Docker Compose

◫◫ Poziom: operacyjny **◫◫ Dotyczy: Docker Compose v2 (plugin docker compose)**

▯ Czas: ~30 min

Compose nie wprowadza żadnej nowej funkcji Dockera — to warstwa deklaratywna nad tym samym `docker run`, która pozwala opisać wiele powiązanych kontenerów jednym plikiem zamiast ciągiem osobnych, długich komend.

◫◫ Skróty użyte na tej stronie (2)

Skrót	Co znaczy
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
YAML	YAML Ain't Markup Language: format plików konfiguracyjnych oparty na wcięciach

Fiszka

◫◫ ZMIENIA SYSTEM

Uruchomienie usług w tle: docker compose up -d

```
docker compose up -d
```

Tworzymy i uruchamiamy wszystkie usługi zdefiniowane w pliku compose w bieżącym katalogu, w tle.

Przykład:

```
[+] Running 3/3
 ✔ Network projekt_default    Created
 ✔ Container projekt-db-1     Started
 ✔ Container projekt-app-1    Started
```

◫◫ **docker** = narzędzie do kontenerów

□□ ZMIENIA SYSTEM

Zatrzymanie i usunięcie usług: docker compose down

```
docker compose down
```

Zatrzymuje i usuwa kontenery oraz sieci utworzone przez `up`, ale pozostawia wolumeny — dane bazy danych przeżywają to polecenie.

□□ **docker** = narzędzie do kontenerów

□□ ZMIENIA SYSTEM

Zatrzymanie usług razem z wolumenami: docker compose down -v

```
docker compose down -v
```

Dokłada usunięcie wolumenów nazwanych zadeklarowanych w pliku — po tym poleceniu dane bazy danych znikają bezpowrotnie, jeśli nie były wolumenem zewnętrznym.

□□ **docker** = narzędzie do kontenerów

□□ TYLKO ODCZYT

Śledzenie logów wszystkich usług naraz: docker compose logs -f

```
docker compose logs -f
```

Łączy logi wszystkich kontenerów projektu w jeden strumień, z prefiksem nazwy usługi przed każdą linią — wygodniejsze niż otwieranie osobnego `docker logs` dla każdej usługi z osobna.

Przykład:

```
db-1   | 2026-09-10 08:12:01.442 UTC [1] LOG:  database system is ready to accept
connections
app-1  | 2026-09-10 08:12:03.108Z [notice] listening on port 80
```

□□ **docker** = narzędzie do kontenerów

□□ TYLKO ODCZYT

Wylistowanie usług projektu: docker compose ps

```
docker compose ps
```

Zwraca stan kontenerów należących do projektu w bieżącym katalogu — odróżnia je od `docker ps`, które pokazuje wszystkie kontenery na hoście, niezależnie z jakiego projektu compose pochodzą.

docker = narzędzie do kontenerów

TYLKO ODCZYT

Pobranie obrazów bez uruchamiania kontenerów: docker compose pull

```
docker compose pull
```

Pobiera najnowsze wersje obrazów wskazanych w polu `image` każdej usługi — przydatne przed `up`, gdy chcemy mieć pewność, że wystartujemy z aktualną wersją, a nie ewentualnie przestarzałą kopią lokalną.

docker = narzędzie do kontenerów

TYLKO ODCZYT

Wyświetlenie w pełni rozwiniętej konfiguracji: docker compose config

```
docker compose config
```

Wypisuje finalną konfigurację po podstawieniu wszystkich zmiennych i scaleniu ewentualnych wielu plików compose — pierwszy krok diagnostyki, gdy usługa startuje inaczej, niż się spodziewaliśmy, bo widzimy dokładnie to, co Compose faktycznie odczytał, a nie to, co wydaje nam się, że napisaliśmy.

docker = narzędzie do kontenerów

Z życia wzięte

Z ŻYCIA WZIĘTE

W poniedziałek rano aplikacja Oli na `docker01` nie może połączyć się z bazą zaraz po `docker compose up -d`, mimo że `depends_on` w pliku wygląda poprawnie. Okazuje się, że `depends_on` czeka tylko na start kontenera bazy, nie na jej gotowość do przyjmowania połączeń. Po dopisaniu `healthcheck` z warunkiem `service_healthy` aplikacja startuje dopiero wtedy, gdy baza faktycznie odpowiada.

