Docker 

Praktični učbenik o Dockerju
na **Ubuntu Linuxu**.

zbral in uredil: Simon Pogorelčnik

Kaj je Docker?

Docker je platforma za **kontejnerizacijo**, ki nam omogoča, da aplikacije in njihove odvisnosti zapakiramo v prenosljive enote, imenovane **kontejnerji**. Ta pristop je lažji in hitrejši od tradicionalne virtualizacije, saj kontejnerji delijo jedro gostiteljskega operacijskega sistema (Ubuntu).

Docker je tudi platforma za razvijalce in sistemske administratorje za razvoj, pošiljanje in izvajanje aplikacij z uporabo kontejnerjev. Za razliko od tradicionalne virtualizacije, kjer se virtualizira celotna operacijska sistem, Docker deli jedro OS z gostiteljem.

Ključne prednosti:

- Hitrejši zagon kot VM
- Manjša poraba virov
- Enostavno pakiranje aplikacij
- Dosledno okolje med razvojem in produkcijo

Vsebine, ki so v tem učbeniku (na kratko):

1. Osnove delovanje Docker-ja
2. Teoretične osnove: Razlika med **sliko (Image)** in **kontejnerjem**.
3. Pripravo okolja: **Namestitev** Dockerja na Ubuntu.
4. Prvi korak: **Zagon** prvega kontejnerja in uporaba osnovnih ukazov.
5. Docker Volumes – shranjevanje podatkov
6. Docker Compose - avtomatizacija
7. Docker Omrežja – povezovanje med aplikacijami
8. Praktični del

Osnovni pojmi delovanja docker-ja

Začnimo z najbolj pomembnim delom teorije: razliko med **sliko (Image)** in **kontejnerjem (Container)**. To je temelj delovanja Dockerja.

Slika/Kontejner

1. **Slika (Image):**

- Je nespremenljiva (immutable) predloga, ki vsebuje vse, kar je potrebno za zagon aplikacije: kodo, knjižnice, okoljske spremenljivke in konfiguracijske datoteke.
- Slika je sestavljena iz sklopa bralnih plasti (**read-only layers**). Ko enkrat ustvarimo sliko, se ta ne spremeni.
- Deluje kot predloga za naše kontejnerje.

2. **Kontejner (Container):**

- Je **živa, (run-time) instanca** slike. Ko zaženemo sliko, se ustvari kontejner.
- Kontejner doda pisalno plast (**writable layer**) nad plasti slike, kar mu omogoča, da med delovanjem shranjuje podatke in izvaja spremembe.
- Je popolnoma izolirano okolje, kjer se dejansko izvede vaša aplikacija.

Slika služi kot statična osnova, medtem ko je kontejner dinamičen zagon te osnove.

Teoretične osnove

Če ste odgovorili, *da je dovolj, da spremenimo samo kontejner*… preberite odgovor.

Tehnično gledano imaš prav: Ko se kontejner izvaja, se vse spremembe (kot je spreminjanje konfiguracijske datoteke) shranijo v **pisalno plast** (writable layer) znotraj tega kontejnerja.

Vendar se moramo tukaj naučiti ključnega Dockerjevega načela: **začasnost** (efemernost) kontejnerjev.

Če ta kontejner ustavimo in nato izbrišemo (kar je pogosta praksa), se vse spremembe, ki smo jih naredili, **izgubijo**. Kontejnerji niso namenjeni za trajno shranjevanje podatkov, ki so ključni za delovanje aplikacije.

Osnove ponovljivosti (Reproducibility)

Za trajne, kontrolirane in **ponovljive** spremembe (da lahko točno določeno konfiguracijo ponovno zaženemo kjerkoli in kadarkoli), moramo **ustvariti novo sliko**.

To nas pripelje do vprašanja: **kako ustvarimo lastno sliko, ki vključuje našo spremenjeno konfiguracijo ali kodo?**

Če želimo ustvariti svojo sliko, moramo napisati recept zanjo. Ta recept se imenuje **Dockerfile** .

Priprava okolja

Dockerfile: navodila za docker image

Dockerfile je navadna besedilna datoteka, ki vsebuje serijo ukazov (navodil), ki jih Docker samodejno izvede, da ustvari vaš docker image. Je kot skripta za gradnjo, ki zagotavlja, da je vsaka nova slika zgrajena natančno enako.

