Podstawy

Obrazy, kontenery, codzienne komendy i Compose.

- Obraz, kontener, warstwa, wolumen: cztery pojęcia
- Codzienne komendy Dockera
- Docker Compose

Obraz, kontener, warstwa, wolumen: cztery pojęcia

Poziom: podstawy **Dotyczy: Docker Engine 24+** **Czas: ~20 min**

Te cztery pojęcia tłumaczą się nawzajem — bez zrozumienia, czym jest warstwa, trudno pojąć, dlaczego budowanie obrazu bywa szybkie albo wolne, a bez zrozumienia różnicy między obrazem a kontenerem trudno w ogóle rozmawiać o Dockerze.

Skróty użyte na tej stronie (4)

Skrót	Co znaczy
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
ISO	International Organization for Standardization: organizacja, która wydaje normy techniczne używane na całym świecie
RAM	Random Access Memory: pamięć operacyjna
SNAT	Source NAT: podmiana adresu nadawcy w ruchu wychodzącym

Z życia wzięte

> **Z ŻYCIA WZIĘTE**
>
> Tomek kasuje w piątek po południu testowy kontener na `docker01` poleceniem `docker rm`, żeby zwolnić miejsce przed weekendem. W poniedziałek Ola pyta, gdzie podziały się dane aplikacji, którą akurat testowała, bo baza w kontenerze nie miała podpiętego żadnego wolumenu. Dane zniknęły razem z zapisywalną warstwą kontenera, bezpowrotnie, dokładnie tak, jak opisuje ta strona.

Obraz i kontener — dlaczego to nie to samo

> **Pojęcie: obraz.** Statyczny, tylko-do-odczytu szablon zawierający system plików aplikacji — biblioteki, zależności, skompilowany kod, plik wykonywalny — razem z metadanymi mówiącymi, jaki proces uruchomić przy starcie. Obraz sam z siebie nic nie robi; leży na dysku (albo w rejestrze) jako zestaw plików gotowych do użycia, dokładnie tak jak plik ISO instalatora systemu operacyjnego nie jest samym zainstalowanym systemem.

> **Pojęcie: kontener.** Uruchomiona instancja obrazu — proces (a właściwie grupa procesów) odizolowany od reszty systemu przez mechanizmy jądra Linuksa (przestrzenie nazw, `cgroups`), z własnym systemem plików nadbudowanym na obrazie, z którego powstał. Z jednego obrazu można uruchomić dowolnie wiele kontenerów naraz, tak jak z jednego pliku ISO można zainstalować system na wielu maszynach — każdy kontener ma własny, niezależny stan, mimo że wszystkie startują z identycznego punktu wyjścia.

Różnica sprowadza się do tego samego rozróżnienia, co między plikiem programu na dysku a uruchomionym procesem w pamięci — obraz jest przepisem, kontener jest gotowym daniem.

Warstwy i cache budowania

> **Pojęcie: warstwa (layer).** Obraz nie jest jednym monolitycznym blokiem danych, tylko sekwencją warstw ułożonych jedna na drugiej — każda instrukcja w pliku definiującym obraz (`FROM`, `RUN`, `COPY`...) tworzy nową warstwę, zawierającą tylko różnicę względem warstwy poprzedniej. System plików kontenera to widok złożony ze wszystkich tych warstw naraz, od najstarszej (bazowej) do najnowszej.

To rozwarstwienie ma bezpośredni wpływ na szybkość budowania. Docker cache'uje każdą warstwę osobno i przy kolejnym budowaniu pomija ponowne wykonanie instrukcji, jeśli ani ona, ani żadna wcześniejsza się nie zmieniła — stąd praktyczna zasada układania instrukcji od najrzadziej zmieniających się (instalacja zależności systemowych) do najczęściej zmieniających się (kopiowanie kodu aplikacji), żeby zmiana jednej linijki kodu nie unieważniała cache'u całej instalacji zależności poprzedzającej ją w pliku definicji obrazu.

Kontener dokłada do tych warstw jeszcze jedną, własną, zapisywalną warstwę na wierzchu — to w niej ląduje wszystko, co proces zapisuje w trakcie działania, i to ona znika bezpowrotnie razem z kontenerem, jeśli nie została skierowana do wolumenu.

