Sieci i utrzymanie

Sterowniki sieci, SNAT oraz utrzymanie hosta.

- Sieci kontenerów i SNAT
- Utrzymanie hosta dockerowego
- Kopia zapasowa aplikacji dockerowych

Sieci kontenerów i SNAT

Poziom: operacyjny **Dotyczy: Docker Engine 24+** **Czas: ~25 min**

Domyślna sieć Dockera wystarcza do pierwszych testów, ale prawdziwa infrastruktura zwykle wymaga świadomej decyzji, jak kontenery mają się komunikować między sobą i jak ich ruch wychodzący ma się prezentować reszcie sieci.

Skróty użyte na tej stronie (7)

Skrót	Co znaczy
DNAT	Destination Network Address Translation: podmiana adresu docelowego pakietu, używana do przekierowania ruchu z zewnątrz do serwera wewnątrz sieci
DNS	Domain Name System: książka telefoniczna internetu, zamienia nazwy na adresy IP
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
MAC	Media Access Control: unikalny adres fizyczny karty sieciowej, przypisany fabrycznie
NAT	Network Address Translation: podmiana adresów IP na granicy sieci
SNAT	Source NAT: podmiana adresu nadawcy w ruchu wychodzącym
WMS	Warehouse Management System: system zarządzania magazynem

Fiszka

ZMIENIA SYSTEM

Utworzenie własnej sieci: docker network create

```
docker network create NazwaSieci
```

Przykład:

```
a1b2c3d4e5f67890abcdef1234567890abcdef1234567890abcdef12345678
```

docker = narzędzie do kontenerów

□□ TYLKO ODCZYT

Wylistowanie sieci: docker network ls

```
docker network ls
```

Przykład:

```
NETWORK ID     NAME          DRIVER    SCOPE
a1b2c3d4e5f6   bridge        bridge    local
f6e5d4c3b2a1   NazwaSieci    bridge    local
c3b2a1f6e5d4   host          host      local
d4c3b2a1f6e5   none          null      local
```

□□ **docker** = narzędzie do kontenerów

□□ TYLKO ODCZYT

Wylistowanie reguł NAT dodanych przez Dockera: iptables -t nat -L

```
sudo iptables -t nat -L DOCKER -n
```

Reguły dodane przez Dockera widać bezpośrednio w tablicach `iptables`.

Przykład:

```
Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            tcp dpt:8080
to:172.17.0.2:80
```

□□ **NAT** = Network Address Translation: podmiana adresów IP na granicy sieci · **iptables** = starsza zapora sieciowa jądra Linuksa, sterowana regułami w tabelach i łańcuchach

□□ TYLKO ODCZYT

Włączenie przekazywania ruchu IPv4 na hoście: sysctl -w

```
sudo sysctl -w net.ipv4.ip_forward=1
```

Warunkiem działania SNAT jest włączone przekazywanie pakietów w jądrze — bez tego host nie routuje ruchu między interfejsami w ogóle.

IPv4 = Internet Protocol wersja 4: adresy zapisane czterema liczbami z kropkami, na przykład 10.20.10.77 · **SNAT** = Source NAT: podmiana adresu nadawcy w ruchu wychodzącym · **sysctl** = odczytaj albo zmień parametr jądra Linuksa w czasie działania systemu, tu limit vm.max_map_count wymagany przez OpenSearch

TYLKO ODCZYT

Utrwalenie ustawienia ip_forward po restarcie

```
net.ipv4.ip_forward = 1
```

(wpis dodajemy do `/etc/sysctl.conf` albo do własnego pliku w `/etc/sysctl.d/`)

TYLKO ODCZYT

Ponowne wczytanie konfiguracji sysctl: sysctl -p

```
sudo sysctl -p
```

Wczytujemy konfigurację `sysctl` ponownie, bez restartu.

sysctl = odczytaj albo zmień parametr jądra Linuksa w czasie działania systemu, tu limit vm.max_map_count wymagany przez OpenSearch

TYLKO ODCZYT

Zmapowanie ruchu wychodzącego z sieci kontenera na adres interfejsu: iptables SNAT

```
sudo iptables -t nat -I POSTROUTING -p all -s 172.18.0.0/29 ! -d 172.18.0.0/29 -j SNAT
--to-source 10.20.10.42
```

Reguła mówi: cały ruch (`-p all`) pochodzący z podsieci dockerowej `172.18.0.0/29`, którego adres docelowy leży poza tą samą podsiecią (`! -d 172.18.0.0/29` — warunek wykluczający ruch wewnętrzny między kontenerami tej sieci), ma dostać podmieniony adres źródłowy na `10.20.10.42` — adres `docker01` przypisany do `NazwaInterfejsu` hosta, przez który ten ruch faktycznie ma wyjść na zewnątrz.

> Reguła SNAT dopisana przed regułą `MASQUERADE`, którą Docker dodaje automatycznie dla własnych sieci, ma pierwszeństwo — kolejność reguł w łańcuchu `POSTROUTING` decyduje, która z nich faktycznie obsłuży dany pakiet, dlatego dopisujemy ją poleceniem `-I` (insert na początek), nie `-A` (append na koniec).

