Webtechnológia és webalkalmazás-fejlesztés - Docker

Docker

Magda Donát

Széchenyi István Egyetem, Győr

https://git.mdnd-it.cc/Donat/GKNB$_M$STM071

2026. február 17.

Mi a Docker?

Docker – Konténerizációs platform

- 2013-ban indult nyílt forráskódú projekt
- Alkalmazások csomagolása és futtatása izolált környezetben
- Könnyűsúlyú alternatíva a virtuális gépeknek

Docker – Definíció

Definíció

A Docker egy platform, amely lehetővé teszi alkalmazások és függőségeik **konténerekbe** csomagolását, amelyek bárhol futtathatók.

Filozófia

„Build once, run anywhere” – egyszer csomagolva, bárhol fut.

A probléma, amit megold

„Works on my machine” probléma

- Fejlesztői gépen működik, production-ön nem
- Különböző operációs rendszerek
- Eltérő függőség verziók

A probléma – Következmények

További problémák

- Bonyolult telepítési folyamatok
- Nehezen reprodukálható hibák
- Inkonzisztens viselkedés környezetek között

Docker megoldás

Hogyan oldja meg?

- Azonos környezet mindenhol (dev, test, prod)
- Összes függőség a konténerben
- Gyors telepítés és indítás

Lényeg

Egyszer csomagolva, bárhol fut – garantáltan!

Miért használjunk Dockert? – Fejlesztőknek

Fejlesztői előnyök

- Gyors setup – percek helyett másodpercek
- Konzisztens környezet minden fejlesztőnél
- Izoláció – egymástól független projektek

Miért használjunk Dockert? – Gyakorlatban

Gyakorlati előnyök

- Új csapattag: `docker compose up` és kész
- Nincs „nálam működik” probléma
- Különböző Node.js verziók párhuzamosan

Miért használjunk Dockert? – DevOps

DevOps előnyök

- Automatizálás – CI/CD pipeline egyszerűsítése
- Skálázhatóság – könnyű horizontális skálázás
- Mikroszolgáltatások – ideális architektúra

Docker fő előnyök

Összesítve

- Hordozhatóság – bárhol fut
- Könnyűsúlyú – MB-ok VM GB-jai helyett
- Gyors indítás – másodpercek

Docker vs Virtuális gépek – VM jellemzői

Virtuális gép (VM)

- Teljes operációs rendszer minden VM-nek
- Hypervisor szükséges (VMware, VirtualBox)
- Lassabb indítás – perceket vehet igénybe

Szerkezet

Hardware → Host OS → Hypervisor → Guest OS → App

Docker vs Virtuális gépek – VM hátrányok

VM hátrányok

- Nagy méret – több gigabájt
- Több erőforrás igény – CPU, memória, tárhely
- Teljes OS virtualizálás – minden VM-nek saját kernel

Docker vs Virtuális gépek – Docker jellemzői

Docker konténer

- Osztott OS kernel a host-tal
- Docker Engine futtatja
- Gyors indítás – másodpercek

Szerkezet

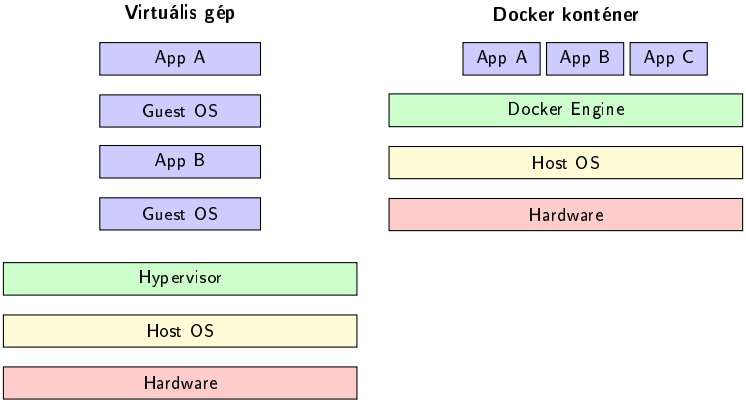
Hardware → Host OS → Docker Engine → App

Docker vs VM – Kulcskülönbség

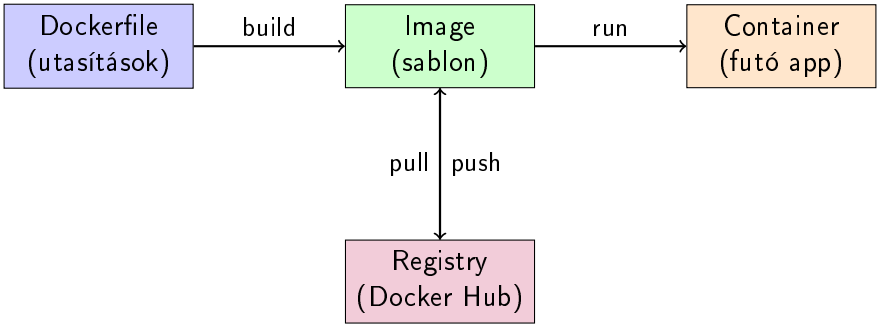
Kulcskülönbség

- Alkalmazás szintű izoláció az OS kernelen
- Konténerek osztják a host kernel-t
- Kisebb méret – megabájtok vs gigabájtok

Architektúra összehasonlítás



Docker életciklus



Docker Engine

Docker Engine – A fő komponens

- A Docker fő komponense
- Kliens-szerver architektúra
- 3 fő rész: Daemon, REST API, CLI

Docker Engine – Részei

1. **Docker Daemon (dockerd)**: Háttérfolyamat, kezeli a konténereket
2. **Docker CLI (docker)**: Parancssor interfész
3. **REST API**: Kommunikáció daemon és CLI között

Működés

Parancs: `docker run nginx`
CLI → REST API → Daemon → Image letöltés & Konténer indítás

Használati esetek – Fejlesztés

Fejlesztési környezet

- Gyors projekt setup másodpercek alatt
- Különböző verziók párhuzamosan (Node 16, 18, 20)
- Csapat tagjai azonos környezetben dolgoznak

Példa

Új fejlesztő csatlakozik: `git clone` + `docker compose up`
⇒ Azonnal dolgozhat, telepítés nélkül

Használati esetek – Mikroszolgáltatások

Mikroszolgáltatások

- Minden szolgáltatás saját konténerben fut
- Független skálázás és deployment
- Könnyű verziókezelés

Használati esetek – CI/CD

CI/CD pipeline

- Automatizált build és tesztelés
- Konzisztens deployment minden környezetben
- Gyors rollback hibás verzióról

Mi kerül konténerbe? – Alapok

Alapvető elemek

- **Alap OS réteg**: Linux disztribúció (Alpine, Ubuntu)
- **Runtime**: Node.js futtatókörnyezet (pl. 18.x)
- **Függőségek**: npm csomagok (node_modules mappa)

Mi kerül konténerbe? – Alkalmazás

Alkalmazás specifikus elemek

- **Alkalmazás kód**: forrásfájlok (src/, index.js)
- **Konfigurációk**: package.json, környezeti beállítások
- **Indítási parancs**: npm start vagy node index.js

Eredmény

Önálló, hordozható csomag, amely bárhol futtatható

Mi NEM kerül konténerbe? – Adatok

Perzisztens adatok

- **Adatbázis adatok** – volume-ban kell tárolni
- **Nagy méretű log fájlok** – host-ra vagy log service-re
- **Feltöltött fájlok** – user uploads, media files

Miért?

Konténer törléskor ezek az adatok elvesznének!

Mi NEM kerül konténerbe? – Biztonság

Biztonsági és development elemek

- **Érzékeny konfigurációk** – titkos kulcsok, jelszavak
- **Fejlesztői eszközök** – debugging, testing tools
- **Build artifacts** – ideiglenesen használt fájlok

Jó gyakorlat

Titkos kulcsok: Docker secrets vagy környezeti változók
Dev tools: csak development image-ben

Docker Desktop – Mi az?

Docker Desktop alkalmazás

- Grafikus felületű alkalmazás Windows és macOS rendszerekhez
- Tartalmazza az összes szükséges Docker komponenst
- Beépített GUI konténerek és image-ek kezeléséhez

Docker Desktop – Technológia

Technológiai előnyök

- Integrált Kubernetes támogatás
- WSL 2 backend Windows-on (Linux kernel)
- Automatikus frissítések

Docker Desktop – Funkciók

Fő funkciók

- Konténerek vizuális monitorozása
- Resource használat követése (CPU, RAM)
- Volume és network kezelés GUI-n keresztül

Docker Desktop – Kiegészítők

Kiegészítők

- Extension marketplace – további funkciók
- Dev Environments támogatás
- Docker Scout – biztonsági elemzés

Docker Desktop – Telepítés

Telepítési lépések

1. Letöltés: `docker.com/products/docker-desktop`
2. Telepítő futtatása
3. Rendszer újraindítása (ha szükséges)

Docker Desktop – Ellenőrzés

Ellenőrzés

1. Docker Desktop indítása
2. Ellenőrzés terminálban: `docker -version`

Docker Desktop – Windows és macOS

Windows és macOS

- **Docker Desktop ajánlott** – grafikus telepítő
- Tartalmaz mindent: Docker Engine, CLI, Compose
- Automatikus frissítések

Docker Desktop – Linux

Linux

- **Docker Engine** közvetlenül telepíthető
- Docker Desktop opcionális (beta)
- Parancssorból: `apt install docker.io`

Összefoglalás – Docker lényege

Docker = Modern konténerizáció

- Alkalmazások izolált, hordozható csomagolása
- Könnyűsúlyú alternatíva a virtuális gépeknek
- „Build once, run anywhere” filozófia

Összefoglalás – Előnyök

Miért használjuk?

