Zaczynamy

Opowieść, konwencje Zielonej, instalacja Dockera i pierwsza aplikacja.

- Docker: kontenery, które zmieniły serwerownie
- Jak trzymamy aplikacje w Dockerze: konwencje Zielonej
- Instalacja Dockera na Rocky Linux 9 z osobnym wolumenem /docker
- Pierwsza aplikacja w Compose krok po kroku

Docker: kontenery, które zmieniły serwerownie

Poziom: podstawy **Dotyczy: Docker Engine 24+** **Czas: ~15 min**

Ta strona otwiera książkę o Dockerze: skąd wziął się pomysł zamykania aplikacji w izolowanych pudełkach, dlaczego Zielona w ogóle postawiła osobny serwer pod kontenery i co czeka na kolejnych stronach.

Skróty użyte na tej stronie (1)

Skrót	Co znaczy
LXC	Linux Containers: wczesny mechanizm izolacji procesów w jądrze Linuksa, na którym pierwotnie oparto Dockera

Skąd się wzięło

> **Pojęcie: kontener.** Sposób uruchamiania aplikacji w izolowanym środowisku, które dzieli jądro systemu operacyjnego z hostem, ale ma własny system plików, własne procesy i własną sieć, tak jakby stało na osobnej maszynie. W odróżnieniu od maszyny wirtualnej kontener nie uruchamia własnego systemu operacyjnego: pożycza go od hosta.

Pomysł na izolowane pudełko do transportu jest starszy niż informatyka. 26 kwietnia 1956 roku Amerykanin Malcolm McLean załadował na przerobiony tankowiec SS Ideal-X pięćdziesiąt osiem identycznych metalowych skrzyń i wysłał je z Newark do Houston. Przed tym dniem towar na statek trafiał sztuka po sztuce, ręcznie przekładany przez dokerów z ciężarówki na pokład; po tym dniu wystarczyło przełożyć całą skrzynię dźwigiem, niezależnie od tego, co było w środku. Standardowy rozmiar i standardowe zaczepy sprawiły, że ten sam kontener przechodził bez przeładunku ze statku na ciężarówkę, a z ciężarówki na pociąg.

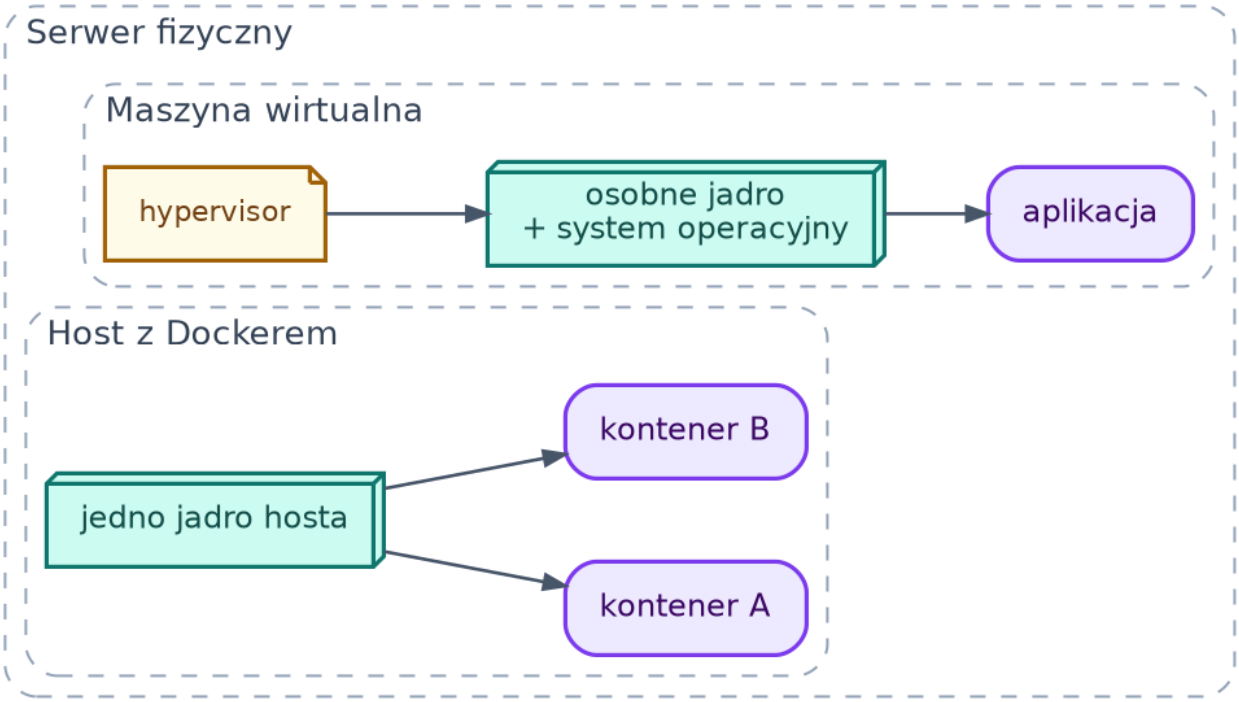
W świecie serwerów podobny problem, jak uruchomić wiele aplikacji na jednej maszynie, nie pozwalając im się nawzajem zaśmiecać, rozwiązywano stopniowo. W 1979 roku, przy pracach nad siódmą wersją Uniksa, pojawił się `chroot`: mechanizm zamykający proces w wybranym podkatalogu, który z jego punktu widzenia wygląda jak cały dysk. W 2000 roku system FreeBSD

rozwinął ten pomysł do jails, dokładając izolację sieci i procesów, nie tylko systemu plików. W 2008 roku w jądrze Linuksa pojawił się LXC, korzystający z dwóch nowych mechanizmów jądra: przestrzeni nazw i grup kontrolnych, które w końcu pozwoliły odizolować od siebie nie tylko pliki, ale i widok na procesy, sieć czy użytkowników.

> **Pojęcie: Docker.** Narzędzie, które w 2013 roku zapakowało LXC, a później własny mechanizm izolacji, w prostą komendę i czytelny format opisu aplikacji, dzięki czemu uruchomienie kontenera przestało wymagać znajomości szczegółów jądra Linuksa. Solomon Hykes pracował wtedy nad dotCloud, firmą hostingową wywodzącą się z programu Y Combinator z lata 2010 roku. Docker powstał jako wewnętrzne narzędzie dotCloud do pakowania aplikacji klientów, a w marcu 2013 roku Hykes pokazał je publicznie na konferencji PyCon w Santa Clara i tego samego roku wydał je jako otwarte oprogramowanie. Sama firma dotCloud jako hosting zniknęła z rynku, ale narzędzie, które miało być jej wewnętrznym dodatkiem, zmieniło sposób, w jaki cała branża stawia aplikacje na serwerach.

