O'REILLY®

Wydanie III

Docker

Niezawodne kontenery produkcyjne

Praktyczne zastosowania

Sean Kane
Karl Matthias

Helion

Tytuł oryginału: Docker: Up & Running: Shipping Reliable Containers in Production, 3rd Edition

Tłumaczenie: Andrzej Stefański

ISBN: 978-83-289-0371-5

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/docpra
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 6.

Poznawanie Dockera

Mając już pewne doświadczenie w pracy z kontenerami i obrazami Dockera, możemy zacząć badać inne jego możliwości. W tym rozdziale będziemy dalej korzystać z narzędzi wiersza poleceń Dockera do komunikacji z uruchomionym serwerem `dockerd`, który skonfigurowałeś. Jednocześnie zapoznamy się z kilkoma podstawowymi poleceniami.

Docker dostarcza polecenia, które pozwalają na łatwe wykonywanie dodatkowych czynności:

- wyświetlanie wersji Dockera,
- przeglądanie informacji o serwerze,
- pobieranie aktualizacji obrazów,
- sprawdzanie kontenerów,
- wchodzenie do działającego kontenera,
- zwracanie wyniku,
- przeglądanie logów,
- monitorowanie statystyk
- i wiele więcej…

Przyjrzyjmy się niektórym z tych narzędzi oraz innym dodatkowym, tworzonym przez społeczność narzędziom zwiększającym możliwości mechanizmów wbudowanych w Dockera.

Wyświetlanie wersji Dockera

Jeśli przeczytałeś poprzedni rozdział, to masz działający demon Dockera na serwerze linuksowym lub w maszynie wirtualnej, a także uruchomiłeś bazowy kontener, aby upewnić się, że wszystko to działa. Jeśli nie przygotowałeś tego jeszcze i chcesz wypróbować kroki przedstawione w dalszej części książki, to zanim wykonasz kroki z tego podrozdziału, musisz wykonać czynności opisujące instalację z rozdziału 3.

Zdecydowanie najprostszą czynnością, jaką można wykonać w Dockerze, jest wyświetlenie wersji poszczególnych komponentów. Może nie brzmi to imponująco, ale jest przydatne, ponieważ Docker zbudowany jest z wielu komponentów, których numer wersji można bezpośrednio przetłumaczyć na

dostępny zakres funkcjonalności. Wiedza o tym, w jaki sposób wyświetlić wersję, pomoże Ci również w usuwaniu niektórych problemów z połączeniami. Czasem na przykład klient Dockera może wyświetlić tajemniczy komunikat o niewłaściwych wersjach API i miło byłoby móc odnieść to do wersji Dockera, by ustalić, który komponent wymaga aktualizacji. To polecenie komunikuje się ze zdalnym serwerem Dockera, a więc jeśli klient z jakiegoś powodu nie może połączyć się z serwerem, zwróci komunikat o błędzie i wyświetli jedynie informację o wersji klienta. Jeśli zobaczysz, że masz problem z połączeniem, powinieneś prawdopodobnie ponownie wykonać kroki z poprzedniego rozdziału.

Jeśli usuwasz problemy lub po prostu nie chcesz korzystać z klienta Dockera do łączenia się ze zdalnym systemem, możesz zawsze bezpośrednio zalogować się na serwerze z Dockerem i uruchomić polecenia Dockera z powłoki serwera. W przypadku większości serwerów Dockera będzie wymagało to uprawnień użytkownika `root` lub dołączenia konta użytkownika do grupy `docker`, co pozwoli na podłączenie się do gniazdka Unix, na którym nasłuchuje Docker.

Ponieważ zainstalowaliśmy wszystkie komponenty Dockera jednocześnie, po uruchomieniu polecenia `docker version` zobaczymy, że wszystkie wersje są zgodne:

```
$ docker version
Client:
  Cloud integration: v1.0.24
  Version: 20.10.17
  API version: 1.41
  Go version: go1.17.11
  Git commit: 100c701
  Built: Mon Jun 6 23:04:45 2022
  OS/Arch: darwin/amd64
  Context: default
  Experimental: true

Server: Docker Desktop 4.10.1 (82475)
  Engine:
  Version: 20.10.17
  API version: 1.41 (minimum version 1.12)
  Go version: go1.17.11
  Git commit: a89b842
  Built: Mon Jun 6 23:01:23 2022
  OS/Arch: linux/amd64
  Experimental: false
containerd:
  Version: 1.6.6
  GitCommit: 10c12954828e7c7c9b6e0ea9b0c02b01407d3ae1
runc:
  Version: 1.1.2
  GitCommit: v1.1.2-0-ga916309
docker-init:
  Version: 0.19.0
  GitCommit: de40ad0
```

Zauważ, że mamy oddzielne sekcje opisujące wersje klienta i serwera. W tym przypadku wersje klienta i serwera są takie same, ponieważ zostały razem zainstalowane. Ważne jest, by pamiętać, że nie zawsze tak będzie. Na szczęście w swoich systemach produkcyjnych będziesz mógł utrzymywać na większości serwerów tę samą wersję Dockera. Nie jest jednak niczym nietypowym, że środowiska deweloperskie i systemy do budowania korzystają z odrobinę innych wersji.

Aplikacje klienckie korzystające z API i biblioteki zazwyczaj pracują z dużą liczbą wersji Dockera i jest to zależne od tego, jakiej wersji API wymagają. W sekcji poświęconej serwerowi możemy zobaczyć informację, że obecna wersja API to 1.41, a minimalna obsługiwana to 1.12. Jest to przydatna informacja, jeśli używasz zewnętrznych aplikacji klienckich i teraz już wiesz, jak to sprawdzić.

Informacje o serwerze

Możemy też dowiedzieć się wiele na temat samego serwera Dockera poprzez klienta Dockera. Później opowiemy o tym więcej — możesz ustalić, z jakiego mechanizmu obsługującego system plików korzysta serwer Dockera, w jakiej wersji jądra oraz w jakim systemie operacyjnym działa, a także jak wiele kontenerów oraz obrazów jest tam obecnie zapisanych. Jeśli uruchomisz `docker system info`, zobaczysz informacje podobne do zaprezentowanych poniżej, które zostały skrócone w celu zwiększenia czytelności:

```
$ docker system info
Client:
…
  Plugins:
    buildx: Docker Buildx (Docker Inc., v0.8.2)
    compose: Docker Compose (Docker Inc., v2.6.1)
    extension: Manages Docker extensions (Docker Inc., v0.2.7)
    sbom: View the packaged-based Software Bill Of Materials (SBOM) …
    scan: Docker Scan (Docker Inc., v0.17.0)

  Server:
    Containers: 11
…
    Images: 6
    Server Version: 20.10.17
    Storage Driver: overlay2
…
  Plugins:
    Volume: local
    Network: bridge host ipvlan macvlan null overlay
    Log: awslogs fluentd gcplogs gelf journald json-file local logentries …
…
    Runtimes: io.containerd.runc.v2 io.containerd.runtime.v1.linux runc
    Default Runtime: runc
…
    Kernel Version: 5.10.104-linuxkit
    Operating System: Docker Desktop
    OSType: linux
    Architecture: x86_64
…
```

W zależności od ustawień demona Dockera może to wyglądać odrobinę inaczej. Nie martw się tym, ponieważ to tylko przykład. Możemy tutaj zobaczyć, że na naszym serwerze zainstalowany jest Docker Desktop z uruchomionym jądrem Linuksa 5.10.104, wspierany przez sterownik systemu plików `overlay2`. Mamy też na serwerze kilka obrazów i kontenerów. Przy nowej instalacji ta liczba powinna być równa zero.

Warto tutaj też wskazać informacje na temat wtyczek. Mówią one nam o wszystkim, co wspiera dana instalacja Dockera. Przy nowej instalacji będzie to wyglądało mniej więcej tak jak tutaj, w zależności od tego, jakie nowe wtyczki są dystrybuowane razem z Dockerem. Sam Docker składa się z wielu różnych wtyczek współpracujących ze sobą. To niezwykle istotna właściwość, ponieważ oznacza też, że możliwe jest zainstalowanie kilku innych wtyczek dostarczanych przez członków społeczności. Możliwość sprawdzenia, co jest zainstalowane, przydaje się choćby do upewnienia się, czy Docker widzi to, co ostatnio dodałeś.

W większości instalacji domyślnym katalogiem do zapisywania obrazów i kontenerów powinien być katalog */var/lib/docker*. Jeśli musisz to zmienić, możesz wyedytować skrypty startowe Dockera w taki sposób, by uruchamiać demon z parametrem `--data-root`, wskazującym na odpowiednie miejsce do przechowywania danych. Aby sprawdzić to ręcznie, powinieneś uruchomić coś takiego:

```
$ sudo dockerd
-H unix:///var/run/docker.sock \
--data-root="/data/docker"
```

Domyślnie plik konfiguracyjny serwera Dockera (*https://oreil.ly/jp7iK*)[1] to */etc/docker/daemon.json*. Większość argumentów, jakie omawiamy i przekazujemy bezpośrednio do polecenia `dockerd`, można na stałe zapisać w tym pliku. Jeśli korzystasz z Docker Desktop, zalecane jest, by modyfikować ten plik za pomocą interfejsu Docker Desktop.

Więcej na temat środowisk uruchomieniowych (ang. *runtime*) powiemy później, ale tutaj możesz zobaczyć, że zainstalowane mamy trzy takie środowiska. `runc` to domyślne środowisko uruchomieniowe Dockera. Myśląc o kontenerach linuksowych, zazwyczaj masz na myśli taki typ kontenerów, który jest budowany przez `runc`. W tym serwerze mamy również zainstalowane środowiska `io.containerd.runc.v2` i `io.containerd.run.time.v1.linux`, które działają odrobinę inaczej. Więcej na temat innych środowisk uruchomieniowych powiemy w rozdziale 11.

Pobieranie aktualizacji obrazów

Dla zaprezentowanych dalej przykładów zamierzamy użyć podstawowego obrazu Ubuntu. Nawet jeśli już raz pobrałeś obraz bazowy `ubuntu:latest`, możesz pobrać go jeszcze raz — pobrane zostaną aktualizacje opublikowane od ostatniego uruchomienia. Dzieje się tak, ponieważ znacznik `latest` powinien wskazywać najnowszą wersję obrazu kontenera. Znacznik `latest` jest kontrowersyjny, ponieważ nie jest na stałe związany z konkretnym obrazem i może mieć różne znaczenie w różnych projektach. Niektórzy używają go, by wskazywać najnowszą stabilną wersję, inni wykorzystują go do wskazania ostatniej wersji zbudowanej przez ich system CI/CD, a jeszcze inni po prostu nie oznaczają żadnego ze swoich obrazów jako `latest`. Pomimo to nadal jest on szeroko wykorzystywany i może być przydatny w środowiskach nieprodukcyjnych, gdzie korzyści wynikające z wygody w korzystaniu z niego są ważniejsze niż korzyści ze wskazywania konkretnej wersji.