- Gyors és könnyű deployment
- Hatékony erőforrás kihasználás
- Skálázható mikroszolgáltatások

Összefoglalás – Fogalmak

Build és futtatás

- **Dockerfile**: Utasítások az image építéséhez
- **Image**: Csak olvasható sablon az alkalmazásról
- **Container**: Futó példány egy image-ből

Folyamat

Dockerfile → build → Image → run → Container

Összefoglalás – Tárolás

Tárolás és kommunikáció

- **Registry**: Image-ek tárolása (pl. Docker Hub)
- **Volume**: Perzisztens adattárolás
- **Network**: Konténerek közötti kommunikáció

Következő lépések

Dockerfile készítés, image build, konténer kezelés, Docker Compose

Mi a Dockerfile?

Dockerfile – Image építési utasítások

- Szöveges fájl, amely leírja hogyan kell egy image-et építeni
- Tartalmazza az összes parancsot sorrendben
- Verziókezelhető (Git-ben tárolva)

Dockerfile – Jellemzők

Tulajdonságok

- Reprodukálható: ugyanaz a Dockerfile = ugyanaz az image
- Neve: `Dockerfile` (kiterjesztés nélkül)

Analógia

A Dockerfile olyan, mint egy **sütési recept** – lépésről lépésre leírja, hogyan készüljön az image.

Első Dockerfile – Kód

Legegyszerűbb példa

```
FROM ubuntu:22.04

CMD echo "Hello World!"
```

Első Dockerfile – Build és futtatás

Build és futtatás

```
# Image építése
docker build -t my-hello .

# Konténer futtatása
docker run my-hello
# Kimenet: Hello World!
```

Első Dockerfile – Magyarázat

Utasítások

- `FROM`: Alap image megadása
- `CMD`: Alapértelmezett parancs futtatáskor

Build paraméterek

- `-t`: Tag (név) adása az image-nek
- `.`: Build context (jelenlegi mappa)

Dockerfile alaputasítások – 1. rész

Alapépítő utasítások

1. `FROM` – Alap image kiválasztása
   - Minden Dockerfile ezzel kezdődik
   - Meghatározza a kiinduló környezetet
2. `WORKDIR` – Munkakönyvtár beállítása
   - Létrehozza és beállítja a dolgozó könyvtárat
   - Minden következő utasítás itt fut

Dockerfile alaputasítások – 2. rész

Fájl és csomag kezelés

3. `COPY` / `ADD` – Fájlok másolása
   - Fájlok hozzáadása az image-hez
   - COPY javasolt, ADD extra funkciókkal
4. `RUN` – Parancs futtatása build közben
   - Csomagok telepítése, fájlok létrehozása
   - Minden RUN új réteget hoz létre

Dockerfile alaputasítások – 3. rész

Konfiguráció és indítás

5. `ENV` – Környezeti változó beállítása
6. `EXPOSE` – Port deklarálása
7. `CMD` / `ENTRYPOINT` – Indítási parancs

Sorrend fontos!

A Dockerfile utasításai felülről lefelé hajtódnak végre

FROM – Alap image kiválasztása

Hivatalos image-ek

```
# Hivatalos Node.js image
FROM node:18

# Specifikus verzió
FROM node:18.16.0

# Alpine Linux alapú (kisebb méret)
FROM node:18-alpine
```

FROM – További példák

Más nyelvek és rendszerek

```
# Ubuntu
FROM ubuntu:22.04

# Python
FROM python:3.11
```

Ajánlás

Használj **specifikus verziót** (pl. `node:18.16`) a `latest` helyett!
Alpine verzió: kisebb méret, gyorsabb build.

WORKDIR – Kód

Beállítja a munkakönyvtárat

```
FROM node:18

# Munkakönyvtár létrehozása és beállítása
WORKDIR /usr/src/app

# Minden utána következő parancs itt fut
COPY package.json ./
RUN npm install
```

WORKDIR – Magyarázat

Funkcionalitás

- Létrehozza a könyvtárat, ha nem létezik
- Minden következő `RUN`, `COPY`, `CMD` itt fut
- Konténerben `docker exec` is ide lép be

Jó gyakorlat

Használj dedikált könyvtárat: `/app` vagy `/usr/src/app`

COPY – Fájl másolás

Egy és több fájl másolása

```
# Egy fájl
COPY package.json ./

# Több fájl
COPY file1.txt file2.txt ./
```

COPY – Mappák és teljes tartalom

Mappák másolása

```
# Mappa teljes tartalma
COPY src/ ./src/

# Minden a build context-ből
COPY . .
```

Jellemzők

- Egyszerű fájl és mappa másolás
- Látható és érthető
- **Ajánlott használni**!

ADD – Speciális másolás

ADD utasítás

```
# Automatikus kicsomagolás
ADD archive.tar.gz /app/

# URL letöltés (nem ajánlott)
ADD http://example.com/file ./
```

ADD vs COPY – Összehasonlítás

ADD vs COPY

- ADD: automatikus kicsomagolás, URL támogatás
- COPY: egyszerű, explicit, ajánlott

Best Practice

Használj `COPY`-t, kivéve ha kifejezetten kell az `ADD` extra funkciója!

URL-hez

URL letöltéshez jobb RUN + curl/wget használata

RUN – Csomagok telepítése

Csomag telepítés

```
# Csomag telepítés
RUN apt-get update && apt-get install -y curl

# NPM függőségek
RUN npm install

# Python csomagok
RUN pip install -r requirements.txt
```

RUN – Több parancs kombinálása

Több parancs egyetlen RUN-ban

```
# Több parancs (shell form)
RUN apt-get update && \
    apt-get install -y git curl && \
    apt-get clean
```

Fontos

Minden `RUN` egy új **réteg** (layer) az image-ben.
Kombináld parancsokat `&&`-del a rétegek csökkentéséhez!

ENV – Környezeti változók

Környezeti változók beállítása

```
# Egy változó
ENV NODE_ENV=production

# Több változó
ENV NODE_ENV=production \
    PORT=3000 \
    DB_HOST=localhost
```

ENV – Használat utasításokban

Használat későbbi utasításokban

```
ENV APP_HOME=/usr/src/app
WORKDIR $APP_HOME
```

Mikor érhető el?

- Build időben: későbbi `RUN` parancsokban
- Runtime-ban: futó konténerben
- Felülírható: `docker run -e NODE_ENV=dev`

EXPOSE – Port deklaráció

Dokumentálja, melyik porton fut az app

```
# HTTP port
EXPOSE 3000

# Több port
EXPOSE 3000 5000 8080

# UDP port
EXPOSE 53/udp
```

EXPOSE – Magyarázat

Fontos megértés

`EXPOSE` **NEM** nyitja meg a portot!

Funkció

- Csak **dokumentáció**, hogy melyik porton fut az alkalmazás
- Tényleges mapping: `docker run -p 3000:3000`

Hasznos

Docker Compose és orchestration eszközök használják ezt az információt.

CMD – Alapértelmezett parancs

CMD utasítás

```
# Shell form
CMD npm start

# Exec form (ajánlott)
CMD ["npm", "start"]

# Paraméterekkel
CMD ["node", "server.js"]
```

CMD – Felülírás futtatáskor

Felülírás futtatáskor

```
# Alapértelmezett CMD fut
docker run my-app

# CMD felülírása
docker run my-app npm test
```

Tippek

- Exec form ajánlott: `["cmd", "param"]`
- Shell form: stringek közvetlenül

ENTRYPOINT – Fő végrehajtható

ENTRYPOINT utasítás

```
# Exec form (ajánlott)
ENTRYPOINT ["node"]

# CMD paramétereket ad hozzá
CMD ["server.js"]
```

ENTRYPOINT – Paraméter változtatás

Paraméter változtatás

```
# server.js fut (CMD alapértelmezett)
docker run my-app

# app.js fut (CMD felülírva, de node marad)
docker run my-app app.js
```

Best Practice

Használj `CMD`-t egyszerű esetekben, `ENTRYPOINT + CMD`-t összetettebb esetekben.