Czym jest plik compose

> **Pojęcie: plik compose.** Plik YAML (domyślnie `docker-compose.yml`), w którym deklarujemy zestaw usług (`services`) składających się na jedną aplikację — każda usługa odpowiada temu, co inaczej trzeba by uruchomić osobnym `docker run`, razem z portami, wolumenami, zmiennymi środowiskowymi i siecią.

Różnica względem `docker run` jest praktyczna, nie koncepcyjna: zamiast pamiętać i za każdym razem wpisywać ręcznie długą komendę z dziesiątkiem flag, opisujemy ją raz w pliku, a `docker compose up` odtwarza cały zestaw kontenerów w jednym wywołaniu — łącznie z siecią łączącą je ze sobą, którą Compose tworzy automatycznie.

Struktura pliku

Podstawowa konfiguracja usług opartych na gotowych obrazach:

```
version: "3.9"

services:
  app:
    image: nginx:latest
    ports:
      - "8080:80"

  db:
    image: postgres:15
    environment:
      POSTGRES_PASSWORD: HasloUzytkownika
```

Po uruchomieniu `docker compose up -d` Compose pobiera obrazy, tworzy kontenery i domyślną sieć. Każda usługa jest dostępna dla pozostałych usług pod nazwą swojej sekcji — usługa `db` jest osiągalna z usługi `app` po prostu jako host `db`, bez znajomości jej adresu IP.

> **Pojęcie: `image` i `build`.** `image` wskazuje gotowy obraz do pobrania z rejestru, `build` każe Composeowi zbudować obraz lokalnie z Dockerfile leżącego we wskazanym katalogu (`build: ./aplikacja`) — obie opcje wykluczają się przy tej samej usłudze, chyba że `build` jest użyte razem z `image` po to, żeby nadać zbudowanemu obrazowi konkretną nazwę i tag.

> **Pojęcie: `depends_on`.** Deklaracja kolejności startu kontenerów — usługa z `depends_on: [db]` czeka, aż kontener `db` zostanie uruchomiony, zanim Compose wystartuje ją samą. To **nie** jest gwarancja, że baza danych jest już gotowa do przyjmowania połączeń, tylko że jej proces w ogóle wystartował — do faktycznego czekania na gotowość służy `healthcheck`.

> **Pojęcie: `healthcheck`.** Definicja komendy sprawdzającej, czy usługa wewnątrz kontenera jest faktycznie gotowa do pracy, nie tylko uruchomiona:

```
services:
  db:
    image: postgres:15
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
      timeout: 3s
      retries: 5
```

Połączone z `depends_on: { db: { condition: service_healthy } }` w innej usłudze pozwala faktycznie odczekać na gotowość bazy, a nie tylko na start jej procesu.

> **Pojęcie: `restart`.** Ten sam mechanizm polityki restartu, co flaga `--restart` w `docker run` — `unless-stopped`, `always`, `on-failure`, `no` (domyślnie).

Sieci w pliku compose

Jeśli sieci nie zostaną jawnie zdefiniowane, Compose tworzy jedną sieć typu bridge i podłącza do niej wszystkie usługi z pliku. W bardziej złożonych przypadkach definiujemy własne sieci, żeby

ograniczyć, które usługi w ogóle się widzą:



Frontend widzi tylko backend, backend widzi frontend i db, a db nie ma dostępu do sieci front (kliknięcie otwiera pełny rozmiar)

```
services:
  frontend:
    image: nginx
    networks:
      - front

  backend:
    image: nginx
    networks:
      - front
      - back

  db:
    image: postgres
    networks:
      - back

networks:
  front:
  back:
```

W tym układzie `frontend` nie ma dostępu do `db`, bo obie usługi nie współdzielą żadnej wspólnej sieci — `backend` pełni rolę pośrednika, widocznego w obu sieciach naraz. Jedna usługa może należeć do wielu sieci jednocześnie.