Jak to działa w jednym obrazku

Maszyna wirtualna i kontener rozwiązują ten sam problem (odizolowanie aplikacji od reszty serwera) na dwóch różnych piętrach. Maszyna wirtualna uruchamia pełny, osobny system operacyjny nad warstwą wirtualizacji sprzętu, więc dwie maszyny wirtualne na jednym serwerze to dwa jądra, dwa zestawy sterowników i dwa razy więcej pamięci zajętej samym systemem, zanim jeszcze wystartuje aplikacja. Kontener dzieli jedno jądro hosta ze wszystkimi innymi kontenerami i uruchamia tylko to, co potrzebne samej aplikacji, więc startuje w sekundach zamiast minut i mieści się w ułamku pamięci maszyny wirtualnej.



Maszyna wirtualna z osobnym jądrem systemowym obok kilku kontenerów dzielących jedno jądro hosta (kliknięcie otwiera pełny rozmiar)

Dlaczego Zielona

> **Z ŻYCIA WZIĘTE**
>
> Zanim Zielona miała `docker01`, każda nowa aplikacja zespołu Oli zaczynała się tym samym zgłoszeniem do nas: „potrzebujemy maszyny". Zamawialiśmy maszynę wirtualną na `hv01`, instalowaliśmy system, aktualizacje, runtime i zależności, a Ola i tak czekała tydzień, zanim cokolwiek w ogóle ruszyło. Gdy trzy miesiące później zespół chciał postawić drugą, mniejszą usługę obok pierwszej, cała procedura zaczynała się od nowa: kolejna maszyna, kolejny tydzień, kolejny system do łatania.

Ta sama aplikacja jako kontener na `docker01` startuje w kilkanaście sekund, bez zamawiania niczego u nas: Ola dostaje dostęp do serwera raz, a każdą kolejną usługę stawia sama, plikiem Compose, który sama pisze i sama rozumie.

Zysk nie jest tylko czasowy. Aktualizacja jednej aplikacji nie wymaga już dotykania systemu operacyjnego pod spodem, bo obraz kontenera niesie swoje zależności ze sobą, a awaria jednej usługi (przepełniony dysk, wyciek pamięci) nie zabiera zasobów pozostałym, dopóki ustawiliśmy

limity opisane na stronie Utrzymanie hosta dockerowego. Zamiast dziesięciu maszyn wirtualnych z osobnym systemem każda, `docker01` trzyma dzisiaj Giteę, NetBoksa, Grayloga, MailCow i Vaultwardena na jednym systemie operacyjnym, każdą aplikację w osobnym, odizolowanym kontenerze.

Mapa tej książki

Strona	Na jakie pytanie odpowiada
Docker: kontenery, które zmieniły serwerownie	Skąd wziął się Docker i dlaczego go używamy
Jak trzymamy aplikacje w Dockerze: konwencje Zielonej	Gdzie w naszym środowisku leżą dane i pliki Compose każdej aplikacji
Instalacja Dockera na Rocky Linux 9 z osobnym wolumenem /docker	Jak postawić hosta dockerowego zgodnie z tą konwencją
Pierwsza aplikacja w Compose krok po kroku	Jak uruchomić swoją pierwszą aplikację od zera
Obraz, kontener, warstwa, wolumen: cztery pojęcia	Czym różni się obraz od kontenera i gdzie żyją dane
Codzienne komendy	Jakich komend Dockera używamy najczęściej
Docker Compose	Jak opisać wiele kontenerów jednym plikiem
Sieci kontenerów i SNAT	Jak kontenery się komunikują i wychodzą do sieci
Utrzymanie hosta dockerowego	Jak dbać o miejsce na dysku, logi i limity zasobów
Kopia zapasowa aplikacji dockerowych	Jak zabezpieczyć dane, gdy coś pójdzie nie tak

Powiązane: Jak trzymamy aplikacje w Dockerze: konwencje Zielonej · Świat przykładów: Zielona

Źródła: en.wikipedia.org/wiki/Docker_(software) · en.wikipedia.org/wiki/Malcom_McLean · en.wikipedia.org/wiki/Chroot · en.wikipedia.org/wiki/LXC

Jak trzymamy aplikacje w Dockerze: konwencje Zielonej

Poziom: referencyjny **Dotyczy: wszystkie aplikacje na docker01** **Czas: ~20 min**

Ta strona nie uczy nowej komendy, tylko opisuje jeden wspólny sposób trzymania każdej aplikacji dockerowej w Zielonej: gdzie leżą jej dane, gdzie plik Compose i dlaczego akurat tak. Każda kolejna strona instalacyjna w tej książce się do niej odwołuje.

Skróty użyte na tej stronie (7)

Skrót	Co znaczy
GID	Group Identifier: numer grupy systemowej, drugi obok UID element decydujący o prawach dostępu do pliku
LVM	Logical Volume Manager: warstwa w Linuksie składająca dyski w elastyczne wolumeny
SSH	Secure Shell: szyfrowane zdalne logowanie do serwera
UID	User ID: numer użytkownika w Linuksie
UISP	Ubiquiti ISP: aplikacja producenta Ubiquiti do zarządzania jego urządzeniami sieciowymi, dawniej pod nazwą UNMS
UniFi	linia sprzętu Ubiquiti do sieci firmowych i domowych: punkty dostępowe Wi-Fi, przełączniki i bramy, zarządzana z aplikacji UniFi Network
Wi-Fi	Sieć bezprzewodowa: wygodna w biurze, ale w serwerowni dokłada kolejną możliwą przyczynę awarii

Fiszka

ZMIENIA SYSTEM

Założenie katalogów pod nową aplikację: mkdir -p

```
sudo mkdir -p /docker/apps/NazwaAplikacji /docker/compose/NazwaAplikacji
```

Zakładamy oba katalogi razem, zanim napiszemy jakikolwiek plik Compose: jeden na dane, drugi na sam przepis uruchomieniowy.

Przykład:

```
docker01:~$ ls -ld /docker/apps/netbox /docker/compose/netbox
drwxr-xr-x. 2 root root 6 wrz 27 10:02 /docker/apps/netbox
drwxr-xr-x. 2 root root 6 wrz 27 10:02 /docker/compose/netbox
```

mkdir = make directory: utwórz katalog

ZMIENIA SYSTEM

Ograniczenie praw do pliku z hasłami: chmod 600

```
chmod 600 /docker/compose/NazwaAplikacji/.env
```

Plik `.env` trzyma hasła i tokeny w postaci jawnej, więc ograniczamy odczyt wyłącznie do właściciela pliku, zwykle roota albo konta wdrażającego.