□□ **SNAT** = Source NAT: podmiana adresu nadawcy w ruchu wychodzącym · **iptables** = starsza zapora sieciowa jądra Linuksa, sterowana regułami w tabelach i łańcuchach

□□ TYLKO ODCZYT

Weryfikacja reguły w tablicy POSTROUTING: iptables -t nat -L

```
sudo iptables -t nat -L POSTROUTING -n
```

Weryfikujemy, że reguła faktycznie trafiła do tablicy.

Przykład:

```
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
SNAT       all  --  172.18.0.0/29        !172.18.0.0/29       to:10.20.10.42
MASQUERADE  all  --  172.18.0.0/29         0.0.0.0/0
```

□□ **iptables** = starsza zapora sieciowa jądra Linuksa, sterowana regułami w tabelach i łańcuchach

□□ ZMIENIA SYSTEM

Utrwalenie reguł iptables po restarcie hosta: iptables-save

```
sudo iptables-save > /etc/sysconfig/iptables
```

Utrwalamy reguły `iptables` po restarcie hosta (na systemach z `iptables-services`).

Z życia wzięte

Z ŻYCIA WZIĘTE

System zewnętrzny dostawcy WMS filtruje ruch przychodzący po adresie IP, a kontenery na `docker01` łączą się z nim z losowego adresu wybranego przez routing hosta, więc połączenie jest odrzucane. Tomek dodaje regułę SNAT wymuszającą stały adres `10.20.10.42` dla ruchu z podsieci dockerowej, ale wstawia ją komendą `-A` zamiast `-I`. Reguła `MASQUERADE` Dockera obsługuje ruch wcześniej i SNAT dalej nie działa, dopóki Tomek nie przenosi jej na początek łańcucha.

Czym jest sieć kontenera

Pojęcie: sieć kontenera. Każdy kontener domyślnie dostaje własną przestrzeń nazw sieciową jądra Linuksa — własny interfejs sieciowy, własną tablicę routingu, własny adres IP, odizolowane od hosta i od innych kontenerów. Docker zarządza tą izolacją przez własny model sieci, w którym kontenery komunikują się ze sobą wyłącznie wtedy, gdy jawnie podłączymy je do tej samej sieci wirtualnej — sama obecność dwóch kontenerów na tym samym hoście nic o ich wzajemnej widoczności nie mówi.

Sterowniki sieci

Docker udostępnia kilka sterowników (`driver`) sieci, różniących się tym, na jakim poziomie odbywa się izolacja i routing ruchu.

bridge to domyślny sterownik dla pojedynczego hosta — Docker tworzy wirtualny przełącznik (most) w jądrze systemu, do którego podłącza interfejsy sieciowe kontenerów; ruch między kontenerami w tej samej sieci bridge przechodzi przez ten most, a ruch na zewnątrz hosta — przez NAT skonfigurowany automatycznie w `iptables`.

host usuwa izolację sieciową całkowicie — kontener współdzieli stos sieciowy bezpośrednio z hostem, widzi te same interfejsy i porty, bez żadnego mapowania. Zysk to brak narzutu na NAT i możliwość użycia dowolnego zakresu portów bez jawnego publikowania, kosztem utraty izolacji — dwa kontenery w trybie `host` nie mogą nasłuchiwać na tym samym porcie.

none wyłącza sieć kontenera całkowicie, poza interfejsem `loopback` — stosowany, gdy kontener nie potrzebuje żadnej łączności sieciowej albo gdy sieć ma zostać skonfigurowana ręcznie z zewnątrz.

macvlan nadaje kontenerowi własny adres MAC i traktuje go jak fizyczne urządzenie bezpośrednio podpięte do sieci lokalnej hosta — kontener dostaje adres IP z tej samej puli co inne urządzenia w

sieci, widoczny dla reszty infrastruktury bez pośrednictwa NAT Dockera. Przydatny, gdy urządzenia zewnętrzne (np. system monitoringu) muszą widzieć kontener jako pełnoprawny host w sieci, a nie jako coś ukrytego za adresem hosta Dockera.

overlay rozciąga jedną sieć wirtualną ponad wieloma hostami fizycznymi — kontenery na różnych maszynach w tej samej sieci overlay komunikują się tak, jakby siedziały na jednym hoście. Ma sens w trybie Swarm albo innej orkiestracji wielohostowej; na pojedynczym hoście bez orkiestracji nie ma zastosowania.

Domyślny bridge kontra sieć użytkownika

> **Pojęcie: rozwiązywanie nazw w sieci Dockera.** Domyślna sieć `bridge`, do której trafia kontener uruchomiony bez jawnie wskazanej sieci, **nie** zapewnia rozwiązywania nazw kontenerów — dwa kontenery w tej sieci widzą się nawzajem tylko po adresie IP, który może się zmienić przy każdym restarcie. Sieć użytkownika (utworzona przez `docker network create` albo automatycznie przez `docker compose`) ma wbudowany wewnętrzny DNS — kontenery w niej widzą się po nazwie kontenera albo nazwie usługi, niezależnie od tego, jaki adres IP akurat dostały.

