriz.
Ör: USER poweruser

COPY | İmaj içine dosya veya klasör kopyalamak için kullanırız
Ör: COPY /source /user/src/app

ADD | COPY ile aynı işi yapar yani dosya ya da klasör kopyalarsınız. Fakat ADD bunun yanında dosya kaynağının bir url olmasına da izin verir. Ayrıca ADD ile kaynak olarak bir .tar dosyası belirtilirse bu dosya imaja .tar olarak sıkıştırılmış haliyle değil de açılarak kopyalanır.
Ör: ADD https://wordpress.org/latest.tar.gz /temp

ENV | Imaj içinde environment variable tanımlamak için kullanılır
Ör: ENV TEMP_FOLDER="/temp"

ARG | ARG ile de variable tanımlarsınız. Fakat bu variable sadece imaj oluşturulurken yani build aşamasında kullanılır. Imajın oluşturulmuş halinde bu variable bulunmaz. ENV ile imaj oluşturulduktan sonra da imaj içinde olmasını istediğiniz variable tanımlarsınız, ARG ile sadece oluştururken kullanmanız gereken variable tanımlarsınız.
Ör: ARG VERSION:1.0

VOLUME | Imaj içerisinde volume tanımlanamızı sağlayan talimat. Eğer bu volume host sistemde varsa container bunu kullanır. Yoksa yeni volume oluşturur.

Ör: VOLUME /myvol

EXPOSE | Bu imajdan oluşturulacak containerların hangi portlar üstünden erişilebileceğini yani hangi portların yayınlanacağını bu talimatla belirtirsiniz.
Ör: EXPOSE 80/tcp

ENTRYPOINT | Bu talimat ile bir containerın çalıştırılabilir bir uygulama gibi ayarlanabilmesini sağlarsınız.
Ör: ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]

CMD | Bu imajdan container yaratıldığı zaman varsayılan olarak çalıştırmasını istediğiniz komutu bu talimat ile belirlersiniz.
Ör: CMD java merhaba

HEALTHCHECK | Bu talimat ile Docker'a bir konteynerin hala çalışıp çalışmadığını kontrol etmesini söylebiliriz. Docker varsayılan olarak container içerisinde çalışan ilk processi izler ve o çalıştığı sürece container çalışmaya devam eder. Fakat process çalışsa bile onun düzgün işlem yapıp yapmadığına bakmaz. HEALTHCHECK ile buna bakabilme imkanına kavuşuruz.
Ör: HEALTHCHECK --interval=5m --timeout=3s CMD curl -f http://localhost/ || exit 1

SHELL | Dockerfile'ın komutları işleyeceği shell'in hangisi olduğunu belirtiriz. Linux için varsayılan shell ["/bin/sh", "-c"],Windows için ["cmd", "/S", "/C"]. Bunları SHELL talimatı ile değiştirebiliriz.
Ör: SHELL ["powershell", "-command"]

Örnek

```
FROM ubuntu:18.04
RUN apt-get update -y
RUN apt-get install default-jre -y
WORKDIR /merhaba
COPY /myapp .
CMD ["java", "merhaba"]
```

```
$ docker image build -t talhaalkan/merhaba -f Dockerfile .
```

*Bu komutu Dockerfile' ın olduğu klasördeyken girmen gerekir.
** "Dockerfile" dockerfile'ın ismini bu şekilde yazarsan "-f" komutunu gerek kalmaz, sonuna ek koyarsan(Dockerfile.txt gibi) -f yazman gerekir.

***Komutun sonunda nokta işareti var, her şeyi bu klasörde ara(build context) diyoruz docker' a bu noktayla,

->bu image dan container oluşturmak için,
$ docker container run talhaalkan/merhaba

***bir imajın geçmişini görebilmek ve katmanlarını incelemek için;
$ docker image history <imagename>

**alpine imajlı örnek Dockerfile
FROM python:alpine
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
EXPOSE 5000
CMD python ./index.py

=>>bir uygulamayı container imajı haline getirmek istiyorsan mutlaka ilk önce bu uygulamanın runtime 'ını docker hub'da aramalısın, mesela python tabanlı bir uygulama kullanacaksan onun resmi imajını bulup onu base imaj olarak kullanın ve uygulamanızı bu imaj üzerine ekleyerek kendi imajınızı oluşturun.