Teljes Node.js példa – Dockerfile

Dockerfile Node.js alkalmazáshoz

```
# Alap image
FROM node:18-alpine

# Munkakönyvtár
WORKDIR /usr/src/app

# Függőségek másolása és telepítése
COPY package*.json ./
RUN npm ci --only=production
```

Teljes Node.js példa – Kód és indítás

Alkalmazás és futtatás

```
# Alkalmazás kódjának másolása
COPY . .

# Környezeti változó
ENV NODE_ENV=production

# Port deklarálás
EXPOSE 3000

# Indítási parancs
CMD ["node", "server.js"]
```

Layer caching – Probléma

Mi a probléma?

Minden Dockerfile sor egy **layer** (réteg).
Ha egy layer változik, az összes utána következő újraépül!

Layer caching – Nem hatékony megoldás

Nem hatékony megközelítés

```
FROM node:18
WORKDIR /app

# Minden változásra újra
COPY . .
RUN npm install

CMD ["npm", "start"]
```

Layer caching – Következmény

Következmény

Bármilyen kód módosítás után az `npm install` is újrafut,
pedig a `package.json` nem változott!

Példa

- 1 sor kód változik a src/app.js-ben
- `COPY . .` layer változik
- `RUN npm install` újrafut (pár perc!)

Layer caching – Optimalizálás

Jó megoldás

```
FROM node:18
WORKDIR /app

# Cache-elhető - csak ha package.json változik
COPY package*.json ./
RUN npm install

# Gyakran változik - de nem érinti a fenti layer-eket
COPY . .

CMD ["npm", "start"]
```

Layer caching – Alapelvek

Alapelv

Ritkábban változó dolgok előre, gyakran változók hátra!

Példa szervezet

1. FROM – alap image (soha nem változik)
2. Függőségek (package.json) → ritkán változnak
3. Forráskód → gyakran változik

Layer caching – Eredmény

Eredmény

Kód változtatás ⇒ csak COPY és CMD layer újraépül
npm install cache-elve marad!

.dockerignore – Mi ez?

Mi ez és miért fontos?

Megadja, mely fájlokat **NE** másolja be az image-be.
Hasonló a `.gitignore`-hoz.

Előnyök

- Kisebb image méret
- Gyorsabb build folyamat
- Biztonságosabb (nincs .env, .git)

.dockerignore – Node.js példa

Node.js projekt .dockerignore fájlja

```
# Node.js
node_modules
npm-debug.log

# Git
.git
.gitignore

# Fejlesztői fájlok
*.md
.env
.env.local
```

.dockerignore – További kizárások

Teszt és IDE fájlok

```
# Teszt fájlok
*.test.js

# IDE
.vscode
.idea

# Logs
logs
*.log
```

Multi-stage build – Alapok

Mi a multi-stage build?

- Több `FROM` utasítás egy Dockerfile-ban
- Különböző szakaszok különböző célokra
- Végső image: csak a szükséges fájlok

Multi-stage build – Előnyök

Miért jó?

- Build eszközök nem kerülnek a végső image-be
- Fejlesztői függőségek nem kellenek production-ben
- Kisebb méret = gyorsabb deploy, kevesebb tárhely

Multi-stage build – Méretösszehasonlítás

Méretösszehasonlítás

- Egy-stage build: ~500MB (Node.js + dev deps + TS compiler)
- Multi-stage build: ~150MB (csak Node.js + prod deps + JS)
- Méretcsökkenés: 70%!

Példa

TypeScript: `tsc` fordítás és csak a JS megy a végső image-be

Multi-stage build – Builder stage

Stage 1: Build

```
FROM node:18 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
```

Multi-stage build – Production stage

Stage 2: Production

```
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY --from=builder /app/dist ./dist
EXPOSE 3000
CMD ["node", "dist/server.js"]
```

Kulcs elemek

- `AS builder`: Első stage elnevezése
- `-from=builder`: Fájl másolás előző stage-ből

Multi-stage build – Go példa

Go alkalmazás – drasztikus méretcsökkentés

```
# Stage 1: Build
FROM golang:1.20 AS builder
WORKDIR /app
COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -o main .

# Stage 2: Production (scratch = üres image!)
FROM scratch
COPY --from=builder /app/main /main
EXPOSE 8080
ENTRYPOINT ["/main"]
```

Eredmény

Golang image: ~800MB $\Rightarrow$ Végső image: ~10MB

ARG – Build-time változók

ARG vs ENV

- `ARG`: Csak build közben érhető el
- `ENV`: Build és runtime is

ARG – Használat

ARG használata

```
# Alapértelmezett érték
ARG NODE_VERSION=18
FROM node:${NODE_VERSION}

ARG VERSION=1.0.0

# Convert to ENV (ha runtime kell)
ENV APP_VERSION=${VERSION}
```

ARG – Build argument átadása

Parancssorból

```
# Build argument átadása
docker build --build-arg NODE_VERSION=20 \
  --build-arg VERSION=2.0 .
```

USER – Biztonság

Biztonság

Soha ne futtass produkciós konténert root-ként!

Miért veszélyes a root?

- Ha a konténer kompromittálódik, a támadó teljes hozzáféréssel rendelkezik
- Biztonsági rések kihasználása könnyebb
- Host rendszer veszélyeztetett lehet

USER – Implementáció

Root user elkerülése

```
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production

# Nem privilegizált user létrehozása
RUN addgroup -S appgroup && \
    adduser -S appuser -G appgroup

# Fájlok tulajdonjogának változtatása
RUN chown -R appuser:appgroup /app
```

USER – Váltás és használat

User váltás

```
# User váltás
USER appuser

COPY --chown=appuser:appgroup . .

CMD ["node", "server.js"]
```

HEALTHCHECK – Állapot ellenőrzés

Konténer egészség ellenőrzés

```
HEALTHCHECK --interval=30s --timeout=3s \
  --start-period=5s --retries=3 \
  CMD node healthcheck.js
```

HEALTHCHECK – HTTP endpoint

HTTP endpoint ellenőrzés

```
HEALTHCHECK --interval=30s --timeout=3s \
  CMD wget --no-verbose --tries=1 \
  --spider http://localhost:3000/health || exit 1
```

HEALTHCHECK – Paraméterek

Paraméterek magyarázata

- `-interval=30s`: Ellenőrzések közötti idő
- `-timeout=3s`: Max várakozás egy ellenőrzésre
- `-start-period=5s`: Kezdeti várakozási idő

HEALTHCHECK – Retries

Újrapróbálkozás

- `-retries=3`: Hányszor próbálja újra hiba esetén
- 3 sikertelen után unhealthy státusz

Build parancsok – Alapok

docker build alapok

```
# Alap build
docker build -t my-app:latest .

# Specifikus Dockerfile
docker build -f Dockerfile.prod -t my-app:prod .

# Build argument
docker build --build-arg NODE_ENV=production \
  -t my-app .
```

Build parancsok – Haladó

Haladó build opciók

```
# No cache (teljes újraépítés)
docker build --no-cache -t my-app .

# Több tag egyszerre
docker build -t my-app:latest -t my-app:1.0.0 .
```

Best Practices – Verzió és méret

Követendő gyakorlatok

- Használj **specifikus verziókat** (`node:18.16` nem `latest`)
- **Alpine image**: kisebb méret
- **.dockerignore**: felesleges fájlok kizárása

Best Practices – Cache és struktúra

Követendő gyakorlatok

- **Layer cache**: függőségek előre, kód hátra
- **Multi-stage build**: kisebb production image
- **Egy konténer = egy folyamat** (ne futtass többet)

Best Practices – Biztonság

Követendő gyakorlatok

- **USER**: ne root userként futtass
- **HEALTHCHECK**: alkalmazás monitorozás
- RUN utasítások **kombinálása** (&&)

Anti-patterns – Biztonsági hibák

Biztonsági problémák

- Jelszavak, API kulcsok a Dockerfile-ban
- Root user production-ben
- sudo használata Dockerfile-ban

Anti-patterns – Alap image hibák

Alap image hibák

- `FROM ubuntu` és minden manuális telepítés
- `latest` tag production-ben

Megoldás

Használj official image-et (pl. `node:18-alpine`)

Anti-patterns – Cache hibák

Cache és build context hibák

- `COPY . .` a Dockerfile elejére (cache miss)
- `RUN apt-get update` külön sorban
- Nagy fájlok (logs, cache) az image-ben

Megoldások

- Függőségek előbb, kód később
- RUN parancsok összevonása

Production Dockerfile – Builder

Stage 1: Builder

```
FROM node:18-alpine AS builder
WORKDIR /app

# Függőségek telepítése
COPY package*.json ./
RUN npm ci

# Alkalmazás fordítása
COPY . .
RUN npm run build && npm prune --production
```

Production Dockerfile – Builder magyarázat

Mit csinál?