Przykład:

```
docker01:~$ ls -l /docker/compose/netbox/.env
-rw-------. 1 root root 214 wrz 27 10:05 /docker/compose/netbox/.env
```

chmod = change mode: zmień prawa dostępu

TYLKO ODCZYT

Sprawdzenie wersji obrazu faktycznie uruchomionej: docker compose images

```
docker compose images
```

Sprawdzamy przed każdą aktualizacją, jaki tag obrazu jest aktualnie uruchomiony, zamiast zgadywać na podstawie pliku compose, który mógł się zmienić od ostatniego `up -d`.

Przykład:

```
docker01:~$ cd /docker/compose/netbox && docker compose images
CONTAINER   REPOSITORY                TAG       IMAGE ID       SIZE
netbox      netboxcommunity/netbox    v4.1.4    3f8a2c9e1b7d   612MB
```

◫ **docker** = narzędzie do kontenerów

Z życia wzięte

◫ Z ŻYCIA WZIĘTE

Wtorek, 9:20. Tomek chce podnieść wersję UISP, aplikacji, którą zespół trzyma na `docker01` do zarządzania sprzętem operatorskim Ubiquiti (punkty Wi-Fi to UniFi, patrz UISP i UniFi: dwa systemy zarządzania sprzętem Ubiquiti i czym się różnią). Zamiast zmienić numer wersji w `.env`, robi szybkie `docker compose pull && docker compose up -d` na obrazie wskazanym w starym pliku jako `ui-com/uisp:latest`. Rejestr pobiera najnowsze duże wydanie, które zmieniło format bazy danych, i po restarcie kontenera panel UISP w ogóle nie startuje, a Tomek do wieczora odtwarza kopię sprzed aktualizacji, żeby przywrócić podgląd stanu urządzeń.

Ten sam scenariusz z przypiętą wersją obrazu (`ui-com/uisp:2.4.4`) nie miałby prawa się zdarzyć: `pull` ściągnąłby dokładnie ten sam tag co wcześniej, a świadoma aktualizacja do nowej wersji byłaby jedną zmianą w `.env`, zaplanowaną i możliwą do cofnięcia.

Jak to działa

◫ Rozwiń: jak to działa pod spodem

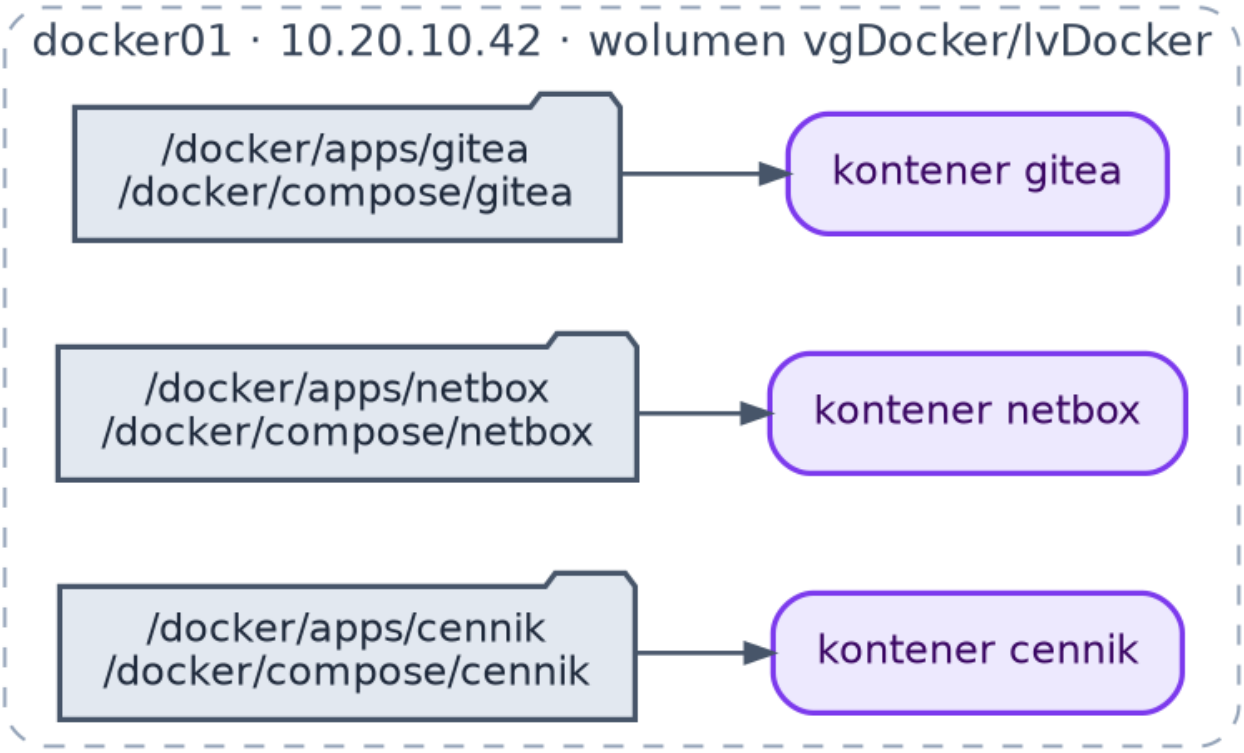
Drzewo katalogów

Wszystkie aplikacje dockerowe w Zielonej trzymają się jednego układu katalogów pod `/docker`, niezależnie od tego, ile ich jest akurat na `docker01`:

```
/docker
├── apps
```

```
│   ├── gitea
│   │   ├── data
│   │   └── postgres
│   ├── netbox
│   │   ├── postgres
│   │   └── redis
│   └── cennik
│       └── data
└── compose
    ├── gitea
    │   ├── compose.yaml
    │   └── .env
    ├── netbox
    │   ├── compose.yaml
    │   └── .env
    └── cennik
        ├── compose.yaml
        └── .env
```

Nazwa katalogu pod `apps` i pod `compose` jest zawsze taka sama, więc szukanie danych konkretnej aplikacji sprowadza się do jednego skojarzenia, nie do pamiętania osobnej konwencji dla każdej aplikacji z osobna.



Katalogi /docker/apps i /docker/compose dla trzech aplikacji, każdy podpięty do własnego kontenera (kliknięcie otwiera pełny rozmiar)

> 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.

Dlaczego osobny wolumen LVM

Pojęcie: wolumen logiczny. Fragment przestrzeni dyskowej zarządzany przez LVM, montowany pod wybranym katalogiem tak jak zwykła partycja, ale możliwy do powiększenia później bez wyłączania systemu. `/docker` na `docker01` leży na własnym wolumenie logicznym `vgDocker/lvDocker`, oddzielnym od wolumenu systemowego `vgOS/lvOS`.

Rozdzielenie ma jeden konkretny powód: aplikacja, która zapełni cały przydzielony jej dysk (rosnąca baza NetBoksa, zapomniana rotacja logów UISP), zapełnia wtedy tylko `/docker`, nie cały system. Proces `systemd`, logi systemowe i możliwość zalogowania się po SSH zostają nietknięte, bo leżą na osobnym wolumenie `vgOS/lvOS`, z własnym, niezależnym wolnym miejscem. Pełną procedurę założenia takiego wolumenu opisuje strona Instalacja Dockera na Rocky Linux 9 z osobnym wolumenem /docker.

