Docker

Hogyan lesz egy virtuális gépből „pehelykönnyű'' — és mi mindent épít erre a mai konténer-ökoszisztéma.

Honnan indulunk — és hova jutunk el

1. blokk — Motiváció: miért nem elég a hagyományos szerver-üzemeltetés

2. blokk — VM vs. konténer: architektúra, namespace-ek, cgroups, OverlayFS

3–5. blokk — Alapfogalmak, tárolás/hálózat/Compose, majd Swarm/Kubernetes

6–8. blokk — Életciklus/parancsok, biztonság, és az ökoszisztéma (OCI, Portainer stb.)

A 6. óra zárásában ígértük: *a mai óra a Docker-ről szól.* Az 5. órán pedig külön megjegyeztük, hogy az OverlayFS-re *„a félév második felében, a Docker-blokkban még visszatérünk"* — ma ezt is beváltjuk.

1. BLOKK

Motiváció

Miért nem elég ráengedni mindent egyetlen szerverre — és miért drága egy virtuális gép minden apró feladatra?

Egy szerver, öt weboldal, öt igény

Képzeljünk el egy szervert, amin 5 weboldal fut: az egyik nginx alatt, a másik Node.js-sel, a harmadik PHP7-tel, a negyedik PHP8-cal, mindegyik más adatbázist (MySQL, PostgreSQL, MSSQL...) kíván.

Ezeket **egymás mellett, ugyanazon a gépen** üzemeltetni — eltérő csomagverziókkal, eltérő függőségekkel — gyakorlatban rendkívül nehézkes, gyakran egyenesen lehetetlen konfliktusok nélkül.

Megoldás lehetne 5 külön virtuális gép is — de ez komoly erőforrás-többletet (RAM, lemez, bootolási idő) jelent 5 teljes operációs rendszer fenntartásáért, miközben valójában csak egy-egy alkalmazást futtatnánk bennük.

Amire igazából szükségünk van

- **Elszigeteltség** — az egyik alkalmazás frissítése/összeomlása ne vigye magával a többit
- **Hordozhatóság** — ami a fejlesztői gépen működik, működjön ugyanúgy az éles szerveren is
- **Gyorsaság** — induljon másodpercek alatt, ne percek/bootolási idő alatt
- **Alacsony overhead** — ne kelljen 5 teljes operációs rendszert fenntartani 5 alkalmazáshoz

A virtuális gép az első kettőt megadja, de a harmadik-negyedik pont ellen dolgozik. A **konténer** — és ezen belül a Docker — pont ezt a négyet egyszerre próbálja biztosítani.

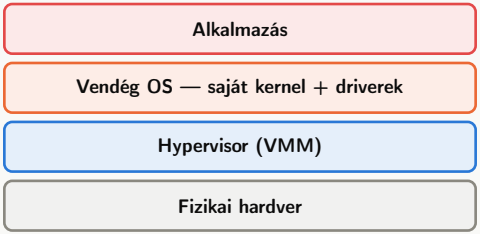
2. BLOKK

VM vs. konténer — architektúra

Ugyanazt a célt (elszigetelt alkalmazás-környezet) két gyökeresen más úton érik el.

A virtuális gép rétegei

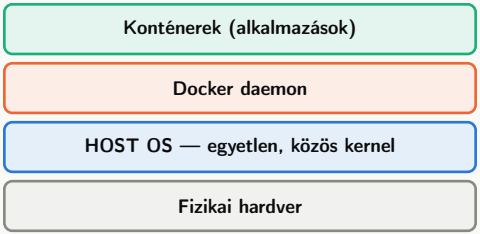
Minden virtuális gép saját, teljes **vendég operációs rendszert** (saját kernellel) futtat a hypervisor fölött.



⚠ A hypervisor minden VM-nek egy (akár fiktív, kompatibilitás miatt „hamisított") hardverkörnyezetet mutat — ezt a 4. órán **HVM/paravirtualizáció** néven már tárgyaltuk.

