Beskrivelse:

Nu har vi en rimelig ide om hvad Docker er. I denne opgave skal vi se om vi kan få lavet ét system der indeholder 3 websider. Det skulle ikke være så svært, men for at der skal være lidt kød på skal der jo også laves en reverse proxy til HVER af de 3 websider. Men bare rolig. Det er ikke så svært som det lyder.

Kapitel 1: Forberedelse.

Det første vi skal, er at lave underbiblioteket vi skal arbejde i, Vi vil kalde det 3web fordi vi skal lave 3 websider. HUSK at stå i din brugers home bibliotek, inden du starter (dette kan gøres ved at skrive **`cd ~`** Eller under Ubuntu bare skrive **`cd`** og trykke enter) vi sætter begge kommandoer ind samtidig:

```
mkdir 3web
cd 3web
```

Foraklaring:
Første kommando laver biblioteket. Den anden kommando hopper ned i biblioteket.

Eksempel output:

```
dtmek@docker2:~$ mkdir 3web
cd 3web
dtmek@docker2:~/3web$
```

Vi fortsætter med at lave forberedelser. Denne gang skal der bare overføres det materiale der findes til Opgave 3. dette kan som altid gøre med f.eks. winscp, som beskrevet i indledningen.

Efter at materialet er overført, må vi hellere lige se efter at alt er overført med følgende kommando:

```
ls -l
```

Eksempel output:

```
dtmek@docker2:~/3web$ ls -l
total 44
-rw-rw-r-- 1 dtmek dtmek  205 Oct 24 20:35 81default.conf
-rw-rw-r-- 1 dtmek dtmek  205 Oct 24 20:35 82default.conf
-rw-rw-r-- 1 dtmek dtmek  205 Oct 24 20:35 83default.conf
-rw-rw-r-- 1 dtmek dtmek  205 Oct 24 20:35 default.conf
-rw-rw-r-- 1 dtmek dtmek 2030 Oct 31 08:18 docker-composepush.yml
-rw-rw-r-- 1 dtmek dtmek 1695 Nov 11 11:54 docker-compose.yml
drwxrwxr-x 2 dtmek dtmek 4096 Nov 11 12:12 nginx
-rw-rw-r-- 1 dtmek dtmek  100 Oct 24 20:35 reverse-proxy.conf
drwxrwxr-x 2 dtmek dtmek 4096 Nov 11 12:12 web1
drwxrwxr-x 2 dtmek dtmek 4096 Nov 11 12:12 web2
drwxrwxr-x 2 dtmek dtmek 4096 Nov 11 12:12 web3
dtmek@docker2:~/3web$
```

Alle filer der er afbildet i ovenstående eksempel output skal være til stede, for at alle opgaver i opgave 3 vil kunne virke.

Slut Kapitel 1: Forberedelse.

Kapittel 2: Containere

Vi starter med noget simpelt, og det er også lidt repetition. Vi starter de 3 websider som en container hver (Du kan kopiere alle kommandoer på en gang):

```
docker run -d -p 81:80 --name web1 -v ./web1:/usr/share/nginx/html nginx
docker run -d -p 82:80 --name web2 -v ./web2:/usr/share/nginx/html nginx
docker run -d -p 83:80 --name web3 -v ./web3:/usr/share/nginx/html nginx
```

Forklaring (Kun Container 1):
`docker run`: Kommando der kører en container.
`-d`: Detached = Frigiver vores SSH commando line igen.
`-p 81:80`: Mounter port 81 på host til port 80 på container
`--name web1`: Containernavnet er web1.
`-v ./web1:/usr/share/nginx/html`: Mounter biblioteket **`./web`** på host til **`/usr/share/nginx/html`** biblioteket på container. Det er her at en nginx standard container leder efter sin webside.
`nginx`: Det er det image vi vil bruge (kan også læses som nginx:latest)

Eksempel output:

```
dtmek@docker2:~/3web$ docker run -d -p 81:80 --name web1 -v ./web1:/usr/share/nginx/html nginx
docker run -d -p 82:80 --name web2 -v ./web2:/usr/share/nginx/html nginx
docker run -d -p 83:80 --name web3 -v ./web3:/usr/share/nginx/html nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
a480a496ba95: Pull complete
f3ace1b8ce45: Pull complete
11d6fdd0e8a7: Pull complete
f1091da6fd5c: Pull complete
40eea07b53d8: Pull complete
6476794e50f4: Pull complete
70850b3ec6b2: Pull complete
Digest: sha256:28402db69fec7c17e179ea87882667f1e054391138f77ffaf0c3eb388efc3ffb
Status: Downloaded newer image for nginx:latest
8f966c399157865f32934e320a69a2bf94111212d30b9ee3ebf65c29ced3e461
2528bb4447ee890ad4d743fb74cfc0de03ca690aa3ad2a23409126a1b19d29ab
3e3348032048c053721b7d840fd2ac189c41ffbf7acd749098f21195c66a1c7d
dtmek@docker2:~/3web$
```

Læg mærke til at den starter 3 containere til sidst, men henter kun ét image: nginx. Kan det være rigtig? Lad os lige se efter. Først lad os se hvad der er af kørende containere:

```
docker ps
```

Eksempel output:

```
dtmek@docker2:~/3web$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS                                 NAMES
3e3348032048   nginx     "/docker-entrypoint.…"   2 minutes ago   Up 2 minutes   0.0.0.0:83->80/tcp, [::]:83->80/tcp   web3
2528bb4447ee   nginx     "/docker-entrypoint.…"   2 minutes ago   Up 2 minutes   0.0.0.0:82->80/tcp, [::]:82->80/tcp   web2
8f966c399157   nginx     "/docker-entrypoint.…"   2 minutes ago   Up 2 minutes   0.0.0.0:81->80/tcp, [::]:81->80/tcp   web1
dtmek@docker2:~/3web$
```

Jo... der er 3 containere kørende. Men hvor mange images har vi mon? Lad os se efter:

```
docker image ls -a
```

Eksempel output:

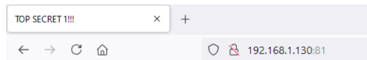
```
dtmek@docker2:~/3web$ docker image ls -a
REPOSITORY   TAG       IMAGE ID       CREATED       SIZE
nginx        latest    3b25b682ea82   5 weeks ago   192MB
dtmek@docker2:~/3web$
```

Jo. Som forventet, er der er kun et image. Her får vi så bekræftet at man kan have lige så mange containere af et image kørende som man har ressourcer til. Eller? Vi havde jo 3 forskellige websider. Mon de nu også er

forskellige, selvom de er lavet på samme image. Lad os se efter og besøge websiderne, åben følgende adresser i en webbrowser:

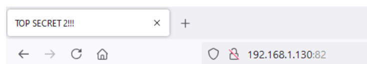
```
http://ip_på_worker_makine:81
http://ip_på_worker_makine:82
http://ip_på_worker_makine:83
```

Eksempel output (Viser alle 3 websider, med adresser):
Webside:81



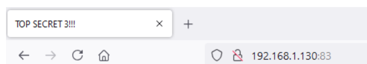

```
TOP SECRET PAGE!!!
Nr 1 (Et!!)
Læs ikke denne tekst
Denne tekst er hemelig!
```

Webside:82




```
TOP SECRET PAGE!!!
Nr 2 (To!!)
Læs ikke denne tekst
Denne tekst er hemelig!
```

Webside:83




```
TOP SECRET PAGE!!!
Nr 3 (Tre!!)
Læs ikke denne tekst
Denne tekst er hemelig!
```

Som man kan se at siderne forskellige, så man kan godt have forskellige websider. Det er jo websidens indhold vi bestemmer med den del af kommandoen der hedder:
-v ./web1:/usr/share/nginx/html

Man kan selvfølgelig også stoppe, og starte alle containere. Med henholdsvis **docker start**, og **docker stop/kill** Det prøver vi lige:

```
docker stop web1
```

Eksempel output:

```
dtmek@docker2:~/3web$ docker stop web1
web1
dtmek@docker2:~/3web$
```

Prøv websiden

```
http://ip_på_worker_makine:81
```

Der vil selvfølgelig ikke kunne oprettes forbindelse (husk at trykke CTRL + F5 for genopfrisk hele siden).
Vi starter den lige igen:

```
docker start web1
```

Eksempel output:

```
dtmek@docker2:~/3web$ docker start web1
web1
dtmek@docker2:~/3web$
```

Prøv websiden igen:

`http://ip_på_din_docker_maskine:81`

Den er tilbage. Det var jo meget godt.