Osnovni Dockerfile za naš praktični učbenik bo vseboval vsaj tri ključne ukaze:

Ukaz (Instrukcija)	Namen
FROM	Določa osnovno sliko (bazo), na kateri gradimo.
RUN	Izvede ukaze **med gradnjo** slike (npr. namestitev programske opreme ali ustvarjanje map).
CMD	Določa, kateri ukaz se izvede ob **zagonu** kontejnerja.

Začnimo z najbolj temeljnim ukazom.

V Docker terminologiji to imenujemo **Osnovna Slika** (Base docker Image) ali **Starševska** Slika (Parent Image). Vsaka slika mora izhajati iz druge slike, saj se s tem zagotovi operacijsko okolje (npr. miniaturna različica Ubuntujevega OS-a).

Najpogosteje uporabljamo majhne in učinkovite osnovne slike, kot so: ubuntu, alpine ali node:alpine. Uporaba manjših slik je bistvena za hitre kontejnerje.

Primer uporabe ukaza FROM:

```
FROM ubuntu:latest
# Na tej točki je naša slika le čist Ubuntu OS.
```

Namestitev programske opreme

Odlično. Zdaj, ko imamo našo osnovo (ubuntu), želimo na ta OS namestiti spletni strežnik **Nginx**.

Kateri od naslednjih ukazov (instrukcij) v Dockerfile bi morali uporabiti, da izvedemo namestitev v času gradnje slike?

a) CMD
b) RUN
c) COPY

Točno tako! Uporabimo ukaz **RUN**.

Ukaz RUN izvede ukaz **med gradnjo slike**. To je bistveno, saj:

1. **Ustvari plast (Layer):** Vsak ukaz RUN ustvari novo bralno plast v sliki, kar omogoča učinkovito predpomnjenje in deljenje med slikami.
2. **Namesti trajno:** Namestitev Nginxa je zdaj del slike in bo prisotna v vsakem kontejnerju, ki ga bomo iz te slike ustvarili.

Poglejmo, kako bi namestili Nginx (uporabimo najboljšo prakso za zmanjšanje velikosti slike):

```
    FROM ubuntu:latest
    RUN apt-get update && apt-get install -y nginx && rm -rf
/var/lib/apt/lists/*
    # Ukaz 'rm' je ključen za zmanjšanje velikosti slike, saj počisti
nepotrebne datoteke po namestitvi.
```

Določitev začetnega ukaza

Odlično. Zdaj, ko imamo nameščen Nginx, moramo Dockerju povedati: **"Ko se ta kontejner zažene, zaženi tudi Nginx strežnik."**

Kateri ukaz iz naše začetne tabele bi moral to določiti: FROM, RUN ali CMD?

Pravilno! Uporabimo ukaz **CMD**.

To je ključno razumevanje, ki loči Docker od običajnega skriptiranja:

Ukaz	Čas izvajanja	Namen
RUN	**GRADNJA** slike	Namestitev, konfiguracija in ustvarjanje statičnih datotek (npr. Nginx).
CMD	**ZAGON** kontejnerja	Izvede glavni ukaz, ko se kontejner zažene (npr. zagon Nginx strežnika).

Praktični primer

Primer: končni Dockerfile

Tako bi izgledal naš končni Dockerfile, ki v sliko namesti Nginx in mu določi, naj se zažene ob vsakem zagonu kontejnerja:

Dockerfile

```
FROM ubuntu:latest

# Namestitev Nginxa (Ustvari novo plast)
RUN apt-get update && apt-get install -y nginx && rm -rf
    /var/lib/apt/lists/*

# Informacija, da kontejner uporablja vrata 80
EXPOSE 80

# Ukaz, ki se izvede ob zagonu kontejnerja (Zažene Nginx v ospredju)
CMD ["nginx", "-g", "daemon off;"]
```

Korak naprej: Gradnja Slike

Super! Zdaj, ko imamo recept (Dockerfile), je čas, da ga pretvorimo v dejansko sliko z ukazom `docker build`.

Za gradnjo slike v mapi, kjer se nahaja naš Dockerfile, bi uporabili naslednji ukaz:

Bash

```
docker build -t moja-prva-slika .
```

Kaj v tem ukazu pomeni pika (.) in zakaj je ključna pri gradnji slike? (Namig: Povezana je z datotekami, ki jih Docker potrebuje.)