[1] URL: *https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file.*

Wywołanie `docker image pull` powinno wyglądać tak:

```
$ docker image pull ubuntu:latest

latest: Pulling from library/ubuntu
405f018f9d1d: Pull complete
Digest: sha256:b6b83d3c331794420340093eb706a6f152d9c1fa51b262d9bf34594887c2c7ac
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
```

Polecenie to spowodowało pobranie jedynie warstw, które zmieniły się od ostatniego uruchomienia polecenia. Mogłeś zobaczyć dłuższą lub krótszą listę albo nawet listę pustą — w zależności od tego, kiedy ostatnio pobierałeś obraz, jakie zmiany zostały wprowadzone do rejestru od tego czasu oraz ile warstw zawiera dany obraz.

Warto pamiętać, że nawet jeśli pobrałeś wersję ze znacznikiem `latest`, Docker nie będzie automatycznie utrzymywał aktualności lokalnej kopii obrazu. Musisz samodzielnie się tym zająć. Jeśli jednak wdrażasz obraz oparty na nowszej kopii `ubuntu:latest`, zgodnie z oczekiwaniami Docker pobierze brakujące warstwy podczas wdrażania. Pamiętaj, że jest to zachowanie klienta Dockera — inne biblioteki czy narzędzia korzystające z API mogą zachowywać się w inny sposób. Zdecydowanie zalecane jest, by zawsze wdrażać kod produkcyjny, korzystając ze znaczników wskazujących wersję, a nie ze znacznika `latest`. To pomaga zagwarantować, że otrzymasz wersję, jakiej oczekiwałeś, i że nie pojawi się nic niespodziewanego.

Poza wykorzystaniem znacznika `latest` lub znacznika z konkretną wersją możesz też odwoływać się do elementów w rejestrze, korzystając ze znacznika związanego z treścią który wygląda w taki sposób:

```
sha256:b6b83d3c331794420340093eb706a6f152d9c1fa51b262d9bf34594887c2c7ac
```

Są one w rzeczywistości generowane jako suma kontrolna zawartości obrazu i stanowią bardzo precyzyjny identyfikator. Jest to jak dotąd najbezpieczniejszy sposób odwoływania się do obrazów Dockera, gdy musisz mieć pewność, że otrzymasz dokładnie taką wersję, jakiej oczekujesz, ponieważ takie identyfikatory nie mogą być przenoszone jak znaczniki opisujące wersje. Składnia pozwalająca na pobranie ich z rejestru jest bardzo podobna, ale zwróć uwagę na znak `@` w znaczniku.

```
docker image pull
ubuntu@sha256:b6b83d3c331794420340093eb706a6f152d9c1fa51b262d9bf34594887c2c7ac
```

Inaczej niż w przypadku większości innych poleceń Dockera, w przypadku których możesz skrócić znacznik, nie możesz tego robić ze znacznikami SHA-256. Tutaj musisz wpisywać całą sumę kontrolną.

Pobieranie informacji o kontenerze

Gdy masz już utworzony kontener, działający lub nie, możesz użyć Dockera, by sprawdzić, w jaki sposób został on skonfigurowany. Jest to często przydatne podczas debugowania, a dostarcza też trochę innych informacji, które mogą być przydatne do identyfikowania kontenera.

Na potrzeby tego przykładu przejdziemy dalej i uruchomimy kontener:

```
$ docker container run --rm -d -t ubuntu /bin/bash
3c4f916619a5dfc420396d823b42e8bd30a2f94ab5b0f42f052357a68a67309b
```

Możemy wyświetlić listę wszystkich naszych uruchomionych kontenerów za pomocą polecenia `docker container ls`, aby upewnić się, że wszystko działa zgodnie z oczekiwaniami, oraz skopiować identyfikator kontenera:

```
$ docker container ls
CONTAINER ID  IMAGE         COMMAND     ... STATUS        ...  NAMES
3c4f916619a5  ubuntu:latest "/bin/bash" ... Up 31 seconds ...  angry_mestorf
```

W tym przykładzie nasz identyfikator to `3c4f916619a5`. Moglibyśmy też wykorzystać `angry_mestorf`, czyli nazwę dynamicznie przypisywaną do naszego kontenera. Wiele działających w niższych warstwach narzędzi wymaga unikalnego identyfikatora kontenera, dlatego warto wyrobić w sobie nawyk patrzenia na niego w pierwszej kolejności. Jak już wcześniej wspomnieliśmy, pokazany tutaj identyfikator jest skróconą (krótką) wersją, ale Docker używa ich wymiennie z dłuższą wersją. Tak jak w wielu systemach kontroli wersji, ta suma kontrolna jest w rzeczywistości jedynie prefiksem dużo dłuższej sumy kontrolnej. Wewnętrznie jądro do oznaczania kontenera korzysta z 64-bajtowej sumy kontrolnej. Użytkownikom trudno byłoby jednak tego używać, dlatego Docker obsługuje skrócone sumy kontrolne.

Wynik działania polecenia `docker container inspect` jest dość obszerny, dlatego skrócimy go w poniższym bloku kodu do kilku wartości, które warto wskazać. Powinieneś przejrzeć wszystkie zwracane dane, aby sprawdzić, co jeszcze może być interesujące:

```
$ docker container inspect 3c4f916619a5

[{
    "Id": "3c4f916619a5dfc420396d823b42e8bd30a2f94ab5b0f42f052357a68a67309b",
    "Created": "2022-07-17T17:26:53.611762541Z",
    ...
    "Args": [],
    ...
    "Image": "sha256:27941809078cc9b2802deb2b0bb6feed6c…7f200e246535337O1ee",
    ...
    "Config": {
        "Hostname": "3c4f916619a5",
        ...
        "Env": [
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
        ],
        "Cmd": [
            "/bin/bash"
        ],
        ...
        "Image": "ubuntu",
        ...
    },
    ...
}]
```

Zwróć uwagę na ten długi ciąg znaków `"Id"`. Jest to pełny unikalny identyfikator tego kontenera. Na szczęście możemy korzystać ze skróconej wersji, nawet jeśli nie jest ona zbyt wygodna.

Możemy też zobaczyć dokładny czas utworzenia kontenera z dużo większą dokładnością niż w przypadku `docker container ls`.

Pojawia się tutaj też kilka innych interesujących elementów: główne polecenie wywoływane w kontenerze, zmienna środowiska przekazana do niego podczas tworzenia, obraz, na którym jest on oparty, oraz zmienna z nazwą hosta zapisaną w kontenerze. Wszystkie te zmienne są konfigurowalne podczas uruchamiania kontenera, jeśli zachodzi taka potrzeba. Typową metodą przekazywania do kontenerów informacji o konfiguracji jest na przykład zastosowanie zmiennych środowiska, dlatego możliwość sprawdzenia za pomocą polecenia `docker container inspect`, w jaki sposób kontener został skonfigurowany, może bardzo pomóc przy debugowaniu.

Możesz też zatrzymać bieżący kontener, wykonując polecenie takie jak `docker container stop 3c4f916619a5`.

Wykorzystanie powłoki

Spróbujmy uruchomić kontener z interaktywną sesją powłoki bash, dzięki której będziemy mogli się trochę rozejrzeć. Uzyskamy to, jak wcześniej, uruchamiając coś takiego:

```
$ docker container run --rm -it ubuntu:22.04 /bin/bash
```

Takie polecenie powinno uruchomić kontener z Ubuntu 22.04 LTS z powłoką bash jako procesem najwyższego poziomu. Wskazując znacznik 22.04, możemy upewnić się, że dostaniemy wskazaną wersję obrazu. Zatem jakie procesy działają po uruchomieniu kontenera?

```
root@35fd1ad27228:/# ps -ef
UID      PID  PPID  C STIME TTY        TIME CMD
root 1 0 0 17:45 pts/0 00:00:00 /bin/bash
root 9 1 0 17:47 pts/0 00:00:00 ps -ef
```

Cóż, nie za wiele, prawda? Okazuje się, że gdy kazaliśmy Dockerowi uruchomić bash, nie otrzymaliśmy nic więcej. Jesteśmy wewnątrz obrazu całej dystrybucji Linuksa, ale nie wystartował dla nas automatycznie żaden inny proces. Dostaliśmy tylko to, czego zażądaliśmy. Warto to zapamiętać.

W odróżnieniu od pełnych maszyn wirtualnych kontenery Dockera domyślnie nie uruchamiają niczego w tle. Są dużo lżejsze niż maszyny wirtualne i dlatego nie uruchamiają systemu `init`. Możesz oczywiście uruchomić cały system `init` lub wbudowany w Dockera system `init` o nazwie `tini` (*https://github.com/krallin/tini*), jeśli go potrzebujesz, ale musisz tego zażądać. Powiemy o tym więcej w rozdziale 7.

W taki sposób otrzymujemy powłokę działającą w kontenerze. Możesz swobodnie porozglądać się i zobaczyć, czy nie ma jeszcze czegoś interesującego wewnątrz kontenera. Może być dostępny dość ograniczony zestaw poleceń. Jesteś jednak w dystrybucji bazowej Ubuntu, więc możesz to naprawić, wykonując polecenie `apt-get update`, a następnie `apt-get install…`, by zainstalować więcej pakietów. Będą one jednak dostępne tylko do końca działania tego kontenera. Modyfikujesz najwyższą warstwę kontenera, a nie obrazu bazowego! Kontenery są z natury efemeryczne, a więc nic, co wykonujesz wewnątrz kontenera, nie pozostanie po jego usunięciu.

Gdy zakończysz pracę z kontenerem, opuść powłokę, wykonując polecenie `exit`, co w sposób naturalny zatrzyma działanie kontenera:

```
root@35fd1ad27228:/# exit
```

Zwracanie wyniku

Jak nieefektywne byłoby uruchamianie całej maszyny wirtualnej w celu uruchomienia pojedynczego polecenia, które zwróci wynik działania? Zazwyczaj byś tego nie zrobił, ponieważ byłoby to bardzo czasochłonne i wymagałoby uruchomienia całego systemu operacyjnego w celu wykonania jednego polecenia. Kontenery dockerowe i linuksowe nie działają jednak w taki sam sposób jak maszyny wirtualne: kontenery są bardzo lekkie i nie muszą uruchamiać się jak system operacyjny. Uruchomienie czegoś w stylu krótkiego zadania w tle i czekanie na zwrócenie kodu zakończenia jest typowym przypadkiem użycia dla kontenerów linuksowych. Możesz uważać to za sposób uzyskania zdalnego dostępu do skonteneryzowanego systemu i wykorzystywać dostęp do każdego z dostępnych wewnątrz tego kontenera poleceń z możliwością przekierowania potoków danych z nich i do nich, a także zwracania kodów zakończenia.

Może się to przydać w wielu przypadkach: możesz na przykład w ten sposób uruchamiać zdalnie wywoływane testy poprawności działania systemu (ang. *health checks*) lub też przygotować szereg maszyn z uruchamianymi za pomocą Dockera procesami do przetworzenia zadanego wsadu i zwrócenia pojedynczego wyniku. Narzędzia uruchamiane za pomocą programu `docker` w wierszu poleceń przekazują wyniki lokalnej maszynie. Jeśli uruchomisz zdalne polecenie jako proces na pierwszym planie i jawnie nie zdefiniujesz, że ma być inaczej, polecenie `docker` przekieruje swoje standardowe wejście do zdalnego procesu, a standardowe wyjście oraz standardowe wyjście błędów na Twój terminal. Jedyną rzeczą, jaką musimy zrobić, aby to uzyskać, jest uruchomienie polecenia na pierwszym planie i niezaalokowanie terminala (TTY) w zdalnym systemie. Taka jest w rzeczywistości domyślna konfiguracja! Nie trzeba dodawać żadnych parametrów w wierszu poleceń.

