

# Docker alapok

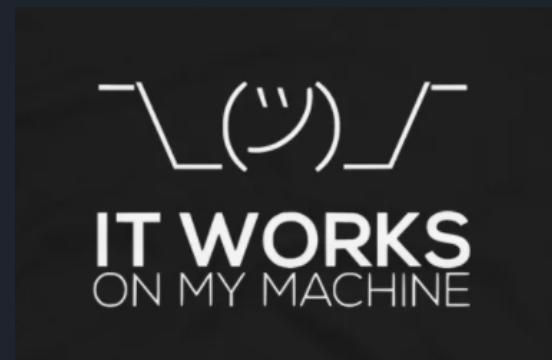


MOSZE előadás

Csapó Ádám Balázs -  
[csapo.adam@sze.hu](mailto:csapo.adam@sze.hu)

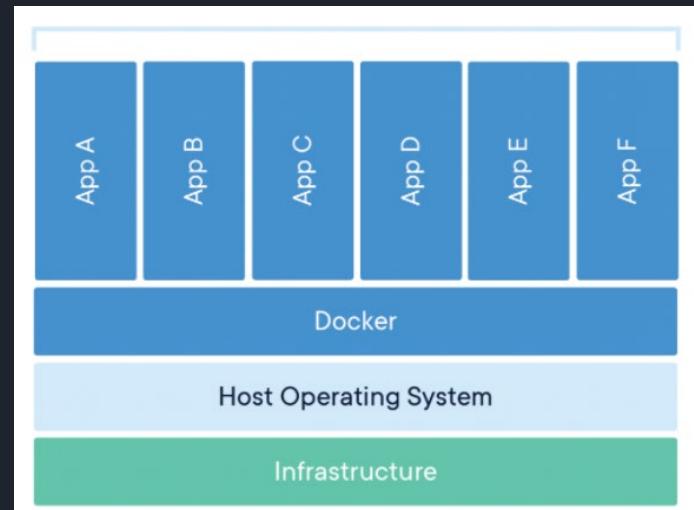
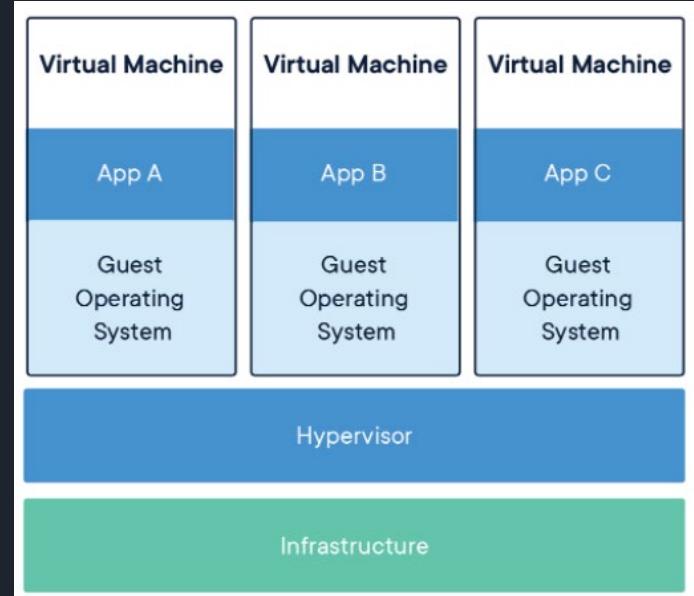
# Milyen problémára született a Docker?

- Mindennapos élmény, hogy készítünk egy sw-t ami a teszt és deploy környezetben másképpen működik
  - A probléma, hogy semmilyen sw nem önmagában fut, vannak dependency-k (könyvtárak, futtathatók, static erőforrások, ...)
  - Gyakran hallani, bár nem túl felelősségteljes gondolkodásra vall, hogy:



# Mi a Docker? - <https://docker.com>

- "Egy konténerizációs technológia"
- Gyakran hasonlítják virtuális gépekhez (VM), de a hasonlóságok felszínesek:
  - VM-nél van a host OS fölött egy hypervisor, ami futtatja a guest OS-t, a fölött vannak a binárisok, lib-ek majd az alkalmazások. A guest OS maga 700-800 Mb méretű is lehet akár, és minden izolált alkalmazáshoz egy ilyet el kell indítani
  - A Docker esetében a hypervisor szerepét a Docker daemon veszi át, viszont e fölött nincs guest OS, hanem minden alkalmazásnak saját Docker konténer van, amelyben minden alkalmazásnak saját binárisai vannak.
  - A Docker direktben kommunikál a host OS-szel (nincs guest OS) és a konténerek a host OS olyan processzei, melyek egymástól, valamint az OS egyéb processzeitől izoláltak!