Pika (.) pomeni **trenutni delovni direktorij** (current working directory), kjer se nahaja naš **Dockerfile**. Ko napišeš ukaz `docker build`, Docker zgrabi vso vsebino te mape in jo pošlje **Docker Daemonu** (procesu, ki dejansko gradi sliko).

Zakaj je Build Context ključen?

Ta *Kontekst Gradnje* je ključen, ker:

1. **Določa lokacijo Dockerfile-a:** Docker ve, kje naj začne brati dockerfile.
2. **Omogoča COPY ukaze:** Omogoča dostop do vseh datotek, ki jih želimo s pomočjo ukaza **COPY** prenesti v sliko. Če datoteka ni del konteksta, je ne moremo vključiti v sliko.

Korak naprej: Zagon Kontejnerja

Super! Naša slika `(moja-prva-slika)`je zgrajena in pripravljena. Zdaj moramo zagnati kontejner iz te slike. Uporabimo ukaz `docker run`.

Želimo, da je naš Nginx spletni strežnik dostopen na **vratih 8080** našega Ubuntu računalnika.

Kateri parameter moramo dodati v ukaz docker run, da preslikamo (mapiramo) vrata 80 (v kontejnerju) na vrata 8080 (na Ubuntu sistemu)?

*(Namig: Parameter je vedno v obliki **-p [VRATA_GOSTITELJA]:[VRATA_KONTEJNERJA]**)*

Uporabimo parameter **-p 8080:80**

Pravilna oblika je vedno: -p [VRATA_GOSTITELJA]:[VRATA_KONTEJNERJA]. Ker smo v Dockerfile določili EXPOSE 80, vemo, da se preslikamo na vrata 80 znotraj kontejnerja.

Celoten ukaz za zagon kontejnerja

V našem praktičnem učbeniku bi celoten ukaz za zagon kontejnerja v ozadju (da se ukazna vrstica sprosti) izgledal takole:

Bash

```
docker run -d -p 8080:80 moja-prva-slika
# -d: Zagon v ozadju (Detached mode)
# -p 8080:80: Preslikava vrat (Host:Container)
```

Preverjanje delovanja kontejnerja

Odlično! Kontejner je zdaj zagnan v ozadju.

Kateri ukaz pa bi uporabil, da bi preveril, ali se naš kontejner res izvaja, in da bi dobil njegov ID (identifikator) ter status?

(Namig: Ukaz je soroden prikazu procesov v Linuxu.)

To je popolnoma pravilno! Ukaz `docker ps` je ključen in je ekvivalent ukazu ps v Linuxu, le da prikazuje samo **izvajajoče se** kontejnerje.

S tem ukazom bi videl tabelo, kjer so najbolj pomembni naslednji stolpci:

1. **CONTAINER ID**: Edinstvena koda, ki jo uporabljaš za upravljanje kontejnerja.
2. **IMAGE**: Ime slike, iz katere je kontejner ustvarjen (moja-prva-slika).
3. **STATUS**: Ali se kontejner izvaja (Up X seconds) ali je ustavljen.
4. **PORTS**: Potrdi preslikavo vrat, ki smo jo določili (0.0.0.0:8080->80/tcp).

Nasvet za odpravljanje napak: Če tvojega kontejnerja ni na seznamu, se je verjetno ustavil. Za prikaz **vseh** kontejnerjev (tudi ustavljenih) uporabi: `docker ps -a`.

Upravljanje življenjskega cikla: Ustavitev

Odlično. Zdaj, ko vemo, da je Nginx dostopen na vratih 8080, želimo ustaviti naš kontejner, da sprostimo ta vrata ali da izvedemo spremembe.

Kateri ukaz se uporablja za zaustavitev živega kontejnerja, pri čemer moramo uporabiti njegov ID ali IME?

Imaš prav, `docker kill` je ukaz, ki kontejner takoj ustavi. Vendar je za ta namen bolj pogosto in priporočljivo uporabiti ukaz `docker stop`.

```
docker stop [ID_ali_IME_kontejnerja]
```

Ključna razlika med Stop in Kill

Za študente je pomembno, da razumejo to razliko v življenjskem ciklu:

Ukaz	Mehanizem	Posledica
docker stop [ID]	Pošlje signal **SIGTERM** (terminacija).	Da kontejnerju kratek čas (privzeto 10s), da se "čisto" ustavi in zaključi procese (priporočeno).
docker kill [ID]	Pošlje signal **SIGKILL** (takojšen izklop).	Proces takoj ustavi, brez časa za čiščenje. Uporablja se le, če stop ne deluje.