Gdy uruchamiamy te polecenia, Docker tworzy nowy kontener, wykonuje polecenie, jakie mu przekazaliśmy wewnątrz przestrzeni nazw i cgroup kontenera, a następnie usuwa kontener, więc pomiędzy jego wywołaniami nie pozostają uruchomione żadne dodatkowe procesy ani nie jest niepotrzebnie zajmowana przestrzeń na dysku. Poniższy kod pokazuje, co możesz w taki sposób wykonać:

```
$ docker container run --rm ubuntu:22.04 /bin/false
$ echo $?
1

$ docker container run --rm ubuntu:22.04 /bin/true
$ echo $?
0

$ docker container run --rm ubuntu:22.04 /bin/cat /etc/passwd

root:x:0:0:root:/root:/bin/bash
daemon:x:2:2:daemon:/sbin:/sbin/nologin
…
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
```

```
$ docker container run --rm ubuntu:22.04 /bin/cat /etc/passwd | wc -l
19
```

Uruchomiliśmy tutaj na zdalnym serwerze program `/bin/false`, który zawsze kończy działanie, zwracając status 1. Zauważ, w jaki sposób polecenie `docker` przekazało ten wynik do nas, do lokalnego terminala. Aby udowodnić, że w ten sam sposób przekazuje też inne wyniki, wywołaliśmy również polecenie `/bin/true`, które zawsze zwraca 0. I taki też wynik otrzymaliśmy.

Następnie użyliśmy polecenia `docker`, by uruchomić `cat /etc/passwd` w zdalnym kontenerze. Otrzymanym wynikiem jest wydruk pliku */etc/passwd* znajdującego się wewnątrz systemu plików kontenera. Ponieważ jest to po prostu zwykły wynik wyświetlany na standardowym wyjściu, możemy go przekierować do lokalnych poleceń, tak jak każdy inny wynik.

W pokazanym wcześniej kodzie standardowe wyjście przekierowywane jest do lokalnego polecenia `wc`, a nie polecenia `wc` w kontenerze. Sam znak przekierowania potoku (*pipe*) nie jest przekazywany do kontenera. Jeśli chcesz przekazać całe polecenie razem ze znakami przekierowania potoku do serwera, musisz wywołać pełną powłokę po drugiej stronie i przekazać do niej „zacytowane" polecenie, czyli użyć postaci `bash -c "<twoje polecenie> | <inne polecnie>"`. Pokazany wcześniej przykład miałby w tej sytuacji postać `docker container run ubuntu:22.04 /bin/bash -c " /bin/cat /etc/passwd | wc -l"`.

Wnętrze działającego kontenera

Możesz stosunkowo łatwo uzyskać dostęp do powłoki uruchomionej w nowym kontenerze utworzonym z prawie każdego obrazu, wykorzystując polecenie `docker container run`, co zaprezentowaliśmy powyżej. Nie jest to jednak to samo, co uzyskanie dostępu do nowej powłoki wewnątrz istniejącego kontenera z uruchomioną Twoją aplikacją. Przy każdym użyciu `docker container run` otrzymujesz nowy kontener. Jeśli jednak masz istniejący kontener, na którym jest uruchomiona aplikacja, i musisz ją zdebugować z wnętrza kontenera, potrzebujesz czegoś innego.

Za pomocą polecenia `docker container exec` można w natywny dla Dockera sposób utworzyć nowy interaktywny proces w kontenerze, ale istnieje także bardziej naturalne dla Linuksa polecenie, a mianowicie `nsenter`. W tym podrozdziale omówimy polecenie `docker container exec`, a polecenie `nsenter` poznasz w rozdziale 11., w podrozdziale „nsenter".

Zapewne się zastanawiasz, do czego może Ci się to przydać.

Podczas programowania bardzo przydatne może to być do modyfikowania i testowania aplikacji. Jest to mechanizm wykorzystywany przez Development Containers (*https://containers.dev*) w środowiskach IDE takich jak Visual Studio Code (*https://code.visualstudio.com/docs/devcontainers/containers*).

W systemach produkcyjnych nie jest dobrą praktyką logowanie się przez SSH bezpośrednio do serwerów produkcyjnych, a to jest mniej więcej to samo. Zdarzają się jednak sytuacje, gdy bardzo ważne jest sprawdzenie, co dzieje się w danym środowisku, i w takich przypadkach może być to pomocne.

docker container exec

Najpierw popatrzmy na najłatwiejszy i najlepszy sposób dostania się do wnętrza działającego kontenera. Zarówno serwer `dockerd`, jak i narzędzia wiersza poleceń Dockera obsługują zdalne uruchomienie nowego procesu w działającym kontenerze za pomocą polecenia `docker container exec`. Zacznijmy więc od uruchomienia kontenera w tle, a następnie wejdźmy do niego za pomocą `docker container exec` i wywołajmy powłokę systemową.

Wywoływane przez Ciebie polecenie nie musi być powłoką: możliwe jest uruchamianie pojedynczych poleceń wewnątrz kontenera i obserwowanie wyników ich działania poza kontenerem za pomocą `docker container exec`. Jeśli jednak chcesz dostać się do kontenera, by się rozejrzeć, wykorzystanie powłoki jest najłatwiejszym sposobem, aby to zrobić.

Aby wykorzystać `docker container exec`, będziemy potrzebowali identyfikatora naszego kontenera. Na potrzeby tego eksperymentu utwórzmy kontener, który uruchomi polecenie `sleep` na 600 sekund:

```
$ docker container run -d --rm ubuntu:22.04 sleep 600
9f09ac4bcaa0f201e31895b15b479d2c82c30387cf2c8a46e487908d9c285eff
```

Skrócony identyfikator tego kontenera to `9f09ac4bcaa0`. Możemy go użyć, by dostać się do kontenera za pomocą polecenia `docker container exec`. Wiersz poleceń przy wywołaniu `docker container exec`, co nie jest zaskoczeniem, bardzo przypomina wiersz poleceń przy wywołaniu `docker container run`. Żądamy uruchomienia pseudo-TTY oraz trybu interaktywnego, dodając parametry `-t` i `-i`:

```
$ docker container exec -it 9f09ac4bcaa0 /bin/bash
root@9f09ac4bcaa0:/#
```

Zauważ, że pojawił się znak zachęty, zawierający identyfikator kontenera, do którego się dostaliśmy. Jest to dość przydatne w śledzeniu, gdzie aktualnie się znajdujemy. Możemy teraz uruchomić zwykłe linuksowe polecenie `ps`, by sprawdzić, co jeszcze działa wewnątrz naszego kontenera. Powinniśmy zobaczyć proces `sleep`, który został utworzony podczas uruchamiania kontenera:

```
root@9f09ac4bcaa0:/# ps -ef
UID          PID  PPID  C  STIME TTY          TIME CMD
root           1     0  0  20:22 ?        00:00:00 sleep 600
root           7     0  0  20:23 pts/0    00:00:00 /bin/bash
root          15     7  0 2 0:23 pts/0    00:00:00 ps -ef
```

Wpisz `exit`, aby wydostać się z kontenera, gdy skończysz.

Możesz też uruchomić dodatkowe procesy w tle za pomocą `docker container exec`. Parametr `-d` wykorzystuje się tak jak przy `docker container run`. Powinieneś jednak dobrze się zastanowić przed zrobieniem tego w celu innym niż debugowanie, ponieważ jeśli wykorzystasz ten mechanizm, stracisz powtarzalność wdrażania obrazu. Inni użytkownicy będą musieli w takiej sytuacji wiedzieć, co przekazać do `docker container exec`, by uzyskać oczekiwaną funkcjonalność. Jeśli kusi Cię, by tak zrobić, pamiętaj, że bardziej korzystne będzie przebudowanie obrazu kontenera w celu uruchomienia obu procesów w powtarzalny sposób. Jeśli musisz wyzwolić w programie działającym wewnątrz kontenera podjęcie jakiegoś działania, takiego jak rotowanie logów czy przeładowanie konfiguracji, bardziej przejrzyste będzie skorzystanie z polecenia `docker container kill -s <SYGNAŁ>` ze standardową nazwą uniksowego sygnału, by przekazać odpowiednią informację do procesu znajdującego się w kontenerze.

docker volume

Docker zawiera moduł `volume` pozwalający wyświetlić listę wszystkich woluminów zapisanych w Twoim katalogu głównym, a następnie zebrać o nich dodatkowe informacje, wśród nich informację o fizycznej lokalizacji na serwerze.

Te woluminy nie są zbindowane (ang. *bind-mounted*), ale są to specjalne kontenery, które pozwalają na trwałe zapisanie danych.

Jeśli uruchomimy zwykłe polecenie `docker` montujące katalog, zauważymy, że nie tworzy ono żadnych woluminów Dockera.

```
$ docker volume ls
DRIVER              VOLUME NAME

$ docker container run --rm -d -v /tmp:/tmp ubuntu:latest sleep 120
6fc97c50fb888054e2d01f0a93ab3b3db172b2cd402fc1cd616858b2b5138857

$ docker volume ls
DRIVER              VOLUME NAME
```

Możesz jednak łatwo utworzyć nowy wolumin za pomocą takiego polecenia:

```
$ docker volume create my-data
```

Jeśli następnie wyświetlisz listę wszystkich swoich woluminów, powinieneś zobaczyć coś takiego:

```
$ docker volume ls

DRIVER              VOLUME NAME
local               my-data

$ docker volume inspect my-data

[
    {
        "CreatedAt": "2022-07-31T16:19:42Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/my-data/_data",
        "Name": "my-data",
        "Options": {},
        "Scope": "local"
    }
]
```

Możesz teraz uruchomić kontener z podłączonym do niego woluminem danych, wydając polecenie:

```
$ docker container run --rm \
    --mount source=my-data,target=/app \
    ubuntu:latest touch /app/my-persistent-data
```

Ten kontener utworzył plik w woluminie danych i natychmiast zakończył działanie.

Jeśli zamontujemy ten wolumin w innym kontenerze, zobaczymy, że nasze dane nadal się tam znajdują:

```
$ docker container run --rm \
    --mount source=my-data,target=/app \
    fedora:latest ls -lFa /app/my-persistent-data

-rw-r--r-- 1 root root 0 Jul 31 16:24 /app/my-persistent-data
```

Możesz też w końcu usunąć wolumin danych, gdy nie będzie już potrzebny, wydając polecenie:

```
$ docker volume rm my-data

my-data
```

Jeśli próbujesz skasować wolumin wykorzystywany przez kontener (działający lub nie), otrzymasz komunikat o błędzie:

```
Error response from daemon: unable to remove volume:
   remove my-data: volume is in use - [
   d0763e6e8d79e55850a1d3ab21e9d...,
   4b40d52978ea5e784e66ddca8bc22...]
```

Polecenia te pozwalają dość szczegółowo zbadać Twoje kontenery. Gdy szerzej omówimy przestrzenie nazw w rozdziale 11., lepiej zrozumiesz, jak właściwie wszystkie te elementy współpracują i wspólnie tworzą kontener.