# Mi a Docker? - <https://docker.com>

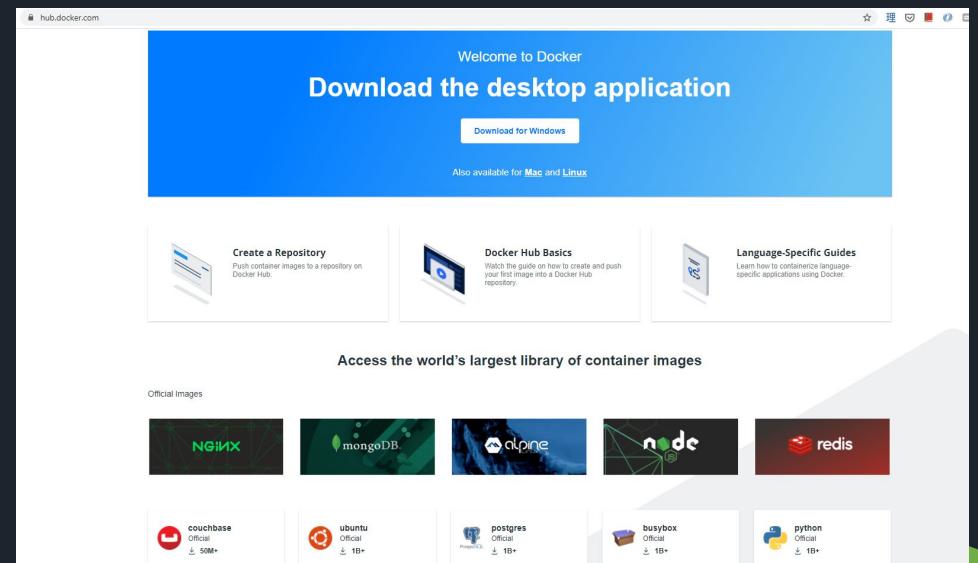
- Fontos tudni, hogy a Docker egy sokszintű szolgáltatás, az ingyenestől a havidíjas szolgáltatásokig több szintet magában foglal
  - Az alap-technológia egy nyílt forráskódú industry standardre, a containerd-re (container daemon) épül
  - Ingyenesen is telepíthetjük a dockert és készíthetünk vele saját image-t (amiből container példányosítható), bizonyos feltételek teljesülése esetén - ld. lent
  - A Docker ún. image repository-val is rendelkezik. Az ingyenes csomagban csak publikus image-eket lehet létrehozni, és limitálva van, hogy naponta hány image-t szedhetünk le. Fizetős esetben privát image-ek is létrehozhatóak.
  - A Docker fontossága napjainkban egyre csak nő, hiszen számos cloud szolgáltató is támogatja (AWS, Azure, Google Cloud, ...)

\*Docker Desktop is free to use, as part of the Docker Personal subscription, for individuals, non-commercial open source developers, students and educators, and small businesses of less than 250 employees AND less than \$10 million in revenue. Commercial use of Docker Desktop at a company of more than 250 employees OR more than \$10 million in annual revenue requires a paid subscription (Pro, Team, or Business) to use Docker Desktop. While the effective date of these terms is August 31, 2021, there is a grace period until January 31, 2022 for those that require a paid subscription to use Docker Desktop.

# Hogyan induljunk el?

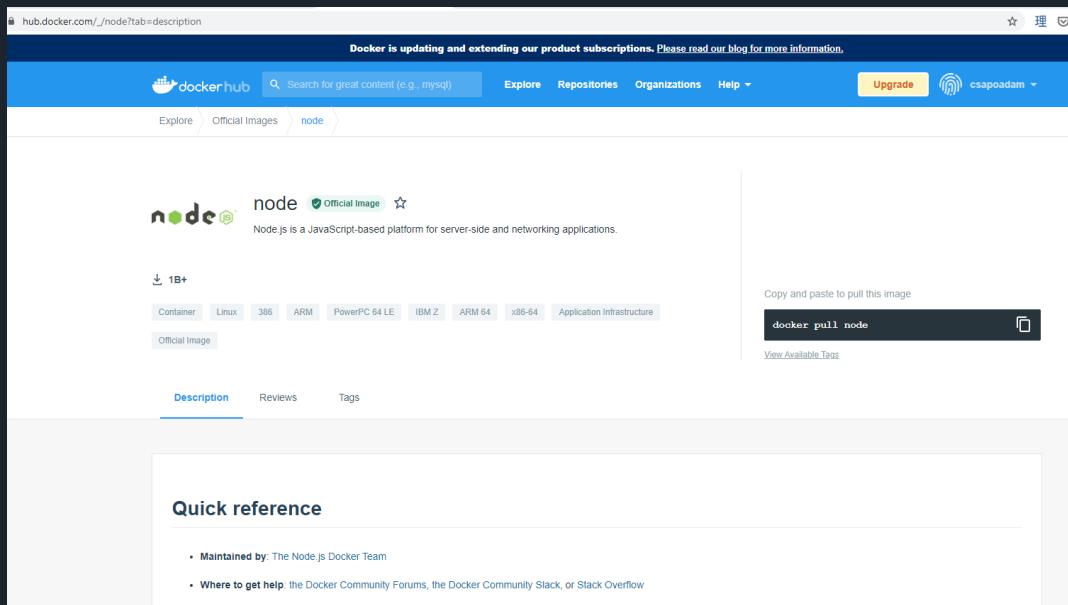
- A www.docker.com oldalon a personal subscription-nél nyomjuk meg a “get started” gombot, hozzunk létre saját id-t, jelszót és erősítsük meg az email címünket.
- Ezt követően töltök le a desktop alkalmazást (létezik alkalmazás Windows, Mac és Linux OS-ekre is). Telepítés után újra kellhet indítani a gépet, plusz Windowson külön telepíteni a WSL 2-t (és defaultnak beállítani a WSL 2-t).

- Ehhez a telepítő fel fog dobni egy linket nekünk, miután újraindítottuk a gépet
- Ez a Windows Subsystem for Linux azért kell, hogy Linux-alapú image-eket is futtathassunk containerként a Windows-on virtuális gép nélkül.