==>> NODEJS imajlı Dockerfile örneği:

FROM node:10

 # Create app directory
WORKDIR /usr/src/app

 # Install app dependencies
 # A wildcard is used to ensure both package.json AND package-lock.json are copied
 # where available (npm@5+)
COPY package*.json ./

RUN npm install
 # If you are building your code for production
 # RUN npm ci --only=production

 # Bundle app source
COPY server.js .

EXPOSE 8080
CMD ["node", "server.js"]

*package.json--> bu dosya node dünyasında mutlaka bulunması gereken manifest dosyasıdır, temel metadata bilgileri ve bu uygulamanın bağımlı olduğu(dependencies) modüller tanımlanır,

$ docker image build -t talhaalkan/node .

**dockerın katman özelliği, daha önce indirdiği bir veriyi cache den almasını sağlar ve bandwidth avantajı sağlar. Değişiklik olan katman varsa cache'den almaz,
***bu nedenle Dockerfile'daki sıralama önemlidir, değişmeyecek veya az değişecek talimatlar Dockerfile' ın üst kısmında yazılmasına ve böylece daha çok ön bellek kullanılmasına çalışılması best practice dir.

==>>> NGINX IMAGE; ** healthcheck örneği

 # Bu örnek uygulama https://github.com/docker/labs/blob/master/beginner/chapters/webapps.md adresinden alınmıştır
 # base imaj olarak nginx resmi imajını kullanıyoruz --
 # https://github.com/nginxinc/docker-nginx/blob/5971de30c487356d5d2a2e1a79e02b2612f9a72f/mainline/buster/Dockerfile
FROM nginx:latest

 # label ile de key value pair olarak bu imajla ilgili bilgileri giriyoruz. her iki talimat da imajın içindeki sistemle alakalı değil.
 # her ikisi de imajın metadatasında saklanıyor. imajla alakali bilgiler.
LABEL maintainer="Ozgur Ozturk @ozgurozturknet"
LABEL version="1.0"
LABEL name="hello-docker"

 # bir adet env variable tanımlıyoruz. Daha sonra bunu runtime'da değiştireceğiz
ENV KULLANICI="Dunyali"

 # nginx debian-slim imajı baz alınarak oluşturulmuştur.
 # bu imajda geriye doğru uyumluluk sorunları var ve o nedenle update

olmuyor.
#Bu url'leri sources.list'e ekleyerek update olabilmesini sağlıyoruz
#RUN printf "deb http://archive.debian.org/debian/ jessie main\ndeb-src http://archive.debian.org/debian/ jessie main\ndeb http://security.debian.org jessie/updates main\ndeb-src http://security.debian.org jessie/updates main" > /etc/apt/sources.list

 # Uygulama depolarından mevcut paketlerin en son listesini çekip güncelliyoruz.
RUN apt-get update

 # debian-slim'de bir çok tool mevcut değil. Bizim curl kullanmamız gerektiği için curl kuruyoruz
RUN apt-get install curl -y

 # nginx web sayfalarını /usr/share/nginx/html folder'ında barındırıyor. O nedenle bu folder'a geçiyoruz
WORKDIR /usr/share/nginx/html

 # web sitemizin açılış sayfası olan Hello_docker.html dosyasını buraya kopyalıyoruz.
COPY Hello_docker.html /usr/share/nginx/html

 # sistemin düzgün çalıştığını ve nginx daemon'ının web sitesini publish etmekte bir sorun yaşamadığını test ediyoruz
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 CMD curl -f http://localhost/ || exit 1