Dlaczego dane i compose rozdzielone

Katalog `/docker/apps/<nazwa>` i katalog `/docker/compose/<nazwa>` mają różny cykl życia i różną wagę. Dane aplikacji to zwykle jedyna nieodtwarzalna rzecz na całym serwerze: baza, załączniki, repozytoria. Plik Compose to za to tylko przepis, możliwy do odtworzenia z repozytorium konfiguracji albo, w ostateczności, do napisania od nowa w kilka minut.

Rozdzielenie tych dwóch katalogów sprawia, że kopia zapasowa, opisana na stronie Kopia zapasowa aplikacji dockerowych, może traktować je inaczej: `/docker/apps` kopiujemy zawsze i z najwyższym priorytetem, `/docker/compose` kopiujemy też, ale jego utrata boli dużo mniej.

Plik .env i prawa 600

Hasła, tokeny i inne sekrety nigdy nie stoją wprost w `compose.yaml`, tylko w osobnym pliku `.env` obok niego, podstawianym przez Compose przez zapis `${ZMIENNA}`. Sam plik compose może bez obaw trafić do repozytorium konfiguracji, `.env` nigdy.

Prawa `600` (odczyt i zapis wyłącznie dla właściciela) ograniczają dostęp do pliku z hasłami do konta, które faktycznie uruchamia kontenery, zamiast do każdego, kto ma jakiekolwiek konto na `docker01`.

Nazewnictwo usług, sieci i kontenerów

Element	Konwencja	Przykład
Usługa w compose.yaml	nazwa roli w aplikacji, liczba pojedyncza	`server`, `db`
container_name	nazwa aplikacji, ewentualnie z dopiskiem roli	`netbox`, `netbox-postgres`
Sieć compose	domyślna, bez jawnej nazwy, chyba że aplikacja łączy się z inną	`netbox_default`

Element	Konwencja	Przykład
Wolumen danych	bind mount pod `/docker/apps`, nie wolumen nazwany	`/docker/apps/netbox/postgres:/var/lib/postgresql/data`

Nazwa `container_name` powtarza nazwę katalogu pod `/docker/apps` i `/docker/compose`, żeby wynik `docker ps`, ścieżka na dysku i wpis w dokumentacji zawsze zgadzały się ze sobą bez tłumaczenia.

Przypięte wersje obrazów zamiast latest

Każdy obraz w `compose.yaml` ma jawny numer wersji, na przykład `netboxcommunity/netbox:v4.1.4`, nigdy tag `latest`. Tag `latest` nie oznacza „najbardziej przetestowana wersja", tylko „cokolwiek autor obrazu akurat oznaczył jako najnowsze w chwili budowania", co bywa świeżym wydaniem sprzed godziny, z niesprawdzoną jeszcze migracją, jak w scence z UISP wyżej.

Aktualizacja u nas to zawsze świadoma zmiana numeru w `.env` albo w `compose.yaml`, sprawdzenie informacji o wydaniu i dopiero potem `docker compose pull` razem z `docker compose up -d`, nigdy odruchowe uruchomienie `pull` na środowisku z tagiem `latest`.

Porty tylko na 127.0.0.1 za Nginksem

Gdy przed aplikacją stoi reverse proxy Nginksa, opisany na stronie Reverse proxy i równoważenie obciążenia, port aplikacji publikujemy wyłącznie na adresie lokalnym hosta:

```
ports:
  - "127.0.0.1:8080:80"
```

Bez adresu przed dwukropkiem Docker publikuje port na wszystkich interfejsach hosta, czyli też na tych widocznych z sieci biurowej albo z internetu, z pominięciem Nginksa i jego certyfikatu, autoryzacji czy limitów ruchu. Ograniczenie do `127.0.0.1` sprawia, że aplikacja jest osiągalna wyłącznie przez proxy stojące na tym samym hoście.

Właściciel katalogów: UID z kontenera

Proces wewnątrz kontenera działa jako konkretny użytkownik z konkretnym UID, niezależnie od tego, jakie konto systemowe na `docker01` uruchomiło `docker compose up`. Katalog na hoście podpięty pod ścieżkę, do której ten proces zapisuje, musi należeć do tego samego UID, inaczej zapis kończy się błędem `Permission denied`, mimo że z poziomu hosta katalog wygląda na dostępny.

Numer UID sprawdzamy w dokumentacji obrazu albo w jego pliku definiującym, i ustawiamy go na katalogu przed pierwszym uruchomieniem:

```
sudo chown -R 999:999 /docker/apps/NazwaAplikacji
```

> (gdzie `999:999` to UID i GID procesu wewnątrz konkretnego obrazu, różne dla każdej aplikacji)

To konwencja, nie wymóg

Nic w Dockerze nie wymaga akurat takiego układu katalogów. Firma, która zaczyna od zera, może równie dobrze trzymać wszystko pod `/srv/docker/<nazwa>`, w jednym katalogu na dane i compose naraz, albo zupełnie inaczej. Jedyny prawdziwy wymóg to spójność: jedna konwencja, spisana w jednym miejscu, zamiast pięciu wariantów wymyślonych przez pięciu administratorów na przestrzeni lat.

Zmiana konwencji w istniejącej firmie oznacza migrację: zatrzymanie kontenerów, przeniesienie katalogów na nowe miejsce, poprawienie ścieżek w każdym `compose.yaml`, dopiero potem `docker compose up -d` na nowej lokalizacji. Nie jest to nieodwracalne, ale nie jest też coś, co robi się bez przygotowania na hoście z dziesięcioma działającymi aplikacjami naraz.