To jest jeden z praktycznych powodów, dla których w realnych wdrożeniach niemal zawsze tworzy się własną sieć zamiast polegać na domyślnej — bez tego każda zmiana adresu IP kontenera (a zdarza się to przy każdym jego restarcie) wymagałaby ręcznej aktualizacji konfiguracji łączących się z nim usług.

Publikowanie portów i iptables

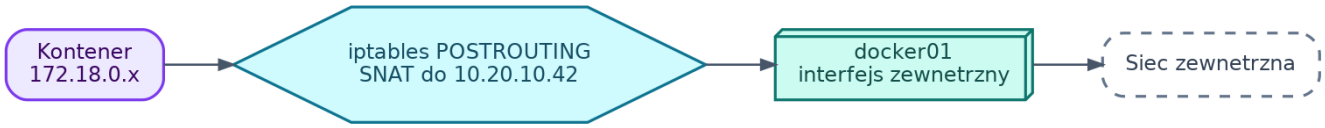
> **Pojęcie: publikowanie portu.** Flaga `-p NumerPortu:80` (albo sekcja `ports` w compose) nie jest zwykłym „otwarciem" portu — to polecenie dla Dockera, żeby dopisał regułę NAT do `iptables` hosta, przekierowującą ruch przychodzący na wskazany port hosta do adresu i portu kontenera wewnątrz sieci bridge. Bez tego wpisu port kontenera jest osiągalny wyłącznie z wnętrza sieci dockerowej, nigdy z zewnątrz hosta, niezależnie od tego, na czym proces wewnątrz kontenera faktycznie nasłuchuje.

SNAT ruchu wychodzącego na konkretny interfejs

Publikowanie portu (DNAT) rozwiązuje ruch przychodzący do kontenera. Osobnym zagadnieniem jest ruch wychodzący z kontenera, gdy host ma więcej niż jeden interfejs sieciowy i zależy nam na tym, żeby aplikacje zewnętrzne widziały ten ruch jako pochodzący z konkretnego adresu IP przypisanego do konkretnego interfejsu, nie z domyślnego adresu, który wybrałby routing hosta.

> **Pojęcie: SNAT (Source NAT).** Reguła `iptables`, która podmienia adres źródłowy pakietu wychodzącego na wskazany adres — w kontekście Dockera pozwala to wymusić, żeby cały ruch z sieci danego kontenera (albo całej podsieci dockerowej) wychodził z konkretnego adresu IP przypisanego do wybranego interfejsu hosta, zamiast adresu, który jądro wybrałoby domyślnie na podstawie tablicy routingu.

Typowy przypadek użycia: kontener musi łączyć się z systemem zewnętrznym, który filtruje ruch przychodzący po adresie źródłowym (whitelisting IP) — jeśli host ma kilka interfejsów, a domyślny routing wysłałby ruch z „niewłaściwego" adresu, jawna reguła SNAT wymusza właściwy.



Ruch z podsieci kontenerów przechodzi przez regułę SNAT w łańcuchu POSTROUTING, wychodząc na zewnątrz z adresu interfejsu docker01 (kliknięcie otwiera pełny rozmiar)

Częste błędy

Zakładanie, że kontenery w domyślnej sieci bridge widzą się po nazwie. Rozwiązywanie nazw działa tylko w sieciach użytkownika — w domyślnym bridge trzeba znać adres IP, który może się zmienić przy każdym restarcie kontenera.

Dodanie reguły SNAT komendą `-A` zamiast `-I`. Reguła dopisana na koniec łańcucha `POSTROUTING` może nigdy nie zostać wykonana, jeśli wcześniejsza reguła `MASQUERADE` dodana przez Dockera już dopasowała i obsłużyła ten sam ruch.

Zapomnienie o `net.ipv4.ip_forward=1`. Bez włączonego przekazywania pakietów host w ogóle nie routuje ruchu między interfejsami — reguła SNAT jest wtedy poprawnie zapisana w `iptables`, ale nigdy nie zostaje wywołana, bo pakiet nie dotrze do łańcucha `POSTROUTING` w oczekiwany sposób.

□□ Ćwiczenia

> Zadania sprawdzające.

1. Wyjaśniamy różnicę między domyślną siecią `bridge` a siecią użytkownika w kontekście rozwiązywania nazw kontenerów.
2. Dobieramy sterownik sieci (bridge, host, macvlan, overlay) do trzech scenariuszy: pojedynczy kontener na jednym hoście bez specjalnych wymagań, kontener, który system monitoringu ma widzieć jako pełnoprawne urządzenie w sieci lokalnej, aplikacja rozproszona na kilku hostach w trybie Swarm.
3. Wyjaśniamy, po co jest warunek `! -d 172.18.0.0/29` w regule SNAT z tej strony i co by się stało bez niego.
4. Host ma dwa interfejsy sieciowe, a ruch wychodzący z kontenerów wychodzi przez „zły" interfejs mimo dodanej reguły SNAT. Wskazujemy dwie rzeczy, które warto sprawdzić w pierwszej kolejności.