Wolumeny — trwałość ponad cyklem życia kontenera

> 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.

> **Pojęcie: wolumen.** Mechanizm przechowywania danych poza zapisywalną warstwą kontenera, dzięki któremu dane przeżywają usunięcie kontenera, który je zapisał. Bez wolumenu każdy `docker rm` oznacza bezpowrotną utratę wszystkiego, co proces zapisał od momentu startu.

Docker udostępnia trzy sposoby trzymania danych poza kontenerem, różniące się tym, kto nimi zarządza i gdzie faktycznie leżą na dysku hosta.



Obraz to warstwy tylko do odczytu, kontener dokłada warstwę zapisywalną, która znika po docker rm, chyba że dane trafiły do wolumenu (kliknięcie otwiera pełny rozmiar)

Wolumen nazwany (named volume) jest tworzony i zarządzany całkowicie przez Dockera, a jego dane leżą pod `/var/lib/docker/volumes/NazwaWolumenu/_data` — nie musimy znać ani dbać o tę ścieżkę, odwołujemy się do wolumenu przez jego nazwę. To domyślny wybór dla danych aplikacji (bazy danych, kolejki), które mają przeżyć restart czy podmianę kontenera.

Bind mount to bezpośrednie podpięcie konkretnego katalogu z hosta do kontenera — w przeciwieństwie do wolumenu nazwanego, ścieżka na hosta jest jawna i w pełni pod naszą kontrolą, co jest wygodne przy pracy deweloperskiej (edycja kodu na hoście widoczna od razu w kontenerze), ale przenosi na nas odpowiedzialność za uprawnienia i strukturę tego katalogu.

tmpfs trzyma dane wyłącznie w pamięci RAM hosta, nigdy nie zapisując ich na dysk — znikają natychmiast przy zatrzymaniu kontenera. Stosujemy go do danych tymczasowych i wrażliwych, których obecność na dysku w ogóle nie jest pożądana (np. pliki tymczasowe z sekretami rozpakowywanymi w trakcie działania procesu).

Sieć kontenera

Każdy kontener domyślnie dostaje własny, izolowany stos sieciowy — własny adres IP i własną przestrzeń nazw sieciową jądra, oddzieloną od hosta i od innych kontenerów, chyba że jawnie połączymy je wspólną siecią. Rodzaje sieci dockerowych, publikowanie portów na hosta i przekierowanie ruchu (SNAT) rozwija strona Sieci kontenerów i SNAT.

Cykl życia kontenera

Kontener przechodzi przez kilka stanów widocznych w kolumnie `STATUS` polecenia `docker ps -a`:

Stan	Znaczenie
`Created`	kontener utworzony (`docker create`), ale jeszcze nieuruchomiony — system plików istnieje, proces nie działa
`Up`	proces działa
`Paused`	wszystkie procesy w kontenerze zamrożone (`docker pause`), stan i pamięć zachowane
`Exited`	proces zakończył działanie (sam się zamknął albo został zatrzymany) — system plików wciąż istnieje, dane w zapisywalnej warstwie są nadal dostępne, dopóki kontener nie zostanie usunięty
(usunięty)	`docker rm` kasuje zapisywalną warstwę kontenera bezpowrotnie — wolumeny podpięte do kontenera przeżywają, jeśli nie zostały usunięte jawnie osobną komendą

`docker run` w praktyce łączy dwa kroki naraz — tworzy kontener z obrazu (`create`) i od razu go uruchamia (`start`). Konkretne komendy do zarządzania tym cyklem, z realnymi przykładami wyniku, zbiera strona Codzienne komendy.

Częste błędy

Traktowanie `docker stop` jako usunięcia kontenera. Zatrzymany kontener (`Exited`) wciąż istnieje na dysku razem z całą swoją zapisywalną warstwą — widać go w `docker ps -a`, zajmuje miejsce, a jego dane (o ile nie były w wolumenie) są nadal tam, gdzie je zostawił proces przed zatrzymaniem. Dopiero `docker rm` usuwa go naprawdę.