Logi

Korzystanie z logów jest bardzo ważną częścią każdej aplikacji produkcyjnej. Gdy coś pójdzie nie tak, logi mogą mieć krytyczne znaczenie dla przywrócenia działania usługi, dlatego muszą być dobrze przygotowane. Istnieje kilka typowych sposobów korzystania z logów aplikacyjnych w systemach linuksowych, przy czym niektóre są lepsze od innych. Jeśli proces aplikacji działa w pudełku, możesz oczekiwać, że wynik jego działania zostanie umieszczony w lokalnym pliku z logami, który będziesz mógł odczytać. Możesz też oczekiwać, że wynik jego działania będzie zapisywany do bufora jądra, gdzie będzie mógł zostać odczytany za pomocą `dmesg`. Albo możesz oczekiwać — jak w wielu nowoczesnych dystrybucjach Linuksa z `systemd` — że logi będą dostępne za pomocą `journalctl`. Z powodu ograniczeń kontenerów oraz tego, w jaki sposób skonstruowany jest Docker, żaden z tych sposobów nie zadziała bez przygotowania przynajmniej podstawowej konfiguracji. Można to zaakceptować, ponieważ wsparcie logowania w Dockerze stoi na najwyższym poziomie.

Docker upraszcza obsługę logów na kilka najważniejszych sposobów. Po pierwsze przechwytuje wszystkie dane tekstowe zwracane przez aplikacje działające w zarządzanych przez niego kontenerach. Wszystko, co zostanie przesłane na standardowe wyjście oraz standardowe wyjście błędów w kontenerze, jest przechwytywane przez demona Dockera i przesyłane strumieniem do skonfigurowanego do logowania backendu. Po drugie, jak wiele innych części Dockera, ten system jest rozszerzalny — wiele potężnych opcji jest dostępnych w postaci wtyczek. Na razie nie wnikajmy w to jednak za głęboko.

Polecenie docker container logs

Zacznijmy od najprostszego przypadku użycia w Dockerze: domyślnego mechanizmu zapisywania logów. Mechanizm ten ma ograniczenia, które za chwilę omówimy, ale w większości przypadków działa dobrze i jest bardzo wygodny. Jeśli uruchamiasz Dockera w środowiskach dla programistów, jest to prawdopodobnie jedyna strategia zapisywania logów, z jakiej będziesz tutaj korzystał. Ta metoda istniała od samego początku i jest zarówno dobrze rozumiana, jak i wspierana. Wykorzystywane są tutaj pliki `json`, a większość użytkowników korzysta z niej za pomocą polecenia `docker container logs`.

Jak sama nazwa wskazuje, gdy uruchamiasz domyślną wtyczkę tworzenia logów wykorzystującą pliki `json`, logi Twojej aplikacji przesyłane są w strumieniu przez demona Dockera do pliku JSON tworzonego dla każdego z kontenerów. Pozwala nam to na pobranie logów dowolnego kontenera w dowolnym momencie.

Aby wyświetlić jakieś logi, uruchomimy kontener z `nginx`:

```
$ docker container run --rm -d --name nginx-test --rm nginx:latest
```

Następnie wydamy polecenie:

```
$ docker container logs nginx-test
…
2022/07/31 16:36:05 [notice] 1#1: using the "epoll" event method
2022/07/31 16:36:05 [notice] 1#1: nginx/1.23.1
2022/07/31 16:36:05 [notice] 1#1: built by gcc 10.2.1 20210110 (Debian 10.2.1-6)
2022/07/31 16:36:05 [notice] 1#1: OS: Linux 5.10.104-linuxkit
…
```

To bardzo wygodne, ponieważ Docker pozwala uzyskać logi zdalnie, bezpośrednio z wiersza poleceń, na żądanie. Jest to naprawdę przydatne przy logach o małej objętości.

Aby ograniczyć zakres wyświetlanych logów do najnowszych, możesz użyć parametru `--since`, pozwalającego wyświetlić jedynie logi wygenerowane po czasie wskazanym zgodnie ze standardem RFC 3339 (np. `2002-10-02T10:00:00-05:00`), za pomocą uniksowego znacznika czasu (np. `1450071961`) lub ciągu opisującego długość odcinka czasu w Go (np. `5m45s`). Możesz też użyć parametru `--tail` z dopisaną liczbą wierszy, jakie mają być wyświetlone (licząc od ostatniego).

Rzeczywiste pliki obsługujące to logowanie znajdują się na samym serwerze Dockera, domyślnie w katalogu */var/lib/docker/containers/<id_kontenera>/*, gdzie *<id_kontenera>* oznacza rzeczywisty identyfikator kontenera. Jeśli przyjrzysz się plikowi *<id_kontenera>.json.log*, zobaczysz, że każdy jego wiersz opisuje obiekt JSON. Plik będzie miał taką postać:

```
{"log":"2022/07/31 16:36:05 [notice] 1#1: using the \"epoll\" event method\n",
"stream":"stderr","time":"2022-07-31T16:36:05.189234362Z"}
```

Pole `log` zawiera treść przesyłaną na standardowe wyjście przez obserwowany proces, pole `stream` wskazuje na to, że informacje pojawiły się na `stdout`, a nie na `stderr`, a z kolei w polu `time` zapisany jest dokładny czas otrzymania informacji przez demon Dockera. Nie jest to typowy format zapisywania logów, ale tworzy to strukturę, a nie czysty strumień, co jest korzystne, jeśli chcesz później przetwarzać takie logi.

Tak samo jak zwykły plik z logami, najnowsze logi Dockera możesz wyświetlać w czasie rzeczywistym, wywołując `docker containter logs -f`:

```
$ docker container logs -f nginx-test
…
2022/07/31 16:36:05 [notice] 1#1: start worker process 35
2022/07/31 16:36:05 [notice] 1#1: start worker process 36
2022/07/31 16:36:05 [notice] 1#1: start worker process 37
2022/07/31 16:36:05 [notice] 1#1: start worker process 38
```

Wygląda to identycznie jak wynik działania zwykłego polecenia `docker container logs`, ale po wyświetleniu danych klient się zatrzymuje, czeka i wyświetla następne logi, gdy tylko otrzyma je z serwera tak jak polecenie linuksowe `tail -f`. W dowolnym momencie możesz przerwać wyświetlanie nowych logów za pomocą kombinacji *Ctrl+C*.

Na koniec możesz zatrzymać testowy kontener:

```
$ docker container stop nginx-test
```

Konfigurując wartość przekazywaną z parametrem `--log-opt tag="{{.ImageName }}/{{.ID }}"`, można zmienić domyślny format znacznika logowania (od którego będzie się zaczynał każdy wiersz logów) na coś bardziej użytecznego. Domyślnie logi Dockera będą oznaczane za pomocą pierwszych 12 znaków identyfikatora kontenera.

W przypadku zapisywania logów na pojedynczym serwerze ten mechanizm działa dość dobrze. Jego wady związane są z rotacją logów, zdalnym dostępem do nich po wykonaniu rotacji oraz zużyciem przestrzeni dyskowej w przypadku dużej objętości logów. Mimo że mechanizm ten opiera się na zapisywaniu w pliku JSON, jest w rzeczywistości wystarczająco wydajny, by większość aplikacji produkcyjnych mogła w ten sposób tworzyć logi, jeśli w Twoim przypadku rozwiązanie to będzie Ci odpowiadało. Jeśli jednak masz bardziej złożone środowisko, będziesz potrzebował czegoś solidniejszego z możliwością scentralizowanego zarządzania logami.

Domyślne ustawienia `dockerd` nie konfigurują obecnie rotacji logów. Powinieneś pamiętać o dodaniu opcji `--log-opt max-size` i `--log-opt max-file` w wierszu poleceń lub pliku *daemon.json*, gdy uruchamiasz aplikacje produkcyjnie. Parametry te określają, odpowiednio, maksymalną wielkość pliku przed wykonaniem rotacji oraz maksymalną liczbę plików z logami, jaka będzie przechowywana. Parametr `max-file` nic nie wnosi, jeśli nie masz ustawionej wartości `max-size`, określającej, kiedy wykonywać rotowanie logów. Zauważ, że gdy jest to włączone, mechanizm `docker container logs` będzie zwracał dane jedynie z bieżącego pliku logów.

Zaawansowane mechanizmy obsługi logów

W przypadku, gdy domyślny mechanizm nie wystarcza — a przy skalowaniu jest to bardzo prawdopodobne — można wykorzystać wspierane przez Dockera konfigurowalne mechanizmy obsługi logów. Lista dostępnych wtyczek tego typu ciągle się powiększa. Obecnie wspierana jest opisywana wcześniej wtyczka `json-file`, a także wtyczki `syslog`, `fluentd`, `journald`, `gelf`, `awslogs`, `splunk`, `gcplogs`, `local` oraz `logentries`, które pozwalają na przesyłanie logów do różnych popularnych frameworków i usług.

Wymieniliśmy tutaj długą listę wtyczek. Najprostszą do wykorzystania obecnie przy skalowaniu Dockera jest wtyczka dająca możliwość przesyłania logów kontenera bezpośrednio z Dockera do sysloga. Możesz ją skonfigurować, dodając opcję `--log-driver=syslog` w wierszu poleceń przy uruchamianiu Dockera, lub ustawić tę wartość jako domyślną w pliku *daemon.json* dla wszystkich kontenerów.

Plik *daemon.json* zawiera konfigurację serwera `dockerd`. Można go zazwyczaj znaleźć w katalogu */etc/docker* na serwerze. W przypadku Docker Desktop ten plik można edytować w interfejsie graficznym w *Preferences/Docker Engine*. Jeśli zmodyfikujesz ten plik, konieczne będzie zrestartowanie Docker Desktop lub demona `dockerd`.

Istnieje również duża liczba niezależnie tworzonych wtyczek. Widzieliśmy różne skutki działania takich dodatkowych wtyczek, przede wszystkim dlatego, że komplikują one instalację i utrzymanie Dockera. Może się jednak zdarzyć, że istnieje zaimplementowana zewnętrzna wtyczka, która idealnie odpowiada potrzebom Twojego systemu i opłaca się trudzić z jej instalacją i utrzymaniem.

Istnieją pewne ograniczenia związane ze sterownikami obsługi logów. Docker potrafi na przykład korzystać tylko z jednego sterownika w danej chwili. Oznacza to, że możesz korzystać ze sterownika logowania `syslog` lub `gelf`, ale nie razem ze sterownikiem `json-file`. Jeśli zmienisz sterownik logowania na inny niż `json-file` lub `journald`, nie będziesz mógł używać polecenia `docker container logs`! Może to być niespodzianką, którą należy wziąć pod uwagę przy zmienianiu sterownika.

Istnieją wtyczki umożliwiające wysyłanie logów do zdalnego punktu i jednoczesne zachowanie kopii w formacie JSON dla polecenia `docker container logs`, ale zawsze należy się upewnić, czy wybrana wtyczka obsługuje taki tryb. W przypadku każdego ze sterowników trzeba uwzględnić zbyt wiele detali, by teraz je wszystkie omawiać, ale przy wybieraniu sterownika powinieneś pamiętać o zależności pomiędzy gwarancją zapisania logów a potencjalną możliwością zablokowania instancji Dockera. Zalecane jest korzystanie z rozwiązań opartych na UDP i innych nieblokujących.