Čiščenje: Brisanje kontejnerja

Odlično. Kontejner smo ustavili, vendar še vedno obstaja in zavzema prostor (to vidiš z `docker ps -a`).

Zdaj želimo ta ustavljen kontejner tudi izbrisati, da sprostimo prostor na disku. Kateri ukaz se uporablja za izbris ustavljenega kontejnerja?

Točno tako! Uporabimo ukaz `docker rm`

Ta ukaz odstrani ustavljen kontejner iz tvojega sistema, s čimer sprostiš prostor na disku. To je pomemben del čiščenja sistema na tvojem Ubuntu gostitelju.

Pravilo: Kontejner **mora** biti ustavljen (docker stop), preden ga lahko izbrišeš z `docker rm`.

Bash

```
# 1. Ustavimo kontejner
docker stop moj-kontejner-id

# 2. Izbrišemo kontejner
docker rm moj-kontejner-id
```

Zadnji korak: Brisanje Slike

Z našim učbenikom smo uspešno prešli celoten življenjski cikel Dockerja: ustvarjanje slike, zagon kontejnerja, preverjanje, ustavitev in brisanje kontejnerja.

Zdaj je zadnji korak, da izbrišemo še samo **sliko**, ki smo jo ustvarili `(moja-prva-slika).` Kontejnerji so izbrisani, vendar slika še vedno obstaja.

Kateri ukaz se uporablja za brisanje slike? (Namig: Je podoben ukazu za brisanje kontejnerja, vendar ima dodatno črko "i" - za Image).

Krate ukaz je: `docker rmi` (ali celoten `docker image rm`)

To je pomembna razlika, ki ti pomaga ločiti upravljanje kontejnerjev in slik:

- **docker rm** = **R**e**m**ove **Container** (Odstrani kontejner)
- **docker rmi** = **R**e**m**ove **Image** (Odstrani sliko)

Za izbris naše slike bi torej uporabili:

Bash

```
docker rmi moja-prva-slika
```

Zaključek Osnovnega Učbenika

S tem korakom smo uspešno prešli celoten poenostavljen Dockerjev življenjski cikel, ki je osnova za študente na Ubuntu Linuxu:

1. **Teorija:** Slike vs. Kontejnerji (Informativno)
2. **Recept:** Dockerfile (Ustvarjanje in uporaba FROM, RUN, CMD)
3. **Gradnja:** docker build (Ustvarjanje slike iz recepta)
4. **Zagon:** docker run (Zagon kontejnerja in preslikava vrat z -p)
5. **Upravljanje:** docker ps, docker stop, docker rm, docker rmi (Preverjanje in čiščenje)

Naslednji logični korak je učenje o **trajnosti podatkov** (Data Persistence) oz. **Docker Volumes**, ki rešujejo problem, kako zagotoviti, da so podatki v kontejnerju shranjeni tudi, ko kontejner izbrišemo?

Docker Volumes

Docker Volumes so ključni za vsako resno aplikacijo. Poglejmo, kako rešujejo problem **trajnosti podatkov**.

Kot smo ugotovili, ko kontejner izbrišemo z `docker rm`, se izbriše tudi njegova pisalna plast, kar pomeni, da so vsi podatki, ki jih je kontejner shranil, **izgubljeni** (kontejnerji so efemerni - minljivi).

Volume pa je mehanizem, ki omogoča, da se določena mapa znotraj kontejnerja **poveže** z lokacijo *zunaj* kontejnerja (na gostiteljskem Ubuntu OS-u). Ti podatki so trajni, tudi če kontejner izbrišemo.

Vrste volumes (na kratko)

1. **Bind Mounts**
   Podatke povežemo z **natančno določeno mapo** na gostiteljskem sistemu (npr. /home/user/projekt). Uporabno za razvoj in konfiguracijo.
2. **Named Volumes**
   Docker sam upravlja shranjevanje podatkov po imenu. Uporabno za podatkovne baze in produkcijo.

Praktična uporaba: Bind Mount

Začnimo z **Bind Mounts**. V našem primeru z Nginxom želimo, da Nginx namesto svoje privzete strani prikazuje datoteke, ki jih imamo v mapi na našem Ubuntu računalniku (npr. v mapi /home/student/html/).