 # bu imajdan container yaratıldığı zaman çalışmasını istediğimiz komutu buraya giriyoruz
 # /usr/share/nginx/html klasörüne geçiyor ve sed aracılığıyla Hello_docker.html dosyasının içerisindeki
 # Kullanici kelimesini $KULLANICI env variable'ının değeriyle Hostname kelimesini de $HOSTNAME ile değiştiriyor ve
 # dosyanın içeriğini index.html'e adlı yeni bir dosyaya yazıyoruz.
 # ardından nginx daemon'ı çalıştırıyoruz
CMD sed -e s/Kullanici/"$KULLANICI"/ Hello_docker.html > index1.html && sed -e s/Hostname/"$HOSTNAME"/ index1.html > index.html ; rm index1.html Hello_docker.html; nginx -g 'daemon off;'

****son çalıştırdığım containerın hostname ve kullancı ismini ayarlayabileyim diye son komutları CMD talimatıyla girdik, daha önce

mesala RUN komutuyla girseydik bu değişiklikler imaj oluşturulurken yapılacaktı, bunu istemiyoruz, CMD ile bu imajdan container oluşturulurken yapılacak işlemleri girmiş oluyoruz.

--> örnek healthcheck komutu:
$ HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 CMD curl -f http://localhost/ || exit 1

-opsiyonlar:
-interval-> ne kadar aralıklarla kontrol yapılacağını gösterir
-timetout-> ne kadar süre cevap alamazsa sağlıksız kabul edileceğini gösterir
-start period-> starttan kaç saniye sonra health check yapmaya başlayacağını gösteriyor
-retries-> kaç deneme sonunda olumsuz yanıt alırsa unhealthy olarak işaretleneceğini gösterir.
*-curl komutuyla da web sitesinin çalışıp çalışmadığını kontrol ediyoruz, bu komut uygulamadan uygulamaya değişebilir.

**Linux de uzun satırları bölmek için ters slash "\" kullanılır.

***Dockerfile içindeki env variable' ı değiştirmek için;
$ docker container run -d -p 80:80 --name xyz -e KULLANICI="talhaalkan" ozgurozturknet/hello-docker

==>> ADD ve COPY talimatlarının farkı:

ADD talimatı, COPY talimatıyla aynı işi yapar, yani imaj içerisine dosya ya da klasör kopyalamamızı sağlar, buna ilaveten;
1-ADD uzak bir lokasyondan(örn.bir web sitesinden) dosya kopyalamamız imkan sağlar(copy sadece imaj oluşturma işlemi yaptığımız sistemdeki dosyaları kopyalar),
2-ADD tar, jar gibi sıkıştırılmış dosyaları açar ve açılmış halini kopyalar(copy sıkıştırılmış dosyaları sıkıştırılmış olarak kopyalar fakat açmaz) ayrıca ADD uzak sunucundan sıkıştırılmış dosya kopyalarken bu dosyayı açmaz sadece localdeki dosyayı kopyalarken açar

=> ADD - COPY farkını gösteren Dockerfile örneği:
FROM alpine:latest
WORKDIR /app1
COPY wordpress.tar.gz .

ADD wordpress.tar.gz .
ADD https://wordpress.org/latest.tar.gz .

==>>> **CMD ve ENTRYPOINT talimatlarının farkı:**

****CMD talimatı** bir imajdan container yarattığımız zaman başkasını belirtmediysek burada(Dockerfile de) girdiğimiz komutu çalıştır demekti. CMD talimatı kullanıldığında Run time da çalışmasını istediğimiz komutu değiştirebiliriz.(runtime da override edebiliriz--> $ docker container run <imagename> ls dersek "ls" komutunu çalıştırır.)

****ENTRYPOINT talimatı** da CMD talimatıyla aynı işi yapar. Fakat CMD den farkı ENTRYPOINT runtime da değiştirilemez, bu nedenle ENTRYPOINT talimatını sanki birer uygulamaymış gibi çalışmasını istediğimiz containerları oluşturulurken kullanırız.

**her dockerfile belgesinde bu iki komuttan biri mutlaka olmalıdır.
***aynı anda hem CMD hem ENTRYPOINT talimatı varsa; docker CMD talimatında yazan komutları ENTRYPOINT talimatına parametre olarak ekler.

FROM centos:latest
ENTRYPOINT ["ping", "127.0.0.1"]
-> bu dockerfile aşağıdaki gibi de yazılabilir.