A Docker rétegei

A konténerek a **HOST operációs rendszer kernelét használják** — nincs saját vendég-kernel, nincs hardvervirtualizáció.



Nincs „memory map”, amit minden hozzáférésnél forwardolni kellene, és a konténerben futó operációs rendszer-réteg is minimális — innen a **„pehelykönnyű”** jelző.

A döntő különbség: 3 példányban

Ha **3 alkalmazást** akarunk elszigetelve futtatni, a két modell gyökeresen más mennyiségű operációs rendszert igényel.



Balra (VM): minden alkalmazás **saját, teljes kernelt** cipel magával. Jobbra (Docker): mindhárom alkalmazás **ugyanazt az egy kernelt** osztja meg — ez adja a kisebb erőforrásigényt és a gyorsabb indulást.

Mitől „elszigetelt” egy konténer, ha nincs külön kernel?

A Linux kernel két beépített mechanizmusa teszi lehetővé az izolációt hardvervirtualizáció nélkül.

Namespace-ek

Elválasztják, hogy egy folyamat **mit lát** — saját PID-listát, hálózati interfészeket, mountolt fájlrendszereket, hostname-et stb. A konténer „azt hiszi”, egyedül van a gépen.

Cgroups

Korlátozzák, hogy egy folyamat(csoport) **mennyi erőforrást** (CPU, memória, I/O) használhat — ez előzi meg, hogy egy konténer „megegye” a gazdagép összes erőforrását.

Ezek **nem Docker-specifikus** találmányok — önmagukban is régóta léteznek a Linux kernelben, a Docker (és a többi konténer-futtatókörnyezet) csak kényelmes, egységes felületet ad a kezelésükhöz.

Image-rétegek — amit az 5. órán már láttunk

Minden Dockerfile-utasítás (`RUN`, `COPY...`) egy új, **csak olvasható réteget** hoz létre — a futó konténer kap egy saját, **írható** réteget is, ami csak a változásokat tárolja.



Ez ugyanaz az **OverlayFS**, amit az 5. órán, a Linux fájlrendszerek kitekintőjénél megígértünk — itt derül ki, miért ez a Docker image-rétegezésének technikai alapja, és miért olyan gyors egy image build (a már meglévő rétegek cache-elve újrafelhasználhatók).

3. BLOKK

Docker alapfogalmak

Image, konténer, registry, Dockerfile — a szókincs, amire minden további dia épül.

Image vs. konténer

Image

Egy **statikus, csak olvasható sablon** — rétegek sorozata, ami tartalmazza az alkalmazást és minden függőségét. Önmagában nem „fut”.

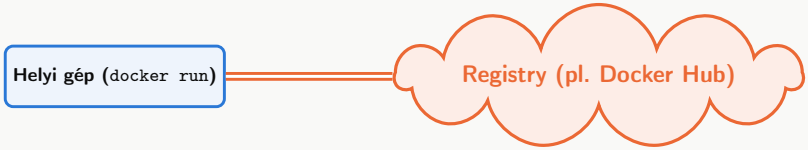
Konténer

Az image **futó példánya** — kap egy saját, írható réteget (előző dia), saját namespace-eket és cgroup-limiteket.

Ugyanabból az image-ből **egyszerre több konténer** is indítható — mindegyik a saját, egymástól független írható rétegével.

Registry — honnan jönnek az image-ek

A **registry** tárolja és szolgálja ki az image-eket — alapértelmezetten a Docker Hub-ot (`docker.com`) használjuk, de üzemeltethetünk sajátot is.



⚠ A Docker Hub-ra **bárki** tölthet fel publikus image-et — ez kényelmes, de kockázat is: mindig ellenőrizzük, megbízható forrásból (hivatalos/verified kiadótól) származik-e, amit letöltünk és futtatunk.

`docker.io` vs. Docker CE — melyiket telepítsd?

`apt install docker.io`

Ubuntu/Debian **saját tárolójából** települ — egyszerű, de gyakran **régebbi** Docker-verziót ad.