Zapisywanie danych aplikacji bez wolumenu, bo „na razie działa". Działa dokładnie do pierwszego `docker rm` albo do podmiany obrazu na nowszą wersję — oba te zdarzenia kasują zapisywalną warstwę kontenera razem z całą jego zawartością.

Mylenie bind mounta z wolumenem nazwanym w dokumentacji projektu. Skrypt wdrożeniowy zakładający konkretną ścieżkę na hoście przestaje działać na maszynie, gdzie ta ścieżka nie istnieje — named volume nie ma tego problemu, bo Docker sam zarządza lokalizacją.

□□ Ćwiczenia

> Zadania sprawdzające.

1. Wyjaśniamy, dlaczego z jednego obrazu można uruchomić wiele niezależnych kontenerów naraz.
2. Plik definiujący obraz kopiuje najpierw kod aplikacji, a dopiero potem instaluje zależności systemowe. Wyjaśniamy, dlaczego taka kolejność szkodzi wykorzystaniu cache'u warstw, i proponujemy poprawną kolejność.
3. Aplikacja w kontenerze zapisuje logi do `/var/log/app` bez podpiętego wolumenu. Opisujemy, co się z nimi stanie po `docker rm` tego kontenera.
4. Dobieramy typ trwałości danych (wolumen nazwany, bind mount, tmpfs) do trzech scenariuszy: dane bazy produkcyjnej, kod aplikacji edytowany lokalnie w trakcie developmentu, plik z tymczasowym tokenem sesji.

Powiązane: Codzienne komendy · Docker Compose · Sieci kontenerów i SNAT · Utrzymanie hosta dockerowego

Codzienne komendy Dockera

Poziom: podstawy **Dotyczy: Docker Engine 24+, Docker CLI** **Czas: ~30 min**

Komendy wykonujemy z CLI bash jako user z uprawnieniami sudo albo jako user dodany do grupy `docker` — bez jednego z tych dwóch warunków każde polecenie kończy się błędem `permission denied` na gnieździe demona.

Skróty użyte na tej stronie (6)

Skrót	Co znaczy
CLI	Command Line Interface: praca przez wpisywanie komend w terminalu
CPU	Central Processing Unit: procesor
GB	Gigabyte: miliard bajtów, jednostka pojemności pamięci i plików, tu najczęściej rozmiar modelu albo pamięci karty graficznej
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
JSON	JavaScript Object Notation: tekstowy format danych z klamrami i cudzysłowami
NAT	Network Address Translation: podmiana adresów IP na granicy sieci

Fiszka

TYLKO ODCZYT

Wylistowanie uruchomionych kontenerów: docker ps

```
docker ps
```

Zwracamy listę aktualnie działających kontenerów wraz z ich identyfikatorem, obrazem źródłowym, czasem działania i zmapowanymi portami.

Przykład:

```
CONTAINER ID   IMAGE          COMMAND                  CREATED         STATUS
PORTS                    NAMES
a1b2c3d4e5f6   nginx:latest   "/docker-entrypoint.…"   3 hours ago     Up 3 hours
0.0.0.0:8080->80/tcp     app_web
f6e5d4c3b2a1   postgres:15    "docker-entrypoint.s…"   3 hours ago     Up 3 hours
5432/tcp                 app_db
```

□□ **docker** = narzędzie do kontenerów

□□ TYLKO ODCZYT

Wylistowanie wszystkich kontenerów, łącznie z zatrzymanymi: docker ps -a

```
docker ps -a
```

Dokładamy do wyniku kontenery w stanie `Exited` — przydatne przy szukaniu kontenera, który się zatrzymał albo nigdy poprawnie nie wystartował.

Przykład:

```
CONTAINER ID   IMAGE          COMMAND                  CREATED
STATUS                     PORTS     NAMES
a1b2c3d4e5f6   nginx:latest   "/docker-entrypoint.…"   3 hours ago     Up 3
hours                  80/tcp    app_web
9f8e7d6c5b4a   redis:7        "docker-entrypoint.s…"   1 day ago       Exited (137) 2
hours ago              app_cache_old
```

□□ **docker** = narzędzie do kontenerów

□□ TYLKO ODCZYT

Wylistowanie pobranych obrazów: docker images

```
docker images
```

Zwracamy obrazy dostępne lokalnie, z ich rozmiarem i identyfikatorem — rozmiar dotyczy unikalnych warstw obrazu, nie sumy warstw współdzielonych z innymi obrazami.