Tradycyjnie większość systemów Linux posiada jakiś rodzaj mechanizmu odbierającego logi. Może być to `syslog`, `rsyslog` lub jedna z wielu innych opcji. Ten protokół istnieje w różnych postaciach od długiego czasu i jest dość dobrze wspierany w większości wdrożeń. Przy migrowaniu do Dockera z tradycyjnych środowisk typu Linux lub Unix wiele firm posiada już infrastrukturę `sysloga`, co oznacza, że jest to też często najłatwiejsza ścieżka migracji.

Wiele nowszych dystrybucji Linuksa korzysta z systemu uruchamiania `systemd` i dlatego domyślnie wykorzystuje do obsługi logów mechanizm `journald`, który różni się od `sysloga`.

Choć klasycznym rozwiązaniem jest syslog, również on ma swoje wady. Sterownik `syslog` w Dockerze wspiera połączenia TLS, TCP i UDP, co brzmi wspaniale, ale powinieneś ostrożnie podchodzić do przesyłania strumienia logów z Dockera do zdalnego serwera logów za pomocą połączenia TCP lub TLS. Problemem jest tutaj to, że oba te mechanizmy korzystają z sesji TCP obsługujących połączenie i Docker próbuje podczas uruchamiania kontenera połączyć się ze zdalnym serwerem obsługującym logi. Jeśli nie uda mu się nawiązać połączenia, zablokuje uruchamianie kontenera. Jeśli jest to Twój domyślny mechanizm obsługi logów, może zawieść w każdej chwili w każdej instancji.

Nie jest to szczególnie korzystna sytuacja w przypadku systemów produkcyjnych i dlatego zalecane jest, by korzystać z opcji UDP dla sysloga, gdy używasz sterownika `syslog`. Oznacza to, że Twoje logi nie będą szyfrowane i nie ma gwarancji, że zostaną dostarczone. Istnieją różne zdania na temat

obsługi logów i musisz znaleźć równowagę pomiędzy koniecznością posiadania logów a niezawodnością Twojego systemu. My skłaniamy się w kierunku niezawodności, ale jeśli uruchamiasz bezpieczne środowisko wymagające specjalnej kontroli, możesz mieć inne priorytety.

Możesz przekazywać logi bezpośrednio do zdalnego serwera kompatybilnego z syslogiem z pojedynczego kontenera, ustawiając opcję logowania `syslog-address` w taki sposób: `--log-opt syslog-address=udp://192.168.42.42:123`.

Ostatnią pułapką, o której trzeba wiedzieć, występującą w większości wtyczek do logowania, jest fakt, że ich domyślnym zachowaniem jest blokowanie, co oznacza, że zwiększony napór ze strony logów może wywołać problemy z Twoją aplikacją. Możesz zmienić to zachowanie, ustawiając `--log-opt mode=non-blocking`, a następnie ustawiając maksymalny rozmiar bufora podobnie do tego: `--log-opt max-buffer-size=4m`. Przy takich ustawieniach aplikacja nie będzie już blokowała się po zapełnieniu bufora. Zamiast tego najstarsze wiersze logów z pamięci będą usuwane. I ponownie: musisz tutaj wziąć pod uwagę swoje potrzeby związane z niezawodnością i biznesową potrzebę posiadania wszystkich logów.

Niektóre zewnętrzne biblioteki i programy zapisują dane w systemie plików z różnych (i czasem nieoczekiwanych) przyczyn. Jeśli spróbujesz zaprojektować czyste kontenery, które nie zapisują danych bezpośrednio w systemie plików kontenera, powinieneś rozważyć wykorzystanie podczas uruchamiania `docker container run` parametrów `--read-only` i `--mount type=tmpfs`, które omówiliśmy w rozdziale 4. Zapisywanie logów *wewnątrz* kontenera nie jest zalecane. Sprawia to, że trudno się do nich dostać, utrudnia ich zachowanie po zakończeniu życia kontenera i może spowodować spustoszenie w systemie plików obsługującym Dockera.

Monitorowanie Dockera

Wśród najważniejszych wymagań stawianych systemom produkcyjnym jest to, by były one łatwe do obserwowania i kontrolowania. System produkcyjny, którego zachowania nie możesz monitorować, na dłuższą metę nie będzie się sprawdzał. W nowoczesnych środowiskach monitorujemy wszystkie znaczące parametry i raportujemy tak wiele użytecznych danych, ile tylko możemy. Docker udostępnia mechanizm pozwalający sprawdzić stan kontenera oraz kilka prostych możliwości raportowania za pomocą `docker container stats` i `docker system events`. Pokażemy Ci je, a następnie przyjrzymy się propozycji firmy Google, która też pozwala na tworzenie przyjaznych informacji graficznych, a także — obecnie eksperymentalnemu — mechanizmowi Dockera eksportującemu parametry kontenera do systemu monitorowania Prometheus.

Statystyki kontenerów

Zacznijmy od narzędzi wiersza poleceń, które są dostarczane z samym Dockerem. CLI Dockera zawiera mechanizm pozwalający przeglądać ważne statystyki działających kontenerów. Narzędzie wiersza poleceń może przechwytywać strumień danych z tego mechanizmu i co kilka sekund przekazywać informacje na temat jednego kontenera lub ich większej liczby, dostarczając prostych

informacji statystycznych o tym, co się dzieje. Program `docker container stats`, tak jak linuksowe polecenie `top`, przejmuje bieżący terminal i aktualizuje wyświetlane na ekranie informacje. Trudno pokazać to w książce, tak więc przedstawimy tylko przykład, w którym aktualizacja danych następuje co kilka sekund.

Statystyki dostępne w wierszu poleceń

Uruchom przykładowy kontener:

```
$ docker container run --rm -d --name stress \
docker.io/spkane/train-os:latest \
stress -v --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 60s
```

Następnie użyj polecenia `stats`, by przyjrzeć się nowemu kontenerowi:

```
$ docker container stats stress
CONTAINER ID NAME CPU % MEM USAGE/LIMIT MEM % NET I/O BLOCK I/O PIDS
1a9f52f0855f stress 476.50% 36.09MiB/7.773GiB 0.45% 1.05kB/0B 0B/0B 6
```

W dowolnym momencie możesz przerwać wyświetlanie statystyk za pomocą kombinacji klawiszy *Ctrl+C*.

Możesz też użyć opcji `--no-stream`, by uzyskać informację o statystykach w pojedynczym punkcie w czasie — nie będzie ona aktualizowana i polecenie zakończy działanie zaraz po wyświetleniu informacji.

Spróbujmy podzielić te skondensowane informacje na mniejsze, łatwiejsze do opanowania kawałki. Widzimy tutaj:

- Identyfikator kontenera (ale nie nazwę).
- Część mocy procesora, jaką obecnie on zużywa. Sto procent oznacza cały jeden rdzeń procesora.
- Ilość pamięci, jaką on w tej chwili zużywa, wraz z informacją, ile maksymalnie będzie mógł wykorzystać.
- Statystyki dotyczące sieciowych i blokowych operacji zapisu i odczytu.
- Liczbę procesów aktywnych wewnątrz kontenera.

Podczas debugowania niektóre z tych informacji będą bardziej przydatne, inne mniej, dlatego zobaczmy, do czego możesz je wykorzystać.

Jedną z bardziej użytecznych informacji jest wielkość wykorzystanej pamięci porównana z limitem ustawionym dla kontenera. Jednym z typowych problemów obserwowanych w produkcyjnie działających kontenerach jest to, że zbyt agresywnie ustawione ograniczenia pamięci mogą powodować, że mechanizm jądra dbający o przestrzeganie tego limitu będzie ciągle zatrzymywał kontener. Polecenie `stats` może pomóc w identyfikowaniu i eliminowaniu tego typu problemów.

Jeśli chodzi o statystyki I/O, to gdy uruchamiasz wszystkie swoje aplikacje w kontenerach, takie zestawienie pozwoli dokładnie zobaczyć, gdzie wykorzystywane są systemowe operacje I/O. Poza kontenerami było to dużo trudniejsze do ustalenia!

Informacją przydatną przy debugowaniu jest również liczba aktywnych procesów w kontenerze. Jeśli masz aplikację, która tworzy procesy potomne bez ich usuwania, można to dość szybko zauważyć.

Jedną ze wspaniałych cech `docker container stats` jest to, że polecenie to może pokazać nie tylko informacje na temat jednego kontenera, ale w jednym zestawieniu pokazuje informacje na temat wszystkich kontenerów. Może to wiele wyjaśniać nawet w przypadku komputerów, co do których wydaje Ci się, że wiesz, co się dzieje.

To wszystko jest przydatne i łatwe do strawienia, ponieważ jest sformatowane do łatwego czytania przez człowieka i dostępne z wiersza poleceń. W Docker API istnieje jednak dodatkowy interfejs, udostępniający o *wiele* więcej informacji niż te pokazywane przez klienta działającego w wierszu poleceń. Dotąd w tej książce unikaliśmy bezpośredniego korzystania z API, ale w tym przypadku dane dostarczane przez API są o tyle bardziej bogate, że wywołamy je za pomocą programu `curl`, aby zobaczyć, co robi nasz kontener. Dane uzyskane w ten sposób czyta się dużo mniej wygodnie, ale znajduje się w nich o wiele więcej szczegółów.

Pamiętaj że praktycznie wszystko co można wykonać za pomocą klienta Dockera można też wykonać za pomocą API Dockera. Oznacza to że możesz programowo wykonywać w swojej aplikacji bardzo podobne operacje jeśli będzie to potrzebne.

W kolejnym punkcie znajdziesz dobre wprowadzenie do samodzielnego korzystania z API.

Interfejs stats

Węzeł `/stats/`, z którego będziemy korzystali w API, będzie przesyłał do nas strumień z danymi o statystykach tak długo, jak długo będziemy utrzymywali otwarte połączenie. Ponieważ, jako ludzie, nie jesteśmy zbyt wprawieni w parsowaniu formatu JSON, będziemy pobierali pojedyncze wiersze i skorzystamy z narzędzia `jq`, by je odpowiednio sformatować — zajmiemy się tym niedługo. Aby to polecenie zadziałało, musisz mieć je zainstalowane. Jeśli nie masz go i chcesz zobaczyć dane JSON, możesz pominąć przekierowanie strumienia do `jq`, ale w takiej sytuacji zobaczysz jedynie prosty, niesformatowany JSON. Jeśli używasz innego ulubionego programu do formatowania JSON, możesz go tutaj wykorzystać.

Większość demonów Dockera po zainstalowaniu udostępnia API jedynie na gniazdku typu Unix, bez dostępu przez TCP. W takiej sytuacji uruchomimy `curl` na samym serwerze Dockera, aby dostać się do API. Jeśli planujesz monitorowanie tego interfejsu na produkcji, będziesz musiał udostępnić API Dockera na porcie TCP. Nie zalecamy tego, ale dokumentacja Dockera (*https://dockr.ly/2Lzuox2*) przeprowadzi Cię przez ten proces.

Jeśli nie pracujesz bezpośrednio na serwerze Dockera lub korzystasz lokalnie z Docker Desktop, to do ustalenia nazwy lub adresu IP wykorzystywanego przez siebie serwera Dockera może być konieczne sprawdzenie zawartości zmiennej środowiska `DOCKER_HOST` za pomocą polecenia takiego jak `echo $DOCKER_HOST`.