Men hvis vi nu sidder og udvikler på vores image, ville det så ikke være nemmere hvis containeren vi startede, selv slettede sig, når vi laver en **`docker stop/kill`** kommando, uden vi skal fjerne containeren hver gang? Lad os prøve, først rydder vi lige op og sletter alle vores kørende containere, imaget lader vi ligge, vi skal nok bruge det igen, vi kører alle kommandoer på en gang igen:

```
docker kill web1 web2 web3
docker container rm web1 web2 web3
```

Eksempel output:

```
dtmek@docker2:~/3web$ docker kill web1 web2 web3
docker container rm web1 web2 web3
web1
web2
web3
web1
web2
web3
dtmek@docker2:~/3web$
```

Så starter vi lige vores containere igen, men med en lille ændring denne gang (Fremhævet):

```
docker run --rm -d -p 81:80 --name web1 -v ./web1:/usr/share/nginx/html nginx
docker run --rm -d -p 82:80 --name web2 -v ./web2:/usr/share/nginx/html nginx
docker run --rm -d -p 83:80 --name web3 -v ./web3:/usr/share/nginx/html nginx
```

Forklaring:
Alt er som første gang, bortset fra **--rm**. Dette er et flag, der indikerer at containeren skal slette sig selv når den ikke kører. Man kan selvfølgelig ikke bruge både **--restart XXXX (f.eks. -restart unless-stopet)** og **--rm** da de ligesom modarbejder hinanden.

Eksempel output:

```
dtmek@docker2:~/3web$ docker run --rm -d -p 81:80 --name web1 -v ./web1:/usr/share/nginx/html nginx
docker run --rm -d -p 82:80 --name web2 -v ./web2:/usr/share/nginx/html nginx
docker run --rm -d -p 83:80 --name web3 -v ./web3:/usr/share/nginx/html nginx
7bf6d901fed2338114129d29a9d3c8d61a7cdd1255060e82fc32089fac24b6c7
1a016408683a702d38351b301b4493a8e4f2e4788c57d92d3f9a774586621749
369863cb75a640a1b497e956ee341571038f95550e1310fdec0b5592b204bddd
dtmek@docker2:~/3web$
```

Så er containerne startet, gå til websiderne for at bekræfte at websiderne er startet.

Lad os prøve en stop kommando på en af containerne:

```
docker stop web1
```

Ekemspel output:

```
dtmek@docker2:~/3web$ docker stop web1
web1
dtmek@docker2:~/3web$
```

Så prøver vi at starte containeren igen:

```
docker start web1
```

Eksempel output:

```
dtmek@docker2:~/3web$ docker start web1
Error response from daemon: No such container: web1
Error: failed to start containers: web1
dtmek@docker2:~/3web$
```

Containeren er væk! Det var jo også det vi forventede.

Slut Kapitel 2: Containere

Start Kapitel 3: Netværk

Nu har vi et andet problem, for at kunne lave en reverse-proxy kræver det at vi kender web containernes IP adresse? Ellers kan vi jo ikke videresende trafik fra vores proxyserver til vores webserver. Hvordan gør vi det? Kan vi egentlig selv bestemme en IP adresse til en container? Selvfølgelig kan vi det, vi skal bare selv oprette et netværk i docker. Først fjerner vi lige de 2 sidste containere:

`docker kill web2 web3`

Eksempel output:

```
dtmek@docker2:~/3web$ docker kill web2 web3
web2
web3
dtmek@docker2:~/3web$
```

Nu må vi hellere oprette vores netværk. Det er faktisk meget nemt. Vi bruger et subnet der hedder **`192.168.180.0/24`** For at oprette netværket bruger vi kommandoen:

`docker network create --subnet=192.168.180.0/24 web`

Forklaring:
`docker network`: Er kommandoen der behandler netværk i docker.
`create`: Vi vil oprette et netværk.
`--subnet=192.168.180.0/24`: Hvad skal vores subnet være? Og IP adresse område (scope).
`web`: Navnet på vores netværk

Eksempel output:

```
dtmek@docker2:~/3web$ docker network create --subnet=192.168.180.0/24 web
bd3d13ffd39dcdc40b73891bd6f8b1f9cc14db47a6d7e562da87b959bb56eee2
dtmek@docker2:~/3web$
```

Så blev netværket oprettet, det lange nummer er et unikt ID til vores netværk.

Skulle vi ikke lige se om netværket også eksisterer:

`docker network ls`

Forklaring:
Kommandoen minder meget om de andre kommandoer der viser information. Så jeg vil ikke gå nærmere ind på den.

Eksempel output:

```
dtmek@docker2:~/3web$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
e449794f46d8   bridge    bridge    local
59b197d894b3   host      host      local
c999b5115ee6   none      null      local
bd3d13ffd39d   web       bridge    local
dtmek@docker2:~/3web$
```

Nederst kan vi se vore ”web” netværk. Driver vil jeg ikke komme nærmere ind på. Det er ikke nødvendigt at vide for resten af eksemplerne. Nærmere omkring driver kan findes på nettet. Denne driver er tilsluttet til hosten via en form for NAT.

Vi tildeler IP adresse og netværk i vores, efterhånden meget lange, kommando streng (fremhævet), og starter alle vores 3 containere.. Igen.. Og vi kan køre alle vores kommandoer på en gang:

```
docker run --rm -d --network web --ip 192.168.180.81 --hostname web1 -p 81:80 --name web1 -v ./web1:/usr/share/nginx/html nginx

docker run --rm -d --network web --ip 192.168.180.82 --hostname web1 -p 82:80 --name web2 -v ./web2:/usr/share/nginx/html nginx

docker run --rm -d --network web --ip 192.168.180.83 --hostname web1 -p 83:80 --name web3 -v ./web3:/usr/share/nginx/html nginx
```

Forklaring:
JA de er lange de kommandoer! Men vi har nu tildelt én unik kendt IP adresse til hver af vores web servere, og valgt hvilket Docker internt netværk de skal være på. HUSK at IP adresser selvfølgelig skal være i det scope vi har oprettet i vores netværk.

Eksempel output:

```
dtmek@docker2:~/3web$ docker run --rm -d --network web --ip 192.168.180.81 --hostname web1 -p 81:80 --name web1 -v ./web1:/usr/share/nginx/html nginx

docker run --rm -d --network web --ip 192.168.180.82 --hostname web1 -p 82:80 --name web2 -v ./web2:/usr/share/nginx/html nginx

docker run --rm -d --network web --ip 192.168.180.83 --hostname web1 -p 83:80 --name web3 -v ./web3:/usr/share/nginx/html nginx
a75bae019676848b5ba09e70034dd43ea281f4ae02190eaa9ed9a66aa04db34e
a39b5d51e0c60a0c37f3ac8c4d15f01820dda57512128419df0bd458fa873779
52aeac8d5bb87a8c349271c022ffe4d263b6e3821d850b1fd402358df4d59a63
dtmek@docker2:~/3web$
```

For at se om de er online, besøg en af websiderne:

`http://ip_på_din_docker_maskine:81`

Det er den selvfølgelig. Lad os se om vi kan se containerne i vores netværk:

`docker network inspect web`

Forklaring:
`docker network`: Kommando der behandler netværk.
`Inspect`: Vi vil se oplysningerne omkring vores netværk (*HINT: inspect findes også til andre commandoer f.eks. docker container inspect!*).
`web`: navnet på vores netværk.

Eksempel output (Der er meget tekst, jeg viser kun udvalgte dele):

Vi kan se vores subnet:

```
"Config": [
    {
        "Subnet": "192.168.180.0/24"
    }
]
```

Vi kan også se alle vores containere (Med de rigtige IP adresser):

```
"Containers": {
    "52aeac8d5bb87a8c349271c022ffe4d263b6e3821d850b1fd402358df4d59a63": {
        "Name": "web3",
        "EndpointID": "4db72eaccd4fb19afeeaf26e76eb808c7619d78ab4352e6bf49a90071fb7d2d4",
        "MacAddress": "02:42:c0:a8:b4:53",
        "IPv4Address": "192.168.180.83/24",
        "IPv6Address": ""
    },
    "a39b5d51e0c60a0c37f3ac8c4d15f01820dda57512128419df0bd458fa873779": {
        "Name": "web2",
        "EndpointID": "ea4d01340cfb1e1ed983410478ca96fe110b951aeb4e85188bcc90da16bf20a8",
        "MacAddress": "02:42:c0:a8:b4:52",
        "IPv4Address": "192.168.180.82/24",
        "IPv6Address": ""
    },
    "a75bae019676848b5ba09e70034dd43ea281f4ae02190eaa9ed9a66aa04db34e": {
        "Name": "web1",
        "EndpointID": "7da4207ceefed06d336968e1de0eff295a578ca3df3cbbd13406b2803e85f5ae",
        "MacAddress": "02:42:c0:a8:b4:51",
        "IPv4Address": "192.168.180.81/24",
        "IPv6Address": ""
    }
},
```

Vi gør lige klar til næste kapitel:

```
docker kill web1 web2 web3
```

Eksempel output:

```
dtmek@docker2:~$ docker kill web1 web2 web3
web1
web2
web3
dtmek@docker2:~$
```

Slut Kapitel 3: Netværk

Kapitel 4: Reverse proxy til intern container.

Så skal vi se om vi kan lave en reverse proxy til en af vores web server container. Det kan vi sagtens. Til at starte med starter vi lige én og denne gang kun én webserver container:

`docker run --rm -d --network web --ip 192.168.180.81 --hostname web1 --name web1 -v ./web1:/usr/share/nginx/html nginx`

Forklaring:
Læg mærke til at vi ikke har brugt **`-p`** for tildeling af port til vores container. Dette er jo den container der ikke skal være ”synlig” fra hosten. Det er jo vores reverse proxy der skal være synlig. Og da det er en nginx container, og nginx er en webserver container, er port 80 allerede åben på det image som containeren bygger på, så det behøver vi ikke specificere.