Powiązane: Obraz, kontener, warstwa, wolumen: cztery pojęcia · Docker Compose · Codzienne komendy

Utrzymanie hosta dockerowego

Poziom: operacyjny **Dotyczy: Docker Engine 24+, RHEL/CentOS Stream**

Czas: ~35 min

Host dockerowy zostawiony bez przeglądu ma dwa typowe końce — pełny dysk pod `/var/lib/docker` albo kontener, który zjada całą dostępną pamięć hosta, bo nikt mu nie ustawił limitu.

Skróty użyte na tej stronie (5)

Skrót	Co znaczy
CLI	Command Line Interface: praca przez wpisywanie komend w terminalu
GB	Gigabyte: miliard bajtów, jednostka pojemności pamięci i plików, tu najczęściej rozmiar modelu albo pamięci karty graficznej
JSON	JavaScript Object Notation: tekstowy format danych z klamrami i cudzysłowami
OOM	Out Of Memory: brak pamięci; OOM killer to mechanizm jądra zabijający proces, gdy jej zabraknie
RHEL	Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa

Fiszka

Komendy wykonujemy z CLI bash z uprawnieniami root (sudo).

ZMIENIA SYSTEM

Instalacja Docker Engine z repozytorium

> 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, razem z instalacją tego wolumenu od zera, na stronach Jak trzymamy aplikacje w Dockerze: konwencje Zielonej i Instalacja Dockera na Rocky Linux 9 z osobnym wolumenem /docker.

Sekwencja ma sens wykonana w całości i w tej kolejności — repozytorium przed instalacją pakietu.

```
# Dodanie oficjalnego repozytorium Dockera (RHEL)
sudo dnf config-manager --add-repo https://download.docker.com/linux/rhel/docker-ce.repo

# Na systemach z rodziny CentOS Stream repozytorium ma inny adres:
# sudo dnf config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo

# Instalacja silnika razem z wtyczkami buildx i compose
sudo dnf install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

dnf = Dandified YUM: menedżer pakietów RHEL/Rocky

TYLKO ODCZYT

Edycja pliku konfiguracyjnego demona: vim

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

Tworzymy albo edytujemy plik konfiguracyjny demona.

vim = Vi IMproved: ulepszony edytor vi

TYLKO ODCZYT

Wklejenie konfiguracji nowej lokalizacji danych: data-root

```
{
  "storage-driver": "overlay2",
  "data-root": "/docker/docker-data"
}
```

Wklejamy konfigurację wskazującą nową lokalizację.

> Sterownik pamięci masowej deklarujemy zawsze jako `overlay2` — starszy `overlay` (bez dwójki) nie jest już wspierany przez aktualne wersje silnika.

ZMIENIA SYSTEM

Restart usługi po zmianie konfiguracji: systemctl restart

```
sudo systemctl restart docker
```

Restart usługi, żeby demon wczytał nową konfigurację.

systemctl = system control: sterowanie usługami systemd

ZMIENIA SYSTEM

Włączenie usługi na świeżej instalacji: systemctl enable --now

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

Jeśli instalacja jest świeża i usługa jeszcze nie była uruchamiana, włączamy ją zamiast restartować.

systemctl = system control: sterowanie usługami systemd

TYLKO ODCZYT

Sprawdzenie nowej lokalizacji danych: docker info

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

Sprawdzamy, czy Docker faktycznie używa nowej lokalizacji.

Przykład:

```
/docker/docker-data
```

docker = narzędzie do kontenerów

□□ ZMIENIA SYSTEM

Test poprawności instalacji: docker run hello-world

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

Uruchamiamy kontener testowy, żeby sprawdzić poprawność instalacji.

Przykład:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
Digest: sha256:926fac19d22aa2d60f1a276b66a20eb765fbeea2db5dbdaafeb45ad8ce815d8
Status: Downloaded newer image for hello-world:latest

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

□□ **docker** = narzędzie do kontenerów

□□ TYLKO ODCZYT

Sprawdzenie łącznej zajętości katalogu danych: du -sh

```
sudo du -sh /docker/docker-data
```

Przykład:

```
28G□/docker/docker-data
```

□□ **du** = disk usage: ile miejsca zajmuje katalog

□□ TYLKO ODCZYT

Rozbicie zajętości na kategorie przez Dockera: docker system df -v

```
docker system df -v
```

Flaga `-v` dokłada do standardowego podsumowania (patrz Codzienne komendy) listę pojedynczych obrazów, kontenerów i wolumenów z ich indywidualnym rozmiarem — przydatne, gdy samo zbiorcze podsumowanie nie mówi, który konkretnie element warto

usunąć.

docker = narzędzie do kontenerów

OSTROŻNIE · NIEODWRACALNE

Usunięcie nieużywanych zasobów agresywnie: docker system prune -a --volumes

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

Flaga `-a` dokłada do kasowania wszystkie obrazy niewykorzystywane przez żaden aktualnie działający kontener — łącznie z obrazami oznaczonymi tagiem, które po prostu nie mają w tej chwili uruchomionego kontenera, ale mogą być potrzebne za chwilę (np. poprzednia wersja przed wdrożeniem). Flaga `--volumes` dokłada kasowanie wolumenów nienależących do żadnego istniejącego kontenera — w tym danych, które ktoś zostawił celowo, licząc na ich trwałość między wdrożeniami.

> Komenda z obiema flagami naraz jest nieodwracalna i nie rozróżnia „nieużywane, bo śmieć" od „nieużywane teraz, ale potrzebne za godzinę" — na hoście produkcyjnym uruchamiamy ją wyłącznie po ręcznym przejrzeniu listy z `docker system df -v`, nigdy odruchowo.

docker = narzędzie do kontenerów

TYLKO ODCZYT

Ograniczenie rozmiaru logów globalnie: log-driver w daemon.json