# Hogyan induljunk el?

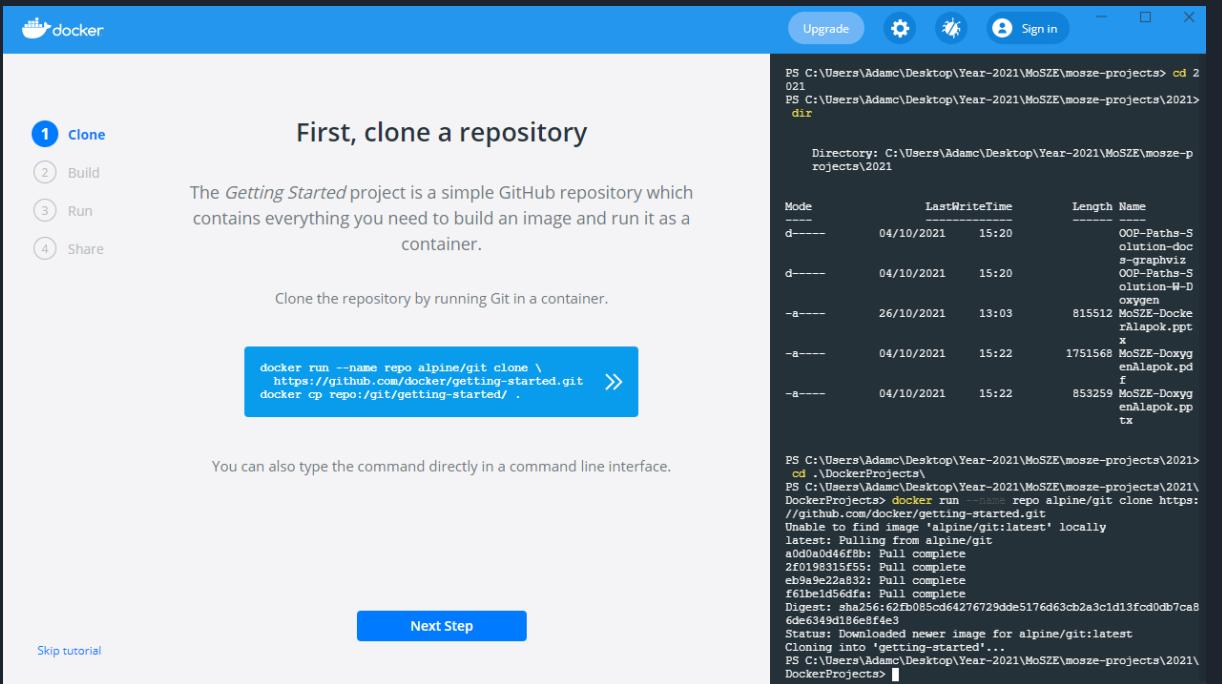
- Látható, hogy számos container image alapból is rendelkezésünkre áll. Ezeket ha kiválasztjuk, olvashatunk róluk leírást és hogy hogyan kell őket telepíteni:
  - Egy lehetőség, hogy pulloljuk az image-t. Egy másik lehetőség, hogy a docker run paranccsal el is indítjuk egy containerben (ld. később)



- Fontos, hogy még a containerek általában egy OS-re hivatkoznak, valójában nincs guest OS, hanem a host OS szimulál egy ilyen környezetet
  - Windows-on pl. a WSL-nek köszönhetően ez megtehető a Linux disztribúciókkal (a kernel ugye minden Linux disztribúcióban ugyanaz, csak a "userland" alkalmazások mások)
  - Linux gépen viszont Windows-os image-eket ma sem lehet futtatni - ebből is látszik, hogy mik a korlátok.

# Hogyan induljunk el?

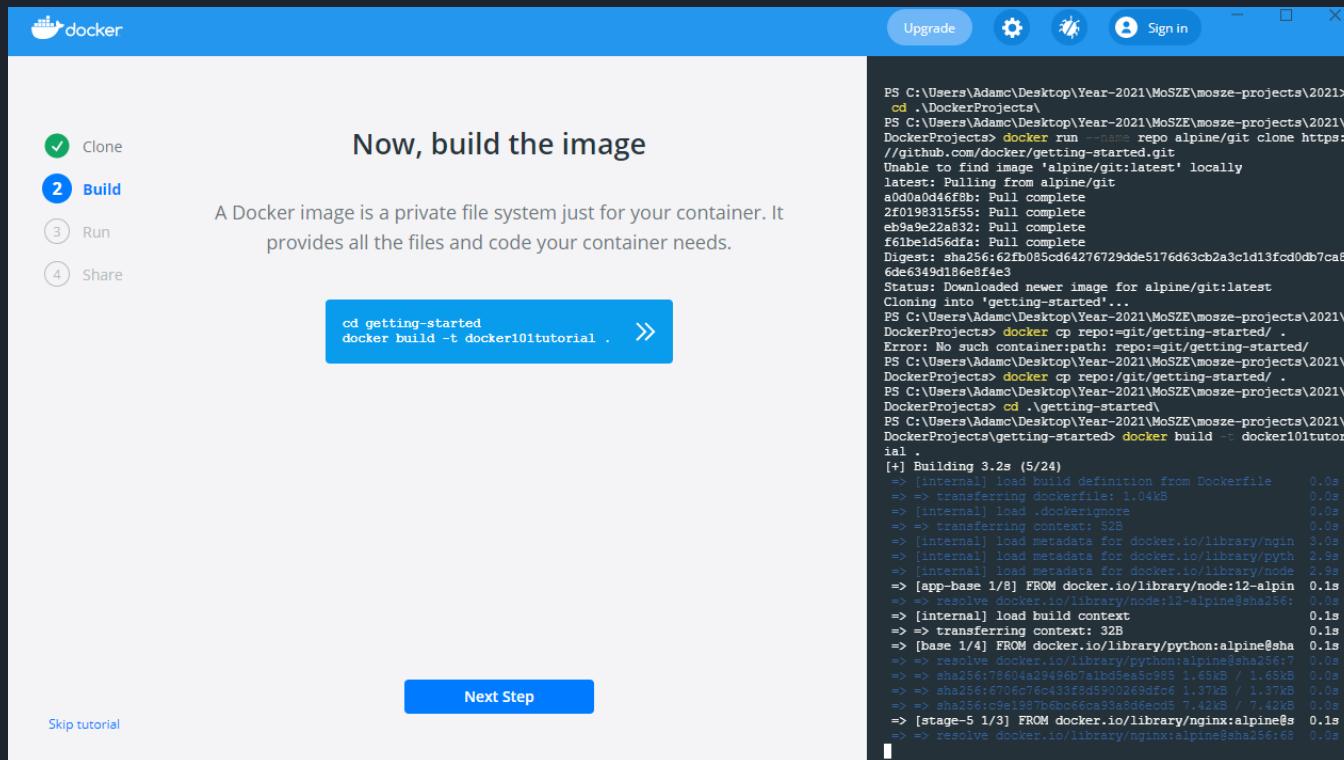
- De hogy könnyebb legyen, mikor először elindítjuk a Desktop alkalmazást, megjelenik benne egy tutorial is. Ehhez a docker run parancssal először elindítunk egy container-t, amiben létezik a git parancs és onnan lépünk tovább. Látható, hogy a docker cp és hasonló parancsok lényegében az OS-hez interface-ek:



- Fontos tudni, hogy a docker valójában csak egy command line parancs, amihez ez a GUI is interfészsel.
  - A docker pull, docker build, docker run stb. parancsokat bármikor elérhetjük sima konzolról is.