Szablon compose.yaml do skopiowania

Punkt startowy dla każdej nowej aplikacji, z komentarzami do usunięcia po wypełnieniu:

```
# Szablon compose.yaml dla nowej aplikacji w Zielonej.
# Kopiujemy do /docker/compose/NazwaAplikacji/compose.yaml
# i usuwamy te komentarze po wypełnieniu.

services:
  server:
    # Obraz zawsze z przypiętą wersją, nigdy z tagiem "latest".
    image: git.zielona.example/zielona/NazwaObrazu:1.0.0
    container_name: NazwaAplikacji
    restart: unless-stopped
    environment:
      # Wartości podstawiane z pliku .env leżącego obok tego pliku.
```

```
      - HASLO=${HASLO_APLIKACJI}
    volumes:
      # Dane aplikacji zawsze pod /docker/apps/<nazwa>, nigdy w wolumenie nazwanym.
      - /docker/apps/NazwaAplikacji/data:/data
    ports:
      # Port tylko na localhost, gdy przed aplikacją stoi Nginx.
      - "127.0.0.1:NumerPortu:80"
```

Częste błędy

Objaw	Przyczyna	Co robimy
Kontener startuje, ale zaraz kończy się błędem zapisu do wolumenu	katalog na hoście należy do roota, proces w kontenerze działa jako inny UID	`chown` katalogu na UID z dokumentacji obrazu, przed pierwszym `up -d`
Po `docker compose pull` aplikacja przestaje działać bez żadnej zmiany w `compose.yaml`	obraz oznaczony tagiem `latest`, rejestr podał nowszą wersję	przypinamy konkretny numer wersji, odtwarzamy kopię sprzed aktualizacji
Aplikacja osiągalna bezpośrednio z internetu mimo Nginksa przed nią	port opublikowany bez adresu `127.0.0.1` przed dwukropkiem	poprawiamy `ports` w `compose.yaml`, `docker compose up -d`
Hasło z `.env` widoczne dla każdego konta na serwerze	plik `.env` z domyślnymi prawami `644` zamiast `600`	`chmod 600` na pliku

□□ Ćwiczenia

> Zadania sprawdzające.

1. Zakładamy katalogi pod nową aplikację `cennik`, zgodnie z konwencją, i sprawdzamy ich prawa dostępu.
2. Wyjaśniamy, dlaczego katalog `/docker` leży na osobnym wolumenie LVM, a nie razem z systemem operacyjnym.
3. Piszemy fragment `ports` w `compose.yaml`, który wystawia port aplikacji tylko dla Nginksa na tym samym hoście.
4. Aplikacja w kontenerze zapisuje dane jako UID `999`, a katalog na hoście należy do `root:root`. Opisujemy, co się stanie przy starcie i jak to naprawić.

Powiązane: Docker: kontenery, które zmieniły serwerownie · Instalacja Dockera na Rocky Linux 9 z osobnym wolumenem /docker · Pierwsza aplikacja w Compose krok po kroku · Kopia zapasowa aplikacji dockerowych

Instalacja Dockera na Rocky Linux 9 z osobnym wolumenem /docker

Typ: procedura **Dotyczy: Rocky Linux 9, Docker CE 27+, wtyczka Compose**

Czas: ~45 min

Efekt: działający Docker Engine na Rocky Linux 9, z własnymi danymi na osobnym wolumenie `vgDocker/lvDocker` zamiast na dysku systemowym, gotowy na pierwszą aplikację opisaną na stronie Pierwsza aplikacja w Compose krok po kroku.

Skróty użyte na tej stronie (5)

Skrót	Co znaczy
CE	Community Edition: darmowa, społecznościowa wersja Dockera, w odróżnieniu od płatnej wersji Enterprise
CLI	Command Line Interface: praca przez wpisywanie komend w terminalu
RHEL	Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa
UUID	Universally Unique Identifier: długi, praktycznie niepowtarzalny identyfikator
XFS	system plików stworzony przez Silicon Graphics, domyślny w Rocky Linux i RHEL

Zanim zaczniemy

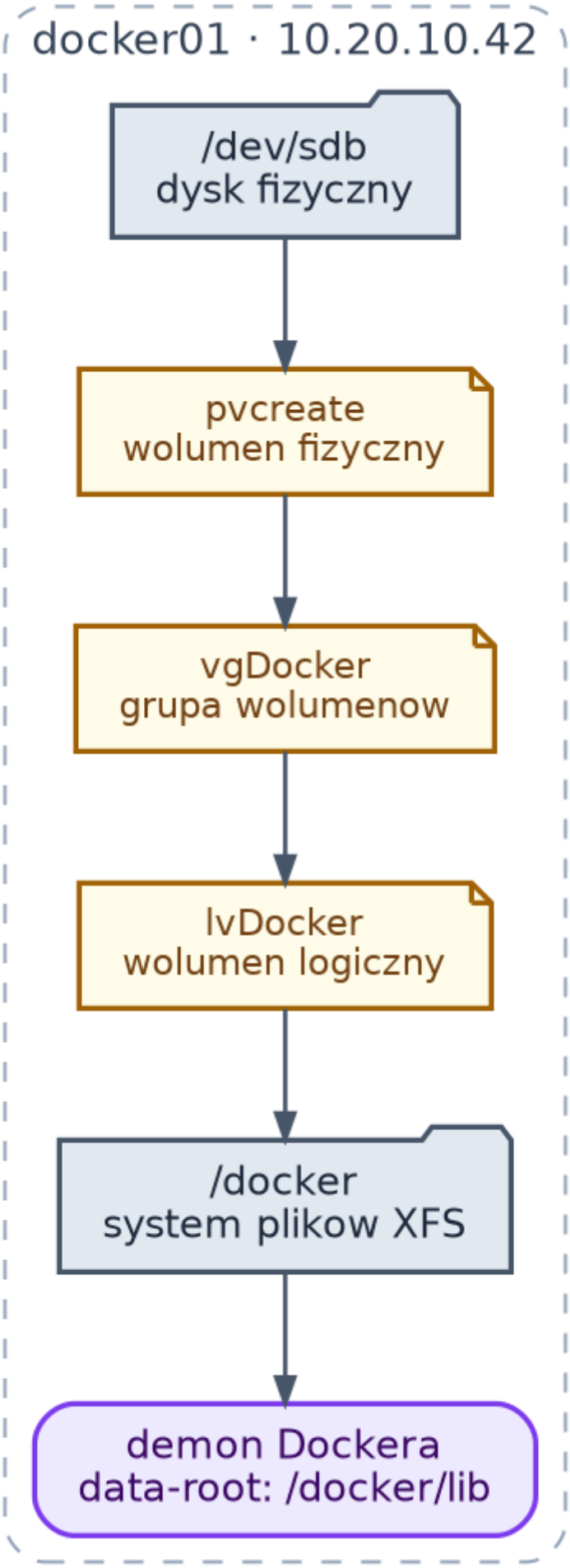
> W naszym środowisku dane aplikacji trzymamy w `/docker/apps/<nazwa>`, a plik Compose w `/docker/compose/<nazwa>/compose.yaml`. To konwencja, nie wymóg Dockera: można ją zmienić, byle w całej firmie była jedna. Katalog `/docker` to osobny wolumen `vgDocker/lvDocker`, żeby pełny dysk aplikacji nie zatrzymał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

Procedura zakłada gotowy system Rocky Linux 9 z jednym wolnym dyskiem pod przyszły `/docker`, zainstalowany według wzorca z Instalacja Rocky Linux z własnym nazewnictwem grup wolumenów:

system operacyjny na `vgOS/lvOS`, `/var` na `vgVAR/lvVAR`. Komendy wykonujemy z CLI bash z uprawnieniami root (sudo).