Przykład:

```
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
nginx        latest    a6bd71f48f68   2 weeks ago     192MB
postgres     15        4c0dcb1c497b   3 weeks ago     412MB
```

docker = narzędzie do kontenerów

ZMIENIA SYSTEM

Uruchomienie kontenera w tle: -d

```
docker run -d nginx:latest
```

Uruchamiamy kontener w tle (detached) i od razu odzyskujemy kontrolę nad terminalem — bez tej flagi proces kontenera zajmuje terminal, w którym wpisaliśmy komendę, dopóki go nie zatrzymamy.

Przykład:

```
a1b2c3d4e5f61234567890abcdef1234567890abcdef1234567890abcdef12
```

docker = narzędzie do kontenerów

ZMIENIA SYSTEM

Uruchomienie kontenera z interaktywnym terminalem: -it

```
docker run -it ubuntu:22.04 bash
```

Łączymy dwie flagi naraz — `-i` utrzymuje otwarte standardowe wejście, `-t` przydziela pseudoterminal — dzięki czemu dostajemy działającą interaktywną powłokę wewnątrz kontenera, tak jakbyśmy się do niego zalogowali.

Przykład:

```
root@a1b2c3d4e5f6:/#
```

docker = narzędzie do kontenerów

ZMIENIA SYSTEM

Zmapowanie portu kontenera na port hosta: -p

```
docker run -d -p 8080:80 nginx:latest
```

Publikujemy port `80` kontenera pod portem `NumerPortu` hosta (tutaj `8080`) — ruch przychodzący na host trafia do kontenera przez regułę NAT dodaną przez Dockera do `iptables`. Bez tej flagi port kontenera jest osiągalny wyłącznie z sieci dockerowej, nie z zewnątrz hosta.

Przykład:

```
CONTAINER ID   IMAGE          PORTS                    NAMES
a1b2c3d4e5f6   nginx:latest   0.0.0.0:8080->80/tcp   app_web
```

NAT = Network Address Translation: podmiana adresów IP na granicy sieci · **docker** = narzędzie do kontenerów

ZMIENIA SYSTEM

Podpięcie wolumenu do kontenera: -v

> 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.

```
docker run -d -v NazwaWolumenu:/var/lib/postgresql/data postgres:15
```

Podpinamy wolumen nazwany pod ścieżkę wewnątrz kontenera — dane zapisywane pod tą ścieżką przeżywają usunięcie kontenera. Ta sama flaga obsługuje też bind mounty, gdy zamiast nazwy wolumenu podamy ścieżkę bezwzględną z hosta (`/SciezkaDoPliku:/var/lib/postgresql/data`).

Przykład:

```
CONTAINER ID   IMAGE          MOUNTS
a1b2c3d4e5f6   postgres:15    NazwaWolumenu
```

docker = narzędzie do kontenerów

◫◫ ZMIENIA SYSTEM

Nadanie kontenerowi nazwy: --name

```
docker run -d --name NazwaKontenera nginx:latest
```

Nadajemy kontenerowi czytelną nazwę zamiast losowo wygenerowanej — dzięki temu w kolejnych komendach (`docker logs`, `docker exec`, `docker stop`) odwołujemy się do niego po nazwie zamiast po identyfikatorze, który trzeba by najpierw sprawdzić.

Przykład:

```
CONTAINER ID   IMAGE          NAMES
a1b2c3d4e5f6   nginx:latest   NazwaKontenera
```

◫◫ **docker** = narzędzie do kontenerów

◫◫ ZMIENIA SYSTEM

Ustawienie polityki automatycznego restartu: --restart

```
docker run -d --restart unless-stopped NazwaObrazu
```

Deklarujemy, że demon Dockera ma sam ponownie uruchomić kontener po jego awarii albo po restarcie hosta — wariant `unless-stopped` robi to zawsze, chyba że kontener został zatrzymany ręcznie przez `docker stop`; wariant `always` ignoruje nawet ręczne zatrzymanie i uruchomi kontener ponownie przy starcie demona; wariant `on-failure` restartuje tylko po zakończeniu z niezerowym kodem wyjścia.