# Hogyan induljunk el?

- Tutorial folytatása. A build parancs egy Dockerfile alapján legenerálja magát az image-t. A következő lépésekben már ezt az image-t fogjuk elindítani egy containerként.

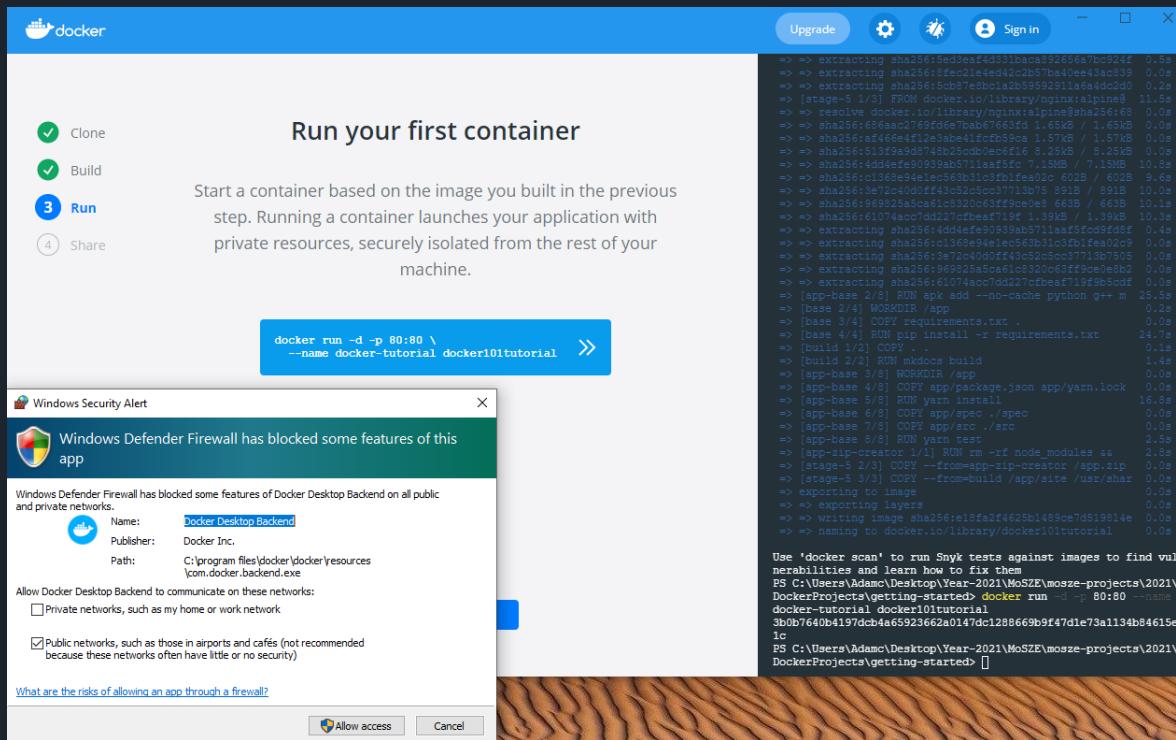


- Az image felfogható úgy mint egy fájlrendszer ami tartalmazza a konténer futásához szükséges libeket, futtathatókat és statikus erőforrásokat.