Wymaganie	Uwaga
Wolny dysk pod `vgDocker/lvDocker`	dołożenie dysku po instalacji systemu, patrz krok 1
Rocky Linux 9 z dostępem do internetu albo lokalnego lustra pakietów	repozytorium Docker CE pobiera pakiety przez sieć
Konto z prawem sudo	do instalacji pakietów i edycji `/etc/fstab`

Droga od gołego dysku do działającego demona Dockera, krok po kroku:


docker01 · 10.20.10.42
/dev/sdb
dysk fizyczny
pvcreate
wolumen fizyczny
vgDocker
grupa wolumenow
lvDocker
wolumen logiczny
/docker
system plikow XFS
demon Dockera
data-root: /docker/lib

Dysk fizyczny zamieniony przez pvcreate, vgcreate i lvcreate w wolumen logiczny, zamontowany jako /docker i wskazany demonowi Dockera jako data-root (kliknięcie otwiera pełny rozmiar)

Przebieg

1. Weryfikacja wolnego dysku

```
lsblk
```

Przykład:

```
NAME         MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
sda            8:0    0    60G  0 disk
├─sda1         8:1    0     1G  0 part /boot
└─sda2         8:2    0    59G  0 part
  └─vgOS-lvOS 253:0    0    50G  0 lvm  /
sdb            8:16   0   500G  0 disk
```

Dysk `/dev/sdb` na przykładzie wyżej to wolny dysk pod `vgDocker`, bez żadnej partycji ani systemu plików.

2. Utworzenie wolumenu fizycznego i grupy wolumenów

```
sudo pvcreate /dev/sdb
```

```
sudo vgcreate vgDocker /dev/sdb
```

Zostawiamy zapas w grupie zamiast od razu zająć całą przestrzeń wolumenem logicznym:

```
sudo lvcreate -L 400G -n lvDocker vgDocker
```

> Nie wykorzystujemy `100%FREE` przy zakładaniu wolumenu, z tego samego powodu co przy każdym innym wolumenie w Zielonej: wolne miejsce w grupie pozwala później powiększyć `lvDocker` jedną komendą, bez dokładania kolejnego

> dysku.

3. Utworzenie systemu plików XFS

```
sudo mkfs.xfs /dev/mapper/vgDocker-lvDocker
```

Przykład:

```
meta-data=/dev/mapper/vgDocker-lvDocker isize=512    agcount=4, agsize=26214400 blks
data     =                       bsize=4096   blocks=104857600, imaxpct=25
naming   =version 2              bsize=4096   ascii-ci=0, ftype=1
log      =internal log           bsize=4096   blocks=51200, version=2
realtime =none                   extsz=4096   blocks=0, rtextents=0
```

XFS jest domyślnym systemem plików Rocky Linux 9 i tego samego używamy na pozostałych wolumenach opisanych w Instalacja Rocky Linux z własnym nazewnictwem grup wolumenów: jeden system plików w całej firmie, żeby nie pamiętać dwóch zestawów narzędzi diagnostycznych.

4. Wpis w /etc/fstab i montowanie

Sprawdzamy UUID nowego systemu plików:

```
sudo blkid /dev/mapper/vgDocker-lvDocker
```

Przykład:

```
/dev/mapper/vgDocker-lvDocker: UUID="3f7a9c1e-8b2d-4e5f-9a1c-6d8e2f4b7a9c" TYPE="xfs"
```

Dopisujemy wpis do `/etc/fstab`:

```
UUID=3f7a9c1e-8b2d-4e5f-9a1c-6d8e2f4b7a9c  /docker  xfs  defaults  0 2
```

> (gdzie `UUID` to wartość odczytana w poprzednim kroku, indywidualna dla każdego wolumenu)

Zakładamy punkt montowania i montujemy bez restartu:

```
sudo mkdir -p /docker
sudo mount -a
```

5. Weryfikacja montowania

```
df -h /docker
```

Przykład:

```
Filesystem                     Size  Used Avail Use% Mounted on
/dev/mapper/vgDocker-lvDocker  400G   30M  400G   1% /docker
```

6. Repozytorium Docker CE

```
sudo dnf config-manager --add-repo https://download.docker.com/linux/rhel/docker-ce.repo
```

> **Pojęcie: Docker CE.** Community Edition, darmowa wersja Docker Engine, w odróżnieniu od płatnej wersji Enterprise sprzedawanej z dodatkowym wsparciem i narzędziami do dużych wdrożeń. W Zielonej korzystamy wyłącznie z CE.

7. Instalacja Docker Engine z wtyczkami

```
sudo dnf install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

8. Konfiguracja demona: data-root i rotacja logów

Zanim pierwszy raz uruchomimy usługę, wskazujemy demonowi własną lokalizację danych i limit logów, żeby nie trzeba było tego robić później na hoście z uruchomionymi już kontenerami:

```
sudo mkdir -p /docker/lib
```

```
sudo vim /etc/docker/daemon.json
```

```
{
  "data-root": "/docker/lib",
  "storage-driver": "overlay2",
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m",
    "max-file": "3"
  }
}
```

> **Pojęcie: `/docker/lib`.** Katalog na własne dane demona Dockera: pobrane obrazy, warstwy, sieci, anonimowe wolumeny. To trzecie miejsce pod `/docker`, obok `/docker/apps` (dane aplikacji) i `/docker/compose` (pliki Compose), bo trzyma wszystko, czym zarządza sam Docker, a nie żadna konkretna aplikacja. Leży na tym samym wolumenie `vgDocker/lvDocker` co reszta, bo obrazy i warstwy potrafią zająć równie dużo miejsca co dane aplikacji.

> Na hoście postawionym przed ujednoliceniem tej konwencji mogła się przyjąć inna nazwa tego katalogu, na przykład `/docker/docker-data`. Nowe instalacje zakładamy z `/docker/lib`, starych hostów nie migrujemy bez potrzeby, patrz Utrzymanie hosta dockerowego.

Limit `max-size` i `max-file` w `log-opts` obowiązuje globalnie dla nowo tworzonych kontenerów: każdy log rotuje po 10 MB, trzymając maksymalnie trzy pliki historii, więc żaden pojedynczy kontener nie zapełni `/docker/lib` samymi logami.

9. Uruchomienie usługi

```
sudo systemctl enable --now docker
```

10. Weryfikacja nowej lokalizacji danych

```
docker info -f '{{ .DockerRootDir }}'
```

Przykład:

```
/docker/lib
```

11. Dodanie użytkownika do grupy docker

```
sudo usermod -aG docker nazwa_usera
```

> Konto dodane do grupy `docker` ma w praktyce pełne uprawnienia roota na tym hoście, mimo że formalnie rootem nie jest: demon Dockera działa jako root, a każdy, kto może z nim rozmawiać, może zamontować dowolny katalog hosta do kontenera i odczytać go bez ograniczeń. Dodajemy do tej grupy wyłącznie konta, które faktycznie mają administrować kontenerami, nigdy odruchowo, „żeby nie pisać sudo".

Zmiana grupy wymaga ponownego zalogowania konta, żeby nowa przynależność została uwzględniona w bieżącej sesji.

12. Firewalld i Docker