- Teljes node_modules telepítés (dev + prod)
- TypeScript/Build folyamat futtatása
- Production függőségek megtartása

Production Dockerfile – Alap

Stage 2: Production alap

```
FROM node:18-alpine
WORKDIR /app

# Biztonsági user létrehozása
RUN addgroup -S appgroup && \
    adduser -S appuser -G appgroup

# Csak a szükséges fájlok másolása
COPY package*.json ./
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/dist ./dist
```

Production Dockerfile – Indítás

Stage 2: Konfiguráció és indítás

```
# Tulajdonjogok beállítása
RUN chown -R appuser:appgroup /app
USER appuser

# Port és healthcheck
EXPOSE 3000
HEALTHCHECK --interval=30s --timeout=3s \
  CMD node healthcheck.js || exit 1

# Indítás
ENV NODE_ENV=production
CMD ["node", "dist/server.js"]
```

Összefoglalás – Főbb utasítások

Dockerfile = Image recept

- `FROM` – Alap image
- `WORKDIR` – Munka könyvtár
- `COPY` – Fájlok másolása

Összefoglalás – Futtatás

Parancsok és indítás

- `RUN` – Parancsok futtatása
- `ENV` – Környezeti változók
- `EXPOSE` – Port deklarálás

Indítás

- `CMD` – Indítási parancs

Összefoglalás – Optimalizálás

Cache és méret optimalizálás

- **Layer caching**: függőségek előre, kód hátra
- **Multi-stage build**: sokkal kisebb production image
- **.dockerignore**: tiszta build context

Összefoglalás – Biztonság

Biztonság és monitorozás

- **Alpine**: kompakt alap image, kisebb méret
- **USER**: ne root-ként futtass (biztonság)
- **HEALTHCHECK**: alkalmazás állapot monitorozás

Következő lépés

Docker Image-ek kezelése, majd Docker Compose

Mi a Docker image?

Image = futtatási sablon

- Csak olvasható sablon egy konténer létrehozásához
- Tartalmazza az alkalmazás kódját és a függőségeket
- Rétegekből (layers) épül fel

Image – Jellemzők

Tulajdonságok

- Egy image-ből több konténer is indulhat
- Immutable – nem módosítható közvetlenül

Analógia

Az image olyan, mint egy telepítő ISO, a konténer pedig a futó rendszer.

Image felépítése – rétegek

Layer alapú működés

- Minden Dockerfile utasítás új réteget hozhat létre
- A rétegek újrahasznosíthatók több image között
- Kisebb tárhelyigény és gyorsabb build a cache miatt

Image rétegek – Konténer indítás

Konténer indulásakor

- Egy írható réteg kerül az image fölé
- Az image rétegek változatlanok maradnak

Miért fontos?

A jó Dockerfile-sorrend jelentősen csökkenti a build időt.

Image azonosítás: név és tag

Formátum

```
[registry/]repository:tag
```

Példák

```
nginx:1.27
node:20-alpine
postgres:16
myrepo/my-api:1.0.0
ghcr.io/acme/payment-service:2026-02-15
```

Best practice

Production környezetben ne használd a `latest` taget.

Image letöltése – docker pull

Pull parancs

```
# Legfrissebb tag letöltése
docker pull nginx

# Konkrét verzió
docker pull nginx:1.27

# Privát registry
docker pull ghcr.io/acme/my-api:1.0.0
```

Image letöltése – Tudnivalók

Automatikus letöltés

- Ha a `docker run` során nincs helyben az image, automatikusan pull történik
- Több réteg párhuzamosan töltődik le

Saját image építése – docker build

Alap build

```
# Aktuális mappából build
docker build -t my-api:1.0.0 .

# Több tag egyszerre
docker build -t my-api:1.0.0 -t my-api:latest .
```

Saját image építése – Haladó

Haladó opciók

```
# Másik Dockerfile használata
docker build -f Dockerfile.prod -t my-api:prod .
```

Tipp

A build context legyen kicsi (`.dockerignore` használata kötelezően ajánlott).

Image lista és információk

Gyakori parancsok

```
# Lokális image-ek listája
docker images

# Csak azonosítók
docker images -q

# Részletes adatok JSON-ben
docker image inspect my-api:1.0.0
```

Image történet

Réteg történet

```
# Réteg történet
docker history my-api:1.0.0
```

Image publikálása – docker push

Folyamat

1. Registry-be bejelentkezés: `docker login`
2. Image tag-elése cél repository-ra
3. Feltöltés `docker push`-sal

Image publikálása – Példa

Példa Docker Hub-ra

```
docker login
docker tag my-api:1.0.0 myuser/my-api:1.0.0
docker push myuser/my-api:1.0.0
```

Image mentés és visszatöltés

Offline terjesztés

```
# Image mentése tar fájlba
docker save -o my-api-1.0.0.tar my-api:1.0.0

# Visszatöltés
docker load -i my-api-1.0.0.tar
```

Mikor hasznos?

Zárt hálózati környezetben, ahol nincs közvetlen registry elérés.

Image takarítás

Felesleges image-ek eltávolítása

```
# Egy image törlése
docker rmi my-api:1.0.0

# Címke nélküli (dangling) image-ek törlése
docker image prune

# Minden nem használt image törlése
docker image prune -a
```

Figyelem

A prune műveletek törlése nem visszavonható.

Best practices – Verzió és méret

Ajánlott gyakorlatok

- Használj fix verzió tag-eket (1.0.0, 2026-02-15)
- Tartsd kicsiben az image méretét (alpine, multi-stage build)
- Kerüld az érzékeny adatok beégetését image-be

Best practices – Build és deploy

Ajánlott gyakorlatok

- Építs reprodukálható módon (determinista build)
- CI-ben automatikusan címkézz és push-olj

Összefoglaló

Jó image stratégia = gyors build, gyors deploy, kevesebb hiba.

Mi a konténer?

Container – Futó alkalmazás példány

- Image-ből létrehozott, izolált futtatási környezet
- A konténer **futó folyamat** a host gépen
- Saját fájlrendszer, hálózat, folyamat tér

Konténer – Jellemzők

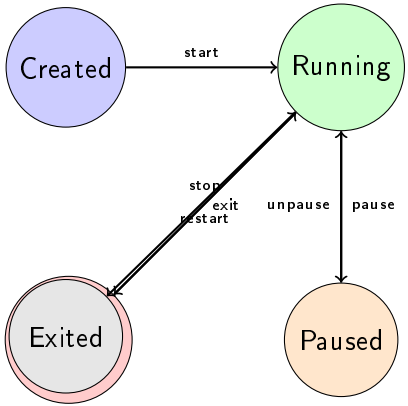
Tulajdonságok

- Könnyűsúlyú, gyorsan indul és áll le
- Egy Image-ből több konténer is indítható párhuzamosan

Analógia

Ha az Image egy **program** (pl. exe fájl), akkor a Container egy **futó folyamat** ebből a programból.

Konténer életciklus állapotok



Konténer állapotok – Magyarázat

- **Running**: Aktívan fut, CPU és memória használ
- **Stopped/Exited**: Leállítva, de nem törölve
- **Paused**: Folyamatok befagyasztva

Konténer indítása – docker run

Alap parancs

```
docker run <image-nev>
```

Egyszerű példa

```
# Nginx webszerver indítása
docker run nginx

# Háttérben futtatás (-d = detached)
docker run -d nginx

# Név adása a konténernek
docker run --name my-nginx nginx
```

Konténer indítása – Megjegyzés

Megjegyzés

Ha az image nincs helyben, Docker automatikusan letölti a Docker Hub-ról.

Port mapping – Miért kell?

Miért kell?

- Konténer saját hálózati térben fut
- Host gépen elérhetetlenek a portok alapértelmezetten
- Port mapping: host port → konténer port

Port mapping – Használat

-p flag használata

```
# Szintaxis: -p <host-port>:<container-port>
docker run -d -p 8080:80 nginx

# Most elérhető: http://localhost:8080
```

Port mapping – Több port

Több port mapping

```
# Több port mapping
docker run -d -p 3000:3000 -p 5000:5000 my-app
```

Példa

-p 8080:80 ⇒ Host 8080-as port ⇒ Konténer 80-as port

Környezeti változók – Alapok

-e flag használata

```
# Egy változó
docker run -e NODE_ENV=production my-app

# Több változó
docker run -e DB_HOST=localhost \
           -e DB_PORT=5432 \
           -e DB_NAME=mydb \
           my-app
```

Környezeti változók – Fájlból

.env fájlból

```
# .env fájlból
docker run --env-file .env my-app
```

Környezeti változók – PostgreSQL példa

PostgreSQL példa

```
docker run -d \
  -e POSTGRES_USER=admin \
  -e POSTGRES_PASSWORD=secret123 \
  -e POSTGRES_DB=myapp \
  postgres:15
```

Volume mounting – Miért kell?

Miért kell volume?