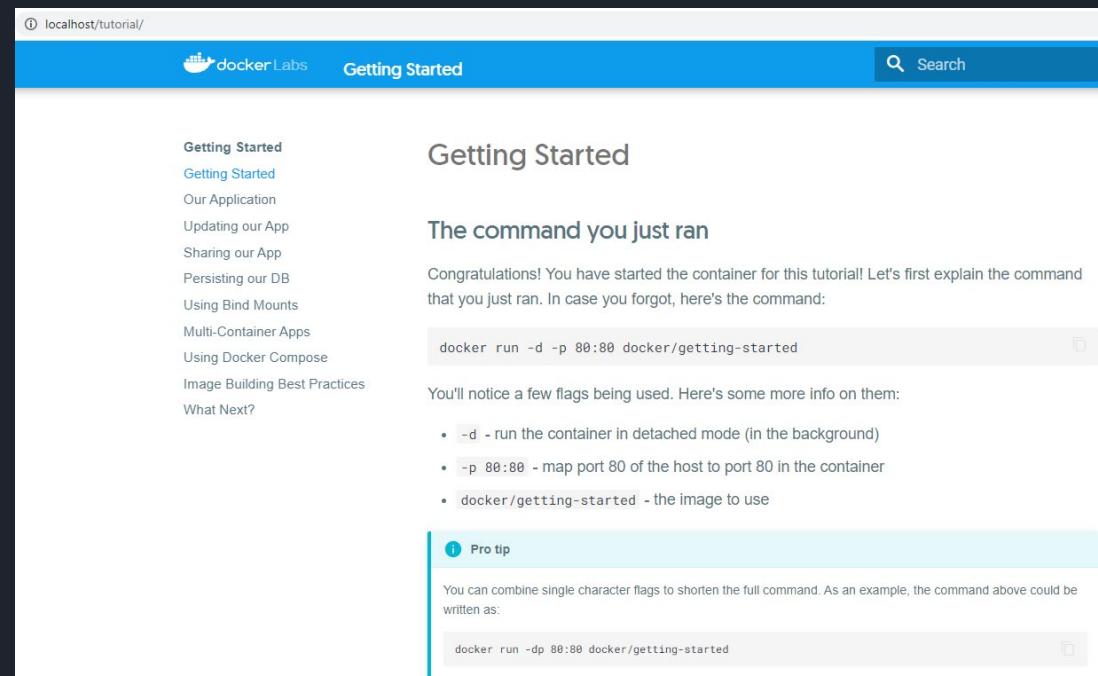
```
PS C:\Users\Adamc\Desktop\Year-2021\MoSZE\mosze-projects\2021>
cd .\DockerProjects\
PS C:\Users\Adamc\Desktop\Year-2021\MoSZE\mosze-projects\2021\DockerProject> docker run --name repo alpine/git clone https://github.com/docker/getting-started.git
Unable to find image 'alpine/git:latest' locally
latest: Pulling from alpine/git
Digest: sha256:62fb805cd64276729dde5176d63cb2a3c1d13fc0db7ca8
6deec349d186e8f4e3
Status: Downloaded newer image for alpine/git:latest
Cloning into 'getting-started'...
PS C:\Users\Adamc\Desktop\Year-2021\MoSZE\mosze-projects\2021\DockerProject> docker cp repo:/git/getting-started/
Error: No such container:path: repo:/git/getting-started/
PS C:\Users\Adamc\Desktop\Year-2021\MoSZE\mosze-projects\2021\DockerProject> docker cp repo:/git/getting-started/
PS C:\Users\Adamc\Desktop\Year-2021\MoSZE\mosze-projects\2021\DockerProject> cd .\getting-started\
PS C:\Users\Adamc\Desktop\Year-2021\MoSZE\mosze-projects\2021\DockerProject> getting-started
PS C:\Users\Adamc\Desktop\Year-2021\MoSZE\mosze-projects\2021\DockerProject> docker build -t docker101tutorial .
[+] Building 3.2s (5/24)
=> [internal] load build definition from Dockerfile      0.0s
=> => transferring dockerfile: 1.04kB                 0.0s
=> [internal] load .dockerignore                         0.0s
=> => transferring context: 52B                         0.0s
=> [internal] load metadata for docker.io/library/nginx 3.0s
=> [internal] load metadata for docker.io/library/python 2.9s
=> [internal] load metadata for docker.io/library/node 2.9s
=> [app-base 1/8] FROM docker.io/library/node:12-alpine 0.1s
=> => resolve docker.io/library/node:12-alpine@sha256: 0.0s
=> [internal] load build context                      0.1s
=> => transferring context: 32B                         0.1s
=> [base 1/4] FROM docker.io/library/python:alpine@sha 0.1s
=> => resolve docker.io/library/python:alpine@sha256:7 0.0s
=> => sha256:78604a29496b7a1bd5ea593a8d6eefc5 1.65kB / 1.65kB
=> => sha256:c9e1987b6bc66ca93a8d6eefc5 7.42kB / 7.42kB
=> => sha256:c9e1987b6bc66ca93a8d6eefc5 7.42kB / 7.42kB
=> [stage-5 1/3] FROM docker.io/library/nginx:alpine@sha 0.1s
=> => resolve docker.io/library/nginx:alpine@sha256:68 0.0s
```

# Hogyan induljunk el?

- Buildelt image futtatása egy containerben:



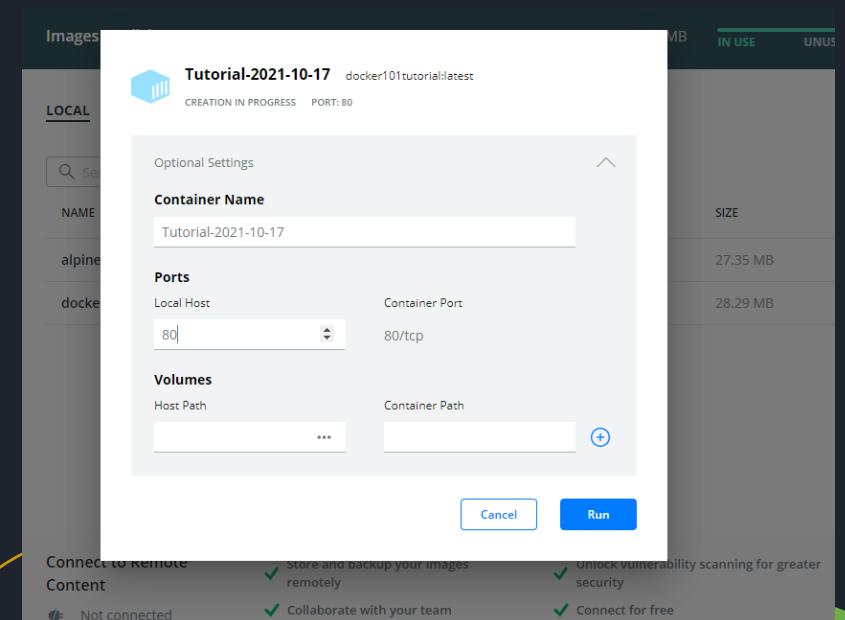
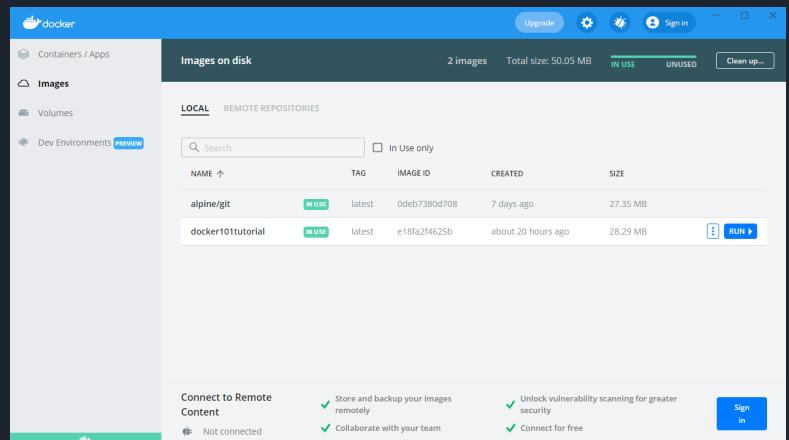
- A localhost betöltésével láthatjuk az alkalmazást futni, és tovább információhoz jutunk:



# Hogyan induljunk el?

- A Docker Dashboard-ot bármikor elérhetjük a desktop appen belülről, és itt megtekinthetjük, hogy milyen containereink futnak éppen / milyen image-ek állnak rendelkezésre.