Eksempel output:

```
dtmek@docker2:~/3web$ docker run --rm -d --network web --ip 192.168.180.81 --hostname web1 --name web1 -v ./web1:/usr/share/nginx/html nginx
088410974da794de5f75d070cb388926918914a641f8940b437d191fafeda53a
dtmek@docker2:~/3web$
```

Vi skal jo lave en reverse proxy. En reverse proxy er faktisk også bare en webserver, der bliver sat op til at være en reverse proxy via en konfigurationsfil, og denne server er placeret foran den rigtige webserver. Nærmere vil jeg ikke komme ind på opbygningen af en reverse proxy, da dette ikke er et webserver kursus.

Så det første vi gøre er at starte en nginx standard container, i hvilken vi så vil modificere en konfigurationfil, så den vil opføre sig som en reverse proxy:

`docker run --rm -d --network web --ip 192.168.180.71 --name nginx-base -p 80:80 nginx`

Forklaring:
Det er prøvet før, så der kommer ikke den store forklaring. Vi lægger containeren i vores netværk kaldet web, og sætter en fast IP, åbner port 80 til host, og kalder imaget for nginx-base. Vores grund image er nginx.

Eksempel output:

```
dtmek@docker2:~/3web$ docker run --rm -d --network web --ip 192.168.180.71 --name nginx-base -p 80:80 nginx
65c79750357d39a00d87f556a682485828fddfb6c115f0ace903c8dc3223ae67
dtmek@docker2:~/3web$
```

Lad os lige se hvilke containere vi har kørende nu:

`docker ps`

Eksempel output:

```
dtmek@docker2:~/3web$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED              STATUS              PORTS                               NAMES
65c79750357d   nginx     "/docker-entrypoint.…"   About a minute ago   Up About a minute   0.0.0.0:80->80/tcp, :::80->80/tcp   nginx-base
088410974da7   nginx     "/docker-entrypoint.…"   16 minutes ago       Up 16 minutes       80/tcp                              web1
dtmek@docker2:~/3web$
```

Der ligger jo de to containere vi har startet (som man kan se har web1 port 80 åben, men ikke bundet til nogen port på hosten.)

Nu skal vi have redigeret i en fil der ligger i vores nginx-base container. Og bare rolig, det kan man faktisk også, vi skal bare hente den fil, vi vil ændre ud af containeren, og så ændre filen, og lægge den tilbage i containeren, og evt. genstarte den service i containeren, hvor vi har ændret noget selvfølgelig. Vi starter med at hente filen ud af containeren:

`docker cp nginx-base:/etc/nginx/conf.d/default.conf ./default.conf`

Forklaring:
`Docker cp`: Er kommandoen, og står for ”docker copy”.
`nginx-base:` Hvilken container du vil kopiere filen fra.
`/etc/nginx/conf.d/default.conf`: Hvor i containeren du vil kopiere fra. Fuld sti, og filnavn.
`./default.conf`: Fuld sti, hvor filen skal ligge. Hvis der ligger en af samme navn bliver den overskrevet (**`./`** betyder aktuel sti.).

Eksempel resultat:

```
dtmek@docker2:~/3web$ docker cp nginx-base:/etc/nginx/conf.d/default.conf ./default.conf
Successfully copied 3.07kB to /home/dtmek/3web/default.conf
dtmek@docker2:~/3web$
```

Så er filen kopieret, vi må nok hellere redigere filen, inden vi lægger den tilbage, det var jo derfor vi hentede ud fra containeren. Først åbner vi filen i vores editor nano:

`nano default.conf`

Når du nu har åbent filen ligger der en hel masse tekst i filen. Alt der står i filen skal SLETTES!! Det kan gøres ved gentagende gange at trykke CTRL + k indtil alle linjer er slettet (CTRL + k sletter aktuel linje).

Når nu alle linjer er slettet indsættes teksten herunder i editoren (kopier tekst, og højre klik i SSH teminalen for indsæt):

```
################## default.conf start #########################
# Complete Nginx Docker reverse proxy config file
server {
  listen 80;

  location / {
    proxy_pass http://192.168.180.81:80/index.html;
  }

} # End of Docker Nginx reverse proxy setup
################## default.conf slut #########################
```

Gem filen ved at trykke "Ctrl+x" -> trykke "Y" -> Tryk Enter ved filnavn. Og du er tilbage til normal promt på Ubuntu.

Forklaring:
Server lytter på port 80, og leder alt trafik fra root (/) videre til under proxy_pass nævnte ip adresse.

Ingen eksempel output.

Nu har vi tilrettet filen. Men hvordan får man så filen tilbage til vores container. Jo vi kopierer filen tilbage til containeren med følgende kommando:

```
docker cp ./default.conf nginx-base:/etc/nginx/conf.d/
```

Forklaring:
`Docker cp`: Er kommandoen, og står for "docker copy"
`./default.conf`: Fuld sti, og filnavn, på filen der skal kopieres over i container (./ betyder aktuel sti.).
`nginx-base`: Hvilken container du vil kopiere filen til.
`/etc/nginx/conf.d/`: Hvor i containeren du vil kopiere til. Hvis der eksisterer en fil med det navn, som du kopierer over i containeren, vil denne fil bliver overskrevet.

Eksempel output:

```
dtmek@docker2:~/3web$ docker cp ./default.conf nginx-base:/etc/nginx/conf.d/
Successfully copied 2.05kB to nginx-base:/etc/nginx/conf.d/
dtmek@docker2:~/3web$
```

Hmmm.. Har den nu også kopieret filen over, og ligger den det rigtige sted? Vi laver da bare lige en SSH session ind i vores kørende container (nginx-base):

`docker exec -i -t nginx-base bash`

Forklaring:
`Docker exec`: Dette er kommandoen. Den executer (udfører) en kommando i en container.
`-i`: interaktiv forbindelse (vi kan taste kommandoer)
`-t`: TTY laver en terminal forbindelse (SSH)
`Nginx-base`: Container navn vi vil have forbindelse til
`Bash`: Hvilken kommando skal udføres i container. Bash er linux udgaven af en commando shell.

Læg mærke til at prompten ændrer sig. Vi er pludselig bruger root i vores container ID, og vi står også i root directory (root@containerid:/#)

Eksempel output:

```
dtmek@docker2:~/3web$ docker exec -i -t nginx-base bash
root@65c79750357d:/#
```

Hvor var det nu vi havde lagt vores fil? Nåh jo her: **`/etc/nginx/conf.d/`** Lad os hoppe ned i det bibliotek med denne kommando:

`cd /etc/nginx/conf.d/`

Eksempel output:

```
root@65c79750357d:/# cd /etc/nginx/conf.d/
root@65c79750357d:/etc/nginx/conf.d#
```

Lad os se om der ligger nogen filer her:

`ls -l`

Eksempel output:

```
root@65c79750357d:/etc/nginx/conf.d# ls -l
total 4
-rw-r--r-- 1 1000 1000 189 Nov 13 08:39 default.conf
root@65c79750357d:/etc/nginx/conf.d#
```

Dato ser ud til at passe, på filnavnet.

Lad os lige se hvad filen indeholder:

`cat default.conf`

Eksempel output:

```
root@65c79750357d:/etc/nginx/conf.d# cat default.conf
# Complete Nginx Docker reverse proxy config file
server {
  listen 80;

  location / {
    proxy_pass http://192.168.180.81:80/index.html;
  }

} # End of Docker Nginx reverse proxy setup
root@65c79750357d:/etc/nginx/conf.d#
```

Jo det var det vi havde ændret filen til. Så det er godt nok! Filen, den vi rettede til, er lagt op i containeren igen.

Nu må vi hellere forlade vores container igen:

`exit`

Eksempel output:

```
root@65c79750357d:/etc/nginx/conf.d# exit
exit
dtmek@docker2:~/3web$
```

Så er vi tilbage fra vores container, og nu kan vi også se om vores reverse proxye virker. Åben en webbrowser og gå til din maskine (**`http:// http://ip_på_worker_makine`**) Vi skal jo til port 80, så vi behøver ikke sætte :80 bag på. Det er standard underforstået, hvis der ikke skrives et portnummer efter adressen på en webside.

Eksempel output:

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

Hvad nu? Nu driller det igen? Vi skulle have set vores webserver container? Hvad er der nu galt? Bare rolig. Som i alt andet ved linux, er en ændring i en konfigurationsfil ikke aktiv før vi har genstartet vores service, og vi skal lige have kontrolleret vores formatering af filen også, og det gør vi selvfølgelig i vores proxy container.