Zacznijmy od uruchomienia kontenera, z którego będziesz mógł odczytać statystyki:

```
$ docker container run --rm -d --name stress \
docker.io/spkane/train-os:latest \
stress -v --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 60s
```

Gdy mamy już działający kontener, możesz uzyskać na jego temat strumień informacji statystycznych w formacie JSON, uruchamiając poniższe polecenie `curl` z dodaną nazwą lub sumą kontrolną działającego kontenera.

W dalszych przykładach wywołujemy polecenie `curl`, podając jako parametr gniazdo Dockera, ale równie dobrze możesz uruchomić je, podając jako parametr port sieciowy Dockera, jeśli masz skonfigurowany.

```
$ curl --no-buffer -XGET --unix-socket /var/run/docker.sock \
        http://docker/containers/stress/stats
```

Strumień statystyk w formacie JSON się nie zakończy. W naszym przypadku możemy użyć kombinacji *Ctrl+C*, by zakończyć ich pobieranie.

Aby pobrać pojedynczy zestaw danych ze statystykami, możemy uruchomić takie polecenie:

```
$ curl -s -XGET --unix-socket /var/run/docker.sock \
        http://docker/containers/stress/stats | head -n 1
```

W końcu, jeśli mamy `jq` (*https://stedolan.github.io/jq*) lub inne narzędzie, które może sformatować dane JSON, możemy sprawić, że uzyskane dane będą bardziej czytelne dla człowieka, co pokazujemy niżej:

```
$ curl -s -XGET --unix-socket /var/run/docker.sock \
    http://docker/containers/stress/stats | head -n 1 | jq
{
    "read": "2022-07-31T17:41:59.10594836Z",
    "preread": "0001-01-01T00:00:00Z",
    "pids_stats": {
        "current": 6,
        "limit": 18446744073709552000
    },
    "blkio_stats": {
        "io_service_bytes_recursive": [
            {
                "major": 254,
                "minor": 0,
                "op": "read",
                "value": 0
            },
…
        ]
    },
    "num_procs": 0,
    "storage_stats": {},
    "cpu_stats": {
        "cpu_usage": {
            "total_usage": 101883204000,
            "usage_in_kernelmode": 43818021000,
            "usage_in_usermode": 58065183000
…
        },
    },
```

```
        "memory_stats": {
            "usage": 183717888,
            "stats": {
                "active_anon": 0,
                "active_file": 0,
…
            },
            "limit": 8346021888
        },
        "name": "/stress",
        "id": "9be7c9de26864ac97e07fc3d8e3ffb5bb52cc2ba49f569d4ba8d407f8747851f",
        "networks": {
            "eth0": {
                "rx_bytes": 1046,
                "rx_packets": 9,
…
            }
        }
    }
```

Pojawia się tu *wiele* informacji. Skróciliśmy je, by nie marnować więcej drzew ani elektronów, niż jest to potrzebne, ale nadal jest tutaj dużo do przetrawienia. Główną ideą jest pokazanie Ci, ile danych na temat każdego kontenera jest dostępnych w API. Nie będziemy poświęcać zbyt dużo czasu na zagłębianie się w detale, ale możesz tutaj znaleźć dość szczegółowe informacje o użyciu pamięci, a także o operacjach blokowych i użyciu procesora.

Jest to wspaniałe źródło informacji również wtedy, gdy tworzysz swoje własne mechanizmy monitorowania, ale dla każdego kontenera istnieje oddzielny adres, przez co nie możesz uzyskać statystyk dotyczących wszystkich kontenerów za pomocą jednego zapytania.

Sprawdzanie stanu kontenera

Tak jak w przypadku każdej innej aplikacji, po uruchomieniu kontenera możliwe jest, że zacznie on działać, ale w rzeczywistości nigdy nie osiągnie właściwego stanu, w którym będzie mógł obsłużyć zapytania. Systemy produkcyjne też czasem zawodzą i Twoja aplikacja może znaleźć się w niewłaściwym stanie w dowolnym momencie swojego życia, co sprawia, że musisz mieć możliwość poradzenia sobie z taką sytuacją.

Wiele środowisk produkcyjnych ma ustandaryzowane sposoby sprawdzania stanu aplikacji. Niestety, nie istnieje jasny standard określający, w jaki sposób ujednolicać to w ramach całej organizacji, i mało prawdopodobne, by wiele firm podchodziło do tego tematu w taki sam sposób. Z tego powodu zbudowano systemy monitoringu radzące sobie z tymi komplikacjami w taki sposób, że mogą one działać w wielu różnych systemach produkcyjnych. Jest to miejsce, gdzie standaryzacja przyniosłaby wiele dobrego.

Aby pomóc w usuwaniu tych komplikacji i wprowadzić standardowy uniwersalny interfejs, Docker dodał mechanizm pozwalający na sprawdzanie stanu kontenerów. Korzystając z analogii do kontenerów używanych w transporcie, można powiedzieć, że kontenery linuksowe powinny wyglądać na zewnątrz tak samo, niezależnie od tego, co znajduje się wewnątrz, tak więc wbudowany w Dockera mechanizm nie tylko standaryzuje sposób sprawdzania stanu kontenerów, ale również zapewnia

izolację pomiędzy tym, co znajduje się wewnątrz kontenera, a tym, jak wygląda to na zewnątrz. Oznacza to, że kontenery z Docker Hub lub innych publicznych repozytoriów mogą implementować standaryzowany mechanizm do sprawdzania stanu, który będzie działał w każdym środowisku Dockera zaprojektowanym do uruchamiania kontenerów produkcyjnych.

Mechanizmy do sprawdzania stanu są elementem konfigurowanym podczas budowania obrazu kontenera i są tworzone za pomocą definicji `HEALTHCHECK` w *Dockerfile*. Ta dyrektywa mówi demonowi Dockera, jakie polecenie może on wykonać wewnątrz kontenera, aby sprawdzić, czy kontener jest w dobrym stanie. Dopóki polecenie to kończy działanie, zwracając wartość zero (0), Docker będzie przekonany, że kontener działa poprawnie. Każdy inny kod przy zakończeniu działania będzie wskazywał Dockerowi, że kontener nie jest we właściwym stanie — w takiej sytuacji odpowiednie działania mogą być podjęte przez system operacyjny lub system monitorujący.

Będziemy korzystali z poniższego projektu, badając Docker Compose w dalszej części tej książki. Na razie jednak skupmy się na tym, że zawiera on użyteczny przykład wykorzystania mechanizmu sprawdzającego stan kontenera. Pobierz kopię kodu i przejdź do katalogu *rocketchat-hubot-demo/mongodb/docker/*:

```
$ git clone https://github.com/spkane/rocketchat-hubot-demo.git \
    --config core.autocrlf=input
$ cd rocketchat-hubot-demo/mongodb/docker
```

W katalogu tym zobaczysz plik *Dockerfile* oraz skrypt o nazwie `docker-healthcheck`. Jeśli zajrzysz do *Dockerfile*, zobaczysz coś takiego:

```
FROM docker.io/bitnami/mongodb:4.4
# Newer Upstream Dockerfile:
# https://github.com/bitnami/containers/blob/
# f9fb3f8a6323fb768fd488c77d4f111b1330bd0e/bitnami/mongodb
# /5.0/debian-11/Dockerfile

COPY docker-healthcheck /usr/local/bin/

# Useful Information:
# https://docs.docker.com/engine/reference/builder/#healthcheck
# https://docs.docker.com/compose/compose-file/#healthcheck
HEALTHCHECK CMD ["docker-healthcheck"]
```

Jest on bardzo krótki, ponieważ za bazę przyjęliśmy dostępny publicznie obraz Mongo (*https://oreil.ly/Is1yt*)[2] i nasz obraz dziedziczy stamtąd wiele rzeczy, takich jak `ENTRYPOINT`, domyślnie wykonywane polecenie czy port do udostępnienia.

Firma Bitnami znacząco zmodyfikowała swoje repozytoria kontenerów na początku 2023 roku i zamieszczony odnośnik prowadzi do wersji 5.0 MongoDB. W tym przykładzie wykorzystujemy MongoDB 4.4, ale wersja dostępna pod podanym adresem powinna pomóc zrozumieć kontekst.

```
EXPOSE 27017
```

[2] URL: *https://github.com/bitnami/containers/blob/f9fb3f8a6323fb768fd488c77d4f111b1330bd0e/bitnami/mongodb/5.0/debian-11/Dockerfile*.

```
ENTRYPOINT [ "/opt/bitnami/scripts/mongodb/entrypoint.sh" ]
CMD [ "/opt/bitnami/scripts/mongodb/run.sh" ]
```

Miej świadomość, że Docker będzie przekazywał zapytania do portu kontenera nawet wtedy, gdy kontener i jego procesy będą jeszcze startowały.

W naszym pliku *Dockerfile* dodajemy więc jedynie pojedynczy skrypt, który może sprawdzać stan naszego kontenera, i definiujemy polecenie `HEALTHCHECK`, które uruchamia ten skrypt.

Możesz zbudować ten kontener w taki sposób:

```
$ docker image build -t mongo-with-check:4.4 .
=> [internal] load build definition from Dockerfile                    0.0s
=> => transferring dockerfile: 37B                                     0.0s
=> [internal] load .dockerignore                                       0.0s
=> => transferring context: 2B                                         0.0s
=> [internal] load metadata for docker.io/bitnami/mongodb:4.4          0.5s
=> [internal] load build context                                       0.0s
=> => transferring context: 40B                                        0.0s
=> CACHED [1/2] FROM docker.io/bitnami/mongodb:4.4@sha256:9162…ae209   0.0s
=> [2/2] COPY docker-healthcheck /usr/local/bin/                       0.0s
=> exporting to image                                                  0.0s
=> => exporting layers                                                 0.0s
=> => writing image sha256:a6ef…da808                                  0.0s
=> => naming to docker.io/library/mongo-with-check:4.4                 0.0s
```

Następnie można uruchomić kontener i sprawdzić wynik działania polecenia `docker container ls`:

```
$ docker container run -d --rm --name mongo-hc mongo-with-check:4.4
5a807c892428ab0641232c82bd477fc8d1142c9e15c27d5946b8bfe7056e2695

$ docker container ls
… IMAGE                  … STATUS                         PORTS …
… mongo-with-check:4.4 … Up 1 second (health: starting) 27017/tcp …
```

Powinieneś zauważyć, że kolumna `STATUS` zawiera teraz zamkniętą w nawiasy sekcję `health`. Na początku będzie tutaj wyświetlana informacja `health: starting` podczas uruchamiania kontenera. Możesz zmienić czas, przez jaki Docker oczekuje na inicjalizację kontenera, za pomocą parametru `--health-start-period` przekazywanego do `docker container run`. Status zmieni się wtedy na `healthy`, gdy tylko kontener się uruchomi i z sukcesem zakończy sprawdzanie stanu. Może upłynąć więcej niż 40 sekund, zanim ten kontener przejdzie w stan `healthy`:

```
$ docker container ls
... IMAGE                ... STATUS                  PORTS     ...
... mongo-with-check:4.4 ... Up 32 seconds (healthy) 27017/tcp ...
```

Możesz sprawdzić ten status bezpośrednio, korzystając z polecenia `docker container inspect`.

```
$ docker container inspect --format='{{.State.Health.Status}}' mongo-hc
healthy

$ docker container inspect --format='{{json .State.Health}}' mongo-hc | jq

{
  "Status": "healthy",
```

```
    "FailingStreak": 0,
    "Log": [
      ...
    ]
  }
```

Jeśli skrypt sprawdzający stan kontenera zacznie zwracać błędy, status zmieni się na `unhealthy` i będziesz mógł wtedy ustalić, w jaki sposób poradzić sobie z sytuacją.

```
$ docker container ls
... IMAGE                ... STATUS                    PORTS     ...
... mongo-with-check:4.4 ... Up 9 minutes (unhealthy) 27017/tcp ...
```

W tym momencie możesz zatrzymać działanie kontenera, wydając polecenie `docker container stop mongo-hc`.