FROM centos:latest
ENTRYPOINT ["ping"]
CMD ["127.0.0.1"] -->> entrypoint teki komutu çalıştırır, cmd deki komutu buna parametre olarak girer, yani ping komutunu 127.0.0.1 ip adresine(localhost) atar.
--Bu şekilde yazmanın avantajı nedir?
-> bu şekilde değişebilmesini istediğimiz parametreleri CMD komutunun içerisine yazabiliriz. örneğin bu imajdan container oluşturacağımızda docker run ping 8.8.8.8. yazarsak CMD de yazan komutu değiştirmiş ve ping atılacak ip adresine runtime sırasında kendimiz ayarlamış oluyoruz.

==>>EXEC ve SHELL Form Farkı:

-> exec -> CMD ["java", "uygulama"]
-> shell -> CMD java uygulama
**exec formu direk process çalıştırır.

**exec formu env var ‘ları göremez.

1-Komut shell formunda girilirse Docker bu imajdan container yaratıldığı zaman bu komutu varsayılan shell’i çalıştırarak onun içerisine girer, çalışan 1.process(pid1) bu shell process olur
2-Exec formunda girilirse Docker herhangi bir shell çalıştırmaz ve direk process çalışır, pid1 komut olur, shell olmaz
3-Exec formunda çalışan komutlar shell process’i çalışmadığı için Environment Variable gibi bazı değerlere erişemezler
4-Entrypoint ve CMD birlikte kullanılacaksa Exec formu kullanılmalıdır çünkü shell formu kullanıldığında CMD’deki komutlar ENTRYPOINT’ e parametre olarak aktarılmaz

==> MULTI STAGE(bir oluşturmayı kademelere bölme)
FROM mcr.microsoft.com/java/jdk:8-zulu-alpine
COPY /source /usr/src/uygulama
WORKDIR /usr/src/uygulama
RUN javac uygulama.java
CMD ["java", "uygulama"]

--> bu dockerfile’dan imaj oluşturduk, imajda java uygulaması olduğu için aynı zamanda RUN talimatıyla javac komutu çalıştırarak source kodu compile ettik,
--> bu imajdan container oluşturduk, java uygulamamız çalışıyor ama çok büyük hacimli oldu javasdk uygulaması nedeniyle, müşteriye gönderirken sadece compile edilmiş uygulamayı gönderebilmek için containerın içinden sadece uygulamamızı almak istersek docker cp komutunu kullanabiliriz.
**docker cp komutunu kullanabilmek için container’ın çalışıyor durumda olmasına gerek yok, silinmemiş olması yeterli.
**daha sonra başka bir docker file ile compile edilmiş uygulama için yeni bir imaj oluşturabiliriz.
FROM mcr.microsoft.com/java/jre:8-zulu-alpine
COPY /uygulama /uygulama
WORKDIR /uygulama
CMD ["java", "uygulama"]
-->imaj boyutu küçüldü

=> CONTAINERDAN HOSTA KLASÖR KOPYALAMA:
$ docker cp javauygulama:/usr/src/uygulama .
<containername>:path .(destination)

```
FROM mcr.microsoft.com/java/jdk:8-zulu-alpine AS derleyici
COPY /source /usr/src/uygulama
WORKDIR /usr/src/uygulama
RUN javac uygulama.java
CMD ["java" , "uygulama"]

FROM mcr.microsoft.com/java/jre:8-zulu-alpine AS derleyici
WORKDIR /uygulama
COPY --from=derleyici /usr/src/uygulama .
CMD [ "java", "uygulama" ]
```

$ docker image build -t javason .
*imaj oluşurken her FROM da ayrı bir stage başlıyor
*imaj her zaman sadece son aşamadaki imajın aynısı olur.

=> Başka bir imajdan kendi oluşturacağımız imaja dosya dosya kopyalamak için;
COPY --from=nginx:latest /usr/src/uygulama .