Docker hivatalos apt tárolója (Docker CE)

Külön repót kell hozzáadni (lásd a hivatalos telepítési útmutatót) — cserébe mindig **naprakész** Docker verziót kapunk.

Figyelem: ne telepíts `docker.io` **mellé** külön `docker` csomagot is — az egy régi, nem konténeres segédprogram (névütközés miatt kapta ezt a nevet a Docker csomag Debian/Ubuntu alatt). Gyakorláshoz a `docker.io` bőven elég, éles környezetben a Docker CE javasolt.

A Dockerfile öt alapvető kulcsszava

FROM

Melyik meglévő image-ből indulunk ki (pl. `ubuntu`) — ez lesz a legalsó réteg.

RUN

Build időben lefuttatandó parancs (pl. csomagtelepítés) — minden `RUN` egy új réteg.

EXPOSE

Mely portokon kommunikálhat a konténer a HOST felé — önmagában nem nyit portot kifelé.

ENTRYPOINT

A konténer indításakor **mindenképp** lefutó program — amíg ez fut, a konténer is fut.

CMD

Alapértelmezett paraméterek az ENTRYPOINT-hoz — a felhasználó felülírhatja `docker run` végén.

A futtatás képlete:
`${ENTRYPOINT} ${CMD}`

Egy rövid, valódi Dockerfile

Az előző dia öt kulcsszava a gyakorlatban — egy apache2-t futtató image, **nem root** felhasználóval (7. blokk előfutára).

```
FROM ubuntu
RUN apt update && apt install apache2 -y
EXPOSE 80
USER www-data
ENTRYPOINT ["apache2ctl", "-D", "FOREGROUND"]
```

Build: `docker build -tag sajat/apache2 .` — a `.` a build kontextus (az aktuális mappa, ahol a Dockerfile van).

Gyakori csapda: a `CMD` tömb-formája

Exec formában (szögletes zárójelekkel) a `CMD` tömb **minden eleme külön, önálló argumentumként** adódik át.

⚠ `CMD ["-D FOREGROUND"]` — ez **egyetlen**, szóközt tartalmazó argumentumként adódik át, amit a legtöbb program nem tud értelmezni, és a konténer elszáll. Helyesen: `CMD ["-D", "FOREGROUND"]` — két külön tömbelem.

ℹ Apró szintaktikai különbség, de ez az egyik leggyakoribb oka annak, hogy egy frissen megírt Dockerfile-lal induló konténer azonnal leáll.

4. BLOKK

Tárolás, hálózat, Compose

Egy konténer önmagában ritkán elég — hogyan marad meg az adata, hogyan beszélnek egymással, és hogyan indítjuk el őket egyszerre?

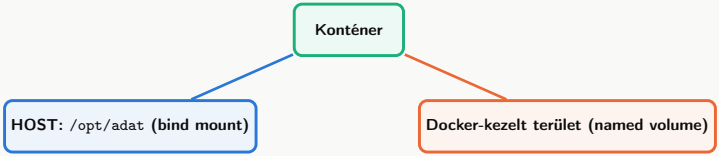
Miért kell egyáltalán volume?

A 2. blokkban látott **írható réteg** (upperdir) a konténeré — ha a konténert töröljük, **ezzel együtt vész el** minden benne keletkezett adat.

⚠ Egy adatbázis-konténernél ez katasztrófa lenne: egy `docker rm`, egy image-frissítés vagy akár egy egyszerű újraépítés az **összes adatot** magával vinné.

✓ A megoldás: a perzisztens adatot **kivisszük** a konténer írható rétegéből egy olyan helyre, ami túléli a konténer törlését — ez a **volume**.

Bind mount vs. named volume



Bind mount

`-v /host/path:/c/path` — egy **konkrét HOST-könyvtárat** kötünk be, a Docker semmit nem kezel belőle.