```
{
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m",
    "max-file": "3"
  }
}
```

Ograniczenie rozmiaru i liczby przechowywanych plików logu globalnie, dla wszystkich nowych kontenerów, przez `/etc/docker/daemon.json`. Po restarcie demona każdy nowo

tworzony kontener rotuje log po osiągnięciu 10 MB, trzymając maksymalnie 3 pliki historii — ustawienie nie działa wstecz na kontenery już istniejące, tylko na nowo tworzone.

□□ ZMIENIA SYSTEM

Ograniczenie rozmiaru logów pojedynczego kontenera: docker run --log-opt

```
docker run -d --log-opt max-size=10m --log-opt max-file=3 NazwaObrazu
```

Ten sam limit da się ustawić pojedynczemu kontenerowi bez zmiany globalnej konfiguracji.

□□ **docker** = narzędzie do kontenerów

□□ ZMIENIA SYSTEM

Ograniczenie pamięci dostępnej dla kontenera: docker run --memory

```
docker run -d --memory=512m NazwaObrazu
```

Proces wewnątrz kontenera, który spróbuje przekroczyć ten limit, zostaje zabity przez OOM killer jądra — nie jest to łagodne ograniczenie, tylko twarda granica kończąca się zakończeniem procesu.

□□ **OOM** = Out Of Memory: brak pamięci; OOM killer to mechanizm jądra zabijający proces, gdy jej zabraknie · **docker** = narzędzie do kontenerów

□□ ZMIENIA SYSTEM

Ograniczenie liczby rdzeni CPU dostępnych dla kontenera: docker run --cpus

```
docker run -d --cpus=1.5 NazwaObrazu
```

Wartość ułamkowa oznacza limit w jednostkach rdzenia — `1.5` pozwala kontenerowi zużyć maksymalnie półtora rdzenia łącznego czasu procesora, niezależnie od tego, ile rdzeni ma cały host.

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

□□ TYLKO ODCZYT

Sprawdzenie ustawionych limitów działającego kontenera: docker inspect

```
docker inspect NazwaKontenera --format '{{ .HostConfig.Memory }} {{ .HostConfig.NanoCpus }}'
```

□□ **docker** = narzędzie do kontenerów

□□ TYLKO ODCZYT

Sprawdzenie, które kontenery wrócą same po restarcie: docker ps -a

```
docker ps -a --format 'table {{.Names}}\t{{.Status}}' --filter 'label!=tymczasowy'
```

Przy planowaniu utrzymania hosta (np. przed zaplanowanym restartem po aktualizacji jądra) sprawdzamy z wyprzedzeniem, które kontenery faktycznie wrócą same.

□□ **docker** = narzędzie do kontenerów

□□ TYLKO ODCZYT

Sprawdzenie aktualnie zainstalowanej wersji: docker version

```
docker version --format '{{.Server.Version}}'
```

□□ **docker** = narzędzie do kontenerów

□□ ZMIENIA SYSTEM

Aktualizacja pakietu silnika: dnf update

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

Aktualizujemy pakiet z tego samego repozytorium, z którego silnik został zainstalowany.

□□ **dnf** = Dandified YUM: menedżer pakietów RHEL/Rocky

Z życia wzięte

Z ŻYCIA WZIĘTE

W piątek wieczorem monitoring budzi Tomka alarmem o pełnym dysku na `docker01`, mimo że nikt niczego tego dnia nie wdrażał. Okazuje się, że kontener Grayloga loguje bardzo intensywnie od tygodnia bez limitu rozmiaru w `daemon.json`, a plik logu urósł do kilkunastu gigabajtów. Po dodaniu `max-size` i restarcie demona miejsce wraca, ale dopiero po ręcznym usunięciu starego pliku logu.

Restart usługi po aktualizacji: systemctl restart