==>> ARG
***build arg --> ARG VERSION**
**Dockerfile içinde değişken kullanmak istersek bunu ARG talimatıyla tanımlayarak yapabiliyoruz. Terminalden image oluştururken bu değişkenin hangi değeri almasını istiyorsak --build-arg yardımıyla onu atıyoruz.
***ENV talimatıyla (ENV key=value)atadığımız değişkene ancak oluşturacağımız container da ulaşabilirken --build-arg opsiyonuyla oluşturduğumuz değişkeni yine Dockerfile içinde oluşturuyoruz, ancak bu değişkene değer atama işlemini imajı build ederken gerçekleştiriyoruz.
**ARG ile oluşturulan değişkenen containerdan erişilemez, bunlar environment değişken değildir, sadece imaj yaratılırken kullanılır.
****ARG özellikle CI/CD sistemlerde otomatik docker image yaratma işlemlerini kurgularken başvurlan bir talimattır.

```
FROM ubuntu:latest
WORKDIR /gecici
ARG VERSION
```
ARG VERSION=3.8.1 #bu şekilde varsayılan bir değerde atayabiliriz.

ADD https://www.python.org/ftp/python/3.7.6/Python-3.7.6.tgz .
(bunun yerine aşağıdaki gibi yazıyoruz)
ADD https://www.python.org/ftp/python/${VERSION}/Python-${VERSION}.tgz .
CMD ls -al

$ docker image build -t vers1 --build-arg VERSION=3.7.1 .
$ docker image build -t vers2 --build-arg VERSION=3.8.1 .

==>>DOCKER COMMIT kullanarak imaj oluşturma:
*önce base bir image kullanarak container oluşturulur
**sonra bu containerdan commit komutu kullanarak imaj oluşturulur

$ docker container run -it --name con1 ubuntu:latest bash
# apt-get update(bazı işlemler yaparız)
$ docker commit <containerName> <tagName>:latest

=> containerdan imaj oluştururken -c opsiyonuyla istediğimiz talimatları girebiliriz.
$ docker commit -c 'CMD ["java", "uygulama"]' <hangicontainer> <imageName>:latest

==>> DOCKER SAVE - LOAD

*Docker save komutunu kullanarak sistemde bulunan bir imajı tar dosyası olarak kaydedebiliyoruz. Daha sonra bu dosyayı internet erişimi olmayan bir makineye örneğin usb ile kopyalayabilir ve sonrasında Docker load komutuyla bu tar dosyasından imaj yaratabiliriz.

$ docker save <imagename>:latest -o con1imaj.tar
$ docker load -i con1imaj.tar

==>> Docker Registry
-->> test ve dev ortamlarında kullanılır, prod için uygun değil
*docker image deposu kurmak istiyorsan reponun imajını indirip bir container oluşturan gerekiyor, (docker pull registry)

$ docker run -d -p 50000:5000 --restart always --name registry(container name) registry(imagename)

*** --restart opsiyonunun alternatifleri

-> always | no | on-failure | unless-stopped
-> default -> no

=>local docker registry repo'ya yeni tag ile image göndermek için;
$ docker image tag ozgurozturknet/hello-app:latest 127.0.0.1/hello-app:latest
$ docker image push 127.0.0.1/hello-app:latest

>>>>63.BÖLÜM ALIŞTIRMALARI TEKRAR YAPILMALI**********

=>> DOCKER COMPOSE

Birden fazla containerdan oluşan uygulamaları tanımlamamıza ve tek bir komut ile bu tanım üzerinden bu containerları ayağa kaldırım çalıştırmamıza imkan sağlayan bir uygulamadır.

**tanım dosyaları .yaml formatında yazılır.
-docker compose production için uygun değildir, daha çok geliştirme aşamasında kullanılır.
--docker desktop kullanmıyorsan sadece docker engine yükleme yeterli değil, docker compose'u da yüklemen gerekir.

-Docker-compose cli
** docker compose cli komutları sadece Docker-compose.yaml dosyasını bulunduğu klasördeyken çalışır
$ docker-compose up -> Docker-compose up, docker-compose.yaml dosyası ile configure edilen sistemi ayağı kaldırdı.

** Docker compose ile oluşan sistemlerde containerlara önce docker compose dosyasının bulunduğu klasörün adı ve servisin adı verilir.