Tak jak w przypadku większości systemów, możesz skonfigurować wiele szczegółów dotyczących sprawdzania stanu, w tym również to, jak często Docker wykonuje testy (`--health-interval`) albo jak wiele błędów będzie potrzebne, by kontener został oznaczony jako niesprawny (`--health-retries`). W razie potrzeby możesz nawet wyłączyć zupełnie sprawdzanie stanu (`--no-healthcheck`).

Ten mechanizm jest bardzo przydatny i powinieneś zdecydowanie rozważyć wykorzystanie go we wszystkich swoich kontenerach. Pomoże Ci to w poprawie zarówno niezawodności środowiska, jak i przejrzystości informacji na temat tego, co w nim działa. Jest on też wspierany przez wiele produkcyjnych mechanizmów do zarządzania i monitorowania, dzięki czemu powinien być łatwy do zaimplementowania.

Jak zawsze użyteczność mechanizmu sprawdzania stanu kontenerów w dużym stopniu zależy od tego, czy dobrze jest on zaimplementowany i czy poprawnie ustala stan usługi.

docker system events

Demon Dockera wewnętrznie generuje strumień zdarzeń związany z cyklem życia kontenera. Dzięki temu różne części systemu uzyskują informacje na temat tego, co się dzieje w innych częściach. Możesz też podłączyć się do tego strumienia, by podejrzeć zdarzenia związane z cyklem życia kontenerów na Twoim serwerze Dockera. Jak już się prawdopodobnie domyślasz, jest to zaimplementowane w narzędziu `docker` jako kolejny parametr wiersza poleceń. Gdy uruchamiasz to polecenie, przejmuje ono terminal i przekazuje Ci ciągły strumień komunikatów. W tle jest utrzymywane długie zapytanie HTTP do API Dockera, które to zapytanie zwraca komunikaty w paczkach JSON, gdy tylko się pojawiają. Narzędzie `docker` dekoduje te dane i wyświetla część z nich w terminalu.

Ten strumień zdarzeń przydaje się w scenariuszach monitorowania lub do wyzwalania dodatkowych działań, takich jak powiadomienie po zakończeniu zadania. Dla ułatwienia debugowania powiadomienie to umożliwia dostrzeżenie momentu zakończenia działania kontenera, nawet jeśli Docker później go zrestartuje. Idąc dalej: może się to przydać, gdy będziesz samodzielnie implementował narzędzia korzystające bezpośrednio z API.

W jednym oknie terminala uruchom polecenie `events`:

```
$ docker system events
```

Zauważysz, że nic się nie wydarzyło.

Przejdź do innego okna terminala i uruchom krótko działający kontener:

```
$ docker container run --rm --name sleeper debian:latest sleep 5
```

W pierwszym oknie terminala z działającym poleceniem `events` powinieneś teraz zobaczyć coś podobnego do poniższych informacji:

```
…09:59.606… container create d6… (image=debian:latest, name=sleeper)
…09:59.610… container attach d6… (image=debian:latest, name=sleeper)
…09:59.631… network connect ea… (container=d60b…, name=bridge, type=bridge)
…09:59.827… container start d6… (image=debian:latest, name=sleeper)
…10:04.854… container die d6… (exitCode=0, image=debian:latest, name=sleeper)
…10:04.907… network disconnect ea… (container=d60b…, name=bridge, type=bridge)
…10:04.922… container destroy d6… (image=debian:latest, name=sleeper)
```

Możesz w dowolnym momencie przerwać wyświetlanie informacji o zdarzeniach, wciskając kombinację klawiszy *Ctrl+C*.

Tak jak w przypadku statystyk Dockera, możesz uzyskać dostęp do zdarzeń Dockera za pomocą narzędzia `curl`, wydając polecenie typu `curl --no-buffer -XGET --unix-socket /var/run/docker.sock http://docker/events`.

W tym przykładzie uruchomiliśmy krótko działający kontener, który odliczył 5 sekund i zakończył działanie.

Komunikaty `container create`, `container attach`, `network connect` oraz `container start` opisują kroki konieczne do uruchomienia kontenera. Gdy kontener kończy działanie, strumień zdarzeń zapisuje kroki takie jak `container die`, `network disconnect` i `container destroy`. Każdy z nich oznacza krok konieczny do zlikwidowania kontenera. Pomocny jest też przekazywany przez Dockera identyfikator obrazu, z którego został utworzony kontener. Może być to na przykład przydatne w łączeniu wdrożeń ze zdarzeniami, ponieważ zazwyczaj nowe wdrożenie korzysta z nowego obrazu.

Jeśli mamy serwer, na którym kontenery nie działają cały czas, strumień zdarzeń systemowych Dockera jest bardzo pomocny w ustaleniu, co i kiedy się dzieje. Jeśli jednak nie obserwujesz wszystkiego na bieżąco, bardzo pomocne jest to, że Docker zapamiętuje niektóre zdarzenia i możesz uzyskać do nich dostęp przez pewien czas po ich wystąpieniu. Możesz zażądać wyświetlenia zdarzeń, jakie zaszły po określonym czasie — za pomocą parametru `--since` — lub przed wskazaną chwilą — za pomocą parametru `--until`. Możesz też użyć obu tych parametrów, aby określić okno zawężające zakres czasu, w którym analizowany problem mógł wystąpić. Do obu tych parametrów informację o czasie można przekazać w formatach takich jak we wcześniej przytaczanym przykładzie (np. `2018-02-18T14:03:31-08:00`).

Istnieje kilka rodzajów zdarzeń, które należy monitorować:

`container oom`

Pojawia się, gdy kontenerowi zabraknie pamięci.

```
container exec_create
container exec_start
container exec_die
```

Pojawiają się, gdy ktoś użyje `docker container exec`, by wejść do kontenera, co może oznaczać naruszenie bezpieczeństwa.

cAdvisor

Programy `docker container stats` oraz `docker system events` są użyteczne, ale nie generują jeszcze wykresów graficznych. Wykresy zaś są bardzo pomocne, gdy próbujemy zobaczyć trendy. Oczywiście inni wypełnili do pewnego stopnia tę lukę. Gdy zaczniesz przeglądać opcje monitorowania Dockera, dowiesz się, że wiele spośród najpopularniejszych narzędzi do monitorowania zawiera już funkcje pozwalające uzyskać lepszy wgląd w dane o wydajności kontenerów i ich bieżącym stanie.

Poza komercyjnymi narzędziami dostarczanymi przez takie firmy jak DataDog, GroundWork czy New Relic istnieje wiele darmowych narzędzi *open source*, takich jak Prometheus czy nawet Nagios. Na temat narzędzia Prometheus powiemy więcej w podrozdziale „Prometheus”. Niedługo po powstaniu Dockera firma Google udostępniła swoje wewnętrzne narzędzie do monitorowania kontenerów jako dobrze utrzymany otwartoźródłowy projekt w serwisie GitHub pod nazwą cAdvisor (*https://github.com/google/cadvisor*). Choć cAdvisor można uruchomić poza Dockerem, nie zdziwi Cię już prawdopodobnie, gdy usłyszysz, że najłatwiej jest po prostu uruchomić kontener Dockera.

Aby zainstalować program cAdvisor w większości systemów Linux, musisz jedynie uruchomić poniższe polecenie.

Polecenie to należy wykonać bezpośrednio na linuksowym serwerze Dockera. Nie zadziała poprawnie uruchomione w systemie Windows lub macOS.

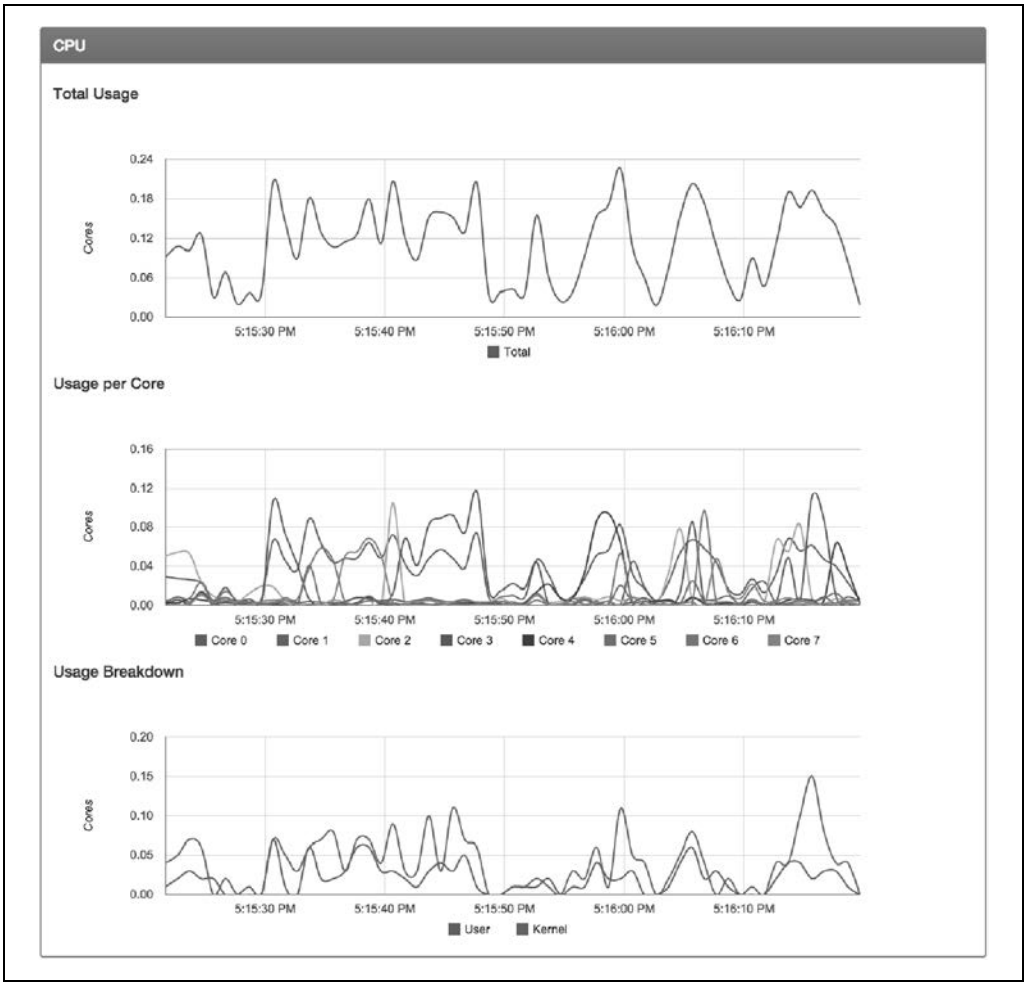
```
$ docker container run \
  --volume=/:/rootfs:ro \
  --volume=/var/run:/var/run:ro \
  --volume=/sys:/sys:ro \
  --volume=/var/lib/docker/:/var/lib/docker:ro \
  --volume=/dev/disk/:/dev/disk:ro \
  --publish=8080:8080 \
  --detach=true \
  --name=cadvisor \
  --privileged \
  --rm \
  --device=/dev/kmsg \
  gcr.io/cadvisor/cadvisor:latest

Unable to find image 'cadvisor/cadvisor:latest' locally
```

```
Pulling repository cadvisor/cadvisor
f0643dafd7f5: Download complete
...
ba9b663a8908: Download complete
Status: Downloaded newer image for cadvisor/cadvisor:latest
f54e6bc0469f60fd74ddf30770039f1a7aa36a5eda6ef5100cddd9ad5fda350b
```

W systemach opartych na RHEL oraz CentOS będziesz musiał do zamieszczonego tutaj polecenia `docker container run` dopisać wiersz `--volume=/cgroup:/cgroup \`.