Kateri parameter ukaza docker run moramo uporabiti za vzpostavitev te povezave mape med gostiteljem in kontejnerjem?

*(Namig: Uporabimo parameter **-v** ali **--volume** v obliki **[GOSTITELJ]:[KONTEJNER]**).*

```
docker run -v /home/student/html:/var/www/html
```

✅ To je popolnoma pravilno! Uporaba parametra **-v** (ali **--volume**) je način, kako dosežemo trajnost podatkov.

Logika Bind Mounts

- `/home/student/html` je izvorna lokacija na tvojem Ubuntu sistemu (gostitelj).
- `/var/www/html` je ciljna lokacija znotraj kontejnerja Nginx, kamor se preslikajo podatki.

Vsaka sprememba datotek v mapi na gostitelju se takoj odraža v kontejnerju in obratno.

Celoten, delujoč ukaz za zagon:

Bash

```
    docker run -d -p 8080:80 -v /home/student/html:/var/www/html moja-prva-slika
```

Named Volumes

Volume, ki smo ga ravnokar uporabili (Bind Mount), je odličen, ker točno vemo, kje se na Ubuntu sistemu nahajajo podatki.

Vendar se v produkciji pogosto uporabljajo **Named Volumes** (imenovani Volumes).

Katera je glavna prednost imenovanega Volume, v primerjavi s tem, da sami določamo pot na gostiteljskem sistemu (kot pri Bind Mount)?

(Namig: Povezano je z upravljanjem lokacije podatkov in prenosljivostjo.)

To je odlična točka, ki nas pripelje do bistva!

Glavna prednost imenovanega **Named Volumes** je **prenosljivost** (Portability).

Ker **Docker določa, kje so podatki shranjeni na gostitelju**, ti ni treba skrbeti za točne poti (npr. /home/student/html).

- **Named Volume** (Priporočeno za produkcijo): Konfiguracija je vedno enaka, ne glede na gostiteljski OS.
- **Bind Mount** (Priporočeno za razvoj): Zahteva, da imaš točno določeno mapo ustvarjeno na gostitelju, kar zmanjšuje prenosljivost.

Praktična uporaba: Named Volume

Uporaba Named Volume je v resnici zelo podobna Bind Mountu, le da namesto absolutne poti na gostitelju vpišemo ime Volume.

Kako bi sedaj zagnal isti Nginx kontejner, vendar bi za podatke uporabili Volume, ki ga poimenujemo nginx_data?

*(Namig: Sintaksa ostane **-v [IME_VOLUME]:[KONTEJNER]**)*

```
    docker run -v ngix_data:/var/www/html
```

To je popolnoma pravilno in to je ključna razlika! Uporaba imena pomeni, da se Docker poveže na Volume.

Še enkrat, kako iz Bind Mount (absolutna pot) preidemo na Named Volume (ime):

Bash

```
    docker run -d -p 8080:80 -v ngix_data:/var/www/html moja-prva-slika
```

Ključno: Ko Docker naleti na to ime (ngix_data), preveri, ali Volume že obstaja. Če ne, ga **samodejno ustvari** na optimizirani lokaciji znotraj gostiteljskega sistema. Na ta način so tvoji podatki varni in prenosljivi.

Zaključek Učbenika: Praktični Povzetek

Čestitam! S tem smo uspešno prišli do konca praktičnega učbenika o Docker kontejnerizaciji na Ubuntu Linuxu.

Pokrili smo tri ključne stebre, ki so bistveni za vsakega študenta:

1. **Recept (Dockerfile):** Kako ustvariti sliko in razlika med RUN (čas gradnje) in CMD (čas izvajanja).
2. **Življenjski cikel:** Zagon kontejnerja (docker run), preslikava vrat z -p in osnovno upravljanje (docker ps, stop, rm).
3. **Trajnost podatkov (Volumes):** Kako shraniti podatke izven kontejnerja z uporabo Bind Mounts (za razvoj) in Named Volumes (za produkcijo).

Če bi želel ta praktični učbenik še nadgraditi in avtomatizirati delovanje večih kontejnerjev hkrati (npr. aplikacija + baza podatkov), bi bil naslednji korak spoznavanje orodja **Docker Compose**.