- Adott image-t bármikor elindíthatjuk konténerként:



# Dockerfile-ok

- Ahogy említettük, az image készítésének lépései egy Dockerfile-ban kerülnek leírásra.
  - Amit a példában leszettünk, az egy sima git repo, ami tartalmaz egy ilyen Dockerfile-t
- Érdemes egy pillantást vetni rá, hogy mi szerepel ebben a Dockerfile-ban!
  - Láthatjuk, hogy egy alap image-t használunk, a python:alpine-t, ami egy Alpine Linux környezetben működő python installation (persze mivel guest OS nincs ezért a Windows-on is a WSL-en keresztül fut az image alapján létrehozott container)
  - Ebbe az alap image-be telepítünk további csomagokat a pip-pel, de felhasználható több image, amikor elkészítjük a saját image-ünket, és bármilyen rendelkezésre álló csomagkezelőt használhatunk a pip mellett is (pl. az apk meg a yarn is ilyenek)

```
1 # Install the base requirements for the app.
2 # This stage is to support development.
3 FROM python:alpine AS base
4 WORKDIR /app
5 COPY requirements.txt .
6 RUN pip install -r requirements.txt
7
8 # Run tests to validate app
9 FROM node:12-alpine AS app-base
10 RUN apk add --no-cache python g++ make
11 WORKDIR /app
12 COPY app/package.json app/yarn.lock ./
13 RUN yarn install
14 COPY app/spec ./spec
15 COPY app/src ./src
16 RUN yarn test
17
18 # Clear out the node_modules and create the zip
19 FROM app-base AS app-zip-creator
20 RUN rm -rf node_modules && \
21     apk add zip && \
22     zip -r /app.zip /app
23
24 # Dev-ready container - actual files will be mounted in
25 FROM base AS dev
26 CMD ["mkdocs", "serve", "-a", "0.0.0.0:8000"]
27
28 # Do the actual build of the mkdocs site
29 FROM base AS build
30 COPY .
31 RUN mkdocs build
32
33 # Extract the static content from the build
34 # and use a nginx image to serve the content
35 FROM nginx:alpine
```

# Példa

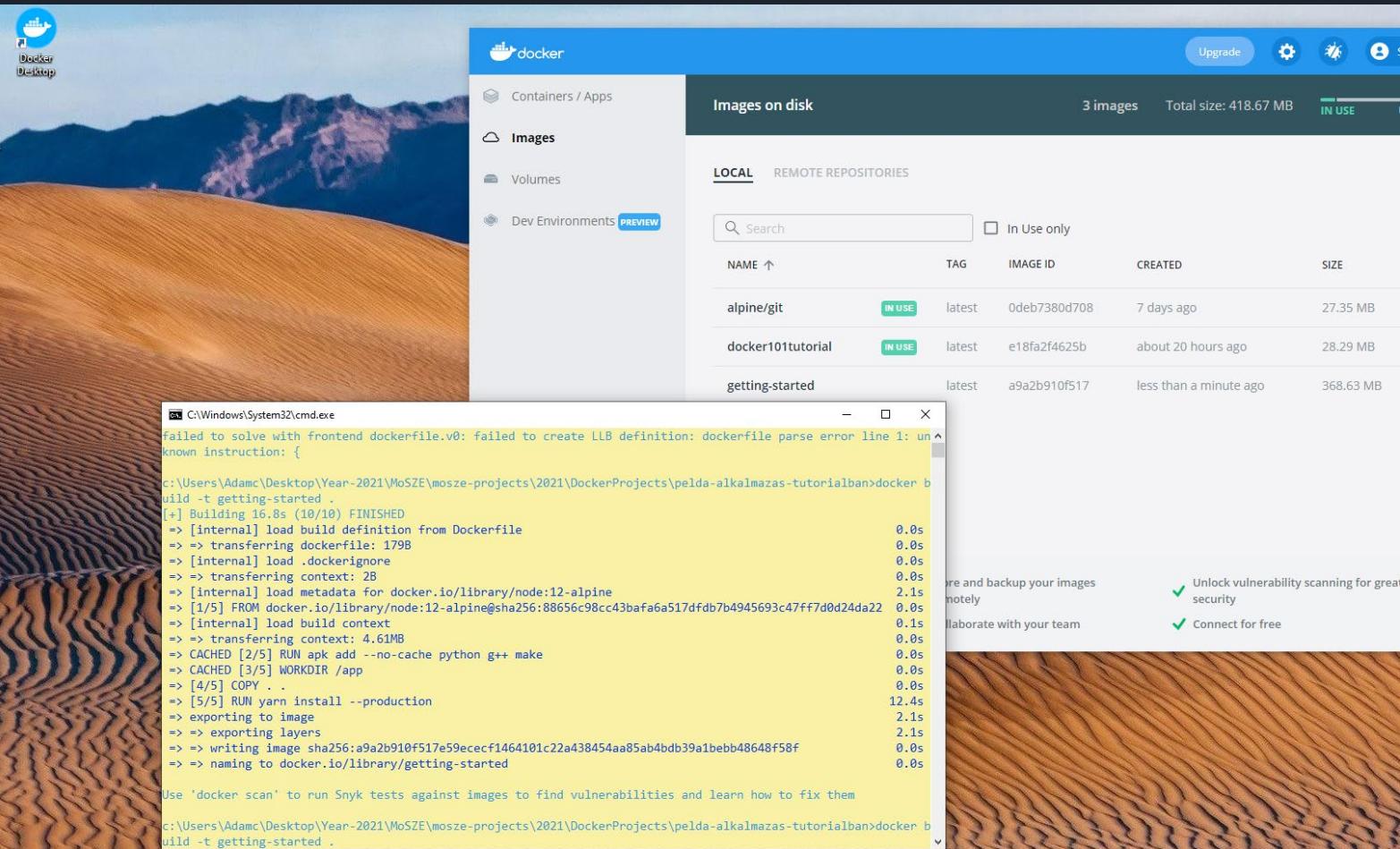
- Ha a tutorialon végig megyünk, egy példa alkalmazást is át tudunk nézni, ami egy példa Docker file készítését is tartalmazza (<http://localhost/tutorial/our-application/>)
- A tutorial először leszedet velünk egy forrás zip-et, ezt ki kell csomagolnunk és benne létre kell hoznunk egy - eleinte részben hibás - Dockerfile-t. (fontos, hogy ennek a file-nak ne legyen kiterjesztése)
- Ezt követően a mappában a

```
docker build -t getting-started .
```

parancssal lebuildeljük az image-t. A következő ábrán látható, hogy az image rögtön megjelenik a desktop app-en belül is:

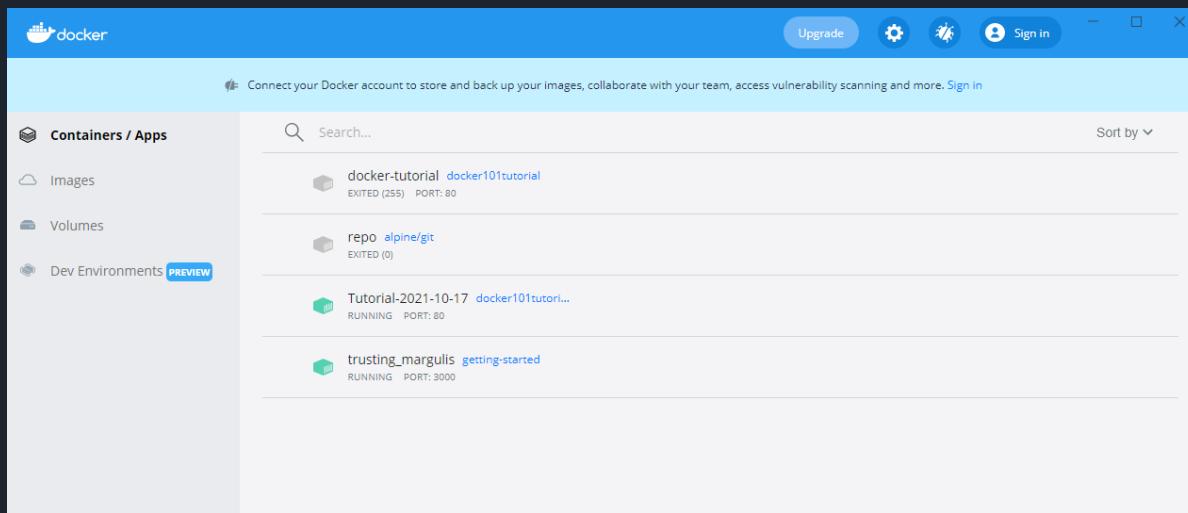
parancssal lebuildeljük az image-t. A következő ábrán látható, hogy az image rögtön megjelenik a desktop app-en belül is:

# Példa



# Példa

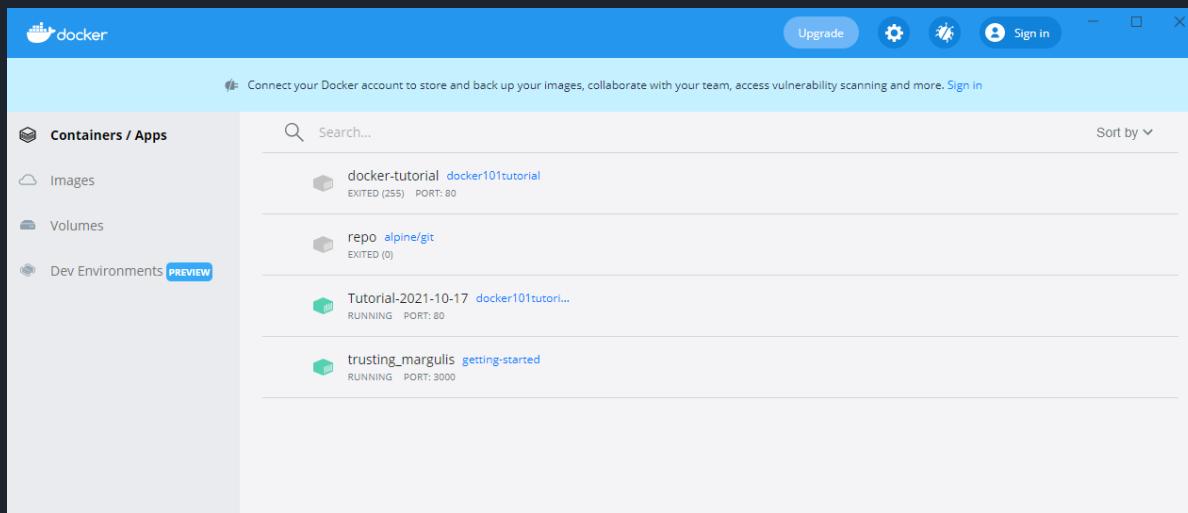
- Ezt követően futtassuk az image-t, akár a konzolról (docker run -dp 3000:3000 getting-started ) parancs, akár a gui-ból
- Látható, hogy most már 2 konténerünk is fut:



- Ha a kódunkat bármikor frissítjük, nem lehet csak úgy újra elindítani a konténert, mert már fut azon a porton!
- Ezért vagy a gui-n állítsuk le és töröljük a korábbi konténert, vagy az alábbi 3 parancsot használjuk:
  - docker ps (ez listázza a processzeket id-vel együtt)
  - docker stop <id>, majd docker rm <id>
  - A kettő egyesíthető is a docker rm -f <id> (force kapcsoló) parancccsal

# Példa

- Ezt követően futtassuk az image-t, akár a konzolról (docker run -dp 3000:3000 getting-started ) parancs, akár a gui-ból
- Látható, hogy most már 2 konténerünk is fut:



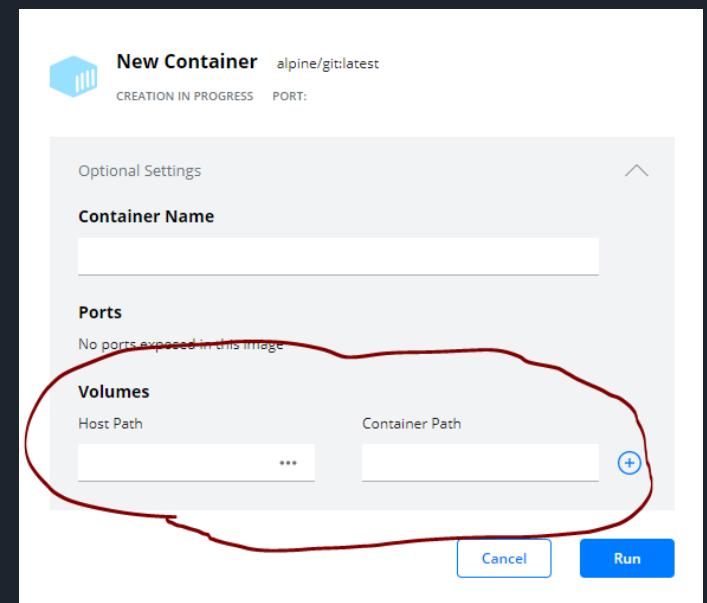
- Ha a kódunkat bármikor frissítjük, nem lehet csak úgy újra elindítani a konténert, mert már fut azon a porton!
- Ezért vagy a gui-n állítsuk le és töröljük a korábbi konténert, vagy az alábbi 3 parancsot használjuk:
  - docker ps (ez listázza a processzeket id-vel együtt)
  - docker stop <id> , majd docker rm <id>
  - A kettő egyesíthető is a docker rm -f <id> (force kapcsoló) parancccsal

# Perzisztálás

- Kis tapasztalattal rá fogunk jönni, hogy adott image minden egyes példányosításakor külön, egyedi fájlrendszer jön létre.
  - Ha a szoftverünk ír egy fájlt vagy adatbázist, ennek nem lesz nyoma, amikor az image-ből új konténert készítünk
- Erre nyújtanak megoldást a kötetek (angolul: volumes)
  - Egy volume azt írja le, hogy a konténer fájlrendszerének mely útvonalát perzisztáljuk, vagy hogy a konténer fájlrendszerének valamely útvonala hogyan párosítható a gazda OS fájlrendszerének valamely útvonalával
  - Kétféle volume típus létezik: named volume és bind mount.
  - A named volume felfogható úgy, mint egy vödör teli adattal ("bucket of data"), amihez nevet társítunk, de nem foglalkozunk azzal, hogy az adatok hol kapnak helyet - a Docker ezt maga megoldja nekünk.
  - A bind mount ezzel szemben meghatároz egy ún. mount pointot a gazda OS fájlrendszerén is, tehát tudjuk azt is, hogy hol férhetünk hozzá a fájlokhoz és szerkeszthetjük is azokat akár "offline" módon is.
    - Igaz, a docker volume inspect parancssal named volume helye is lekérdezhető

# Perzisztálás named volume segítségével

- Named volume-ot a “docker volume create” parancssal hozhatunk létre, melynek további argumentuma a volume neve
  - docker volume create todo-db
- Ezután amikor container-be példányosítjuk az image-t, a -v kapcsolóval lehet megadni, hogy melyik named volume-ot használja a konténer fájlrendszerének mely pontján. Pl. ha az alkalmazásunk a /etc/todos alá menti a todo.db nevű sqlite adatbázist, akkor indíthatjuk a konténert így:
  - docker run -dp 3000:3000 -v todo-db:/etc/todos getting-started
- Természetesen minden mindez a GUI felületén is konfigurálható:



# Perzisztálás bind mount segítségével

- A bind mount nagyon hasonló a named volume-hoz, azzal a kivétellel, hogy mi határozzuk meg a host OS-en levő mount point-ot is
- Gyakori felhasználás, hogy az alkalmazásunk forráskódja egy olyan alkönyvtárában található az image-nek, amit a konténerben felfountolunk a gazda filerendszerre is - így a konténerben futó alkalmazásunk reagálhat arra, ha futás közben módosítjuk a forráskódot!
  - Ilyen megoldást tesz lehetővé például a node környezethez használható nodemon alkalmazás is - ez egy script ami újraindítja pl. a node servert, ha változik a kódja
- Egyébként a kétféle volume típus használata nagyjából megegyezik (a -v kapcsolót használjuk) - további részletek a weben megtalálhatóak.

# Docker a gyakorlatban: multi-konténer alkalmazások

- A gyakorlatban arra célszerű törekednünk, hogy minden konténerünk 1 dolgot csináljon, de azt csinálja jól
- Ha olyan összetett alkalmazást tervezünk, amely adatbázist, backendet, frontendet is tartalmaz, ezeket célszerű külön konténerbe rakni, több ok miatt is:
  - Deploy-t követően a skálázási igények eltérőek lehetnek a különböző komponensekre
  - Ha több konténerünk van, ezek külön verziózhatóak / frissíthetőek
  - Egy-egy konténer más lehet a lokális és a deployolt verzióban (pl. lehet hogy lokálisan van egy saját adatbázisunk, a felhőben viszont már egy menedzselt adatbázist használunk adott felhő szolgáltatón)
  - Ha több processzből áll az alkalmazásunk, ez bonyodalmakat vonhat magával, mivel alapvetően 1 konténer 1 processzben fut.

# Docker a gyakorlatban: multi-konténer alkalmazások

- A konténerek példányosításakor lényegében megadható az is, hogy milyen network-höz kapcsolódnak, és ha két konténer azonos network-höz kapcsolódott, akkor kommunikálhatnak egymással
  - IP-cím illetve egy felhasználó által megadott alias alapján is!
  - Ehhez a -network és -network-alias kapcsolók szükségesek
- A docker compose nevű tool-lal automatizálható akár egy több-konténeres alkalmazás elindítása is – minden össze egy .yml leírásban meg kell adni, hogy az alkalmazás milyen image-ekből áll és ezeket hogyan kell elindítani. Ezt követően a docker compose egy parancssal minden elindít.