- Konténer törléskor az adatok elvesznek
- Volume: perzisztens tárolás a host gépen
- Megosztható több konténer között

Volume mounting – Named volume

Named volume

```
# Named volume
docker run -v mydata:/var/lib/postgresql/data \
  postgres
```

Volume mounting – Bind mount

Bind mount (host útvonal)

```
# Bind mount
docker run -v /host/path:/container/path my-app

# Windows példa
docker run -v C:\projects\app:/usr/src/app node-app

# Csak olvasható
docker run -v mydata:/data:ro my-app
```

Teljes példa – Node.js alkalmazás

Összes opció együtt

```
docker run -d \
  --name my-node-app \
  -p 3000:3000 \
  -e NODE_ENV=production \
  -e DB_HOST=postgres \
  -v $(pwd)/src:/usr/src/app/src \
  --restart unless-stopped \
  node:18-alpine \
  npm start
```

Teljes példa – Paraméterek

Opciók részletesen

- `-d`: Detached módban fut (háttérben)
- `-name`: Konténer neve (my-node-app)
- `-p 3000:3000`: Port mapping (host:container)

Teljes példa – Paraméterek folyt.

Opciók részletesen

- `-e`: Környezeti változók beállítása
- `-v`: Volume mounting (perzisztens adatok)
- `-restart`: Automatikus újraindítás

Konténerek listázása – docker ps

Futó konténerek

```
# Csak futó konténerek
docker ps

# Összes konténer (leállítottak is)
docker ps -a

# Csak ID-k megjelenítése
docker ps -q
```

Konténerek listázása – Formázás

Formázott kimenet

```
docker ps --format \
  "table␣{{.Names}}\t{{.Status}}\t{{.Ports}}"
```

Konténer kezelő parancsok – Leállítás

Leállítás és újraindítás

```
# Leállítás (graceful)
docker stop <container-id>

# Erőszakos leállítás
docker kill <container-id>

# Újraindítás
docker restart <container-id>
```

Konténer kezelő parancsok – Szünet és törlés

Szüneteltetés és törlés

```
# Szüneteltetés
docker pause <container-id>
docker unpause <container-id>

# Törlés (csak leállított konténer)
docker rm <container-id>

# Futó konténer törlése
docker rm -f <container-id>
```

Interaktív mód – Flagek

-it flagek

- `-i`: Interactive (interaktív)
- `-t`: TTY (pseudo-terminal)

Interaktív mód – Példák

Bash shell indítása konténerben

```
# Ubuntu konténer bash-el
docker run -it ubuntu bash

# Node.js REPL
docker run -it node:18

# Alpine Linux shell
docker run -it alpine sh
```

Kilépés

`exit` parancs vagy `Ctrl+D` ⇒ konténer leáll

`Ctrl+P`, `Ctrl+Q` ⇒ konténer fut tovább (detach)

docker exec – Parancs futtatása

docker exec használata

```
# Shell indítása futó konténerben
docker exec -it <container-id> bash

# Parancs futtatása
docker exec <container-id> ls -la /app

# Root userként
docker exec -u root -it <container-id> bash
```

docker exec – Gyakorlati példák

Gyakori használat

```
# PostgreSQL konzol
docker exec -it postgres-container \
  psql -U admin -d mydb

# NPM parancs futtatása
docker exec my-app npm install axios
```

Logok – docker logs

Alap log parancsok

```
# Összes log
docker logs <container-id>

# Utolsó 50 sor
docker logs --tail 50 <container-id>

# Követés (folyamatos, mint tail -f)
docker logs -f <container-id>
```

Logok – Szűrés

Szűrt logok

```
# Időbélyegekkel
docker logs -t <container-id>

# Utolsó 10 perc
docker logs --since 10m <container-id>
```

Tipp

Fejlesztéskor: `docker logs -f` ⇒ valós idejű log követés

docker inspect – Részletes info

docker inspect

```
# Teljes JSON konfiguráció
docker inspect <container-id>

# Specifikus mező (pl. IP cím)
docker inspect -f \
  '{{.NetworkSettings.IPAddress}}' <container-id>
```

docker stats – Erőforrás használat

docker stats

```
# Valós idejű statisztika (CPU, memória, hálózat)
docker stats

# Csak egy konténer
docker stats <container-id>
```

Erőforrás korlátok – Memória

Memória korlátozás

```
# Memória limit (256MB)
docker run -m 256m nginx

# Memória swap limit
docker run -m 256m --memory-swap 512m my-app
```

Erőforrás korlátok – CPU

CPU korlátozás

```
# CPU limit (50% of 1 CPU)
docker run --cpus="0.5" nginx

# Kombináció
docker run -m 512m --cpus="1.0" my-app
```

Miért fontos?

Megakadályozza, hogy egy konténer minden erőforrást elfogyasszon.
Production környezetben kötelező!

Restart policies – Opciók

–restart flag opciók

- `no`: Nem indul újra (alapértelmezett)
- `always`: Mindig újraindul
- `on-failure`: Csak hiba esetén

Restart policies – Használat

Használat

```
# Mindig újraindul (host reboot után is)
docker run -d --restart always nginx

# Csak hiba esetén, max 5 próbálkozás
docker run -d --restart on-failure:5 my-app

# Production ajánlott
docker run -d --restart unless-stopped my-app
```

Fájlok másolása – docker cp

Host és konténer között

```
# Host -> Konténer
docker cp /host/file.txt container-id:/path/

# Konténer -> Host
docker cp container-id:/app/logs/app.log ./logs/
```

Fájlok másolása – Példák

Gyakorlati példa

```
# Konfigurációs fájl feltöltés
docker cp config.json \
  nginx:/etc/nginx/config.json

# Log fájl letöltés elemzéshez
docker cp my-app:/var/log/app.log ./logs/
```

Tisztítás – Konténer törlés

Leállított konténerek törlése

```
# Egy konténer törlése
docker rm <container-id>

# Összes leállított konténer
docker container prune

# Futó konténer törlése
docker rm -f <container-id>
```

Tisztítás – Tömeges törlés

Tömeges műveletek

```
# Minden konténer törlése (VIGYÁZAT!)
docker rm -f $(docker ps -aq)
```

Figyelem

A prune parancsok visszavonhatatlanul törölnek!
Production környezetben óvatosan használd!

Hasznos egy-sorosok – Alapok

Gyakori feladatok

```
# Összes futó konténer leállítása
docker stop $(docker ps -q)

# Konténer IP címének lekérése
docker inspect -f \
  '{{range.NetworkSettings.Networks}}
␣␣{{.IPAddress}}{{end}}' <container>
```

Hasznos egy-sorosok – Haladó

Haladó parancsok

```
# Legutóbbi konténer logjai
docker logs -f $(docker ps -lq)

# Konténer shell gyors elérés
docker exec -it \
  $(docker ps -q --filter name=my-app) bash
```

Összefoglalás – Létrehozás

Létrehozás és futtatás

```
docker run -d -p 8080:80 -name web nginx
```

Kezelés

- `docker ps`: Listázás
- `docker logs -f`: Logok
- `docker exec -it <id> bash`: Shell

Összefoglalás – Fontos opciók

Fontos opciók

- `-p`: Port mapping (hozzáférés)
- `-v`: Volume (perzisztencia)
- `-e`: Env változók (konfiguráció)

Megbízhatóság

- `-restart`: Újraindítás (megbízhatóság)
- `-m / -cpus`: Erőforrás korlátok

Best Practices – Konfiguráció

Ajánlott gyakorlatok

- Mindig adj **nevet** a konténernek (`-name`)
- Használj **restart policy**-t production-ben
- Állíts be **resource limits**-et (CPU, memória)

Best Practices – Adatok és biztonság

Ajánlott gyakorlatok

- Használj **volume**-ot perzisztens adatokhoz
- Ne futtass **root user**-ként
- **Logokat** naplózd és monitorozd

Best Practices – Kerülendő

Kerülendő

- Érzékeny adatok környezeti változóban (használj secrets-et)
- Túl sok folyamat egy konténerben
- Adatok tárolása konténerben (volume nélkül)

Mi a Docker Compose?

Docker Compose – Több konténer kezelése

- Eszköz **több konténerből álló** alkalmazások definiálására
- YAML formátumú konfigurációs fájl (`docker-compose.yml`)
- Egyetlen parancs: teljes alkalmazás indítása/leállítása

Docker Compose – Használati eset

Példa használati eset

Node.js backend + PostgreSQL adatbázis + Redis cache
⇒ 3 konténer, 1 docker-compose.yml, 1 parancs: `docker compose up`

Miért? – Docker Compose nélkül

Docker Compose nélkül

```
docker network create myapp
docker run -d --name postgres \
  --network myapp \
  -e POSTGRES_PASSWORD=secret \
  postgres:15
docker run -d --name redis \
  --network myapp redis:7
docker run -d --name backend \
  --network myapp \
  -p 3000:3000 my-backend
```

Miért? – A probléma

Probléma

Bonyolult, sok parancs, nehezen karbantartható

Miért? – Docker Compose-zal

Docker Compose-zal

```
docker compose up
```

Előnyök

- Egyetlen parancs!
- Minden konfiguráció a `docker-compose.yml`-ben
- Reprodukálható és verziókezelhető

Miért? – Csapatmunka

Csapatmunka előnyök

- Verziókezelhető (Git)
- Csapatmunkában könnyen megosztható
- Konzisztens környezet minden fejlesztőnél