```
sudo firewall-cmd --list-all
```

Docker zarządza własnymi regułami `iptables` niezależnie od `firewalld` i domyślnie omija strefy firewalld przy publikowaniu portów kontenerów: port opublikowany przez `-p` albo przez `ports:` w compose jest osiągalny z zewnątrz, nawet jeśli `firewalld` w ogóle go nie wymienia jako otwartego. Świadome ograniczenie ruchu do konkretnych portów i adresów robimy przez sam Docker, adresem `127.0.0.1` przy publikowaniu portu, zgodnie ze stroną Jak trzymamy aplikacje w Dockerze: konwencje Zielonej, nie przez dopisywanie reguł do `firewalld`, które i tak nie obejmą ruchu przekierowanego przez Docker do kontenera.

13. Test poprawności instalacji

```
sudo docker run hello-world
```

Przykład:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.
```

Weryfikacja

```
docker compose version
```

Przykład:

```
Docker Compose version v2.29.1
```

```
docker info -f '{{ .DockerRootDir }}'
```

Przykład:

```
/docker/lib
```

Wycofanie zmian

Odinstalowanie Dockera bez usuwania danych z `/docker`:

```
sudo systemctl disable --now docker
```

```
sudo dnf remove -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

Wolumen `vgDocker/lvDocker` i wpis w `/etc/fstab` zostają nietknięte: ponowna instalacja pakietu i wskazanie tego samego `data-root` w `daemon.json` odzyskuje wszystkie obrazy i kontenery bez ponownego pobierania.

Gdy coś nie zadziała

Objaw	Przyczyna	Co robimy
`docker info` pokazuje `/var/lib/docker` mimo zmiany w `daemon.json`	plik `daemon.json` ma błąd składni JSON i demon zignorował całą konfigurację	`python3 -m json.tool /etc/docker/daemon.json`, poprawiamy i restartujemy usługę
Kontener publikuje port mimo zamkniętej strefy w firewalld	Docker dopisuje własne reguły do `iptables` z pominięciem firewalld	ograniczamy publikację portu do `127.0.0.1`, nie liczymy na firewalld
`mount -a` zwraca błąd po dopisaniu wpisu do fstab	literówka w UUID albo w nazwie systemu plików	`blkid` ponownie, porównujemy z wpisem w `/etc/fstab` znak po znaku

Objaw	Przyczyna	Co robimy
Po `usermod -aG docker` polecenie `docker ps` nadal zwraca `permission denied`	użytkownik nie wylogował się i nie zalogował ponownie	nowa sesja logowania, ewentualnie `newgrp docker` na czas bieżącej sesji

Powiązane: Jak trzymamy aplikacje w Dockerze: konwencje Zielonej · Instalacja Rocky Linux z własnym nazewnictwem grup wolumenów · Utrzymanie hosta dockerowego · Pierwsza aplikacja w Compose krok po kroku · Połączenie z serwerem z Windowsa: SSH, Windows Terminal i PuTTY

Źródła: docs.docker.com, rozdział „Install Docker Engine on RHEL" · docs.docker.com, rozdział „Configure the Docker daemon" · man 5 fstab

Pierwsza aplikacja w Compose krok po kroku

Typ: procedura **Dotyczy: Docker Compose v2, obraz nginx:1.27-alpine**

Czas: ~25 min

Efekt: własna, prosta strona WWW uruchomiona w kontenerze przez Docker Compose, od założenia katalogów po zatrzymanie i usunięcie, z każdym krokiem sprawdzonym wynikiem, zanim przejdziemy do prawdziwych aplikacji dalej w tej książce.

Skróty użyte na tej stronie (3)

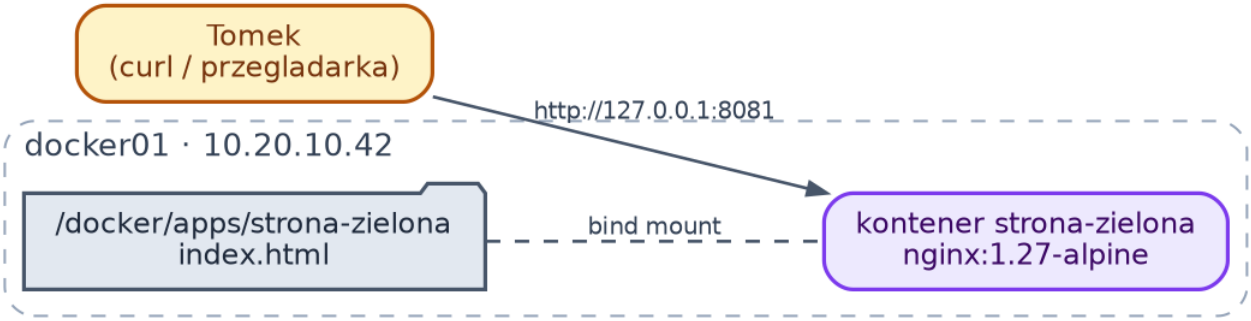
Skrót	Co znaczy
CLI	Command Line Interface: praca przez wpisywanie komend w terminalu
HTML	HyperText Markup Language: format tekstowy stron WWW, tu używany jako format raportu GPO do przeglądania w przeglądarce
WWW	World Wide Web: strony i aplikacje otwierane w przeglądarce po HTTP albo HTTPS

Zanim zaczniemy

Procedura zakłada działający Docker Engine z wtyczką Compose, zainstalowany według strony Instalacja Dockera na Rocky Linux 9 z osobnym wolumenem /docker, i katalogi zakładane zgodnie z konwencją opisaną na stronie Jak trzymamy aplikacje w Dockerze: konwencje Zielonej. Komendy wykonujemy z CLI bash jako user z uprawnieniami sudo albo jako user należący do grupy `docker`.

> 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.

Aplikacją testową jest zwykły serwer WWW Nginx, serwujący jedną stronę HTML z krótką informacją o Zielonej: wystarczająco prosty, żeby skupić się na samym Compose, a nie na konfiguracji aplikacji.



Katalog na hoście podpięty bind mountem do kontenera nginx, odpowiadający na zapytanie curl przez port 127.0.0.1 (kliknięcie otwiera pełny rozmiar)

Przebieg

1. Założenie katalogów

```
sudo mkdir -p /docker/apps/strona-zielona /docker/compose/strona-zielona
```

Przykład:

```
docker01:~$ ls -ld /docker/apps/strona-zielona /docker/compose/strona-zielona
drwxr-xr-x. 2 root root 6 wrz 27 11:00 /docker/apps/strona-zielona
drwxr-xr-x. 2 root root 6 wrz 27 11:00 /docker/compose/strona-zielona
```

2. Plik strony HTML

> **Pojęcie: bind mount.** Bezpośrednie podpięcie katalogu z hosta do wnętrza kontenera: to, co leży w `/docker/apps/strona-zielona` na hoście, kontener widzi pod wskazaną ścieżką u siebie, i odwrotnie. Zmiana pliku na hoście jest widoczna w kontenerze natychmiast, bez ponownego budowania czegokolwiek.