Skriv følgende kommando:

`docker exec nginx-base nginx -t`

Forklaring:
Vi kan se at vi kører en kommando i containeren nginx-base. Det er en nginx kommando der verificerer om vi har lavet nogen skrivefejl i vores fil. (**`nginx -t`**)

Eksempel output:

```
dtmek@docker2:~/3web$ docker exec nginx-base nginx -t
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
dtmek@docker2:~/3web$
```

Det gik jo fint. Alt ok! Nu skal vi kun genstarte vores nginx service i containeren, og det gør vi ved at skrive:

`docker exec nginx-base nginx -s reload`

Forklaring:
Vi udfører igen en kommando. Det er stadig en nginx commando, og den genstarter nginx servicen i containeren. (**`nginx -s reload`**)

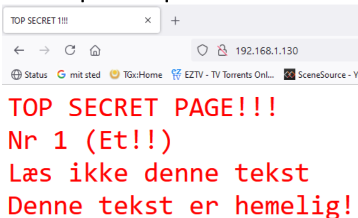
Eksempel output:

```
dtmek@docker2:~/3web$ docker exec nginx-base nginx -s reload
2024/11/13 09:21:22 [notice] 58#58: signal process started
dtmek@docker2:~/3web$
```

Så er servicen genstartet. Lad os genbesøge vores hjemmeside:

`http://ip_på_worker_makine`

Eksempel output:



YES! Så bliver vi ledt videre til vores nginx webserver! Det var jo fantastisk, det virker, så skal vi jo bare.. ... øhh... Lige et øjeblik. Vi har jo lavet nogle ændringer til vores container. Ændringen er jo ikke lavet på vores image. Så hvis vi genstarter bliver alt jo glemt. En container er jo kun kørende indtil den bliver lukket når vi indsætter et **`--rm`**, og det gjorde vi jo. Hvad gør vi nu? Skal vi hver gang igennem alt det vi lige har lavet? For alle de 3 website containere?

Selvfølgelig skal man ikke det. Vi kan faktisk gemme ændringer vi har lavet i et image, så det gør vi lige:

```
docker commit nginx-base proxy1
```

Forklaring:
`docker commit`: Dette er kommandoen der skriver en kørende container i et image.
`Nginx-base`: Navnet på den container der skal gemmes.
`proxy1`: Navnet (tagget) på imaget

Eksempel output:

```
dtmek@docker2:~/3web$ docker commit nginx-base proxy1
sha256:fd056e25ef16f8c2b8eb9d31836ce88583166cdee608e7ec148410d2fc8084b7
dtmek@docker2:~/3web$
```

Så er den gemt som image. Vi må hellere lige se, om vi kan se det nye image:

```
docker image ls -a
```

Eksempel output:

```
dtmek@docker2:~/3web$ docker image ls -a
REPOSITORY    TAG       IMAGE ID       CREATED              SIZE
proxy1        latest    fd056e25ef16   About a minute ago   192MB
nginx         latest    3b25b682ea82   5 weeks ago          192MB
dtmek@docker2:~/3web$
```

Der ligger den jo, vores proxy1 image. Læg mærke til at den har fået et andet image ID. Det er jo heller ikke vores standard nginx image mere ... Men virker den så? Vi må straks prøve det. Vi killer vores nginx-base:

```
docker kill nginx-base
```

Eksempel output:

```
dtmek@docker2:~/3web$ docker kill nginx-base
nginx-base
dtmek@docker2:~/3web$
```

Så er vores base image væk. Hvis vi nu prøvede at besøge vores webside (**`http://ip_på_worker_makine`**) ville man få at vide at det ikke var muligt. Men vi skal jo også have vores proxy1 reverse proxy kørende for at det virker. Lad os starte proxy1 imaget på vores normale måde:

```
docker run --rm -d --network web --ip 192.168.180.71 --name proxy1 -p 80:80
proxy1
```

Eksempel output:

```
dtmek@docker2:~/3web$ docker run --rm -d --network web --ip 192.168.180.71 --name proxy1 -p 80:80 proxy1
651544c285bf4d0047991910126f74049b31c4d718ae48ec3e58eef6cff4f198
dtmek@docker2:~/3web$
```

Nu kører vores reverse proxy.. Teoretisk.. besøg lige websiden, og lad os se om det også er rigtig at siden kommer frem:

`http://ip_på_worker_makine`

Ingen eksempel output.

Nu har vi jo lavet en revese-proxy til en af vores webservere. Vi vil jo gerne have dette her lavet for alle 3 af vores webserver. Det er det, det næste kapitel handler om. Vores containere er fjernet når vi laver en kill, så vi killer lige vores 2 containere. Det gør vi på en gang:

`docker kill web1 proxy1`

Eksempel output:

```
dtmek@docker2:~/3web$ docker kill web1 proxy1
web1
proxy1
dtmek@docker2:~/3web$
```

Vi fjerner også lige det image vi lavede da vi gemte vores ændringer i containeren:

`docker image rm proxy1`

Eksempel output:

```
dtmek@docker2:~/3web$ docker image rm proxy1
Untagged: proxy1:latest
Deleted: sha256:fd056e25ef16f8c2b8eb9d31836ce88583166cdee608e7ec148410d2fc8084b7
Deleted: sha256:6b022d6252b0fe730fdc69517bd6b0c64c24353eec42741113ae8c92a2d6e2ad
dtmek@docker2:~/3web$
```

Slut på Kapitel 4: Reverse proxy til intern container.

Kapitel 5: 3 web og 3 reverse proxy

Er det ikke lidt bøvlet at skulle lave en proxy til hver vores webside, når vi nu har 3 websider? Og hvad hvis man har 100 websider? Det ville tage tid at ændre hver eneste proxy server, og hvad når image bliver opdateret? I stedet for gør vi det, ved at mounte vores fil, som read only, til vores container, på samme måde som vi mounter vores 3 websider til vores webserver. Først starter vi lige vores 3 web containere. Vi starter dem på en gang:

docker run --rm -d --network web --ip 192.168.180.81 --hostname web1 --name web1 -v ./web1:/usr/share/nginx/html nginx

docker run --rm -d --network web --ip 192.168.180.82 --hostname web2 --name web2 -v ./web2:/usr/share/nginx/html nginx

docker run --rm -d --network web --ip 192.168.180.83 --hostname web3 --name web3 -v ./web3:/usr/share/nginx/html nginx

Forklaring:
Vi har startet vores 3 webside containere med vores websider liggende på hosten i vores netværk der hedder web, og vi har ikke ført nogen porte igennem til vores Host.

Eksempel output:

```
dtmek@docker2:~/3web$ docker run --rm -d --network web --ip 192.168.180.81 --hostname web1 --name web1 -v ./web1:/usr/share/nginx/html nginx

docker run --rm -d --network web --ip 192.168.180.82 --hostname web1 --name web2 -v ./web2:/usr/share/nginx/html nginx

docker run --rm -d --network web --ip 192.168.180.83 --hostname web1 --name web3 -v ./web3:/usr/share/nginx/html nginx
6ee0451af409ab2121f9db0e48e8bcba99e97db6fae707ab1357bc53f6b3c6c9
bd3b6107b1f38e7cab60a4535990df94a0c6495190dd0859be73d1f75e7994b6
8e3412b7adf8553eb454744073307694f91dd76f7b74e645225cf1512c13b0d2
dtmek@docker2:~/3web$
```

Nu skal vi starte vores proxy'er, og samtidig bruger vi forskellige konfigurations filer i hver proxy container. Vi starter alle 3 på en gang:

docker run --rm -d -v ./81default.conf:/etc/nginx/conf.d/default.conf:ro --network web --ip 192.168.180.71 --name proxy1 -p 81:80 nginx

docker run --rm -d -v ./82default.conf:/etc/nginx/conf.d/default.conf:ro --network web --ip 192.168.180.72 --name proxy2 -p 82:80 nginx

docker run --rm -d -v ./83default.conf:/etc/nginx/conf.d/default.conf:ro --network web --ip 192.168.180.73 --name proxy3 -p 83:80 nginx

Forklaring:
Som man kan se er der ikke så meget forskel i forhold til alle de andre containere vi har startet med docker run. Forskellen ligger sådan set i at vi sender 3 forskellige porte igennem, til hver sin container. Og at vi har sendt en konfigurationsfil til containeren via hosten. I er velkommen til at kigge på de 3 konfigurationsfiler (**cat filnavn**, husk der er forskel på store og små bogstaver i linux) Det der kan fremhæves i de 3 konfigurationsfiler er at alle containere "lytter" på port 80, da vi først omdirigerer til ny port når vi starter containeren, og at de IP adresser (webservere) der ledes videre til, selvfølgelig er forskellige.

Eksempel output på næste side.