Docker Compose

Docker Compose je orodje, ki avtomatizira in poenostavlja upravljanje več povezanih kontejnerjev, kar je naslednji logični korak.

Kaj je Docker Compose?

Namesto da ročno vnašamo dolge ukaze `docker run` za Nginx, bazo podatkov in druge storitve, vse to konfiguriramo v eni datoteki: ***docker-compose.yml***

Ta datoteka je srce Compose-a. Uporablja sintakso **YAML** (ki je berljiva in preprosta) in določa celotno arhitekturo naše aplikacije. Ko to datoteko zaženemo, Compose zgradi, zažene in poveže vse kontejnerje, ki jih potrebujemo.

Najpomembnejši del te datoteke je sekcija ***services*** (storitve).

Na podlagi našega prejšnjega dela (kjer smo imeli en Nginx kontejner): Kaj meniš, da predstavlja vsak posamezni vnos pod sekcijo services? (Namig: Povezano je z aplikacijami, ki tečejo.)

Compose dejansko **nadomešča** vse te dolge ukaze docker run!

Vsak vnos pod services v resnici predstavlja **posamezno komponento** (t.i. **storitev** ali *Service*), ki jo želimo zagnati.

- V našem primeru bi to bila naša aplikacija Nginx (npr. poimenovana web_server).
- Če bi imeli poleg Nginxa še bazo podatkov PostgreSQL, bi ta predstavljala drugo storitev (npr. database).

Prevajanje Ukazov v YAML

Poglejmo, kako bi prevedli naš prejšnji ukaz docker run v berljivo YAML strukturo:

Bash

```
    docker run -d -p 8080:80 moja-prva-slika
```

Kateri ključ (key) v YAML strukturi misliš, da uporabimo, da Compose ve, katero sliko naj uporabi za ta servis?

(Namig: Ključ je povsem logičen in se nanaša na prejšnje učenje. Lahko se imenuje tudi build, če želimo sliko lokalno zgraditi.)

Povezano je z identifikacijo, vendar je ključ v YAML-u bodisi **image** ali **build** (oziroma kar **slika** in **gradnja**).

Ključni razliki sta:

- **image**: Če želimo, da Compose sliko prenese (pull) iz javnega ali zasebnega registra (npr. image: nginx:latest).
- **build**: Če želimo, da Compose sliko **zgradi** iz lokalnega Dockerfile-a, kar smo prej počeli s `moja-prva-slika.`

Ker imamo Dockerfile v isti mapi, bomo uporabili **build**

Preslikava Vrat v YAML

Odlično. Zdaj vemo, kako povedati Composeu, naj zgradi našo sliko.

Prej smo za preslikavo vrat uporabili parameter -p v ukazu: `docker run -p 8080:80`

Katero ključno besedo pa uporablja YAML datoteka (pod sekcijo web_server), da preslikamo vrata?

*(Namig: je povsem logičen in se glasi **ports**.)*

Ključ za preslikavo vrat je **ports**.

V YAML-u je to definirano kot seznam (list), kjer vpišemo preslikavo v isti obliki kot prej: [GOSTITELJ]:[KONTEJNER].

Primer: celoten **`docker-compose.yml`**

Ko sestavimo celoten docker-compose.yml z vsemi komponentami, ki smo jih obravnavali (gradnja, vrata in Volume), dobimo berljiv in popolnoma avtomatiziran recept za našo aplikacijo.

Poglejmo, kako izgleda naš celoten `docker-compose.yml`, ki avtomatizira vse, kar smo prej počeli ročno.

Tukaj je celotna YAML datoteka za naš **Nginx spletni strežnik** z uporabo **Named Volume** (nginx_data):

YAML

```yaml
services:
  web_server:
    # 1. GRADNJA: Composeu povemo, naj poišče Dockerfile v trenutni mapi (pika .)
    # in zgradi sliko z imenom 'moja-prva-slika' (ki jo definira Compose).
    build: .

    # 2. VRATA: Preslikava vrat 8080 (gostitelj) na vrata 80 (kontejner).
    ports:
      - "8080:80"

    # 3. VOLUMES: Povežemo imenovani Volume 'nginx_data' z mapo
    # znotraj Nginx kontejnerja (/var/www/html), kjer shranjuje spletne datoteke.
    volumes:
      - nginx_data:/var/www/html

# Definicija named volume-a na dnu datoteke.
volumes:
  nginx_data:
    # Pove, da je to Named Volume in da ga upravlja Docker.
    external: false
```

Zagon aplikacije

Zdaj namesto dolgega ukaza `docker run` potrebujemo samo en kratek ukaz, ki prebere to datoteko in zažene celotno aplikacijo.