Docker Compose telepítés

Modern Docker verzió

Docker Desktop és újabb Docker Engine verziók tartalmazzák:

```
# Ellenőrzés
docker compose version
# Docker Compose version v2.24.0
```

Docker Compose – Régi vs új szintaxis

Régi vs új szintaxis

- Régi (v1): `docker-compose up` (kötőjellel, külön telepítés)
- Új (v2): `docker compose up` (szóköz, built-in)

Ajánlott: v2 (docker compose)

Első docker-compose.yml – Kód

Egyszerű webszerver példa

```
version: '3.8'

services:
  web:
    image: nginx:alpine
    ports:
      - "8080:80"
```

Első docker-compose.yml – Parancsok

Indítás és leállítás

```
# Indítás (háttérben)
docker compose up -d

# Leállítás és törlés
docker compose down
```

docker-compose.yml struktúra – Kód

Fő elemek

```
version: '3.8'  # Compose fájl verzió

services:       # Konténerek definíciója
  service1:
    # konfiguráció
  service2:
    # konfiguráció

volumes:        # Named volume-ok (opcionális)
  data:

networks:       # Egyedi hálózatok (opcionális)
  backend:
```

docker-compose.yml struktúra – Magyarázat

Elemek

- `version`: Compose fájl formátum verzió
- `services`: Konténerek (ezeket indítja)
- `volumes`: Megosztott perzisztens tárolók

docker-compose.yml struktúra – Networks

Hálózatok

- `networks`: Hálózati konfiguráció
- Compose automatikusan létrehoz egy default hálózatot

Service definíció – image alapú

Existing image használata

```
services:
  database:
    image: postgres:15-alpine
    environment:
      POSTGRES_DB: myapp
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: secret123
    ports:
      - "5432:5432"
    volumes:
      - postgres-data:/var/lib/postgresql/data
    restart: unless-stopped

volumes:
  postgres-data:
```

Service definíció – build alapú

Saját Dockerfile build-elése

```
services:
  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    environment:
      NODE_ENV: production
      DB_HOST: database
    depends_on:
      - database
    restart: unless-stopped
```

Service definíció – build magyarázat

Build elemek

- `build`: Dockerfile-ból épít
- `context`: Build context útvonal
- `depends_on`: Függőségek (ez után indul)

Environment variables – Inline

Inline környezeti változók

```
services:
  app:
    environment:
      NODE_ENV: production
      PORT: 3000
```

Environment variables – Fájlból

.env fájlból

```
services:
  app:
    env_file:
      - .env
      - .env.local
```

Environment variables – Host változók

Host environment változók

```
services:
  app:
    environment:
      # .env fájlból vagy host-ról
      DB_PASSWORD: ${DB_PASSWORD}
```

Volumes – Named volumes

Named volumes (ajánlott)

```
services:
  db:
    image: postgres:15
    volumes:
      - db-data:/var/lib/postgresql/data

volumes:
  db-data:
```

Volumes – Bind mounts

Bind mounts (fejlesztéshez)

```
services:
  app:
    volumes:
      # Relatív útvonal
      - ./src:/app/src
      # Abszolút útvonal
      - /host/path:/container/path
```

Networks – Automatikus

Automatikus hálózat

```
services:
  backend:
    image: my-app
  database:
    image: postgres
```

Compose automatikusan létrehoz egy hálózatot!
backend eléri database-t a név alapján.

Networks – Egyedi hálózatok

Egyedi hálózatok

```
services:
  backend:
    networks:
      - frontend
      - backend
  database:
    networks:
      - backend

networks:
  frontend:
  backend:
```

MERN stack – Adatbázisok

MongoDB és Redis

```
version: '3.8'

services:
  mongodb:
    image: mongo:7
    volumes:
      - mongo-data:/data/db
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: secret
    restart: unless-stopped
```

MERN stack – Redis és volumes

Redis és adattárolás

```
services:
  redis:
    image: redis:7-alpine
    restart: unless-stopped

volumes:
  mongo-data:
```

MERN stack – Adatbázis magyarázat

Adattárolás

- **MongoDB**: perzisztens volume (mongo-data)
- **Redis**: cache, nincs szükség volume-ra
- **Restart policy**: biztosítja a rendelkezésre állást

MERN stack – Backend

Backend szolgáltatás

```
services:
  backend:
    build: ./backend
    ports:
      - "3001:3001"
    environment:
      MONGO_URL: mongodb://admin:secret@mongodb:27017
      REDIS_URL: redis://redis:6379
    depends_on:
      - mongodb
      - redis
    restart: unless-stopped
```

MERN stack – Frontend

Frontend szolgáltatás

```
services:
  frontend:
    build: ./frontend
    ports:
      - "3000:3000"
    environment:
      REACT_APP_API_URL: http://localhost:3001
    depends_on:
      - backend
    restart: unless-stopped
```

depends_on – Alap használat

Alapvető függőség

```
services:
  backend:
    depends_on:
      - database
  database:
    image: postgres
```

`database` indul először, **de nem vár** amíg kész!

depends_on – Figyelmeztetés

Fontos

depends_on csak az indítási sorrendet szabályozza,
NEM várja meg amíg a szolgáltatás tényleg készen áll!

Megoldás

- Alkalmazásban: retry logika
- Healthcheck használata
- wait-for-it.sh script

Healthcheck a Compose-ban

Szolgáltatás készenlét ellenőrzés

```
services:
  database:
    image: postgres:15
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 10s
      timeout: 5s
      retries: 5

  backend:
    depends_on:
      database:
        condition: service_healthy
```

Healthcheck – Magyarázat

Healthcheck elemek

- `condition: service_healthy`: Vár amíg healthcheck OK
- `test`: Ellenőrző parancs
- `interval`: Ellenőrzések gyakorisága

Compose parancsok – Indítás

Indítás

```
# Indítás (előtérben, logokkal)
docker compose up

# Indítás háttérben
docker compose up -d

# Rebuild és indítás
docker compose up --build
```

Compose parancsok – Leállítás

Leállítás

```
# Leállítás (konténerek megmaradnak)
docker compose stop

# Leállítás és törlés
docker compose down

# Törlés volume-okkal együtt
docker compose down -v
```

Compose parancsok – Státusz

Státusz és logok

```
# Futó szolgáltatások listája
docker compose ps

# Összes szolgáltatás (leállítottak is)
docker compose ps -a

# Logok megjelenítése
docker compose logs
```

Compose parancsok – Log szűrés

Log szűrés

```
# Követés (egyik szolgáltatás)
docker compose logs -f backend

# Utolsó 100 sor
docker compose logs --tail=100
```

Exec és run – Futó konténerben

Parancs futtatása futó szolgáltatásban

```
# Shell futó konténerben
docker compose exec backend sh

# PostgreSQL konzol
docker compose exec database psql -U admin -d myapp
```

Exec és run – Egyszeri parancs

Egyszeri parancs (új konténer)

```
# Database migráció
docker compose run backend npm run migrate

# Teszt futtatás
docker compose run --rm backend npm test
```

-rm: Konténer törlése futás után

Scale – Több példány

Szolgáltatás skálázása

```
# 3 backend példány indítása
docker compose up -d --scale backend=3
```

Scale – Port konfliktus

Port konfliktus

```
services:
  backend:
    ports:
      - "3000:3000"  # Nem skálázható!
```

Megoldás: dinamikus port

```
    ports:
      - "3000-3002:3000"  # 3 példány
```

Override fájlok – Dev config

docker-compose.override.yml

```
# docker-compose.yml (base)
services:
  app:
    image: my-app

# docker-compose.override.yml (dev)
services:
  app:
    volumes:
      - ./src:/app/src  # live reload
    environment:
      DEBUG: "true"
```

Automatikusan egyesül a base-zel fejlesztéskor!

Override fájlok – Production

docker-compose.prod.yml

```
services:
  app:
    image: my-app:1.0.0
    restart: always
    deploy:
      replicas: 3
      resources:
        limits:
          cpus: '0.5'
          memory: 512M
```

Override fájlok – Használat

Használat

```
# Specifikus fájl megadása
docker compose -f docker-compose.yml \
  -f docker-compose.prod.yml up -d
```

Secrets kezelés – .env fájl

Érzékeny adatok .env fájlban

```
# .env fájl (ne commitold!)
POSTGRES_PASSWORD=supersecret123
API_KEY=abc123xyz
JWT_SECRET=random_secret_key
```

Secrets kezelés – Compose-ban

Használat docker-compose.yml-ben

```
services:
  database:
    environment:
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
  backend:
    env_file:
      - .env
```

Biztonság

`.env` legyen a `.gitignore`-ban!

Használj `.env.example` sablont repository-ban!