```
sudo systemctl restart docker
```

Restartujemy usługę po aktualizacji.

> Aktualizacja restartuje demona, co zatrzymuje wszystkie kontenery bez ustawionej polityki automatycznego wznowienia — przed aktualizacją produkcyjnego hosta sprawdzamy dokładnie tą samą komendą co wyżej, które kontenery wymagają ręcznego `docker start` po restarcie.

Zmiana domyślnej lokalizacji plików Dockera

Pojęcie: `data-root`. Katalog, w którym demon Dockera trzyma wszystkie obrazy, kontenery i wolumeny nazwane — domyślnie `/var/lib/docker`. Zmiana tej ścieżki ma sens, gdy dysk systemowy jest mały, a dane Dockera mają rosnąć na osobnym, większym wolumenie.

Ryzyko system prune -a --volumes

Pojęcie: zakres `docker system prune`. Sama komenda bez flag kasuje wyłącznie to, co bezpiecznie nieużywane — zatrzymane kontenery, nieoznaczone obrazy, nieużywane sieci i cache budowania. Każda dodatkowa flaga poszerza ten zakres o coś, co bywa wciąż potrzebne.

Rotacja logów kontenerów

Pojęcie: sterownik logowania `json-file`. Domyślny sposób, w jaki Docker przechowuje standardowe wyjście i wyjście błędów kontenera — każda linia trafia jako wpis JSON do pliku na dysku hosta. Bez limitu rozmiaru plik logu kontenera, który działa miesiącami i intensywnie loguje, potrafi urosnąć do rozmiaru liczonego w dziesiątkach gigabajtów.

Limity pamięci i CPU

Pojęcie: limit zasobów kontenera. Bez jawnie ustawionego limitu kontener może zużyć całą dostępną pamięć albo cały dostępny czas procesora hosta — mechanizm `cgroups` jądra Linuksa, z którego korzysta Docker, pozwala ograniczyć oba te zasoby per kontener.



Docker ogranicza zasoby kontenera przez cgroups jądra, a OOM killer zabija proces dopiero po przekroczeniu limitu pamięci (kliknięcie otwiera pełny rozmiar)

Polityki restartu w kontekście utrzymania

Polityka restartu (opisana jako flaga `--restart` na stronie Codzienne komendy) ma znaczenie utrzymaniowe wykraczające poza pojedynczy kontener — przy restarcie hosta demon Dockera odtwarza wszystkie kontenery z polityką `always` albo `unless-stopped` automatycznie, bez interwencji, natomiast kontenery uruchomione bez tej flagi (albo z domyślnym `no`) zostają po restarcie w stanie zatrzymanym i wymagają ręcznego `docker start`.

Częste błędy

Zmiana `data-root` bez zatrzymania i przeniesienia istniejących danych. Sama zmiana ścieżki w `daemon.json` na hoście, który już ma obrazy i wolumeny pod starą lokalizacją, sprawia, że po restarcie Docker widzi pusty katalog danych — stare dane wciąż fizycznie leżą pod starą ścieżką, ale demon o nich nie wie, dopóki nie zostaną tam skopiowane ręcznie przed restartem.

Traktowanie `docker system prune -a --volumes` jako rutynowego sprzątania. To jest operacja do świadomego, jednorazowego użycia po przejrzeniu listy z `docker system df -v`, nie polecenie do wrzucenia w cykliczne zadanie `cron` bez nadzoru.

Brak limitu pamięci na kontenerach na współdzielonym hoście. Jeden kontener bez `--memory` potrafi w razie wycieku pamięci zabrać zasoby wszystkim pozostałym kontenerom na tym samym hoście, zanim OOM killer w ogóle zainterweniuje.

□□ Ćwiczenia

> Zadania sprawdzające.

1. Instalujemy Docker Engine z oficjalnego repozytorium na świeżym hoście RHEL i zmieniamy domyślną lokalizację danych na `/docker/docker-data`, weryfikując zmianę.
2. Wyjaśniamy, dlaczego `docker system prune -a --volumes` jest ryzykowniejsze niż `docker system prune` bez flag, i opisujemy, co sprawdzamy przed jego uruchomieniem.
3. Kontener loguje bardzo intensywnie i po tygodniu jego plik logu zajmuje 4 GB. Piszemy fragment `daemon.json`, który zapobiegnie temu w przyszłości dla nowych kontenerów.
4. Host ma zaplanowany restart po aktualizacji jądra. Opisujemy, jak sprawdzimy z wyprzedzeniem, które kontenery wrócą same, a które będą wymagać ręcznego uruchomienia.

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

Kopia zapasowa aplikacji dockerowych

Typ: procedura **Dotyczy: wszystkie aplikacje na docker01** **Czas: ~40 min**

Kopia zapasowa aplikacji dockerowej to zawsze dwie osobne rzeczy: pliki spod `/docker/apps` i `/docker/compose` oraz osobny zrzut bazy danych, jeśli aplikacja jej używa, zamiast kopiowania plików bazy w trakcie jej działania. Ta strona opisuje wspólny mechanizm dla wszystkich aplikacji na `docker01`, nie backup jednej konkretnej z nich.

Skróty użyte na tej stronie (1)

Skrót	Co znaczy
SSH	Secure Shell: szyfrowane zdalne logowanie do serwera

Zanim zaczniemy

Z ŻYCIA WZIĘTE

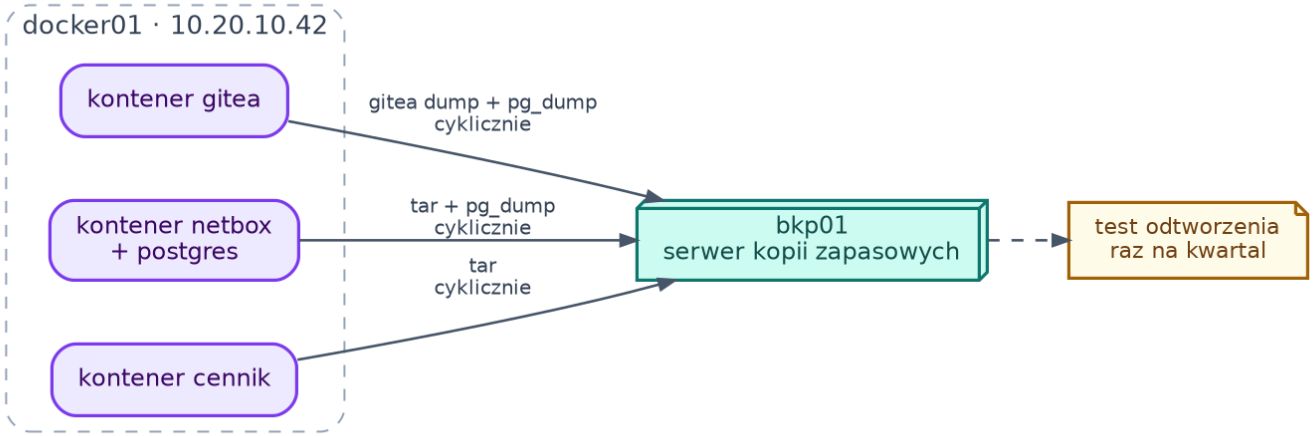
W czwartek Ola prosi o pilne odtworzenie NetBoksa na testowym hoście po nieudanej migracji danych. Zespół sięga po najświeższą kopię z `bkp01`, ale nikt wcześniej nie sprawdzał, czy zrzut bazy faktycznie się odtwarza. Test na czystym hoście kończy się sukcesem dopiero za trzecim razem, po poprawieniu uprawnień do pliku `.env` skopiowanego osobno od `compose.yaml`.

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.

Wymaganie	Uwaga
Dostęp SSH z `docker01` na `bkp01` kluczem, bez hasła	konto dedykowane kopiom, nie konto administratora

Wymaganie	Uwaga
Miejsce na `bkp01` pod kopie z `docker01`	patrz Świat przykładów: Zielona po adres i rolę serwera
Znajomość, które aplikacje trzymają dane w zewnętrznej bazie, a które w plikach	tabela w kroku 1

Uważaj: kopiowanie plików działającej bazy danych, na przykład katalogu `postgres` bind mountowanego pod kontener, w trakcie jej pracy daje niespójny obraz, bo baza w tym samym czasie zapisuje zmiany. Zamiast tego zawsze wykonujemy logiczny zrzut narzędziem samej bazy (`pg_dump` , `mysqldump`) i kopiujemy dopiero gotowy plik zrzutu.



Skrypt na docker01 kopiuje pliki spod /docker/apps i zrzuty baz danych na bkp01, cyklicznie i z okresowym testem odtworzenia (kliknięcie otwiera pełny rozmiar)

Przebieg

1. Rozpoznanie, co kopiujemy dla każdej aplikacji

Aplikacja	Dane w plikach	Baza danych	Co zrzucamy
`gitea`	repozytoria, załączniki	PostgreSQL na `pg01`	`gitea dump` plus `pg_dump`
`netbox`	pliki multimedialne	PostgreSQL w kontenerze na `docker01`	`pg_dump` z kontenera bazy

Aplikacja	Dane w plikach	Baza danych	Co zrzucamy
`cennik`	cały katalog danych	brak, tylko pliki	sam katalog, bez zrzutu bazy

Rozpoznanie robimy raz, przy wdrożeniu każdej aplikacji, i zapisujemy w skrypcie kopii, nie sprawdzamy tego ręcznie przy każdym uruchomieniu.

2. Skrypt kopii dla aplikacji z bazą PostgreSQL w kontenerze