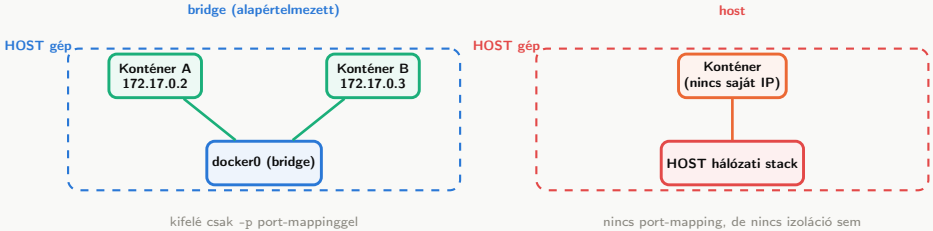
Named volume

`-v nev:/c/path` — a Docker **saját kezelésű** tárterülete; `docker volume ls` listázza, hordozhatóbb.

Hasonló különbség, mint az 5. órán egy konkrét lemezút és egy menedzselt tárolási szolgáltatás (EBS) között — csak lokálisan. Dockerfile-ban a `VOLUME` kulcsszó jelöl ilyen pontot.

Hálózati módok: bridge vs. host

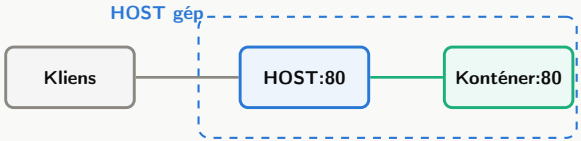
Indításkor eldönthető, a konténer **milyen hálózatot lásson** — ez a `-network` kapcsolóval állítható.



bridge: minden konténer saját, virtuális IP-t kap, elszigetelve — emiatt kell a következő dián látott port-mapping. **host**: a konténer a HOST hálózati namespace-ét használja közvetlenül — gyorsabb, de a konténer pontosan annyira van kitéve, mint maga a HOST. (Van egy harmadik, **none** mód is: teljesen hálózat nélküli konténer.)

Port-mapping — hogyan ér el kívülről egy konténert

bridge módban (előző dia) a konténer portja alapból **csak a Docker belső hálózatán** érhető el — a `-p host_port:container_port` kapcsolóval nyitjuk ki a HOST felé.



`-p 80:80` — a HOST 80-as portját irányítjuk a konténer 80-as portjára. A HOST-oldali IP lehet `0.0.0.0` (mindenhonnan), `127.0.0.1` (csak helyben) vagy egy konkrét belső IP.

Docker Compose — több konténer, egy fájl

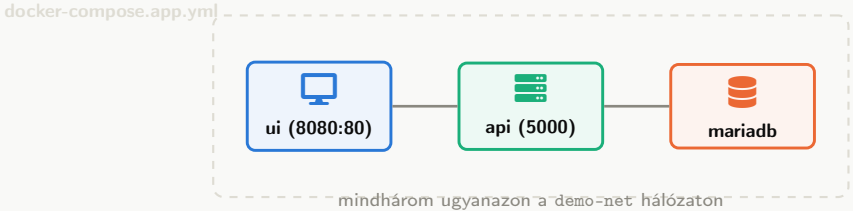
Egy valódi alkalmazás ritkán egyetlen konténer — a gyakorlaton épített demo-app is **UI** + **API** + **adatbázis** hármasból állt.

⚠ Három különálló `docker run` parancsot kézzel, helyes sorrendben, helyes hálózati beállításokkal elindítani — és minden újraindításkor megismételni — hibára és felesleges időveszteségre ad lehetőséget.

✓ A **Compose** egyetlen YAML fájlban írja le az összes szolgáltatást, a köztük lévő hálózatot és a build-lépéseket — egyetlen `docker compose up` paranccsal indul az egész.

Amit mi is használtunk: a demo-app

Ugyanaz a hármas, amit a `docker-compose.app.yml`-ben már megírtunk és lefuttattunk.