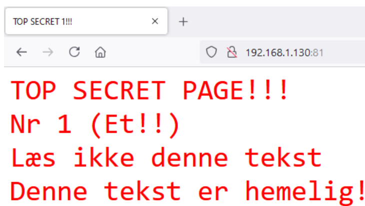
Eksempel output:

```
dtmek@docker2:~/3web$ docker run --rm -d -v ./81default.conf:/etc/nginx/conf.d/default.conf:ro --network web --ip 192.168.180.71 --name proxy1 -p 81:80 nginx

docker run --rm -d -v ./82default.conf:/etc/nginx/conf.d/default.conf:ro --network web --ip 192.168.180.72 --name proxy2 -p 82:80 nginx

docker run --rm -d -v ./83default.conf:/etc/nginx/conf.d/default.conf:ro --network web --ip 192.168.180.73 --name proxy3 -p 83:80 nginx
5cd74c6c88a41084ce5b855df88ff561b6f4eb1779bf79c2d1ac2b3641dbeaaf
0d1307b25b42d75e0aafd6f1f21057592a6d438772564baa7fa1135d266e882a
045f7ccc01701ff55ac7a60034cbca98d6c472568c4a6366317c23f37807d650
dtmek@docker2:~/3web$
```

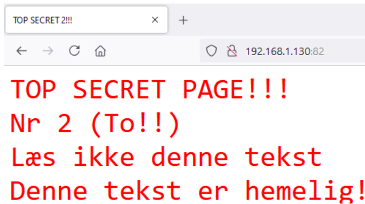
Så skulle de alle køre. Prøv at besøge de 3 sider:

`http://ip_på_din_docker_maskine:81`
`http://ip_på_din_docker_maskine:82`
`http://ip_på_din_docker_maskine:83`

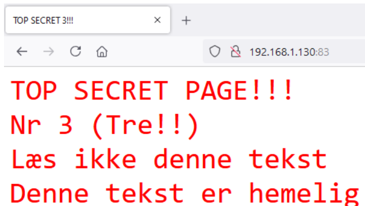
Eksempel output (Viser alle 3 websider, med adresser):
Webside:81



Webside:82



Webside:83



Nu kører alle sider. Men... Glemte vi ikke noget vigtigt? Jo.. vi skrev jo **`--rm`** Vi var jo i gang med at udvikle, Det betyder at containerne stopper når vi genstarter server. Det må vi ændre. Nu skal vores hjemmesider jo helst overleve en genstart. Det første er at stoppe alle vores containere, og fjerne dem:

```
docker kill web1 web2 web3 proxy1 proxy2 proxy3
```

Eksempel output:

```
dtmek@docker2:~/3web$ docker kill web1 web2 web3 proxy1 proxy2 proxy3
web1
web2
web3
proxy1
proxy2
proxy3
dtmek@docker2:~/3web$
```

Hvis Ikke alle containere er væk (Se efter med **`docker container ls -a`**) kan de slettes med:

```
docker container rm navn1 navn2 navn3
```

Nu starter vi alle vores containere igen, hvor vi kun har udskiftet **`--rm`** til **`--restart unless-stopped`**, derefter prøver vi at genstarte vores host server. Vi starter selvfølgelig alle på en gang:

```
docker run --restart unless-stopped -d --network web --ip 192.168.180.81 --hostname web1
--name web1 -v ./web1:/usr/share/nginx/html nginx
docker run --restart unless-stopped -d --network web --ip 192.168.180.82 --hostname web1
--name web2 -v ./web2:/usr/share/nginx/html nginx
docker run --restart unless-stopped -d --network web --ip 192.168.180.83 --hostname web1
--name web3 -v ./web3:/usr/share/nginx/html nginx

docker run --restart unless-stopped -d -v
./81default.conf:/etc/nginx/conf.d/default.conf:ro --network web --ip 192.168.180.71 --
name proxy1 -p 81:80 nginx
docker run --restart unless-stopped -d -v
./82default.conf:/etc/nginx/conf.d/default.conf:ro --network web --ip 192.168.180.72 --
name proxy2 -p 82:80 nginx
docker run --restart unless-stopped -d -v
./83default.conf:/etc/nginx/conf.d/default.conf:ro --network web --ip 192.168.180.73 --
name proxy3 -p 83:80 nginx
```

Eksempel output:

```
dtmek@docker2:~/3web$ docker run --restart unless-stopped -d --network web --ip 192.168.180.81 --hostname web1 --name web1 -v ./web1:/usr/share/nginx/html nginx
docker run --restart unless-stopped -d --network web --ip 192.168.180.82 --hostname web1 --name web2 -v ./web2:/usr/share/nginx/html nginx
docker run --restart unless-stopped -d --network web --ip 192.168.180.83 --hostname web1 --name web3 -v ./web3:/usr/share/nginx/html nginx

docker run --restart unless-stopped -d -v ./81default.conf:/etc/nginx/conf.d/default.conf:ro --network web --ip 192.168.180.71 --name proxy1 -p 81:80 nginx
docker run --restart unless-stopped -d -v ./82default.conf:/etc/nginx/conf.d/default.conf:ro --network web --ip 192.168.180.72 --name proxy2 -p 82:80 nginx
docker run --restart unless-stopped -d -v ./83default.conf:/etc/nginx/conf.d/default.conf:ro --network web --ip 192.168.180.73 --name proxy3 -p 83:80 nginx
de5949c6efbdb2544805598129e02f1100c9132b845553b7122a40d06e2303e3
30d8c31c50fd41dbc3be20afac384b8a76a688f624614addf9aa76a07c29885a
26a0735828bf47f8a92f49c9bd7e36b92d55a9bfa5ee8646869d7aee01ef3604
04ec9c4bcaf8f078c44202fac583a126d02523c8958bb4b9389f3a82f3ea8a70
bbb70d731dc5af3c0f6ba3b9e2595502ecbf621ec161cd2c781b3e6cfdc05f20
49313e7f724e2442113cc979daee7885497d794801dcb8cb86718a021b5f731c
dtmek@docker2:~/3web$
```

GENSTART HOST!! (**`Sudo reboot`**, og indtast password)

Efter genstart verificer at containerne er startet igen, ved at besøge websiderne:

`http://ip_på_din_docker_maskine:81`
`http://ip_på_din_docker_maskine:82`
`http://ip_på_din_docker_maskine:83`

Ingen eksempel output.

Selvfølgelig kører containerne. Husk at logge ind på worker igen efter genstart, og hoppe ned i underbiblioteket 3web (**`cd 3web`**)

Er de der kommandoer vi bruger ikke ved at være lidt lange? Til tider MEGET lange? Jo... hvordan var det nu man kunne starte en masse uden at køre en kommando for hver container? Nåh jo. Det havde vi da om i Opgave 2. Det var ved hjælpe af en fil der skulle hedde: **docker-compose.yml** (alt med små bogstaver!). Vi laver straks den fil... øhhm.. er det ikke forfærdeligt meget tastearbejde? (kopi arbejde?) Filen ligger allerede klar, hvis du har lagt Eksempel3Materiale ind i underbiblioteket. Hvis du ikke har lagt filen ind, kan den laves med editoren, her er indholdet til filen. **VIGTIGT**!!! At alle indrykninger skal være som herunder!!!! Der anbefales at kopier filen fra det materiale der kan downloades:

```
####################### docker-compose.yml start ######################
services:
  web1:
    image: nginx
    hostname: web1
    container_name: web1
    restart: unless-stopped
    expose:
      - 80
    volumes:
      - ./web1:/usr/share/nginx/html
    networks:
      web:
        ipv4_address: 192.168.180.81

  web2:
    image: nginx
    hostname: web2
    container_name: web2
    restart: unless-stopped
    expose:
      - 80
    volumes:
      - ./web2:/usr/share/nginx/html
    networks:
      web:
        ipv4_address: 192.168.180.82

  web3:
    image: nginx
    hostname: web3
    container_name: web3
    restart: unless-stopped
    expose:
      - 80
    volumes:
      - ./web3:/usr/share/nginx/html
    networks:
      web:
        ipv4_address: 192.168.180.83

  proxy1:
    image: nginx
    hostname: proxy1
    container_name: proxy1
    restart: unless-stopped
    ports:
      - 81:80
```

```
    volumes:
      - ./81default.conf:/etc/nginx/conf.d/default.conf
    networks:
      web:
        ipv4_address: 192.168.180.71

  proxy2:
    image: nginx
    hostname: proxy2
    container_name: proxy2
    restart: unless-stopped
    ports:
      - 82:80
    volumes:
      - ./82default.conf:/etc/nginx/conf.d/default.conf
    networks:
      web:
        ipv4_address: 192.168.180.72

  proxy3:
    image: nginx
    hostname: proxy3
    container_name: proxy3
    restart: unless-stopped
    ports:
      - 83:80
    volumes:
      - ./83default.conf:/etc/nginx/conf.d/default.conf
    networks:
      web:
        ipv4_address: 192.168.180.73

networks:
  web:
    name: web
    driver: bridge
    ipam:
      config:
        - subnet: 192.168.180.0/24


######################## docker-compose.yml slut ########################
```

For at starte alt, kører vi selvfølgelig følgende kommando:

```
docker compose up -d
```

Forklaring:
Kommandoen har vi mødt før, den starter jo alt det der er defineret i vores fil (docker-compose.yml). og detacher (-d), så vi kan køre videre i vores ssh terminal.

Eksempel output:

```
dtmek@docker2:~/3web$ docker compose up -d
WARN[0000] a network with name web exists but was not created by compose.
Set `external: true` to use an existing network
network web was found but has incorrect label com.docker.compose.network set to ""
dtmek@docker2:~/3web$
```

Hov! Vi glemte at slette vores hjemmelavede netværk, og containerne! Vi bliver lige nødt til at rydde op først (Alt kan køres samtidigt.):

```
docker kill web1 web2 web3 proxy1 proxy2 proxy3
docker container rm web1 web2 web3 proxy1 proxy2 proxy3
docker network rm web
```

Forklaring:
Stopper og fjerne alle vores containere. Og derefter sletter netværket "web".

