Konteneryzacja: Docker w praktyce

Zarządzanie Systemami Rozproszonymi

mgr inż. Jakub Woźniak

Politechnika Poznańska
Wydział Informatyki i Telekomunikacji

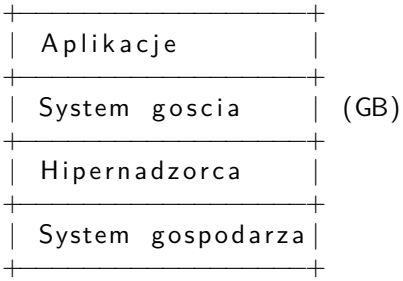
Cele i plan

Cele:

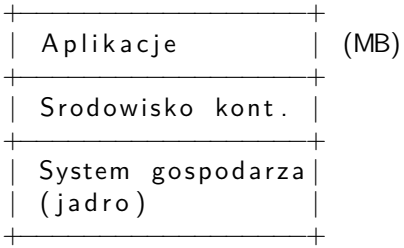
- Zrozumieć różnice między maszynami wirtualnymi a kontenerami oraz poznać architekturę Dockera.
- Opanować wzorce tworzenia Dockerfile: kompilacja wieloetapowa, uruchamianie bez uprawnień administratora, dobór lekkiego obrazu bazowego.
- Poznać podstawy bezpieczeństwa (skanowanie obrazów, utwardzanie środowiska wykonawczego) i skład usług z użyciem Docker Compose.

Maszyny wirtualne a kontenery — istota różnic

Maszyna wirtualna: pełny system gościa nad hipernadzorcą; izolacja silna, narzut zasobów większy.

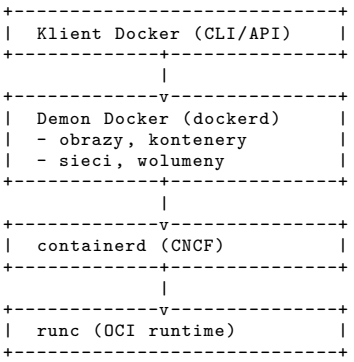


Kontener: wspólny jądro systemu gospodarza; uruchamianie szybkie, zużycie zasobów małe.



Rozmiar: GB kontra MB; start: 30–60 s kontra < 1 s; gęstość uruchomień: dziesiątki kontra setki/tysiące na host.

Architektura Dockera w pigułce



Pojęcia: obraz (tylko do odczytu), kontener (obraz + warstwa zapisu), system warstw z pamięcią podręczną, rejestry obrazów (publiczne i prywatne).

Warstwy, pamięć podręczna i .dockerignore

Kolejność instrukcji a wykorzystanie pamięci podręcznej:

Niekorzystnie (częste przebudowy):

```
FROM node:20-alpine
COPY . .
RUN npm install
RUN npm run build
```

Korzystnie (stabilna pamięć podręczna):

```
FROM node:20-alpine
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build
```

Plik .dockerignore (mniejszy kontekst budowania):

```
node_modules/
.git/
.env
*.md
.vscode/
coverage/
dist/
```

Kompilacja wieloetapowa — smukły obraz wykonawczy

```
# Etap 1: budowanie
FROM node:20 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY . .
RUN npm run build

# Etap 2: uruchomieniowy (lekki)
FROM node:20-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
USER node
EXPOSE 3000
CMD ["node","dist/main.js"]
```

Efekt: znacznie mniejszy obraz i szybsze pobieranie oraz uruchamianie.

Dobór obrazu bazowego

Wielkość i bezpieczeństwo obrazu bazowego mają kluczowe znaczenie:

- **alpine** (7 MB) — bardzo lekki, dobry na etapie deweloperskim (uwaga na musl vs glibc).
- **distroless** (20 MB) — tylko środowisko uruchomieniowe, bez powłoki i menedżera pakietów; mniejsza powierzchnia ataku.
- **scratch** (0 MB) — wyłącznie statyczne binaria; maksymalna prostota i najmniejszy rozmiar.

Praktyka: jeden obraz prowadzony przez wszystkie środowiska, minimalny jak tylko to wygodne - ISO 9k

Uruchamianie bez uprawnień administratora i ograniczanie zasobów

Tworzenie użytkownika i uruchamianie z ograniczeniami:

```
FROM ubuntu:22.04
RUN apt-get update && apt-get install -y python3 && \
    groupadd -g 10001 app && useradd -u 10001 -g app -m app
USER app
WORKDIR /home/app
COPY --chown=app:app app.py .
CMD ["python3","app.py"]
```

Przykładowe uruchomienie z politykami:

```
docker run --read-only --tmpfs /tmp --cap-drop=ALL \
  --cap-add=NET_BIND_SERVICE --pids-limit=200 --memory=256m app:prod
```

Skanowanie obrazów i spis komponentów (SBOM)

Narzędzia: Trivy, Grype, Docker Scout, Snyk.
Szybki start:

```
trivy image --severity HIGH,CRITICAL myapp:latest
grype myapp:latest
```

Zalecenia:

- Regularnie przebudowuj obrazy (np. co miesiąc), aby włączyć poprawki bezpieczeństwa.
- Generuj SBOM (np. CycloneDX/Syft) i egzekwuj polityki w potokach CI/CD.

Docker Compose — zwarta konfiguracja środowiska

```
version: "3.9"
services:
  web:
    build: .
    ports: ["3000:3000"]
    environment:
      DATABASE_URL: "postgresql://db:5432/app"
      REDIS_URL: "redis://cache:6379"
    depends_on:
      db: { condition: service_healthy }
      cache: { condition: service_started }
    user: "10001:10001"
    read_only: true
    tmpfs: ["/tmp"]
    healthcheck:
      test: ["CMD","curl","-f","http://localhost:3000/health"]
      interval: 30s
      timeout: 5s
      retries: 3
  db:
    image: postgres:16-alpine
    environment: { POSTGRES_PASSWORD: ${DB_PASSWORD} }
    volumes: ["db-data:/var/lib/postgresql/data"]
    healthcheck:
      test: ["CMD-SHELL","pg_isready -U postgres"]
      interval: 10s
volumes: { db-data: {} }
```

Alternatywy i środowiska uruchomieniowe

Podman: praca bez demona, tryb bez uprawnień administratora domyślnie, integracja z systemd (generowanie jednostek).
containerd / CRI-O: lekkie i natywne dla Kubernetes (Dockershim usunięty od wersji 1.24).
Praktyka: środowiska deweloperskie — Docker/Podman; produkcja — Kubernetes z containerd/CRI-O.

WebAssembly (Wasm) w zarysie

Zalety: uruchamianie w milisekundach, obrazy rzędu kilobajtów–megabajtów, silna izolacja (piaskownica) — dobre na brzeg/serwerless.
Ograniczenia: młody ekosystem, braki w funkcjonalności w części uruchomień, narzędzia wciąż dojrzewają.
Kierunek: współistnienie kontenerów i Wasm; możliwe uruchamianie Wasm również z narzędziami Dockera.

Lista kontrolna — praktyka od jutra

- Kompilacja wieloetapowa, lekki obraz bazowy, uruchamianie bez uprawnień administratora.
- .dockerignore, ograniczenia zasobów, system plików tylko do odczytu, ograniczenie uprawnień jądra.
- Skanowanie obrazów, podpisy kryptograficzne, tajne dane poza obrazem, polityki sieci, cykliczne przebudowy.
- Dev example: Docker + Compose (środowiska per gałąź). Prod: Kubernetes + obserwowalność.