Egy `services:` blokk szolgáltatásonként (build/image, környezeti változók, portok), a Compose pedig automatikusan létrehozza a köztük lévő belső hálózatot — a konténerek **a szolgáltatás nevén** (pl. `api`, `mariadb`) szólítják meg egymást, nem IP-n.

Ugyanez valódi YAML-ban (röviden)

Az előző dia ábrájának rövidített `docker-compose.yml`-je.

```
services:
  api:
    build: ./api
    environment:
      DB_HOST: mariadb
    networks: [demo-net]
  mariadb:
    image: mariadb:11
    networks: [demo-net]
networks:
  demo-net:
```

Indítás: `docker compose up -d` — egy parancs buildeli/indítja mindkettőt, a hálózattal együtt.

5. BLOKK

Orchestration — Swarm & Kubernetes

A Compose egyetlen gépre szól — mi van, ha a konténereinket *több* gépen, automatikusan skálázva akarjuk futtatni?

Egy gép már nem elég

- Mi van, ha a terhelés túlnő **egyetlen gép** kapacitáján?
- Mi van, ha egy konténer (vagy maga a gép) **összeomlik** — ki indítja újra, és hová?
- Mi van, ha **frissíteni** akarunk leállás nélkül (rolling update)?

Ugyanaz a probléma, mint az AWS Auto Scaling Groupnál (gyakorlat) — csak itt nem EC2 példányokat, hanem konténereket kell automatikusan indítani/leállítani, több gépen elosztva.

Docker Swarm — beépített, egyszerű

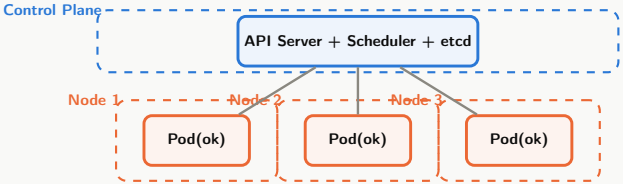
A Docker daemon-ba **eleve beépített** orchestrator — nincs külön telepítés, pár paranccsal (`docker swarm init`) üzembe állítható.

Alacsony tanulási görbe, jól illeszkedik a már megtanult `docker`/`compose` parancsokhoz (`docker stack deploy` a Compose fájlt is tudja hasznosítani).

Cserébe jóval kevesebb funkciót, kisebb ökoszisztémát és közösséget kínál, mint a következő dián látott alternatíva — emiatt mára inkább kisebb/egyszerűbb igényekhez választják.

Kubernetes — a de facto szabvány

Google belső rendszeréből (Borg) nőtte ki magát — ma a legelterjedtebb, leggazdagabb ökoszisztémájú konténer-orchestrator.



A **Pod** a legkisebb ütemezhető egység (egy vagy több, szorosan összetartozó konténer) — a Control Plane dönti el, melyik Node-ra kerüljön, és gondoskodik róla, hogy mindig a kívánt számú Pod fusson.

Swarm vagy Kubernetes?

	Swarm	Kubernetes
Telepítés	Beépített a Dockerbe	Külön kell telepíteni/üzemeltetni
Tanulási görbe	Alacsony	Meredek
Ökoszisztéma, eszközök	Kisebb	Hatalmas (Helm, operátorok stb.)
Felhő-menedzselt verzió	Ritka	EKS, GKE, AKS — mindenhol
Elterjedtség (2026)	Csökkenő	Ipari szabvány

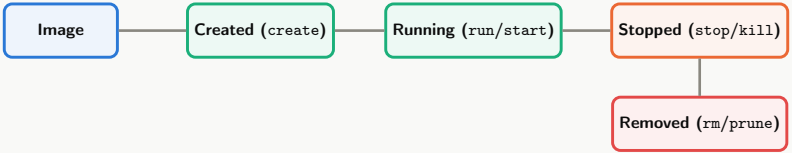
Kis projekthez/tanuláshoz a Swarm (vagy akár csak a Compose) tökéletesen elég — komolyabb, több csapatot kiszolgáló rendszereknél szinte mindenhol Kubernetesszel találkozol majd.