Eksempel output:

```
dtmek@docker2:~/3web$ docker kill web1 web2 web3 proxy1 proxy2 proxy3
docker container rm web1 web2 web3 proxy1 proxy2 proxy3
docker network rm web
web1
web2
web3
proxy1
proxy2
proxy3
web1
web2
web3
proxy1
proxy2
proxy3
web
dtmek@docker2:~/3web$
```

Så er alt væk, lad os køre docker compose up igen:

```
docker compose up -d
```

Eksempel output:

```
dtmek@docker2:~/3web$ docker compose up -d
[+] Running 7/7
 ✔ Network web         Created
 ✔ Container web3      Started
 ✔ Container proxy2    Started
 ✔ Container proxy1    Started
 ✔ Container proxy3    Started
 ✔ Container web1      Started
 ✔ Container web2      Started
dtmek@docker2:~/3web$
```

Hvad nu? Den hentede ikke noget? Den oprettede kun netværket web, og startede alle containere? Vi kigger lige om de er online i vores webbrowser.

Det finder vi ud af at de er. Men hvorfor hentede den så ikke noget på nettet? Der er en simpel forklaring. Da vi kørte alle de kommandoer for at starte vores containere sidst skrev vi jo til sidst nginx. Så det var det

grund image (parrent image) den skulle bruge, og da vores docker-compose.yml fil også kun bruger nginx så lå det image allerede på maskinen. Lad os kigge efter:

```
docker image ls -a
```

Eksempel output:

```
dtmek@docker2:~/3web$ docker image ls -a
REPOSITORY   TAG       IMAGE ID       CREATED        SIZE
nginx        latest    3b25b682ea82   5 weeks ago    192MB
dtmek@docker2:~/3web$
```

Jo… der ligger kun et image. Nu vil vi jo egentlig godt kunne gemme vores opsætning, uden at skulle bygge det hele igen. Så vi skal pushe hele systemet ud til vores registry. Men til det skal der jo oprettes et image for hver af vores containere. Vores image hedder jo ikke andet end nginx, så vi laver lige vores docker-compose.yml fil lidt om. Men først lukker vi lige det hele ned igen, med kommandoen:

```
docker compose down
```

Eksempel output:

```
dtmek@docker2:~/3web$ docker compose down
[+] Running 7/7
 ✔ Container proxy1  Removed
 ✔ Container proxy3  Removed
 ✔ Container web3    Removed
 ✔ Container web2    Removed
 ✔ Container proxy2  Removed
 ✔ Container web1    Removed
 ✔ Network web       Removed
dtmek@docker2:~/3web$
```

############# Udføres kun hvis du har kopieret materiale ind fra Opgave3Materiale ############

Hvis du har kopieret ind fra Opgave3Materiale ligger det hele allerede klar. Vi skal kun omdøbe den nuværende compose fil, og den tilpassede compose fil. Dette gøres med følgende kommandoer, vi kører alle kommandoer på en gang:

```
mv docker-compose.yml docker-composeold.yml
mv docker-composepush.yml docker-compose.yml
```

Forklaring:
mv betyder move, så vi flytter begge filer over i nyt navn.

Eksempel output:

```
dtmek@docker2:~/3web$ mv docker-compose.yml docker-composeold.yml
mv docker-composepush.yml docker-compose.yml
dtmek@docker2:~/3web$
```

Rediger din nye docker-compose.yml (`nano docker-compose.yml`) HUSK at indsætte IP adresse på din registry server de steder der står (Der er 6 steder det skal rettes, se fremhævede pladser i teksten til filen herunder) Hvor det skal ændres står der: **`ip_Registry_server`**.
Og husk at gemme filen igen.

####slut på Udføres kun hvis du har kopieret materiale ind fra Opgave3Materiale ########

############# Udføres kun hvis du ikke har kopieret materialet ind på server ############

Først skal vi have lavet et underbibliotek, og vi skal have oprettet en Dockerfile (Med stort D). vi bruger alle kommandoer på en gang:

```
mkdir nginx
cd nginx
echo FROM nginx >> Dockerfile
cd ..
```

Forklaring:
linje 1: Laver bibliotek nginx
linje 2: hopper ned i biblioteket
linje 3: laver Dockerfile, der kun indeholder en linje: "FROM nginx" det er navnet på vores parrent image.
linje 4: hopper ud af biblioteket nginx igen.

Eksempel output:

```
dtmek@docker2:~/3web$ mkdir nginx
cd nginx
echo FROM nginx >> Dockerfile
cd ..
dtmek@docker2:~/3web$
```

Nu skal vi lave vores nye docker-compose.yml fil, først omdøber vi den gamle:

```
mv docker-compose.yml docker-composeold.yml
```

Eksempel output:

```
dtmek@docker2:~/3web$ mv docker-compose.yml docker-composeold.yml
dtmek@docker2:~/3web$
```

Start editor med navnet på den fil du skal oprette (**`nano docker-compose.yml`**) Sæt efterfølgende tekst ind, og gem filen. HUSK at indsætte IP adresse på din registry server de fremhævede steder (Der er 6 steder det skal rettes) INDEN du gemmer sidste gang **VIGTIGT**!!! At alle indrykninger skal være som herunder!!!! Der anbefales at kopier filen fra det materiale der kan downloades:

Teksten begynder på næste side.

```
####################### docker-compose.yml start ######################
services:
  web1:
    image: ip_Registry_server:5000/web1:latest
    build: ./nginx/
    hostname: web1
    container_name: web1
    restart: unless-stopped
    expose:
      - 80
    volumes:
      - ./web1:/usr/share/nginx/html
    networks:
      web:
        ipv4_address: 192.168.180.81

  web2:
    image: ip_Registry_server:5000/web2:latest
    build: ./nginx/
    hostname: web2
    container_name: web2
    restart: unless-stopped
    expose:
      - 80
    volumes:
      - ./web2:/usr/share/nginx/html
    networks:
      web:
        ipv4_address: 192.168.180.82

  web3:
    image: ip_Registry_server:5000/web3:latest
    build: ./nginx/
    hostname: web3
    container_name: web3
    restart: unless-stopped
    expose:
      - 80
    volumes:
      - ./web3:/usr/share/nginx/html
    networks:
      web:
        ipv4_address: 192.168.180.83

  proxy1:
    image: ip_Registry_server:5000/proxy1:latest
    build: ./nginx/
    hostname: proxy1
    container_name: proxy1
    restart: unless-stopped
    ports:
      - 81:80
    volumes:
      - ./81default.conf:/etc/nginx/conf.d/default.conf
    networks:
      web:
        ipv4_address: 192.168.180.71
```

```
  proxy2:
    image: ip_Registry_server:5000/proxy2:latest
    build: ./nginx/
    hostname: proxy2
    container_name: proxy2
    restart: unless-stopped
    ports:
      - 82:80
    volumes:
      - ./82default.conf:/etc/nginx/conf.d/default.conf
    networks:
      web:
        ipv4_address: 192.168.180.72

  proxy3:
    image: ip_Registry_server:5000/proxy3:latest
    build: ./nginx/
    hostname: proxy3
    container_name: proxy3
    restart: unless-stopped
    ports:
      - 83:80
    volumes:
      - ./83default.conf:/etc/nginx/conf.d/default.conf
    networks:
      web:
        ipv4_address: 192.168.180.73

networks:
  web:
    name: web
    driver: bridge
    ipam:
      config:
        - subnet: 192.168.180.0/24


####################### docker-compose.yml slut #######################
```

Slut på Udføres kun hvis du ikke har kopieret materialet ind på server

Som vi kan se er der 2 linjer der er blevet ændret ved hvert image, det er disse 2 linjer:

```
image: ip_Registry_server:5000/imagenavn:latest
build: ./nginx/
```

Forklaring:
Linje 1: Her fortæller vi hvad det image vi opretter skal hedde og vi har selvfølgelig indtastet vores registrys IP.
Linje 2: fortæller hvor vi finder "byggevejledning" til vores image henne. Det var den vejledning der kun fortalte at vore image bygges på nginx:latest. Alt andet opsætning bliver jo mountet til vores containere/images.

Ingen eksempel output:

Lad os nu starte vores containere:

```
docker compose up -d
```

eksempel output (ikke det fulde output):

```
 ! web2 Warning   manifest for 192.168.1.131:5000/web2:latest not found: manifest unknown: manifest unknown
 ! proxy1 Warning manifest for 192.168.1.131:5000/proxy1:latest not found: manifest unknown: manifest unknown
 ! proxy3 Warning manifest for 192.168.1.131:5000/proxy3:latest not found: manifest unknown: manifest unknown
 ! proxy2 Warning manifest for 192.168.1.131:5000/proxy2:latest not found: manifest unknown: manifest unknown
 ! web1 Warning   manifest for 192.168.1.131:5000/web1:latest not found: manifest unknown: manifest unknown
 ! web3 Warning   manifest for 192.168.1.131:5000/web3:latest not found: manifest unknown: manifest unknown
```

Det har vi set før. Den siger nu igen, at den ikke kan finde disse images på vores registry endnu, men det er nok fordi vikke har pushed endnu. Og ellers er der en hel masse tekst.

```
 ✔ Network web          Created
 ✔ Container proxy3     Started
 ✔ Container web2       Started
 ✔ Container web1       Started
 ✔ Container proxy2     Started
 ✔ Container proxy1     Started
 ✔ Container web3       Started
dtmek@docker2:~/3web$
```

Og som vi kan se afslutter den med at starte alle containere, og oprette netværket web.