```
sudo vim /docker/skrypty/kopia-netbox.sh
```

```
#!/usr/bin/env bash
set -euo pipefail

# Kopia zapasowa NetBoksa: pliki aplikacji plus zrzut bazy z kontenera postgres.
DATA=2026-09-27
CEL=nazwa_usera@bkp01:/backup/netbox/

# Zrzut bazy z kontenera, bez zatrzymywania samej bazy.
docker exec netbox-postgres pg_dump -U netbox netbox > /tmp/netbox_${DATA}.sql

# Archiwum plików aplikacji (media, załączniki), bez katalogu bazy.
tar czf /tmp/netbox-apps_${DATA}.tar.gz -C /docker/apps/netbox --exclude=postgres .

# Wysyłka obu plików na bkp01.
scp /tmp/netbox_${DATA}.sql /tmp/netbox-apps_${DATA}.tar.gz "${CEL}"

# Sprzątanie plików tymczasowych na docker01.
rm -f /tmp/netbox_${DATA}.sql /tmp/netbox-apps_${DATA}.tar.gz
```

```
sudo chmod +x /docker/skrypty/kopia-netbox.sh
```

`set -euo pipefail` na początku skryptu zatrzymuje go od razu przy pierwszym błędzie, zamiast kontynuować z niepełną kopią i cichym wpisem w logu: brak bazy, pełny dysk na `bkp01` albo zerwane SSH przerywa skrypt, zamiast wysyłać częściowe lub puste archiwum.

3. Skrypt kopii dla aplikacji bez bazy danych