$ docker-compose down -->up komutunun tersi, ne varsa siler

$ docker compose config --> compose dosyasını işlem sırasına göre gösterir.

$ docker compose images --> compose.yaml dosyasındaki servislerin hangi imajlarla oluştuğunu gösterir.

$ docker compose logs --> ayaktaki servisin loglarını gösterir.

$ docker compose exec servicename ls -a --> compose dosyası ile oluşturulan servis içinde komut çalıştırabilmek için kullanılır

**==>> docker compose için service, container anlamına gelir. service = container

***volume ‘ler docker compose down komutuyla silinmez, diğer her şey silinir.

=>Docker-compose.yaml dosyası

```
version: '3.7'

services:
  mysqldb:
    image: ozgurozturknet/webdb
    environment:
     MYSQL_DATABASE: proje
     MYSQL_USER: projemaster
     MYSQL_PASSWORD: master1234
     MYSQL_ROOT_PASSWORD: master1234
    networks:
     - webnet

  websrv:
    build: .
    depends_on:
      - mysqldb
    ports:
      - "80:80"
    restart: always
    networks:
      - webnet
    environment:
      DB_SERVER: mysqldb
      DB_USERNAME: projemaster
      DB_PASS: master1234
      DB_NAME: proje
networks:
 webnet:
   driver: bridge
```

**build: . --> bunun sayesinde hem imaj oluşturma hem de container oluşturma içişini docker compose ile yapmış olduk

**>> app de / uygulamada herhangi bir değişiklik yaptıysak yeniden build etmemiz lazım
$ docker-compose build

=>> DOCKER SWARM
* Docker Swarm, Docker Engine'e entegre bir container orchestration çözümüdür. bir Docker ana bilgisayar havuzunu tek bir sanal ana bilgisayara dönüştürür.

*TCP port 2377 -> Cluster yönetimi
*TCP ve UDP port 7946 -> Nodelar arası iletişim
*UDP port 4789 -> Overlay Network için

--> docker-swarm manager
$ docker swarm init

To add a worker to this swarm, run the following command:
$ docker swarm join --token
SWMTKN-1-03dhupdaz.....172.31.59.172:2377

** docker swarm init komutu bir manager biri worker yaratmak için iki tane token oluşturur.
** manager node desired state ile current state'ı karşılaştırır ve gerekirse container ayağa kaldırır veya siler.
**bir swarm sisteminde birden fazla manager node olabilir, fakat bir tanesi aktif olur, diğerleri yedek olur. Her şeyi lider olan manager node yapar.

***lider seçimi RaftConsensus algoritması kullanılarak yapılır.
-Raft algoritmasının doğru çalışabilmesi ve lider seçiminin sorunsuz yapılabilmesi için her zaman tek sayıda manager node kurulmuş olması gerekir. ideali 3, 5 veya 7 manager kurulması.

**>>
$ docker swarm init --advertise-addr 192.168.0.13
*hangi ip adresi üzerinde cluster iletişiminin kurulacağını da belirtmiş olduk

**docker swarm join-token manager
**docker swarm join-token worker

$ docker node ls -->clusterdaki nodeleri görmek için

$ docker service create --name test --replicas=5 8080:80 nginx

$ docker service ps test

==>> Docker Service

-> Docker Swarm modunda oluşturulabilecek en küçük objeye service denir.

**docker service create --name test --network abc --publish 8080:80 --replicas=10 --update-delay=10s --update-parallelism=2 ozgurozturk/alistirma:kirmizi

=>Docker Swarm Modes
1-Replicated = oluşturmak istediğimiz servisin kaç replica içereceğini belirtiriz. Swarm uygun olan nodeler üstünde o sayıda replica oluşturur.

2-Global = oluşturmak istediğimiz servisin kaç replica içereceğini belirtmeyiz, swarm altındaki her node üzerinde 1 adet replica oluşturur.