Profiles – Opcionális szolgáltatások

Fejlesztői eszközök csak manuális indításkor

```
services:
  app:
    image: my-app
  database:
    image: postgres
  adminer:
    image: adminer
    profiles:
      - debug
    ports:
      - "8080:8080"
```

Profiles – Használat

Használat

```
# Normál indítás: app + database
docker compose up

# Debug profillal: app + database + adminer
docker compose --profile debug up
```

Resource limits

CPU és memória korlátok

```
services:
  backend:
    image: my-app
    deploy:
      resources:
        limits:
          cpus: '1.0'
          memory: 1G
        reservations:
          cpus: '0.5'
          memory: 512M
```

Logging konfiguráció

Log beállítások

```
services:
  backend:
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"
```

Driverek

`json-file`, `syslog`, `journald`, `awslogs`

Teljes projekt – PostgreSQL

PostgreSQL szolgáltatás

```
version: '3.8'

services:
  postgres:
    image: postgres:15-alpine
    environment:
      POSTGRES_DB: ${DB_NAME}
      POSTGRES_USER: ${DB_USER}
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    volumes:
      - postgres-data:/var/lib/postgresql/data
    restart: unless-stopped
```

Teljes projekt – PostgreSQL healthcheck

Healthcheck hozzáadása

```
services:
  postgres:
    healthcheck:
      test: ["CMD-SHELL",
        "pg_isready -U ${DB_USER}"]
      interval: 5s
      timeout: 5s
      retries: 5
```

Teljes projekt – Redis és volumes

Redis és adattárolás

```
services:
  redis:
    image: redis:7-alpine
    restart: unless-stopped

volumes:
  postgres-data:
```

Teljes projekt – Backend

Backend hotreload-dal

```
services:
  backend:
    build: ./backend
    ports:
      - "3001:3001"
    environment:
      DATABASE_URL: postgresql://${DB_USER}:
        ${DB_PASSWORD}@postgres:5432/${DB_NAME}
      REDIS_URL: redis://redis:6379
    volumes:
      - ./backend/src:/app/src
    depends_on:
      postgres:
        condition: service_healthy
    restart: unless-stopped
```

Teljes projekt – Frontend

React frontend hotreload-dal

```
services:
  frontend:
    build: ./frontend
    ports:
      - "3000:3000"
    environment:
      REACT_APP_API_URL: http://localhost:3001
    volumes:
      - ./frontend/src:/app/src
      - /app/node_modules
    depends_on:
      - backend
    restart: unless-stopped
```

Teljes projekt – Frontend magyarázat

Fejlesztői környezet

- Volume mount: kód változások azonnal láthatóak
- node_modules izolált a konténerben
- Restart policy védelem összeomlás ellen

Compose vs Orchestration – Compose

Mikor használj Docker Compose-t?

- Fejlesztési környezet
- Teszt környezet
- Kisebb produkciós alkalmazások (1-2 szerver)

Compose vs Orchestration – Kubernetes

Mikor kell Kubernetes vagy Docker Swarm?

- Nagy léptékű production (több szerver)
- Magas rendelkezésre állás (HA)
- Automatikus failover és auto-scaling

Fontos

Kezdd Compose-zal, csak akkor válts orchestration-re, ha tényleg szükséges!

Best Practices – Konfiguráció

Ajánlott gyakorlatok

- **Verziókezeld** a `docker-compose.yml-t`
- **.env fájl** érzékeny adatokhoz (.gitignore!)
- **Named volumes** adatok perzisztálásához

Best Practices – Megbízhatóság

Ajánlott gyakorlatok

- **Healthcheck** használata depends_on-nál
- **Restart policy**: unless-stopped production-ben
- **Resource limits** beállítása

Best Practices – Build

Ajánlott gyakorlatok

- **Specifikus image verziók** (ne latest)
- **Multi-stage Dockerfile** kisebb image-ekhez
- **Service nevek** beszédesek legyenek

Anti-patterns – Kerülendő

Ne csináld

- Érzékeny adatok közvetlenül a YAML-ban
- `latest` tag production-ben
- Adatok a konténerben (volume nélkül)

Anti-patterns – Kerülendő folyt.

Ne csináld

- Egy konténerben több alkalmazás
- Hardcoded portok ha skálázni akarsz
- Depends_on healthcheck nélkül

Összefoglalás – Lényeg

Docker Compose = Egyszerű multi-container kezelés

- YAML fájl: teljes alkalmazás stack leírása
- Egy parancs: `docker compose up`
- Automatikus hálózat, volume, service discovery

Összefoglalás – Előnyök

Előnyök

- Gyors setup – másodpercek alatt
- Konzisztens környezet minden fejlesztőnél
- Reprodukálható, verziókezelhető

Összefoglalás – Kulcs koncepciók

Fő elemek

- `services`: Mi fusson (konténerek definíciója)
- `volumes`: Adatok perzisztálása
- `networks`: Szolgáltatások összekapcsolása

Következő lépés

Docker hálózatok részletesen

Docker hálózatok – Bevezetés

Miért fontosak a Docker hálózatok?

- Konténerek közötti kommunikáció
- Szolgáltatások izolálása
- Biztonságos konténer-architektúra

Probléma hálózatok nélkül

Mi történik hálózatok nélkül?

- Konténerek nem érik el egymást
- Minden port-ot a host-ra kell publikálni
- Nincs belső szolgáltatás-felderítés (DNS)

Megoldás – Docker hálózatok

Docker hálózatokkal

- Konténerek név alapján érik el egymást
- Belső kommunikáció port-publikálás nélkül
- Automatikus DNS feloldás

Hálózati driverek – Áttekintés

Docker hálózati driverek

- `bridge`: Alapértelmezett, izolált hálózat
- `host`: Host hálózat közvetlen használata
- `none`: Nincs hálózat

Hálózati driverek – További

Haladó driverek

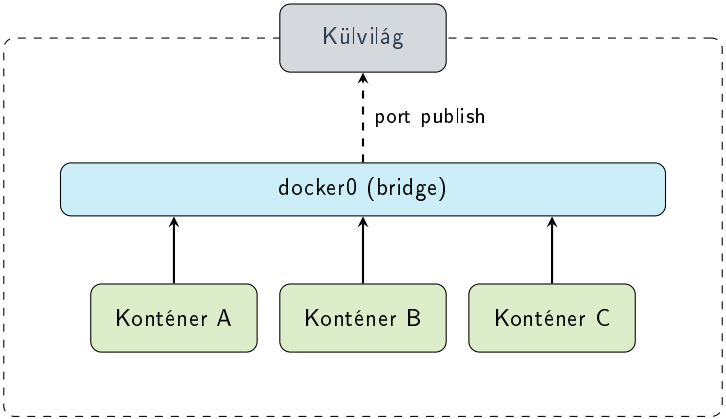
- `overlay`: Több Docker host közötti hálózat (Swarm)
- `macvlan`: Saját MAC cím, fizikai hálózatba illeszkedik

Bridge hálózat – Mi az?

Bridge driver (alapértelmezett)

- Szoftveres hálózati híd a host-on
- Konténerek egymást elérik a bridge-en
- Külvilág felé port-publikálás szükséges

Bridge architektúra



Hálózat parancsok – Listázás

Hálózatok listázása

```
# Összes hálózat
docker network ls

# NETWORK ID    NAME        DRIVER     SCOPE
# abc123        bridge      bridge     local
# def456        host        host       local
# ghi789        none        null       local
```

Hálózat parancsok – Létrehozás

Hálózat létrehozása

```
# Egyszerű bridge hálózat
docker network create mynetwork

# Megadott driver-rel
docker network create --driver bridge mynetwork

# Megadott subnet-tel
docker network create --subnet=172.20.0.0/16 mynetwork
```

Hálózat parancsok – Törlés

Hálózat törlése és tisztítás

```
# Egy hálózat törlése
docker network rm mynetwork

# Nem használt hálózatok törlése
docker network prune

# Részletes információk
docker network inspect mynetwork
```

Konténer csatlakoztatás – Indításkor

Hálózat megadása indításkor

```
# Konténer indítása adott hálózaton
docker run -d --name backend \
  --network mynetwork my-app

# Több hálózat: indítás után csatlakoztatás
docker network connect frontend-net backend
```

Konténer csatlakoztatás – Utólag

Utólagos csatlakoztatás / leválasztás

```
# Futó konténer csatlakoztatása hálózathoz
docker network connect mynetwork mycontainer

# Leválasztás hálózatról
docker network disconnect mynetwork mycontainer
```

Default bridge vs User-defined

Default bridge hálózat

- **Nincs** automatikus DNS feloldás
- Konténerek csak IP-vel érik el egymást
- IP cím változhat újraindításkor

User-defined bridge – Előnyök

User-defined bridge hálózat

- **Van** automatikus DNS feloldás
- Konténerek **név alapján** érik el egymást
- Jobb izoláció, automatikus link

Ajánlás

Mindig user-defined bridge-et használj!

DNS és Service Discovery – Kód

Automatikus névfeloldás

```
# Hálózat létrehozása
docker network create app-net

# Adatbázis indítása
docker run -d --name database \
  --network app-net postgres:15

# Backend indítása
docker run -d --name backend \
  --network app-net \
  -e DB_HOST=database \
  my-backend
```

DNS és Service Discovery – Magyarázat

Hogyan működik?