```
sudo vim /docker/skrypty/kopia-cennik.sh
```

```
#!/usr/bin/env bash
set -euo pipefail

DATA=2026-09-27
CEL=nazwa_usera@bkp01:/backup/cennik/

tar czf /tmp/cennik_${DATA}.tar.gz -C /docker/apps/cennik .
scp /tmp/cennik_${DATA}.tar.gz "${CEL}"
rm -f /tmp/cennik_${DATA}.tar.gz
```

Bez bazy danych skrypt jest krótszy o cały krok zrzutu: samo archiwum katalogu `/docker/apps/cennik` wystarcza, bo to jedyne miejsce, gdzie ta aplikacja trzyma cokolwiek trwałego.

4. Timer systemd zamiast crona

Wybieramy systemd timery: OnCalendar zamiast crona zamiast wpisu w `crontab`, żeby mieć w jednym miejscu log wykonania i status ostatniego uruchomienia.

```
sudo vim /etc/systemd/system/kopia-docker.service
```

```
[Unit]
Description=Kopia zapasowa aplikacji dockerowych na docker01

[Service]
Type=oneshot
ExecStart=/docker/skrypty/kopia-netbox.sh
ExecStart=/docker/skrypty/kopia-cennik.sh
```

```
sudo vim /etc/systemd/system/kopia-docker.timer
```

```
[Unit]
Description=Harmonogram kopii zapasowej aplikacji dockerowych
```

```
[Timer]
OnCalendar=*-*-* 02:30:00
Persistent=true

[Install]
WantedBy=timers.target
```

```
sudo systemctl enable --now kopia-docker.timer
```

5. Weryfikacja ostatniego uruchomienia

```
systemctl status kopia-docker.timer
```

Przykład:

```
● kopia-docker.timer - Harmonogram kopii zapasowej aplikacji dockerowych
     Loaded: loaded (/etc/systemd/system/kopia-docker.timer; enabled)
     Active: active (waiting) since Sun 2026-09-27 02:30:12 UTC
    Trigger: Mon 2026-09-28 02:30:00 UTC
```

```
journalctl -u kopia-docker.service --since today
```

6. Test odtworzenia na czystym hoście

Raz na kwartał odtwarzamy najświeższą kopię jednej aplikacji na osobnym, testowym hoście z tą samą wersją Dockera, niezależnie od `docker01`. Kolejność jest zawsze taka sama: rozpakowanie archiwum plikowego do `/docker/apps/<nazwa>`, odtworzenie zrzutu bazy do pustej bazy (`psql` albo `pg_restore`, zależnie od formatu zrzutu), skopiowanie `compose.yaml` i `.env` z `/docker/compose/<nazwa>`, `docker compose up -d`, sprawdzenie w przeglądarce, że aplikacja wygląda tak jak na `docker01`. Kopia, której nikt nigdy nie spróbował odtworzyć na osobnym hoście, jest tylko przypuszczeniem, że działa.

Weryfikacja

```
ssh nazwa_usera@bkp01 "ls -la /backup/netbox/ /backup/cennik/"
```

Przykład:

```
-rw-r--r--. 1 nazwa_usera nazwa_usera 4823112 wrz 27 02:30 netbox_2026-09-27.sql
-rw-r--r--. 1 nazwa_usera nazwa_usera 118293440 wrz 27 02:31 netbox-apps_2026-09-27.tar.gz
-rw-r--r--. 1 nazwa_usera nazwa_usera 2048512 wrz 27 02:30 cennik_2026-09-27.tar.gz
```

Wycofanie zmian

Wyłączenie harmonogramu bez usuwania już wykonanych kopii:

```
sudo systemctl disable --now kopia-docker.timer
```

Gdy coś nie zadziała

Objaw	Przyczyna	Co robimy
Skrypt kończy się natychmiast bez żadnego pliku na `bkp01`	`set -euo pipefail` zatrzymał go na pierwszym błędzie	`journalctl -u kopia-docker.service`, sprawdzamy dokładny komunikat i krok, w którym padło
`pg_dump` z kontenera zwraca `role does not exist`	nazwa użytkownika w skrypcie inna niż w kontenerze bazy	sprawdzamy zmienne `POSTGRES_USER` w `compose.yaml` aplikacji
Archiwum `.tar.gz` dużo mniejsze niż spodziewany rozmiar katalogu	katalog bazy nieodfiltrowany przez `--exclude`, ale akurat pusty w danym momencie	`du -sh` na katalogu źródłowym przed archiwizacją, porównanie z rozmiarem archiwum
Test odtworzenia na czystym hoście nie startuje kontenera	inna wersja Dockera albo brakujący plik `.env` skopiowany osobno od `compose.yaml`	kopiujemy `.env` razem z `compose.yaml`, sprawdzamy `docker version` na obu hostach

Powiązane: Jak trzymamy aplikacje w Dockerze: konwencje Zielonej · Utrzymanie hosta dockerowego · systemd timery: OnCalendar zamiast crona · pg_dump, pg_dumpall, pg_restore

Źródła: man 1 systemd.timer · dokumentacja PostgreSQL, rozdział „pg_dump"