6. BLOKK

Docker a gyakorlatban

Az életciklus és a leggyakrabban használt parancsok — gyors összefoglalóként.

Egy konténer életciklusa



Fontos szabály: a konténer **addig fut, amíg az ENTRYPOINT/CMD programja fut** — ha az kilép (pl. egy interaktív `bash`-ból `exit`-tel), a konténer is azonnal leáll.

Parancsok — gyors referencia

Parancs	Mire való
`docker run -ti <image>`	Új konténer indítása interaktív terminállal
`docker run -d <image>`	Új konténer indítása a háttérben (daemonizálva)
`docker ps / container ls`	Futó konténerek listázása
`docker exec -ti <id> bash`	Terminál nyitása egy már futó konténerhez
`docker stop/kill <id>`	Normál / erőszakos leállítás
`docker build -tag <név> .`	Image építése a `Dockerfile` alapján
`docker image ls / prune`	Image-ek listázása / nem használtak törlése
`docker system df`	Mennyi lemezt foglalnak az image-ek/konténerek/volume-ok
`docker system prune`	**Minden** nem használt erőforrás törlése egy paranccsal

⚠ A `docker system prune -a` a nem futó konténerek mellett **minden olyan image-et is törli**, amit épp nem használ egy futó konténer sem — nagyon hasznos takarításhoz, de visszavonhatatlan.

7. BLOKK

Biztonság konténerekkel

A 6. órán tanultak nem álltak meg az AWS határánál — a konténerek világában is megvan a megfelelőjük.

Megosztott kernel = megosztott kockázat

A 2. blokkban látott előny (egy közös kernel) egyben a legfőbb biztonsági **kompromisszum** is.

⚠ Egy **kernel-sebezhetőség** kihasználásával egy konténerből elvileg ki lehet törni a HOST rendszerbe — egy VM-nél ehhez a *hypervisort magát* is fel kéne törni, ami jóval nehezebb, mélyebb izolációt ad.

ℹ Ezért gyakori minta: **sandboxing-ra** (gyors, eldobható kipróbálásra) kiválóan jó a konténer, de igazán érzékeny, bizalmatlan kóddal szemben egy VM (vagy kifejezetten erre épülő megoldás, pl. AWS Firecracker a Lambda mögött — lásd 2. óra) erősebb határt ad.

Ne fuss root-ként a konténerben

Alapértelmezetten egy konténerben futó folyamat **root** jogosultsággal fut — pontosan ugyanaz a „legkisebb jogosultság elve” (6. óra, IAM) vonatkozik itt is.

⚠ Ha a konténeren belül root-ként fut az alkalmazás, egy esetleges container escape (előző dia) **root jogosultsággal** jut ki a HOST-ra — ennél rosszabb kiindulás nemigen van.

✓ A Dockerfile-ban a `USER` kulcsszóval állítható be egy nem-root felhasználó — a gyakorlaton látott apache2 példa pontosan ezt mutatta be (`useradd`, `chgrp`/`chmod`, majd `USER admin`).

Honnan jön az image, amit futtatsz?

A 3. blokkban már jeleztük: a Docker Hub-ra **bárki** tölthet fel publikus image-et.

⚠ Egy rosszindulatú/sebezhető image futtatása ugyanolyan kockázat, mint egy ismeretlen forrásból letöltött futtatható fájl — csak itt „csak egy image-ről” van szó, ami megtévesztő biztonságérzetet adhat.

✓ Részesítsük előnyben a **hivatalos (Official Image)** vagy **Verified Publisher** jelzésű image-eket, és — ugyanúgy, ahogy a 6. órán az OpenVAS-nál tárgyaltuk — érdemes a felhasznált image-eket ismert sebezhetőség-adatbázis (CVE) ellen is átvizsgálni.

