

# Docker

DELTA - Střední škola informatiky a ekonomie, s.r.o.

Ing. Luboš Zápotočný

08.01.2026

CC BY-NC-SA 4.0

# Úvod do Dockeru

# Co je Docker?

**Docker** je platforma pro **kontejnerizaci** aplikací

- Umožňuje zabalit aplikaci a její závislosti do jednoho **kontejneru**
- Kontejner běží **izolovaně** od hostitelského systému
- Zaručuje **konzistentní prostředí**  
„funguje mi to lokálně“ ≈ funguje všude
- Usnadňuje **nasazování** (deployment) aplikací

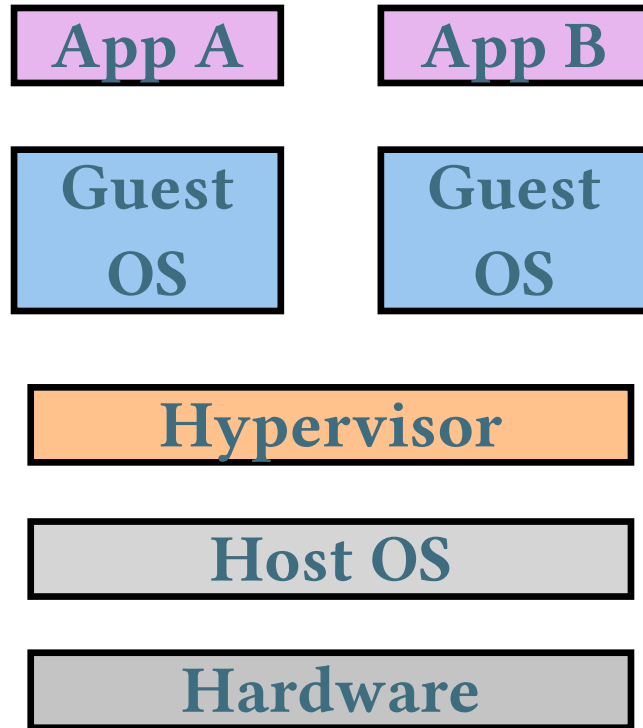
# Základní pojmy

- **Image** (obraz): šablona pro vytvoření kontejneru
- **Container** (kontejner): běžící instance image
- **Dockerfile**: soubor s instrukcemi pro sestavení image
- **Registry**: úložiště pro images (Docker Hub)
- **Volume**: persistentní úložiště dat
- **Network**: virtuální síť pro komunikaci mezi kontejnery

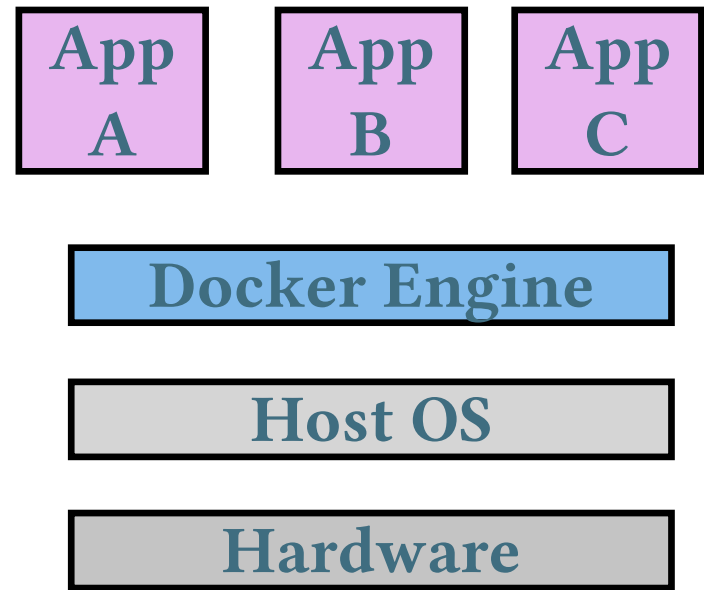
# Kontejner vs Virtuální stroj

Vlastnost	Kontejner	VM
Izolace	Procesová	Hardwarová
Velikost	MB	GB
Start	Sekundy	Minuty
Výkon	Nativní	Overhead
OS	Sdílené jádro	Vlastní OS