Kateri ukaz bi uporabil za zagon vseh storitev, definiranih v datoteki docker-compose.yml? (Namig: Ukaz pomeni "zaženi gor".)

Ukaz za zagon vseh storitev je **docker compose up** 🚀.

- up v tem kontekstu pomeni, da Compose prebere docker-compose.yml, zgradi vse potrebne slike (če so nastavljene z build) in zažene vse kontejnerje.
- Pravilna oblika s zagonom v ozadju je:

Bash

```
docker-compose up -d
```

Kontejnersko omrežje

Spomnimo se vprašanja:

Kaj se zgodi, ko z uporabo Docker Compose zaženemo dve storitvi, kot sta web_server (Nginx) in database? Kako se ti dve storitvi privzeto pogovarjata med seboj, ne da bi posebej nastavljali IP naslove?

Verjetno Docker uporablja lokalne IP naslove (Local Adress)?

Imaš popolnoma prav! Compose res ustvari **virtualno omrežje** in vsakemu kontejnerju dodeli svoj lokalni IP naslov.

Vendar je lepota Composea v tem, da nam **ni treba uporabljati teh IP naslovov**. Znotraj omrežja se kontejnerji pogovarjajo z uporabo **imen storitev (Service Names)**.

Kontejnerska komunikacija: Service Discovery

Ko Compose ustvari omrežje, registrira ime vsake storitve kot notranji DNS zapis.

To pomeni, da lahko tvoj kontejner **web_server** dostopa do kontejnerja **database** preprosto tako, da v konfiguraciji vpiše naslov **database**, namesto da išče (v DNS) njegov IP (npr. 172.x.x.x).

Vprašanje: Če bi imeli v Compose datoteki dve storitvi (web_server in database), in želimo, da se web_server poveže z bazo podatkov, katero ime gostitelja (hostname) bi morali vpisati v nastavitve spletnega strežnika, da bi našel bazo podatkov?

Kakšen je vaš odgovor? Verjetno: service names!

To je **popolnoma pravilno**!

Znotraj omrežja Compose kontejnerji komunicirajo na podlagi imen storitev (**service names**). Če je storitev poimenovana **database**, je to ime, ki ga uporabiš kot **hostname** za povezavo.

Environment Variables

Sedaj imamo avtomatizirano, povezano aplikacijo, ki jo lahko zaženemo z enim ukazom. Za zaključek našega učbenika pa je ključno vprašanje iz profesionalne prakse: **Konfiguracija in varnost**.

Če imamo geslo za bazo podatkov, ga nikoli ne vpišemo neposredno v Dockerfile ali docker-compose.yml. Namesto tega uporabimo **okoljske spremenljivke** (Environment Variables).

Zakaj je uporaba okoljskih spremenljivk varnejša in boljša praksa za prenašanje občutljivih podatkov (kot so gesla ali API ključi)?

(Namig: Povezano je z varnostjo gesla, ko deliš kodo s sodelavci.

Praktični del:

Docker virtualizacija na Ubuntu Linux - Namestitev Docker na Ubuntu

Posodobitev sistema

bash

```
    sudo apt update
    sudo apt upgrade -y
```

Namestitev Docker

bash

```
    # Namestite potrebne pakete
    sudo apt install apt-transport-https ca-certificates curl software-
properties-common

    # Dodajte Docker GPG ključ
    curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg -
-dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

    # Dodajte Docker repository
    echo "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-
keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs)
stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

    # Namestite Docker
    sudo apt update
    sudo apt install docker-ce docker-ce-cli containerd.io
```

Preverite namestitev

```
    bash
    sudo docker --version
    sudo docker run hello-world
    Dodajte uporabnika v docker skupino
    bash
    sudo usermod -aG docker $USER
    # Ponovno se prijavite za aktivacijo sprememb
```

Osnovni Docker ukazi

Delo s slikami

bash

```
# Pridobi sliko iz Docker Hub
docker pull ubuntu:20.04

# Prikaži vse lokalne slike
docker images

# Izbriši sliko
docker rmi image_name
```