8. BLOKK

A konténer-ökoszisztéma

A Docker a legnépszerűbb, de nem az egyetlen szereplő — és a parancssor sem az egyetlen út a kezeléséhez.

OCI — a szabvány a kulisszák mögött

Az **Open Container Initiative** (2015, Linux Foundation — maga a Docker kezdeményezte) két szabványt határoz meg, hogy a konténer-technológia ne egyetlen cégé legyen.

Image Spec

Hogyan épül fel egy image (rétegek, manifest) — emiatt fut egy Dockerrel épített image más eszközön is.

Runtime Spec

Hogyan kell egy image-ből ténylegesen konténert indítani (namespace-ek, cgroups) — ennek referencia-implementációja a `runc`.

Ez a „ragasztó”, ami miatt a következő dián látott eszközök (Podman, containerd) ugyanazokat az image-eket tudják futtatni, mint a Docker — nincs „Docker-lock-in”.

Alternatív konténer-futtatókörnyezetek

LXC / LXD

Linux Container — „rendszer-szintű” konténerek, egy teljes disztribúcióhoz közelebb állnak. A Proxmox is ezt/erre épít.

Podman

Docker-kompatibilis parancssor, de **daemon nélkül** fut, és natívan támogatja a root nélküli (rootless) konténereket — közvetlen válasz az előző blokk kockázatára.

containerd

Alacsony szintű futtatókörnyezet — a Docker daemon is *erre épül* a motorháztető alatt, a Kubernetes is gyakran ezt használja közvetlenül.

runc

A tényleges OCI-szabványos konténer-indító — a namespace-ek és cgroups beállítását végzi; szinte minden fenti eszköz végső soron erre (vagy egy hasonló OCI runtime-ra) támaszkodik.

GUI-k és menedzsment eszközök

Egyik sem helyettesíti a parancssor ismeretét, de napi üzemeltetésben sokat gyorsítanak.

Portainer

Legnépszerűbb webes admin felület — konténerek, image-ek, hálózatok kattintgatással, logok és terminál böngészőből.

Docker Desktop

Hivatalos Windows/macOS fejlesztői környezet — beépített GUI, egy kattintásra bekapcsolható Kubernetesszel.

Lazydocker

Terminál-alapú (TUI) felület — gyors áttekintés SSH-n át, grafikus felület nélkül is.

Rancher / Rancher Desktop

Teljes Kubernetes-menedzsment platform, illetve ennek könnyűsúlyú, helyi fejlesztői változata.

ÖSSZEFOGLALÁS

Mit vittünk haza?

- A konténer **megosztott kernelt** használ hardvervirtualizáció helyett — ettől „pehelykönnyű” a VM-hez képest
- Az izolációt a Linux **namespace-ek és cgroups** adják, a rétegzett image-eket az **OverlayFS** (5. óra callback)
- Alapfogalmak: **image** (sablon) vs. **konténer** (futó példány), **registry**, **Dockerfile** (FROM/RUN/EXPOSE/ENTRYPOINT/CMD)
- A konténer írható rétege **efemer** — perzisztens adathoz **volume** kell (bind mount vagy named volume, 5. óra callback)
- **Compose** egy gépen fog össze több szolgáltatást (lásd a demo-app-ot); **Swarm**/**Kubernetes** ugyanezt több gépen, automatikus skálázással
- Biztonság: megosztott kernel = nagyobb kockázat, ne fuss root-ként, csak megbízható image-et használj (6. óra elvei itt is élnek)
- Az **OCI** szabvány köti össze az ökoszisztémát: **Podman, containerd, runc, LXC**/**LXD** és GUI-k (**Portainer** stb.) mind ugyanazokat az image-eket futtatják

KÖVETKEZŐ ÓRÁK: KISELŐADÁSOK

Kérdések?

Ezzel lezártuk a féléves tananyag-blokkot — a motivációtól a VM/konténer architektúrán, Compose-on és Kubernetesen át a biztonságig és az ökoszisztémáig. Innentől a hallgatói kiselőadásoké a szó.