Przykład:

```
CONTAINER ID   IMAGE         STATUS
a1b2c3d4e5f6   NazwaObrazu   Up 2 seconds (health: starting)
```

◫◫ **docker** = narzędzie do kontenerów

◫◫ ZMIENIA SYSTEM

Podłączenie kontenera do konkretnej sieci: --network

```
docker run -d --network NazwaSieci NazwaObrazu
```

Podpinamy kontener do wskazanej sieci użytkownika zamiast domyślnej sieci bridge — kontenery w tej samej sieci użytkownika widzą się nawzajem po nazwie, bez znajomości adresów IP. Więcej o rodzajach sieci na stronie Sieci kontenerów i SNAT.

Przykład:

```
CONTAINER ID   IMAGE          NETWORKS
a1b2c3d4e5f6   NazwaObrazu    NazwaSieci
```

IP = Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection) · **SNAT** = Source NAT: podmiana adresu nadawcy w ruchu wychodzącym · **docker** = narzędzie do kontenerów

ZMIENIA SYSTEM

Ustawienie zmiennej środowiskowej w kontenerze: -e

```
docker run -d -e POSTGRES_PASSWORD=HasloUzytkownika postgres:15
```

Przekazujemy zmienną środowiskową do procesu wewnątrz kontenera — standardowy sposób konfiguracji obrazów gotowych (baz danych, kolejek), które odczytują parametry startowe właśnie ze zmiennych środowiskowych, nie z pliku konfiguracyjnego.

Przykład:

```
docker exec NazwaKontenera env | grep POSTGRES_PASSWORD
```

```
POSTGRES_PASSWORD=HasloUzytkownika
```

docker = narzędzie do kontenerów

OSTROŻNIE · NIEODWRACALNE

Automatyczne usunięcie kontenera po zakończeniu: --rm

```
docker run --rm -it ubuntu:22.04 bash
```

Kontener znika automatycznie zaraz po zakończeniu procesu — wygodne przy jednorazowych, testowych uruchomieniach, żeby nie zaśmiecać `docker ps -a` dziesiątkami zakończonych kontenerów, o których i tak zapomnimy.

docker = narzędzie do kontenerów

TYLKO ODCZYT

Wykonanie komendy wewnątrz działającego kontenera: docker exec

```
docker exec -it NazwaKontenera bash
```

Uruchamiamy dodatkowy proces wewnątrz już działającego kontenera, bez wpływu na jego główny proces — najczęstsze zastosowanie to wejście do powłoki działającego kontenera w celu diagnostyki, bez jego restartowania.

Przykład:

```
root@a1b2c3d4e5f6:/#
```

docker = narzędzie do kontenerów

TYLKO ODCZYT

Podgląd logów kontenera: docker logs

```
docker logs -f NazwaKontenera
```

Wypisujemy standardowe wyjście i wyjście błędów procesu kontenera, z flagą `-f` śledząc je na bieżąco — to samo wyjście, które widzielibyśmy w terminalu, gdyby kontener nie był uruchomiony w tle.

Przykład:

```
2026-09-10T08:12:03.441Z [notice] nginx worker process started
2026-09-10T08:12:04.102Z [notice] listening on 0.0.0.0:80
```

docker = narzędzie do kontenerów

TYLKO ODCZYT

Sprawdzenie pełnej konfiguracji kontenera: docker inspect

```
docker inspect NazwaKontenera
```

Zwracamy pełną strukturę JSON opisującą kontener — zmienne środowiskowe, zamontowane wolumeny, adres IP, politykę restartu, wszystko, co zostało ustawione przy tworzeniu. Przydatne, gdy trzeba sprawdzić dokładną konfigurację kontenera uruchomionego dawno

temu przez kogoś innego.