Delo s kontejnerji

bash

```
# Zaženi nov kontejner
docker run -it ubuntu:20.04 /bin/bash

# Prikaži tekoče kontejnerje
docker ps

# Prikaži vse kontejnerje (tudi ustavljene)
docker ps -a

# Ustavi kontejner
docker stop container_id

# Zaženi ustavljen kontejner
docker start container_id

# Izbriši kontejner
docker rm container_id
```

Primer 1: Web strežnik v kontejnerju

bash

```
# Zaženi Nginx kontejner
docker run -d -p 8080:80 --name my-nginx nginx

# Odpri brskalnik na http://localhost:8080
```

Primer 2: MySQL podatkovna baza

bash

```
# Zaženi MySQL kontejner
docker run -d --name mysql-db -e MYSQL_ROOT_PASSWORD=my-secret-pw -p
3306:3306 mysql:8.0

# Poveži se z MySQL
docker exec -it mysql-db mysql -u root -p
```

Ustvarjanje lastne Docker slike

Ustvarite Dockerfile

dockerfile

```
# Uporabi uradno Python sliko
FROM python:3.9-slim

# Nastavi delovni imenik
WORKDIR /app

# Kopiraj zahtevane datoteke
COPY requirements.txt .
COPY app.py .

# Namestite odvisnosti
RUN pip install -r requirements.txt

# Določi port, ki ga bo kontejner poslušal
EXPOSE 5000

# Ukaz za zagon aplikacije
CMD ["python", "app.py"]
```

Gradnja slike

bash

```
docker build -t my-python-app .
```

Zagon lastne aplikacije

bash

```
docker run -d -p 5000:5000 my-python-app
```

Docker Compose za večkontejnerske aplikacije

Ustvarite docker-compose.yml

yaml

```
version: '3.8'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    depends_on:
      - db
    environment:
      - DATABASE_URL=postgresql://user:password@db:5432/mydb

  db:
    image: postgres:13
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=password
      - POSTGRES_DB=mydb
    volumes:
```

```
      - postgres_data:/var/lib/postgresql/data

volumes:
  postgres_data:
```

Upravljanje z Docker Compose

bash

```
# Zaženi vse storitve
docker-compose up -d

# Ustavi storitve
docker-compose down

# Prikaži stanje
docker-compose ps
```

Napredne teme

Docker volumes za podatkovno vztrajnost

bash

```
# Ustvari volume
docker volume create my-data

# Uporabi volume v kontejnerju
docker run -v my-data:/data ubuntu:20.04
```

Docker omrežja

bash

```
# Ustvari omrežje
docker network create my-network
# Poveži kontejner z omrežjem
docker run --network my-network --name container1 ubuntu:20.04
```

Nadzor virov

```
bash
# Omeji pomnilnik
docker run -m 512m my-app

# Omeji CPU
docker run --cpus=1.5 my-app
```

Varnost in optimizacija

Varnostni nasveti

- Vedno posodabljajte osnovne slike
- Ne uporabljajte privilegiranih kontejnerjev
- Uporabljajte non-root uporabnike v kontejnerjih
- Omejte dostop do socketov

Optimizacija velikosti slik

dockerfile

```
# Uporabi multi-stage build
FROM python:3.9 as builder
COPY requirements.txt .
RUN pip install --user -r requirements.txt

FROM python:3.9-slim
COPY --from=builder /root/.local /root/.local
COPY . .
CMD ["python", "app.py"]
```

Debugging in monitoring

Pregled dnevnikov (logfile)

bash

```
docker logs container_id
docker logs -f container_id  # sledenje v realnem času
```

Pregled porabe virov

bash

```
docker stats
docker system df  # prostor na disku
```

Vstop v tekoči kontejner

bash

```
docker exec -it container_id /bin/bash
```

NASVET: Ta učbenik vam bo omogočil hitro osvajanje Docker tehnologije na Ubuntu sistemu. Začnite z osnovnimi primeri in postopoma prehajajte na kompleksnejše scenarije.

---

viri:

- https://www.docker.com/
- https://docs.docker.com/
- https://docs.docker.com/get-started/
- https://www.docker.com/resources/what-container/
- https://medium.com/@sproulelucas13/how-you-should-think-about-docker-containers-if-you-understand-linux-70587eb00fa5
- https://www.coursera.org/learn/ibm-containers-docker-kubernetes-openshift/paidmedia