```
sudo tee /docker/apps/strona-zielona/index.html > /dev/null << 'EOF'
<!DOCTYPE html>
```

```
<html lang="pl">
<head><meta charset="utf-8"><title>Zielona sp. z o.o.</title></head>
<body>
<h1>Zielona sp. z o.o.</h1>
<p>Hurtownia artykulow spozywczych, centrala w Pamiatkowie, magazyn w Swarzedzu.</p>
</body>
</html>
EOF
```

3. Plik compose.yaml

```
sudo vim /docker/compose/strona-zielona/compose.yaml
```

```
services:
  server:
    image: nginx:1.27-alpine
    container_name: strona-zielona
    restart: unless-stopped
    volumes:
      - /docker/apps/strona-zielona/index.html:/usr/share/nginx/html/index.html:ro
    ports:
      - "127.0.0.1:8081:80"
```

> **Pojęcie: obraz.** Gotowy szablon systemu plików aplikacji, tutaj `nginx:1.27-alpine`: pobrany raz z internetowego rejestru, leży potem lokalnie na hoście i służy do uruchamiania dowolnej liczby kontenerów bez ponownego pobierania. Numer `1.27-alpine` to przypięta wersja: dokładnie ten sam obraz przy każdym kolejnym `pull`, zgodnie z konwencją z poprzedniej strony.

Port publikujemy na `127.0.0.1`, nie na wszystkich interfejsach, z tego samego powodu co każda inna aplikacja w Zielonej: docelowo stanie przed nią Nginx frontowy z certyfikatem, a na czas tego ćwiczenia sprawdzamy stronę z samego hosta.

4. Pierwsze uruchomienie

```
cd /docker/compose/strona-zielona
docker compose up -d
```

Przykład:

```
[+] Running 2/2
 ✔ Network strona-zielona_default   Created
 ✔ Container strona-zielona         Started
```

5. Sprawdzenie stanu kontenera

```
docker compose ps
```

Przykład:

```
NAME             IMAGE               STATUS
strona-zielona    nginx:1.27-alpine   Up 8 seconds
```

6. Sprawdzenie odpowiedzi serwera

```
curl -s http://127.0.0.1:8081 | head -n 5
```

Przykład:

```
<!DOCTYPE html>
<html lang="pl">
<head><meta charset="utf-8"><title>Zielona sp. z o.o.</title></head>
<body>
<h1>Zielona sp. z o.o.</h1>
```

7. Podgląd logów

```
docker compose logs -f
```

Przykład:

```
strona-zielona  | 10.20.10.42 - - [27/Sep/2026:11:05:02 +0000] "GET / HTTP/1.1" 200 612 "-"
"curl/8.5.0"
```

Przerywamy podgląd kombinacją `Ctrl+C`, kontener działa dalej w tle.

8. Zmiana strony i ponowne zastosowanie

Zmieniamy treść pliku HTML na hoście:

```
sudo sed -i 's/Pamiatkowie/Pamiatkowie (od 1998 roku)/' /docker/apps/strona-zielona/index.html
```

Odświeżenie strony pokazuje zmianę natychmiast, bo plik jest bind mountem, nie kopią zamkniętą wewnątrz obrazu. `docker compose up -d` po samej zmianie pliku HTML nic nie zmienia w stanie kontenera, bo nie zmieniliśmy `compose.yaml`:

```
docker compose up -d
```

Przykład:

```
[+] Running 1/1
 ✔ Container strona-zielona  Running
```

Compose porównuje aktualną konfigurację z tą, z jaką kontener już działa, i nie robi nic, gdy nie ma różnicy: `up -d` jest bezpieczne do wywołania wielokrotnie.

9. Zmiana w compose.yaml i ponowne zastosowanie

Zmieniamy port hosta z `8081` na `8082` w `compose.yaml`, zapisujemy plik i uruchamiamy ponownie:

```
docker compose up -d
```

Przykład:

```
[+] Running 1/1
 ✔ Container strona-zielona  Recreated
```

Tym razem Compose wykrywa różnicę w `ports` i podmienia kontener na nowy, z nową konfiguracją: `Recreated` zamiast `Running` w wyniku.

10. Zatrzymanie i usunięcie

```
docker compose down
```

Przykład:

```
[+] Running 2/2
 ✔ Container strona-zielona       Removed
 ✔ Network strona-zielona_default Removed
```

Plik HTML w `/docker/apps/strona-zielona` przeżywa `down`, bo leży na hoście, nie w kontenerze: usuwamy go ręcznie, tylko jeśli faktycznie kończymy z tym ćwiczeniem.

```
sudo rm -rf /docker/apps/strona-zielona /docker/compose/strona-zielona
```

Weryfikacja

```
docker compose ps
```

Przykład:

```
NAME   IMAGE   STATUS
```

Pusty wynik, bez żadnego wiersza, potwierdza, że żadna usługa z tego katalogu już nie działa.

Wycofanie zmian

Zatrzymanie bez usuwania danych, jeśli chcemy tylko chwilowo wyłączyć stronę:

```
docker compose stop
```

Ponowne uruchomienie tego samego kontenera, bez odtwarzania niczego od zera:

```
docker compose start
```

Gdy coś nie zadziała

Objaw	Przyczyna	Co robimy

`curl` zwraca `Connection refused` na `127.0.0.1:8081`	kontener jeszcze się uruchamia albo port zajęty przez inną usługę	`docker compose logs`, `ss -tlnp \`
Zmiana w `index.html` niewidoczna po odświeżeniu w przeglądarce	przeglądarka pokazuje stronę z pamięci podręcznej	twarde odświeżenie kombinacją `Ctrl+Shift+R` albo sprawdzenie przez `curl` z terminala
`docker compose up -d` po zmianie portu nie tworzy kontenera na nowo	literówka w pliku, Compose wczytał starą, niezmienioną wersję	`docker compose config`, sprawdzamy, co Compose faktycznie odczytał
`Permission denied` przy zapisie do `index.html` przez `sed`	plik należy do innego użytkownika niż ten, który wykonuje komendę	`sudo` przed komendą edytującą plik na hoście

Powiązane: Jak trzymamy aplikacje w Dockerze: konwencje Zielonej · Docker Compose · Codzienne komendy

Źródła: docs.docker.com, rozdział „Compose file reference" · hub.docker.com, obraz `nginx`