Przykład:

```
[
    {
        "Id": "a1b2c3d4e5f6...",
        "State": { "Status": "running", "Running": true },
        "NetworkSettings": { "IPAddress": "172.17.0.2" }
    }
]
```

JSON = JavaScript Object Notation: tekstowy format danych z klamrami i cudzysłowami · **IP** = Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection) · **docker** = narzędzie do kontenerów

TYLKO ODCZYT

Podgląd zużycia zasobów na żywo: docker stats

```
docker stats
```

Pokazujemy zużycie CPU, pamięci i ruchu sieciowego wszystkich działających kontenerów w czasie rzeczywistym, odświeżane co sekundę — pierwsze narzędzie do sprawdzenia, który kontener zjada zasoby hosta.

Przykład:

```
CONTAINER ID   NAME       CPU %     MEM USAGE / LIMIT     MEM %     NET I/O
a1b2c3d4e5f6   app_web    0.12%     24.3MiB / 512MiB      4.75%     1.2kB / 850B
f6e5d4c3b2a1   app_db     3.41%     186MiB / 1GiB         18.16%    412kB / 1.1MB
```

CPU = Central Processing Unit: procesor · **docker** = narzędzie do kontenerów

ZMIENIA SYSTEM

Skopiowanie pliku między kontenerem a hostem: docker cp

```
docker cp NazwaKontenera:/SciezkaDoPliku /SciezkaDoPliku
```

Kopiujemy plik lub katalog z kontenera na hosta (albo odwrotnie, po zamianie kolejności argumentów) bez potrzeby montowania wolumenu — wygodne przy jednorazowym wyciągnięciu pliku diagnostycznego z już działającego kontenera.

docker = narzędzie do kontenerów

ZMIENIA SYSTEM

Zatrzymanie kontenera: docker stop

```
docker stop NazwaKontenera
```

Wysyłamy do głównego procesu kontenera sygnał `SIGTERM` i czekamy (domyślnie 10 sekund) na jego samodzielne zakończenie, zanim Docker doda `SIGKILL` — dajemy procesowi szansę na uporządkowane zamknięcie (np. dokończenie zapisu do bazy).

SIGTERM = sygnał numer 15: prośba do procesu, żeby zakończył się grzecznie, proces może ją zignorować albo posprzątać przed wyjściem · **SIGKILL** = sygnał numer 9: każe jądru natychmiast zabić proces, bez szansy na sprzątnięcie po sobie · **docker** = narzędzie do kontenerów

ZMIENIA SYSTEM

Wymuszone zabicie kontenera: docker kill

```
docker kill NazwaKontenera
```

Wysyłamy `SIGKILL` natychmiast, bez czekania na uporządkowane zamknięcie procesu — stosujemy, gdy `docker stop` nie skutkuje albo gdy zależy nam na natychmiastowym efekcie kosztem ewentualnej utraty niezapisanych danych.

SIGKILL = sygnał numer 9: każe jądru natychmiast zabić proces, bez szansy na sprzątnięcie po sobie · **docker** = narzędzie do kontenerów

OSTROŻNIE · NIEODWRACALNE

Usunięcie kontenera: docker rm

```
docker rm NazwaKontenera
```

Usuwamy zatrzymany kontener razem z jego zapisywalną warstwą — wolumeny podpięte do niego przeżywają, chyba że dodamy flagę `-v`, która usunie też anonimowe wolumeny utworzone razem z tym kontenerem.

docker = narzędzie do kontenerów

TYLKO ODCZYT

Sprawdzenie zajętości miejsca przez Dockera: docker system df

```
docker system df
```

Zwracamy podsumowanie miejsca zajętego przez obrazy, kontenery, lokalne wolumeny i cache budowania — pierwszy krok przed decyzją o sprzątaniu, żeby wiedzieć, co faktycznie zajmuje najwięcej.