Lad os lige se hvilke containere der kører:

```
docker ps
```

Eksempel output:

```
dtmek@docker2:~/3web$ docker ps
CONTAINER ID   IMAGE                                 COMMAND                  CREATED         STATUS         PORTS                                 NAMES
cc3103328d3b   192.168.1.131:5000/web1:latest        "/docker-entrypoint.…"   2 minutes ago   Up 2 minutes   80/tcp                                web1
5c5c007067d7   192.168.1.131:5000/web2:latest        "/docker-entrypoint.…"   2 minutes ago   Up 2 minutes   80/tcp                                web2
cc2ec8e771dd   192.168.1.131:5000/proxy1:latest      "/docker-entrypoint.…"   2 minutes ago   Up 2 minutes   0.0.0.0:81->80/tcp, [::]:81->80/tcp   proxy1
c1ad06ad6891   192.168.1.131:5000/proxy2:latest      "/docker-entrypoint.…"   2 minutes ago   Up 2 minutes   0.0.0.0:82->80/tcp, [::]:82->80/tcp   proxy2
72098883d80e   192.168.1.131:5000/web3:latest        "/docker-entrypoint.…"   2 minutes ago   Up 2 minutes   80/tcp                                web3
21e93cd97e9b   192.168.1.131:5000/proxy3:latest      "/docker-entrypoint.…"   2 minutes ago   Up 2 minutes   0.0.0.0:83->80/tcp, [::]:83->80/tcp   proxy3
dtmek@docker2:~/3web$
```

Under IMAGE kan der ses at der er lavet et image til hver af vores container, kan det mon være rigtig? Lad os lige se efter:

`docker image ls -a`

Eksempel output:

```
dtmek@docker2:~/3web$ docker image ls -a
REPOSITORY                  TAG       IMAGE ID       CREATED       SIZE
192.168.1.131:5000/proxy3   latest    d469abac2cb3   5 weeks ago   192MB
192.168.1.131:5000/proxy1   latest    6344115479c5   5 weeks ago   192MB
192.168.1.131:5000/web2     latest    ae42859940c8   5 weeks ago   192MB
192.168.1.131:5000/proxy2   latest    9a1ce5245de4   5 weeks ago   192MB
192.168.1.131:5000/web1     latest    77344f24249f   5 weeks ago   192MB
192.168.1.131:5000/web3     latest    d809fe31d71b   5 weeks ago   192MB
nginx                       latest    3b25b682ea82   5 weeks ago   192MB
dtmek@docker2:~/3web$
```

Det var rigtig nok at der er et image til hver af vores container, og vores parrent image ligger der også (nginx) så det var derfor det gik så hurtigt!

Slut på Kapitel 5: 3 web og 3 reverse proxy

Kapitel 6: Pushing and pulling

Nu kører det hele, så måske vi skulle have hele vores system over på vores registry server. Det kan vi jo gøre af en omgang med en kommando, som vi har prøvet en enkel gang før:

`docker compose push`

Eksempel output:

```
dtmek@docker2:~/3web$ docker compose push
[+] Pushing 0/42
 Pushing 192.168.1.131:5000/web1:latest: e4e9e9ad93c2 Mounted from web2
 Pushing 192.168.1.131:5000/web1:latest: 6ac729401225 Mounted from web2
 Pushing 192.168.1.131:5000/web1:latest: 8ce189049cb5 Mounted from web2
 Pushing 192.168.1.131:5000/web1:latest: 296af1bd2844 Mounted from web2
 Pushing 192.168.1.131:5000/web1:latest: 63d7ce983cd5 Mounted from web2
 Pushing 192.168.1.131:5000/web1:latest: b33db0c3c3a8 Mounted from web2
 Pushing 192.168.1.131:5000/web1:latest: 98b5f35ea9d3 Mounted from web2
 Pushing 192.168.1.131:5000/web2:latest: e4e9e9ad93c2 Mounted from web
 Pushing 192.168.1.131:5000/web2:latest: 6ac729401225 Mounted from web
 Pushing 192.168.1.131:5000/web2:latest: 8ce189049cb5 Mounted from web
 Pushing 192.168.1.131:5000/web2:latest: 296af1bd2844 Mounted from web
 Pushing 192.168.1.131:5000/web2:latest: 63d7ce983cd5 Mounted from web
 Pushing 192.168.1.131:5000/web2:latest: b33db0c3c3a8 Mounted from web
 Pushing 192.168.1.131:5000/web2:latest: 98b5f35ea9d3 Mounted from php
 Pushing 192.168.1.131:5000/web3:latest: e4e9e9ad93c2 Mounted from web1
 Pushing 192.168.1.131:5000/web3:latest: 6ac729401225 Mounted from web1
 Pushing 192.168.1.131:5000/web3:latest: 8ce189049cb5 Mounted from web1
 Pushing 192.168.1.131:5000/web3:latest: 296af1bd2844 Mounted from web1
 Pushing 192.168.1.131:5000/web3:latest: 63d7ce983cd5 Mounted from web1
 Pushing 192.168.1.131:5000/web3:latest: b33db0c3c3a8 Mounted from web1
 Pushing 192.168.1.131:5000/web3:latest: 98b5f35ea9d3 Mounted from web1
 Pushing 192.168.1.131:5000/proxy1:latest: e4e9e9ad93c2 Mounted from web3
 Pushing 192.168.1.131:5000/proxy3:latest: e4e9e9ad93c2 Mounted from proxy1
 Pushing 192.168.1.131:5000/proxy3:latest: 6ac729401225 Mounted from proxy1
 Pushing 192.168.1.131:5000/proxy3:latest: 8ce189049cb5 Mounted from proxy1
 Pushing 192.168.1.131:5000/proxy3:latest: 296af1bd2844 Mounted from proxy1
 Pushing 192.168.1.131:5000/proxy3:latest: 63d7ce983cd5 Mounted from proxy1
 Pushing 192.168.1.131:5000/proxy3:latest: b33db0c3c3a8 Mounted from proxy1
 Pushing 192.168.1.131:5000/proxy3:latest: 98b5f35ea9d3 Mounted from proxy1
 Pushing 192.168.1.131:5000/proxy1:latest: 6ac729401225 Mounted from web3
 Pushing 192.168.1.131:5000/proxy1:latest: 8ce189049cb5 Mounted from web3
 Pushing 192.168.1.131:5000/proxy1:latest: 296af1bd2844 Mounted from web3
 Pushing 192.168.1.131:5000/proxy1:latest: 63d7ce983cd5 Mounted from web3
 Pushing 192.168.1.131:5000/proxy1:latest: b33db0c3c3a8 Mounted from web3
 Pushing 192.168.1.131:5000/proxy1:latest: 98b5f35ea9d3 Mounted from web3
 Pushing 192.168.1.131:5000/proxy2:latest: e4e9e9ad93c2 Mounted from proxy3
 Pushing 192.168.1.131:5000/proxy2:latest: 6ac729401225 Mounted from proxy3
 Pushing 192.168.1.131:5000/proxy2:latest: 8ce189049cb5 Mounted from proxy3
 Pushing 192.168.1.131:5000/proxy2:latest: 296af1bd2844 Mounted from proxy3
 Pushing 192.168.1.131:5000/proxy2:latest: 63d7ce983cd5 Mounted from proxy3
 Pushing 192.168.1.131:5000/proxy2:latest: b33db0c3c3a8 Mounted from proxy3
 Pushing 192.168.1.131:5000/proxy2:latest: 98b5f35ea9d3 Mounted from proxy3
dtmek@docker2:~/3web$
```

Holy S…! det gik hurtigt!! Men som vi kan se havde den allerede det meste liggende. Så den har kun overført det der var forskelligt. Men vi må heller se om det hele virker, så vi stopper alt med kommandoen:

`docker compose down`

Eksempel output:

```
dtmek@docker2:~/3web$ docker compose down
[+] Running 7/7
 ✔ Container proxy1  Removed
 ✔ Container proxy3  Removed
 ✔ Container web2    Removed
 ✔ Container proxy2  Removed
 ✔ Container web1    Removed
 ✔ Container web3    Removed
 ✔ Network web       Removed
dtmek@docker2:~/3web$
```

Lad os se om vi mangler noget mere:

Hvad med kørende containere?

`docker ps`

Eksempel output:

```
dtmek@docker2:~/3web$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
dtmek@docker2:~/3web$
```

Der var ikke noget, hvad med inaktive containere?:

`docker container ls -a`

Eksempel outpout:

```
dtmek@docker2:~/3web$ docker container ls -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
dtmek@docker2:~/3web$
```

Der var heller ikke noget. Hvad med images?