$ docker service create nginx
$ docker service inspect <servicename> ->desired state 'i görmek için kullanılır

**task --> kaç replica istiyorsak o kadar task oluşur, task sayısı kadar container oluşur. (task=container)

$ docker service scale test1=3 --> yeni bir desired state düzenledik ve container sayısını 3 e çıkardık

-Global service oluşturmak için:
$ docker service create --name global --mode=global nginx

==>>Overlay Network -(overlay driver)

- the overlay driver enable simple and secure multi-host networking
- all containers on the overlay network can communicate
- node'lar hangi network veya subnette olursa olsun, overlay networke bağlanan tüm container'lar birbirleriyle haberleşebilir.
- yani farklı veri merkezlerinde ve farklı hostlardaki nodelerda oluşturulmuş containerlar overlay network(sanal ağ) oluşturularak iletişime geçmesi sağlanır. Her container bu overlay network'ten (belirlenen ip aralığından)bir ip adresi alır.
- routing docker tarafından yapılır.

**bir swarm cluster oluşturulduğunda ingress adında bir overlay network de otomatik olarak oluşturulur. Aksi belirtilmedikçe oluşturulan servisler bu overlay network'e bağlanır.
**user defined overlay network ler oluşturulabilir.
--opt encrypted opsiyonu kullanılarak haberleşme trafiği encrypt edilebilir. Fakat network trafiğini yavaşlatır.
***aynı overlay network'e bağlı servislerin containerları **herhangi bir port kısıtlaması olmadan** sanki aynı ağdaymış gibi haberleşebilirler.
**aynı overlay network üzerinde containerlar isimleriyle haberleşebilirler, yani Docker dns çözümlemesi ve load balancing hizmeti de sunmaktadır.
**overlay network üzerinde port publish yapılabilir. ingress routing mesh desteklenir.

*her uygulama için ayrı overlay oluşturmak güvenlik için daha doğrudur.
$ docker network create -d overlay -overnet
$ docker service create --name db --network over-net ozgurozturknet/fakedb

==>> VERSİYONLAMA - service update
$ docker service update --help
$ docker service update --OPTIONS
**bu komutu girince mevcut container'ı siler ve yeni bir container oluşur.

$ docker service update --update-parallelism uint
aynı anda kaç containerda(task) değişiklik/güncelleme yapmak istiyorsak
$docker service update --detach --update-delay 5s --update-parallelism 2 --image ozgurozturk/web:v2 websrv

=>>update ettiğimiz şeyleri geri almak için kullanılır.
$ docker service rollback --detach websrv

=>SECRETS

Containerlarda plain text olarak tutmamızın güvenlik zafiyeti yaratabileceği kullanıcı adı ve şifre gibi verileri secret objeleri şeklinde encrypted olarak transfer edebiliriz.

1.YÖNTEM
*dosyanın içinden secret yaratma
$ docker secret create kullanici_adi kullanici.txt
$ docker secret ls
$ docker secret inspect

**secret ‘lar /run/secrets/ klasörü altında oluşur.

2.YÖNTEM
$ echo “bu bir denemedir” | docker secret create deneme -
*deneme isimli secret oluşturarak echo komutuyla belirlediğimiz değeri atadık.

==>> bu secretları servislere bağlamak için:
$ docker service create -d --name secretdeneme --secret kullanici_adi --secret sifre --secret deneme ozgurozturknet/web

***> şifreyi güncellemek için**
$ docker service update --secret-rm sifre | --secret-add sifre2 servicename
-mevcut şifrenin içeriği değiştirilemez, şifre güncellenmek isteniyorsa yeni bir şifre oluşturulup eskisiyle değiştirilir.

***Docker Stack**
*docker compose’un swarm için olanı...
**network driver olarak overlay kullanılmak zorunda
**imajın önceden oluşturulması gerekir, Dockerfile üzerinde build yapılamaz.
**deploy anahtarını kullanarak replica sayısı belirlenebilir.
**oluşturma komutunda dosyası -c komutuyla göstermemiz gerekir.

websrv

```
        image:
        deploy:
                replicas: 3
                update_config:
                        parallelism: 2
                        delay: 10s
                        order: stop-first
$ docker stack OPTIONS COMMAND
$ docker stack deploy -c ./docker-compose.yml ilkstack

$ docker stack ls
$ docker stack services serviceName

$ docker stack rm serviceName
```