- `backend` eléri `database`-t a nevével
- Docker beépített DNS szerver: `127.0.0.11`
- Konténer név = hostname a hálózaton

Fullstack példa – Hálózat létrehozás

Hálózat előkészítése

```
# Hálózat létrehozása
docker network create fullstack-net
```

Fullstack példa – Adatbázis

PostgreSQL indítása

```
docker run -d --name postgres \
  --network fullstack-net \
  -e POSTGRES_PASSWORD=secret \
  -e POSTGRES_DB=myapp \
  -v pgdata:/var/lib/postgresql/data \
  postgres:15-alpine
```

Fullstack példa – Backend

Node.js backend

```
docker run -d --name backend \
  --network fullstack-net \
  -p 3001:3001 \
  -e DATABASE_URL=postgresql://postgres:secret
    @postgres:5432/myapp \
  my-backend
```

Fullstack példa – Frontend

React frontend

```
docker run -d --name frontend \
  --network fullstack-net \
  -p 3000:3000 \
  my-frontend
```

Fullstack példa – Magyarázat

Kommunikáció

- Backend eléri az adatbázist: `postgres:5432`
- Frontend eléri a backend-et: `backend:3001`
- Külvilág eléri: `localhost:3000` és `:3001`

Port publishing vs belső hálózat

Mikor kell port publishing?

Csak ha a **host-ról** vagy **külvilágból** kell elérni!

Belső hálózat

- Adatbázis: **NEM** kell port publish
- Backend API: kell, ha host-ról is elérik
- Frontend: kell, mert böngészőből érik el

Port példa – Helyes konfiguráció

Best practice

```
services:
  database:
    image: postgres   # Nincs ports: belső!
    networks:
      - backend

  api:
    ports:
      - "3001:3001"   # Kívülről is elérhető
    networks:
      - backend
      - frontend
```

Host hálózat – Mi az?

Host driver

- Konténer közvetlenül a host hálózatát használja
- Nincs port mapping szükség
- Nincs hálózati izoláció

Host hálózat – Használat

Parancs

```
docker run -d --network host nginx
# nginx elérhető: localhost:80 (port mapping nélkül)
```

Host hálózat – Előnyök és hátrányok

Előnyök

- Jobb hálózati teljesítmény
- Egyszerűbb port-kezelés

Hátrányok

- Nincs izoláció
- Port ütközések lehetségesek
- Csak Linuxon működik jól

None hálózat – Mi az?

None driver

- Teljesen izolált konténer
- Nincs hálózati hozzáférés
- Batch feldolgozáshoz, biztonsági feladatokhoz

None hálózat – Használat

Parancs

```
docker run -d --network none my-batch-processor
```

Network inspect

Hálózat részletes adatai

```
docker network inspect mynetwork
```

Network inspect – Kimenet

Kimenet részlet

```
{
  "Name": "mynetwork",
  "Driver": "bridge",
  "Subnet": "172.20.0.0/16",
  "Containers": {
    "abc123": {
      "Name": "backend",
      "IPv4Address": "172.20.0.2/16"
    }
  }
}
```

Egyedi subnet és IP

Subnet és fix IP beállítás

```
# Hálózat egyedi subnet-tel
docker network create \
  --subnet=172.28.0.0/16 \
  --gateway=172.28.0.1 \
  custom-net

# Konténer fix IP-vel
docker run -d --name db \
  --network custom-net \
  --ip 172.28.0.10 \
  postgres:15
```

Egyedi subnet – Mikor használjuk?

Mikor hasznos a fix IP?

- Tűzfal szabályokhoz
- Legacy alkalmazásokhoz
- Speciális hálózati konfigurációkhoz

Ajánlás

Lehetőleg DNS neveket használj fix IP-k helyett!

Docker Compose hálózatok – Automatikus

Compose automatikus hálózat

```
services:
  backend:
    image: my-app
  database:
    image: postgres
```

Compose automatikusan létrehozza: `<projektnév>_default`
Mindkét szolgáltatás rajta van, név alapján érik el egymást.

Docker Compose hálózatok – Egyedi

Egyedi hálózatok definiálása

```
services:
  frontend:
    networks:
      - frontend-net
  backend:
    networks:
      - frontend-net
      - backend-net
  database:
    networks:
      - backend-net

networks:
  frontend-net:
  backend-net:
```

Compose hálózatok – Magyarázat

Izoláció

- `frontend` ↔ `backend`: OK (frontend-net)
- `backend` ↔ `database`: OK (backend-net)
- `frontend` ↔ `database`: **NEM** megy!

Biztonsági előny

Az adatbázis nem érhető el közvetlenül a frontendből.

Links – Régi módszer

Legacy: links

A `links` kulcsszó elavult!

- Régi Docker verziókban használták
- Helyette: user-defined networks
- Automatikus DNS feloldás jobb megoldás

Network alias

Alternatív nevek a hálózaton

```
docker run -d --name postgres \
  --network app-net \
  --network-alias db \
  --network-alias database \
  postgres:15
```

A konténer 3 néven érhető el: `postgres`, `db`, `database`

Network alias – Compose-ban

Compose alias

```
services:
  postgres:
    image: postgres:15
    networks:
      backend:
        aliases:
          - db
          - database

networks:
  backend:
```

Overlay hálózat

Overlay driver

- Több Docker host közötti hálózat
- Docker Swarm és Kubernetes használja
- Konténerek különböző gépeken kommunikálnak

Egyetemi keretek

Fejlesztéshez bridge hálózat elég, overlay csak produkciós klaszterben szükséges.

Troubleshooting – Alapparancsok

Hálózati hibakeresés

```
# Konténer hálózati beállításai
docker inspect --format \
  '{{json .NetworkSettings.Networks}}' backend

# Ping teszt konténerből
docker exec backend ping database

# DNS feloldás konténerből
docker exec backend nslookup database
```

Troubleshooting – Haladó

Haladó hibakeresés

```
# Hálózati konténer csatlakoztatás
docker run --rm -it --network mynetwork \
  nicolaka/netshoot

# Hálózati port ellenőrzés
docker exec backend curl -v database:5432
```

Gyakori problémák 1 – Nem éri el

Probléma: Konténer nem éri el a másikat

- **Ok**: Nem ugyanazon a hálózaton vannak
- **Megoldás**: `docker network connect` vagy közös hálózat

Gyakori problémák 2 – DNS nem működik

Probléma: DNS névfeloldás nem működik

- **Ok**: Default bridge-en nincs DNS
- **Megoldás**: User-defined bridge hálózat használata

Gyakori problémák 3 – Port ütközés

Probléma: Port already in use

- **Ok**: Host-on már foglalt a port
- **Megoldás**: Másik host port (`-p 3001:3000`)

Gyakori problémák 4 – Connection refused

Probléma: Connection refused

- **Ok**: Alkalmazás csak localhost-on figyel
- **Megoldás**: `0.0.0.0`-ra bind-olás

Biztonsági best practices 1

Hálózati biztonság

- Külön hálózat minden szolgáltatás-rétegnek
- Ne publikálj felesleges portokat
- Adatbázis soha nem érhető el kívülről

Biztonsági best practices 2

Hálózati biztonság – Fejlesztés

- Használj user-defined bridge-et
- Network alias a rugalmasságért
- Prod-ban minimális port exposure

Komplex hálózat – 3-rétegű alkalmazás

Frontend réteg

```
version: '3.8'

services:
  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
    networks:
      - frontend
    depends_on:
      - api
```

Komplex hálózat – API réteg

API réteg

```
services:
  api:
    build: ./api
    networks:
      - frontend
      - backend
    depends_on:
      - postgres
      - redis
    environment:
      DB_HOST: postgres
      REDIS_HOST: redis
```

Komplex hálózat – Adatbázis réteg

Adatbázis réteg

```
services:
  postgres:
    image: postgres:15
    networks:
      - backend
    volumes:
      - pgdata:/var/lib/postgresql/data

  redis:
    image: redis:7-alpine
    networks:
      - backend
```

Komplex hálózat – Hálózat és volume definíciók

Hálózatok és volume-ok

```
networks:
  frontend:
    driver: bridge
  backend:
    driver: bridge

volumes:
  pgdata:
```

Komplex hálózat – Magyarázat

Biztonsági izoláció

- nginx ↔ api: frontend hálózat
- api ↔ postgres/redis: backend hálózat
- nginx ↔ postgres: **NINCS** közvetlen út

Összefoglalás – Lényeg

Docker hálózatok

- Konténerek közötti kommunikáció alapja
- Szolgáltatás-felderítés DNS-sel
- Biztonsági izoláció rétegekkel

Összefoglalás – Kulcs koncepciók

Fő elemek

- **Bridge**: alapértelmezett, legtöbb esetben elég
- **User-defined**: mindig ezt használd (DNS!)
- **Compose**: automatikus hálózat létrehozás

Összefoglalás – Aranyszabályok

Aranyszabályok

- Konténer név = hostname a hálózaton
- Csak szükséges portokat publikáld
- Külön hálózat = jobb biztonság