Przykład:

```
TYPE            TOTAL     ACTIVE    SIZE      RECLAIMABLE
Images          14        6         3.8GB     2.1GB (55%)
Containers      9         3         142MB     98MB (69%)
Local Volumes   5         3         1.2GB     310MB (25%)
Build Cache     42        0         890MB     890MB (100%)
```

docker = narzędzie do kontenerów

OSTROŻNIE · NIEODWRACALNE

Usunięcie nieużywanych zasobów: docker system prune

```
docker system prune
```

Kasujemy zatrzymane kontenery, nieużywane sieci, nieoznaczone obrazy (`<none>`) i cache budowania — po potwierdzeniu pytaniem, bo operacja jest nieodwracalna. Warianty tej komendy i ryzyko flagi `-a --volumes` rozwija strona Utrzymanie hosta dockerowego.

docker = narzędzie do kontenerów

Operacje na rejestrze obrazów

Zalogowanie do prywatnego rejestru: docker login

```
docker login git.zielona.example
```

Logujemy demona Dockera do wskazanego rejestru obrazów — bez tego `docker pull` i `docker push` na obrazy z rejestrów prywatnych kończą się błędem `unauthorized`. Komenda pyta interaktywnie o login i hasło, jeśli nie zostały podane inaczej. W Zielonej rejestrem obrazów jest rejestr pakietów Gitei pod `git.zielona.example`, a obrazy nazywamy `git.zielona.example/zielona/NazwaObrazu:wersja`; w polu hasła wpisujemy token z Gitei z zakresem `read:package` albo `write:package`.

Przykład:

```
Username: nazwa_usera
Password:
Login Succeeded
```

Poświadczenia zapisują się w pliku `~/.docker/config.json` danego użytkownika systemowego — przy przenoszeniu tej samej tożsamości logowania na inny serwer wystarczy skopiować ten plik do analogicznej lokalizacji zamiast logować się tam ręcznie po raz drugi, o ile hasło dla tego konta nie wymaga rotacji między serwerami.

Z życia wzięte

> **□□ Z ŻYCIA WZIĘTE**
>
> Ola dostaje `permission denied` przy pierwszym `docker pull` z rejestru firmowego na nowym laptopie, bo nikt jej nie powiedział, że trzeba się najpierw zalogować przez `docker login git.zielona.example`. Po zalogowaniu obraz pobiera się bez problemu, a poświadczenia zostają zapisane w `~/.docker/config.json` do następnego razu. Tomek kopiuje później ten sam plik na inny serwer, myśląc że to wystarczy, i zapomina, że tamto konto wymaga osobnej rotacji hasła.

Częste błędy

Wpisywanie komend Dockera bez `sudo`, gdy user nie jest w grupie `docker`. Kończy się komunikatem `permission denied while trying to connect to the Docker daemon socket` — trwałym

rozwiązaniem jest dodanie użytkownika do grupy `docker` (`usermod -aG docker nazwa_usera`) i ponowne zalogowanie, nie wpisywanie `sudo` przed każdą komendą.

Mylenie `docker stop` z `docker rm`. Zatrzymany kontener nadal istnieje i zajmuje miejsce — nowa osoba bywa zaskoczona, widząc go dalej w `docker ps -a` po tym, co uważała za jego usunięcie.

Uruchamianie `docker system prune` bez wcześniejszego `docker system df`. Polecenie kasuje bez pytania o każdy element z osobna — sprawdzenie, co faktycznie zajmuje miejsce, zanim skasujemy, chroni przed usunięciem czegoś, co ktoś inny właśnie przygotowywał do użycia.

□□ Ćwiczenia

> Zadania sprawdzające.

1. Piszemy jedną komendę `docker run`, która uruchamia kontener `nginx:latest` w tle, pod nazwą `strona_glowna`, z portem 80 kontenera zmapowanym na port 8081 hosta i polityką restartu `unless-stopped`.
2. Kontener `app_db` przestał odpowiadać. Opisujemy kolejność komend, którymi sprawdzimy jego logi, zużycie zasobów i pełną konfigurację, zanim zdecydujemy się go zrestartować.
3. Wyjaśniamy różnicę między `docker stop` a `docker kill` i podajemy sytuację, w której świadomie wybieramy to drugie.
4. `docker system df` pokazuje 2.1 GB miejsca możliwego do odzyskania w obrazach. Opisujemy, jaką komendą sprawdzimy to dokładniej przed wykonaniem `prune`.

Powiązane: Obraz, kontener, warstwa, wolumen: cztery pojęcia · Docker Compose · Utrzymanie hosta dockerowego

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