`docker image ls -a`

Eksempel output:

```
dtmek@docker2:~/3web$ docker image ls -a
REPOSITORY                   TAG       IMAGE ID       CREATED       SIZE
192.168.1.131:5000/web3      latest    d809fe31d71b   5 weeks ago   192MB
nginx                        latest    3b25b682ea82   5 weeks ago   192MB
192.168.1.131:5000/proxy3    latest    d469abac2cb3   5 weeks ago   192MB
192.168.1.131:5000/proxy1    latest    6344115479c5   5 weeks ago   192MB
192.168.1.131:5000/web2      latest    ae42859940c8   5 weeks ago   192MB
192.168.1.131:5000/proxy2    latest    9a1ce5245de4   5 weeks ago   192MB
192.168.1.131:5000/web1      latest    77344f24249f   5 weeks ago   192MB
dtmek@docker2:~/3web$
```

Ok! Der ligger noget det sletter vi lige, husk at slette ALLE. Et trick er at trykke det første bogstav af navnet på imaget, og så på TAB tasten. Så udfylder maskinen det den kan indtil næste tegn hvor der findes flere muligheder. Tryk endnu et af de næste bogstaver og tryk TAB tasten. Dette kan gentages i det uendelige, indtil hele navnet er stående. Derefter mellemrum, og man fortsætter med det næste navn, indtil alle images er nævnt i linjen. HUSK nginx imaget! (Man kan se om man har slettet alle med kommandoen **`docker image ls -a`**):

`docker image rm image 1 image2 image3` osv.

Eksempel output på næste side.

Eksempel output:

```
dtmek@docker2:~/3web$ docker image rm 192.168.1.131:5000/web1:latest 192.168.1.131:5000/web2:latest 192.168.1.131:5000/web3:latest 192.168.1.131:5000/proxy1:latest 192.168.1.131:5000/proxy2:latest 192.168.1.131:5000/proxy3:latest nginx:l
atest
Untagged: 192.168.1.131:5000/web1:latest
Untagged: 192.168.1.131:5000/web1@sha256:9d5a36c11fde73e24084ed5cc62011a626add9e9b8da753589a00a0404e8763b
Deleted: sha256:77344f24249f18467029714264cee3b8e525c98025ae161dce0634d02f4e1785
Untagged: 192.168.1.131:5000/web2:latest
Untagged: 192.168.1.131:5000/web2@sha256:e0dc3337df5e3ff6361aac47abd9ab4f28629f4ba51474d71b2741f41f8b7f05
Deleted: sha256:ae42859940c8b22a0d109c3e810a12ded3e631afd2591b3ef0e84695974e51bd
Untagged: 192.168.1.131:5000/web3:latest
Untagged: 192.168.1.131:5000/web3@sha256:83eb5677ac5086ab868ae6aa5b988cf5d684e83c03be1c5a83f50a70a7a2d70d
Deleted: sha256:d809fe31d71bff75cc45b2b9500172c789a3b473cf44888f8823ccf86b85fca3
Untagged: 192.168.1.131:5000/proxy1:latest
Untagged: 192.168.1.131:5000/proxy1@sha256:d4ab3fb5566bd8ba66022c114f4eb9102ed8081cd7a8460b8f0989fa4a5ed8c9
Deleted: sha256:6344115479c55d5e8d7f4de1b220f4ae28218d24d6ff7ac078ded81c5d86ce20
Untagged: 192.168.1.131:5000/proxy2:latest
Untagged: 192.168.1.131:5000/proxy2@sha256:1ad23b68c36aedcef809871b84e9ff8b937629917e03fc1f30ace0decd626307
Deleted: sha256:9a1ce5245de45d96097b714bcabfa6c91314167a648f470d1de142e9ab3c39b5
Untagged: 192.168.1.131:5000/proxy3:latest
Untagged: 192.168.1.131:5000/proxy3@sha256:6c30a4aa88899ed4355d1389b3bd562080937ac04b4dd6256374dea93992b92d
Deleted: sha256:d469abac2cb37fc742edd32032a3770680acf4bebc0a69b182b89699498b723c
Untagged: nginx:latest
Untagged: nginx@sha256:28402db69fec7c17e179ea87882667f1e054391138f77ffaf0c3eb388efc3ffb
Deleted: sha256:3b25b682ea82b2db3cc4fd48db818be788ee3f902ac7378090cf2624ec2442df
dtmek@docker2:~/3web$
```

Det var så det sidste der var væk. Men vi må hellere huske at køre en:

`docker system prune`

Eksempel output:

```
dtmek@docker2:~/3web$ docker system prune
WARNING! This will remove:
  - all stopped containers
  - all networks not used by at least one container
  - all dangling images
  - unused build cache

Are you sure you want to continue? [y/N] y
Deleted build cache objects:
w1c2nf6yp3sa3xblvjkqw98kg
71zikjo607plzi7ph8r0ied4b
ql2d3d2l96t4gfuhu77g80prn
scktcqhu7gug5rexsvz8uh46u
lbc3fh0qwm6lomstxmqg32yde
lba8iqsiovg5j53kztoplnkx4
qb9btegad2ny8rw2bsbnkfe8s
mhcbgkb66tgo8r72tocbpl9or
rrhy2xsg307aphdztwhwdplhf
yt9lppoklp317hqbxthqx9mo5
vvhmht7kijo0tptqdvldckv2j
m73csjulihvw4eh2em36cs5hn
x16k0kbn2qax9oa62xbl31nc4
l347ufj7kjzmaiksr78unrueh
yrdqcu23jmjovm07lhggwn9q0
up8nym763ytz44tfh6ailjzrd
2k2lizlq9fn9l99zxlvprzsiv
ho66a5eks3hkxjademfwnqikm
qmcddi3rg845b0ccc0v577z9k

Total reclaimed space: 66B
dtmek@docker2:~/3web$
```

Så er det sidste væk, lad os pulle:

`docker compose pull`

Eksempel output:

```
dtmek@docker2:~/3web$ docker compose pull
[+] Pulling 13/13
 ✔ web2 Pulled
 ✔ web1 Pulled
 ✔ proxy1 Pulled
 ✔ web3 Pulled
 ✔ proxy2 Pulled
   ✔ a480a496ba95 Pull complete
   ✔ f3ace1b8ce45 Pull complete
   ✔ 11d6fdd0e8a7 Pull complete
   ✔ f1091da6fd5c Pull complete
   ✔ 40eea07b53d8 Pull complete
   ✔ 6476794e50f4 Pull complete
   ✔ 70850b3ec6b2 Pull complete
 ✔ proxy3 Pulled
dtmek@docker2:~/3web$
```

Hvorfor I alverden henter den ikke mere? Jo.. der var jo kun ét grund image: nginx, så det var det der fyldte med de få linjer markeret "pull cmplete" alt andet var jo kun små ændringer.

Lad os starte vores containere, og se om det virker:

`docker compose up -d`

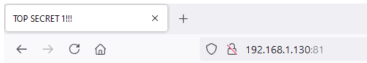
Eksempel output:

```
dtmek@docker2:~/3web$ docker compose up -d
[+] Running 7/7
 ✔ Network web         Created
 ✔ Container web3      Started
 ✔ Container proxy2    Started
 ✔ Container web1      Started
 ✔ Container proxy3    Started
 ✔ Container proxy1    Started
 ✔ Container web2      Started
dtmek@docker2:~/3web$
```

Prøv at hoppe på websiderne igen:

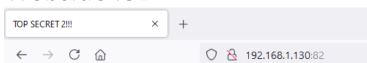
http://ip_på_din_docker_maskine:81
http://ip_på_din_docker_maskine:82
http://ip_på_din_docker_maskine:83

Eksempel output (Viser alle 3 websider, med adresser):
Webside :81



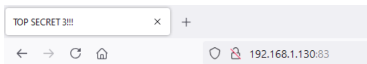
TOP SECRET PAGE!!!
Nr 1 (Et!!)
Læs ikke denne tekst
Denne tekst er hemelig!

Webside :82



TOP SECRET PAGE!!!
Nr 2 (To!!)
Læs ikke denne tekst
Denne tekst er hemelig!

Webside :83



TOP SECRET PAGE!!!
Nr 3 (Tre!!)
Læs ikke denne tekst
Denne tekst er hemelig!

Yes det virker!!

Slut på Kapitel 6: Pushing and pulling, eller ikke helt...

Tilbage i denne opgave er der kun at rydde op. Det er jo nemt, da vi jo har en docker-compose.yml fil:

docker compose down

Eksempel output:

```
dtmek@docker2:~/3web$ docker compose down
[+] Running 7/7
 ✔ Container web2     Removed
 ✔ Container proxy1   Removed
 ✔ Container proxy3   Removed
 ✔ Container web1     Removed
 ✔ Container web3     Removed
 ✔ Container proxy2   Removed
 ✔ Network web        Removed
dtmek@docker2:~/3web$
```

Vi skal lige huske at slette vores images også (**docker image rm image1 image2 image3** osv.)

Du skal ikke slette noget fra biblioteket, vi skal bruge det i opgave 4.

Hermed slut på Opgave 3.