# Kontejner vs Virtuální stroj



Virtuální stroje



Kontejnery

# Server s Dockerem

Na serveru Coder ([coder.delta-topgun.cz](https://coder.delta-topgun.cz)) je možné spustit virtuální počítač s přeinstalovaným Dockerem

- Přihlášení pomocí školního Microsoft účtu

# Instalace Dockeru

**Docker Desktop** - GUI aplikace pro Windows a macOS

- Stáhnout na: [docker.com/products/docker-desktop](https://docker.com/products/docker-desktop)

**Docker Engine** - pro Linux

- Instalace: [docs.docker.com/engine/install/](https://docs.docker.com/engine/install/)

Ověření instalace:

```
docker --version
```

```
docker info
```

# **Základní příkazy**

# **docker run**

Příkaz `docker run` je základní příkaz pro spuštění kontejneru

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

# **docker run**

Nejčastější přepínače:

- `-d`, `--detach` - spustí kontejner na pozadí
- `-p`, `--publish` - mapování portů (host:kontejner)
- `-v`, `--volume` - připojení volume nebo adresáře
- `-e`, `--env` - nastavení proměnných prostředí
- `--name` - pojmenování kontejneru
- `--rm` - automatické odstranění kontejneru po ukončení
- `-it` - interaktivní režim s terminálem

# **docker run**

Spuštění prvního kontejneru:

```
docker run hello-world
```

Docker stáhne obraz z Docker Hub, vytvoří kontejner a spustí ho

# **docker run**

Spuštění nginx webového serveru na pozadí:

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

Otevřete v prohlížeči <http://localhost:8080>

# **docker run**

Spuštění interaktivního shellu:

```
docker run -it alpine sh
```

Uvnitř kontejneru:

```
cat /etc/os-release
```

```
exit
```

# **docker container ls**

Zobrazení **běžících** kontejnerů:

```
docker container ls
```

Zobrazení **všech** kontejnerů (včetně zastavených):

```
docker container ls -a
```

Zobrazení pouze ID kontejnerů:

```
docker container ls -q
```

# **docker container stop, start, rm**

Zastavení kontejneru:

```
docker container stop my-nginx
```

Spuštění zastaveného kontejneru:

```
docker container start my-nginx
```

Odstranění kontejneru:

```
docker container rm my-nginx
```

# **docker container stop, start, rm**

Zastavení a odstranění jedním příkazem:

```
docker container rm -f my-nginx
```

Přepínač -f (force) zastaví běžící kontejner před odstraněním

# **docker container logs**

Zobrazení logů kontejneru:

```
docker container logs my-nginx
```

Sledování logů v reálném čase:

```
docker container logs -f my-nginx
```

## **docker container exec**

Spuštění příkazu uvnitř **běžícího** kontejneru:

```
docker container exec my-nginx cat /etc/nginx/nginx.conf
```

Spuštění interaktivního shellu:

```
docker container exec -it my-nginx /bin/bash
```

# **docker image ls, pull, rm**

Zobrazení stažených obrazů:

```
docker image ls
```

Stažení obrazu z registru:

```
docker image pull node:24
```

Odstranění obrazu:

```
docker image rm hello-world
```

# Dockerfile

# Co je Dockerfile?

**Dockerfile** je textový soubor s instrukcemi pro sestavení Docker image

- Definuje **základní image**
- Kopíruje **zdrojové soubory**
- Instaluje **závislosti**
- Nastavuje **příkaz** pro spuštění aplikace

# Základní instrukce

- FROM - základní obraz, ze kterého se staví
- WORKDIR - nastavení pracovního adresáře
- COPY - zkopírování souborů do obrazu
- RUN - spuštění příkazu během sestavení
- ENV - nastavení proměnné prostředí
- EXPOSE - dokumentace portu (neotvívá port!)
- CMD - výchozí příkaz při spuštění kontejneru

# Příklad: vlastní nginx obraz

Vytvořte HTML soubor:

```
<!DOCTYPE html>  
<html>  
<body>  
    <h1>Hello from Docker!</h1>  
</body>  
</html>
```

# Příklad: vlastní nginx obraz

Vytvořte Dockerfile:

```
FROM nginx:alpine
```

```
COPY index.html /usr/share/nginx/html/index.html
```