Sieć oznaczona jako wewnętrzna blokuje ruch wychodzący poza sieć Dockera — usługi w niej widzą się nawzajem, ale nie mają dostępu do internetu ani do reszty hosta:

```
networks:
  back:
    internal: true
```

Wolumeny w pliku compose

> W naszym środowisku dane aplikacji trzymamy w `/docker/apps/<nazwa>`, a plik Compose w `/docker/compose/<nazwa>/compose.yaml`. To konwencja, nie wymóg Dockera: można ją zmienić, byle w całej firmie była jedna. Katalog `/docker` to osobny wolumen `vgDocker/lvDocker`, żeby pełny dysk aplikacji nie zatrzymał systemu. Pełny opis konwencji na stronie Jak trzymamy aplikacje w Dockerze: konwencje Zielonej.

Bez wolumenów dane zapisane wewnątrz kontenera znikają razem z jego usunięciem — Compose obsługuje te same rodzaje trwałości danych, co `docker run -v`, tylko zadeklarowane w YAML.

Bind mount, czyli mapowanie katalogu z hosta, użyteczne w środowisku deweloperskim, gdzie potrzebny jest bezpośredni dostęp do plików:

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

Wolumen nazwany, zarządzany przez Dockera, niezależny od struktury katalogów hosta:

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

volumes:
  db_data:
```

Docker przechowuje wtedy dane we własnej przestrzeni pod `/var/lib/docker/volumes/...` — nazwane wolumeny są bardziej przenośne między hostami niż bind mounty, bo nie zależą od konkretnej struktury katalogów.

Wolumen anonimowy, bez podanej nazwy źródłowej:

```
services:
  app:
    image: nginx
    volumes:
      - /data
```

Docker tworzy wtedy wolumen bez nazwy i przypisuje go do kontenera — rozwiązanie rzadziej stosowane, bo utrudnia świadome zarządzanie takim wolumenem później.

Wolumeny mogą być współdzielone między usługami, co pozwala na operacje pomocnicze takie jak backup czy migracja danych bez bezpośredniego dostępu do kontenera bazy:

```
services:
  db:
    image: postgres
    volumes:
      - db_data:/data

  backup:
    image: alpine
    volumes:
      - db_data:/data

volumes:
  db_data:
```

Dostęp do wolumenu można ograniczyć do trybu tylko do odczytu dopiskiem `:ro`:

```
services:
  app:
    image: nginx
    volumes:
      - db_data:/data:ro
```

Składnia rozszerzona pozwala opisać te same parametry jawnymi kluczami zamiast skróconego zapisu:

```
services:
  app:
```

```
    image: nginx
    volumes:
      - type: volume
        source: db_data
        target: /data
        read_only: true

volumes:
  db_data:
```

Wolumeny mogą korzystać z różnych sterowników (`driver`), co pozwala na integrację z zewnętrznymi systemami przechowywania danych, z dodatkowymi opcjami przekazywanymi przez `driver_opts`:

```
volumes:
  db_data:
    driver: local
```

Wolumen zewnętrzny (`external: true`) nie jest tworzony przez Compose — musi istnieć wcześniej, co pozwala współdzielić dane między różnymi projektami niezależnie od cyklu życia konkretnego pliku compose:

```
volumes:
  db_data:
    external: true
```

Typowym wzorcem jest inicjalizacja danych przy pierwszym uruchomieniu kontenera (np. obrazy takie jak `postgres` inicjalizują pusty katalog danych przy pierwszym starcie) — wolumen gwarantuje, że ta inicjalizacja nie powtórzy się przy kolejnych uruchomieniach, bo katalog danych nie jest już pusty.

Zmienne w pliku .env

> **Pojęcie: plik `.env`.** Plik w tym samym katalogu co plik compose, z parami `KLUCZ=wartość`, które Compose automatycznie podstawia w miejscach zapisanych jako `${KLUCZ}` w pliku YAML — pozwala trzymać wartości zależne od środowiska (hasła, tagi wersji obrazów) poza samym plikiem compose, żeby ten sam plik działał identycznie na środowisku testowym i produkcyjnym, różniąc się tylko zawartością `.env`.