Sestavte obraz:

```
docker image build -t my-web:v1 .
```

## Příklad: vlastní nginx obraz

Spusťte kontejner:

```
docker run -d -p 8080:80 --name my-web my-web:v1
```

Otevřete <http://localhost:8080>

# Dockerfile pro Node.js

```
FROM node:24-alpine
```

```
WORKDIR /app
```

```
COPY package*.json ./
```

```
RUN npm install
```

```
COPY . .
```

```
EXPOSE 3000
```

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

# Dockerfile pro Node.js

Proč kopírujeme `package*.json` **před** zbytkem?

**Layer caching** - Docker cachuje každou vrstvu

- Pokud se `package.json` nezmění, `npm install` se nespustí znovu
- Zrychluje sestavování při změnách v kódu

# Multi-stage builds

Pro produkci používáme **multi-stage build**:

```
# Build stage
FROM node:24 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build
```

# Multi-stage builds

```
# Production stage
FROM node:24-slim
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY --from=builder /app/dist ./dist
EXPOSE 3000
CMD ["node", "dist/index.js"]
```

# Multi-stage builds

Výhody multi-stage build:

- **Menší** výsledný image
- Neobsahuje dev závislosti (TypeScript, ...)
- Bezpečnější - méně potenciálních zranitelností

Porovnání velikosti:

```
docker image ls | grep my-app
```

# **.dockerignore**

Soubor `.dockerignore` funguje jako `.gitignore`:

`node_modules`

`dist`

`.git`

`.env`

`*.log`

`Dockerfile*`

- Zrychluje sestavování (menší context)
- Zabraňuje kopírování citlivých souborů

# Docker Compose

# Co je Docker Compose?

**Docker Compose** je nástroj pro definování a spouštění **více kontejnerů**

- Konfigurace v souboru `compose.yml`
- Jeden příkaz spustí celou aplikaci
- Automaticky vytváří síť pro služby
- Ideální pro vývoj a testování

# Struktura compose.yml

```
services:  
  api:  
    build: ./api  
    ports:  
      - "3000:3000"  
    depends_on:  
      - db
```

# Struktura compose.yml

```
db:
  image: postgres:18
  environment:
    POSTGRES_USER: user
    POSTGRES_PASSWORD: secret
    POSTGRES_DB: app
  volumes:
    - db-data:/var/lib/postgresql
```

```
volumes:
  db-data:
```

# Klíčové vlastnosti

- `image` - použít existující obraz
- `build` - cesta k Dockerfile
- `ports` - mapování portů
- `environment` - proměnné prostředí
- `volumes` - persistentní úložiště
- `depends_on` - pořadí spouštění služeb
- `networks` - vlastní síť

# Příkazy Docker Compose

Příkaz	Popis
<code>docker compose up</code>	Spustí služby
<code>docker compose up -d</code>	Spustí na pozadí
<code>docker compose down</code>	Zastaví a odstraní
<code>docker compose ps</code>	Zobrazí běžící služby
<code>docker compose logs</code>	Zobrazí logy
<code>docker compose build</code>	Sestaví obrazy
<code>docker compose exec</code>	Spustí příkaz ve službě

# Příkazy Docker Compose

Spuštění všech služeb:

```
docker compose up -d
```

Sestavení a spuštění:

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

Zobrazení logů:

```
docker compose logs -f
```

# Příkazy Docker Compose

Zastavení služeb:

`docker compose down`

Zastavení a smazání volumes:

`docker compose down -v`

# **Praktický příklad**

# Příprava Express.js aplikace

Vytvoření projektu:

```
mkdir my-express-app
```

```
cd my-express-app
```

```
npm init -y
```

# Příprava Express.js aplikace

Instalace závislostí:

```
npm install express
```

```
npm install -D typescript @types/express @types/node
```

```
npm install -D ts-node nodemon
```

# Příprava Express.js aplikace

Inicializace TypeScriptu:

```
npx tsc --init
```

# Příprava Express.js aplikace

tsconfig.json:

```
{  
  "compilerOptions": {  
    "target": "ES2020",  
    "module": "commonjs",  
    "outDir": "./dist",  
    "rootDir": "./src",  
    "strict": true,  
    "esModuleInterop": true  
  }  
}
```

# Příprava Express.js aplikace

package.json - skripty:

```
{  
  "scripts": {  
    "dev": "nodemon --exec ts-node src/index.ts",  
    "build": "tsc",  
    "start": "node dist/index.js"  
  }  
}
```

# Příprava Express.js aplikace

src/index.ts:

```
import express, { Request, Response } from 'express';
const app = express();
const PORT = process.env.PORT || 3000;
app.get('/', (req: Request, res: Response) => {
  res.json({ message: 'Hello from Docker!' });
});
app.listen(PORT, '0.0.0.0', () => {
  console.log(`Server running on port ${PORT}`);
});
```

# Dockerizace aplikace

Struktura projektu:

```
my-express-app/  
├── src/  
│   └── index.ts  
├── package.json  
├── package-lock.json  
├── tsconfig.json  
├── Dockerfile  
├── .dockerignore  
└── compose.yml
```

# Dockerizace aplikace

**.dockerignore:**

node\_modules

dist

.git

.env

\*.log

# Dockerizace aplikace

## Dockerfile:

```
FROM node:24-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build
EXPOSE 3000
CMD ["node", "dist/index.js"]
```

# Dockerizace aplikace

Předchozí Dockerfile není optimální

Optimalizaci pomocí multi-stage build můžete vyzkoušet sami

# Dockerizace aplikace

`compose.yml:`

`services:`

`app:`

`build: .`

`ports:`

`- "3000:3000"`

`environment:`

`NODE_ENV: production`

# Dockerizace aplikace

Sestavení a spuštění:

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

Ověření:

```
docker compose ps
```

```
docker compose logs app
```

Aplikace je dostupná na <http://localhost:3000>

# Přidání databáze

`compose.yml` s PostgreSQL:

```
services:
  app:
    build: .
    ports:
      - "3000:3000"
    environment:
      NODE_ENV: production
      DATABASE_URL: postgres://user:pass@db:5432/mydb
    depends_on:
      - db
```

# Přidání databáze

`compose.yml` s PostgreSQL:

```
db:
```

```
  image: postgres:18
```

```
  environment:
```

```
    POSTGRES_USER: user
```

```
    POSTGRES_PASSWORD: pass
```

```
    POSTGRES_DB: mydb
```

```
  volumes:
```

```
    - database:/var/lib/postgresql
```

```
volumes:
```

```
  database:
```

# Přidání databáze

Spuštění s databází:

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

Připojení k databázi:

```
docker compose exec db psql -U user -d mydb
```

Úklid

# Odstranění nepoužívaných zdrojů

Zastavené kontejnery:

`docker container prune`

Nepoužívané obrazy:

`docker image prune`

Nepoužívané volumes:

`docker volume prune`

# Odstranění nepoužívaných zdrojů

Odstranění **všeho** nepoužívaného:

```
docker system prune -a --volumes
```

**Varování:** Tento příkaz odstraní všechna nepoužívaná data!

**Shrnutí**

# Shrnutí

## Klíčové příkazy:

- `docker run` - spuštění kontejneru
- `docker image build` - sestavení image
- `docker container ls` - zobrazení kontejnerů
- `docker container logs` - zobrazení logů
- `docker container exec` - spuštění příkazu v kontejneru

# Shrnutí

## Docker Compose:

- `docker compose up` - spuštění služeb
- `docker compose down` - zastavení služeb
- `docker compose logs` - zobrazení logů
- `docker compose exec` - příkaz ve službě

# Shrnutí

## Klíčové soubory:

- Dockerfile - instrukce pro sestavení image
- compose.yml - konfigurace více služeb
- .dockerignore - ignorované soubory

# Užitečné zdroje

## Oficiální dokumentace

- Docker: [docs.docker.com](https://docs.docker.com)
- Docker Compose: [docs.docker.com/compose](https://docs.docker.com/compose)
- Dockerfile Best Practices:  
[docs.docker.com/build/building/best-practices](https://docs.docker.com/build/building/best-practices)

## Další zdroje

- Docker Hub: [hub.docker.com](https://hub.docker.com)
- Play with Docker: [labs.play-with-docker.com](https://labs.play-with-docker.com)