```
POSTGRES_PASSWORD=HasloUzytkownika
TAG_OBRAZU=1.4.2
```

```
services:
  app:
    image: git.zielona.example/zielona/aplikacja:${TAG_OBRAZU}
  db:
    image: postgres:15
    environment:
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
```

Osobno od `.env` istnieje klucz `env_file` w samej usłudze — wskazuje dowolny plik z parami `KLUCZ=wartość`, które trafiają jako zmienne środowiskowe do kontenera tej usługi (nie do samego pliku compose):

```
services:
  app:
    image: aplikacja:latest
    env_file:
      - ./config/app.env
```

Pełny przykład

Konfiguracja łącząca sieci i wolumeny w jednym projekcie:

```
version: "3.9"

services:
  app:
    image: nginx
    ports:
      - "8080:80"
    networks:
      - frontend
      - backend

  db:
    image: postgres:15
```

```
    environment:
      POSTGRES_PASSWORD: HasloUzytkownika
    volumes:
      - db_data:/var/lib/postgresql/data
    networks:
      - backend

  cache:
    image: redis:7
    networks:
      - backend

networks:
  frontend:
  backend:

volumes:
  db_data:
```

W tej konfiguracji `app` komunikuje się z `db` i `cache` przez sieć `backend`, natomiast `db` nie jest dostępna z zewnątrz ani z sieci `frontend`. Dane bazy są przechowywane w wolumenie `db_data`, który pozostaje po usunięciu kontenerów zwykłym `docker compose down`.

Częste błędy

Zakładanie, że `depends_on` czeka na gotowość usługi, nie tylko na jej start. Aplikacja próbująca połączyć się z bazą danych zaraz po starcie kontenera bazy, zanim ta zdąży zainicjalizować się i zacząć nasłuchiwać, kończy się błędem połączenia mimo poprawnie zadeklarowanego `depends_on` — potrzebny jest `healthcheck` z warunkiem `service_healthy`.

`docker compose down -v` uruchomiony odruchowo przy każdym restarcie środowiska. Flaga `-v` kasuje wolumeny nazwane razem z kontenerami — na środowisku z prawdziwymi danymi (nie tylko testowymi) to nieodwracalna utrata danych za jedną literę różnicy w komendzie.

Trzymanie haseł wprost w pliku compose zamiast w `.env`. Plik compose zwykle trafia do repozytorium razem z kodem — hasło wpisane wprost w `environment:` trafia więc do historii commitów na stałe, nawet jeśli zostanie później usunięte z najnowszej wersji pliku.

□□ Ćwiczenia

> Zadania sprawdzające.

1. Piszemy plik compose z dwiema usługami — `web` (obraz `nginx`, port 8080→80) i `db` (obraz `postgres:15`) — w tej samej sieci użytkownika o nazwie `backend`.
2. Wyjaśniamy różnicę między `docker compose down` a `docker compose down -v` i wskazujemy, które z nich bezpiecznie uruchomić na produkcji bez ryzyka utraty danych.
3. Usługa `app` ma zależeć od usługi `db` i czekać na jej faktyczną gotowość, nie tylko na start procesu. Dopisujemy odpowiedni `healthcheck` i warunek w `depends_on`.
4. Przenosimy hasło bazy danych z pliku compose do pliku `.env` i podstawiamy je w `environment` przez `${...}`.

Powiązane: Obraz, kontener, warstwa, wolumen: cztery pojęcia · Codzienne komendy · Sieci kontenerów i SNAT

Revision #39
Created 2026-03-28 13:21:29 UTC by Mikołaj Kocemba
Updated 2026-10-26 14:09:25 UTC by Tomasz Konieczny