Po wydaniu tego polecenia i wpisaniu do przeglądarki internetowej adresu serwera Dockera z numerem portu 8080 (np. *http://172.17.42.10:8080/*) powinieneś zobaczyć interfejs programu cAdvisor wraz z różnymi szczegółowymi wykresami przygotowanymi dla serwera oraz pojedynczych kontenerów (patrz rysunek 6.1).



Rysunek 6.1. Wykresy obciążenia procesora z cAdvisor (przykładowe)

Program cAdvisor udostępnia też REST API, które można łatwo odpytywać o szczegółowe informacje na temat monitorowanych systemów:

```
$ curl http://172.17.42.10:8080/api/v2.1/machine/
```

Więcej informacji na temat cAdvisor API można znaleźć w oficjalnej dokumentacji (*https://github.com/google/cadvisor/blob/master/docs/api_v2.md*).

Poziom szczegółowości danych zwracanych przez cAdvisor powinien być wystarczający do zaspokojenia potrzeb związanych z tworzeniem wykresów i monitorowaniem.

Prometheus

System do monitorowania Prometheus (*https://prometheus.io/*) stał się popularnym rozwiązaniem do monitorowania systemów rozproszonych. W dużej części opiera się on na pobieraniu danych, co oznacza nawiązywanie połączenia i zbieranie statystyk z węzłów w ustalonych odstępach czasu. W Dockerze istnieje zbudowany dla systemu Prometheus interfejs, który sprawia, że można łatwo dołączyć statystyki Twojego kontenera do systemu monitorowania Prometheus. W chwili pisania tego tekstu interfejs ten ma status eksperymentalnego i nie jest włączany domyślnie w serwerze `dockerd`. Nasze doświadczenia z nim pokazują, że działa dobrze i jest dość zgrabnym rozwiązaniem, co zaraz zademonstrujemy. Powinniśmy tutaj zaznaczyć, że to rozwiązanie służy do monitorowania serwera `dockerd` — w odróżnieniu od innych rozwiązań, które udostępniają informacje na temat kontenerów.

Aby wyeksportować metryki do systemu Prometheus, musimy przekonfigurować serwer `dockerd`, by włączyć mechanizmy eksperymentalne i dodatkowo udostępnić nasłuchiwanie metryk na wybranym porcie. Jest to wygodne, ponieważ nie musimy udostępniać całego API Dockera na porcie TCP, aby pobrać z naszego systemu metryki — to korzyść z punktu widzenia bezpieczeństwa kosztem odrobiny dodatkowej konfiguracji. Aby to wykonać, możemy albo dodać parametry `--experimental` i `--metrics-addr=` w wierszu poleceń, albo umieścić je w pliku *daemon.json*, którego demon używa do przechowywania swojej konfiguracji. Ponieważ wiele dostępnych aktualnie dystrybucji korzysta z `systemd` i zmiana konfiguracji w tym miejscu w dużej mierze zależy od konkretnej instalacji, skorzystamy z możliwości modyfikacji pliku *daemon.json*, która jest bardziej uniwersalna. Zademonstrujemy to na Ubuntu Linux 22.04 LTS. W tej dystrybucji na początku plik zazwyczaj nie istnieje. Umieśćmy więc go tam, korzystając z Twojego ulubionego edytora.

Jak wcześniej wspomniano, plik *daemon.json* w Docker Desktop może być edytowany za pomocą interfejsu graficznego w menu *Preferences/Docker Engine*. Jeśli zmienisz ten plik, będziesz musiał zrestartować Docker Desktop lub demona `dockerd`.

Wklej poniższy tekst lub dodaj poniższe parametry do istniejącego pliku *daemon.json*:

```
{
  "experimental": true,
  "metrics-addr": "0.0.0.0:9323"
}
```

Za każdym razem, gdy udostępniasz usługę w sieci, musisz zastanowić się, jakie możesz wprowadzać zagrożenia bezpieczeństwa. Wierzymy, że korzyści z udostępnienia metryk są tego warte, ale powinieneś dokładnie przemyśleć następstwa tego kroku w swoim przypadku. Na przykład udostępnienie metryk w otwartym internecie zwykle nie jest dobrym pomysłem.

Gdy zrestartujemy Dockera, będziemy mieli proces nasłuchujący na wszystkich adresach na porcie 9323. To tutaj Prometheus podłączy się, by pobrać nasze metryki. Najpierw jednak musimy zrestartować serwer `dockerd` i przetestować interfejs. Docker Desktop zrestartuje się automatycznie, ale w przypadku serwera linuksowego z Dockerem musisz uruchomić polecenie restartujące demona takie jak np. `sudo systemctl restart docker`. Po restarcie demon nie powinien zwracać żadnych błędów. Jeśli błędy się pojawią, najprawdopodobniej w pliku *daemon.json* jest jakiś błąd.

Następnie możesz przetestować API zwracające metryki za pomocą polecenia `curl`:

```
$ curl -s http://localhost:9323/metrics | head -15

# HELP builder_builds_failed_total Number of failed image builds
# TYPE builder_builds_failed_total counter
builder_builds_failed_total{reason="build_canceled"} 0
builder_builds_failed_total{reason="build_target_not_reachable_error"} 0
builder_builds_failed_total{reason="command_not_supported_error"} 0
builder_builds_failed_total{reason="dockerfile_empty_error"} 0
builder_builds_failed_total{reason="dockerfile_syntax_error"} 0
builder_builds_failed_total{reason="error_processing_commands_error"} 0
builder_builds_failed_total{reason="missing_onbuild_arguments_error"} 0
builder_builds_failed_total{reason="unknown_instruction_error"} 0
# HELP builder_builds_triggered_total Number of triggered image builds
# TYPE builder_builds_triggered_total counter
builder_builds_triggered_total 0
# HELP engine_daemon_container_actions_seconds The number of seconds it
# takes to process each container action
# TYPE engine_daemon_container_actions_seconds histogram
```

Jeśli uruchomisz to polecenie u siebie, otrzymasz bardzo podobne wyniki. Mogą nie być identyczne i jest to poprawne, o ile nie będą to komunikaty o błędach.

Mamy więc teraz miejsce, z którego Prometheus może pobrać nasze statystyki. Musimy jednak także uruchomić gdzieś sam system Prometheus. Możemy wykonać to w łatwy sposób, uruchamiając kontener. Najpierw jednak musimy napisać konfigurację. Umieścimy ją w pliku */tmp/prometheus/prometheus.yaml*. Możesz użyć ulubionego edytora, by umieścić poniższy tekst we wskazanym pliku:

```
# Zbieraj metryki co 5 sekund i użyj nazwy 'stats-monitor'
global:
  scrape_interval: 5s
  external_labels:
    monitor: 'stats-monitor'

# Nazwiemy nasze zadanie 'DockerStats' i udostępnimy do adresu mostka docker0
# gdzie będą dostępne statystyki. Jeśli twój interfejs docker0 ma inny
# adres IP to wpisz go tutaj. 127.0.0.1 i localhost nie zadziałają.
scrape_configs:
  - job_name: 'DockerStats'
    static_configs:
    - targets: ['172.17.0.1:9323']
```

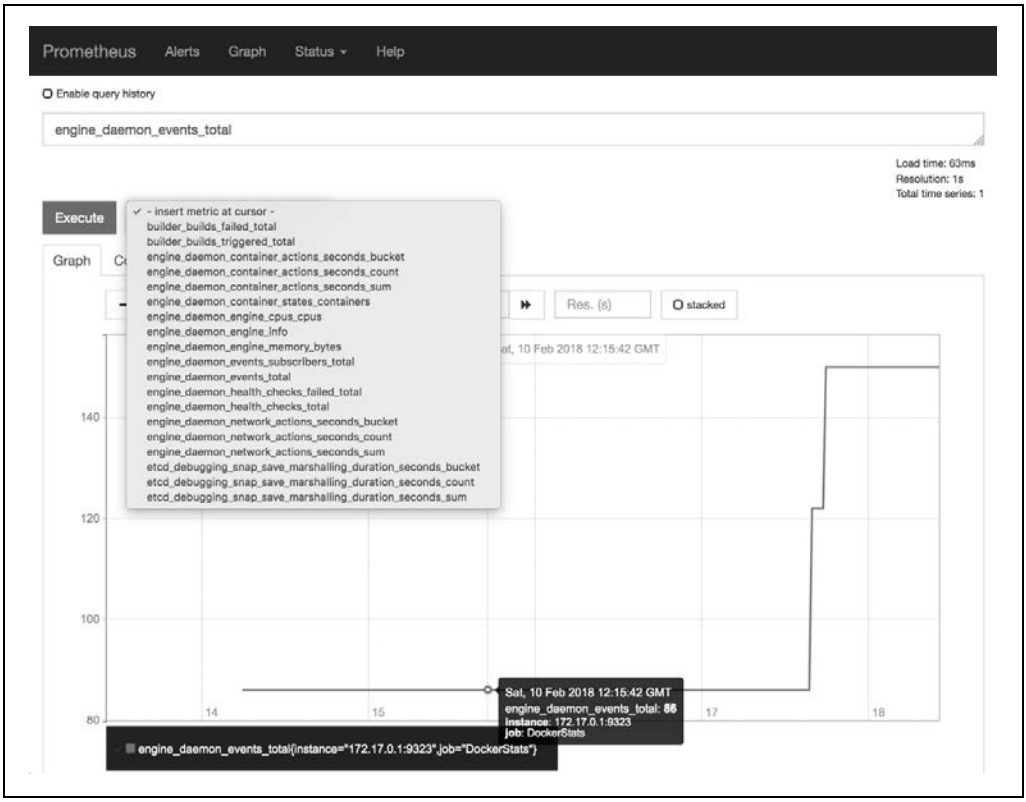
W Docker Desktop możesz też skorzystać z adresu `host.docker.internal:9323` lub `gateway.docker.internal:9323` zamiast pokazanego tutaj `172.17.0.1:9323`. Obie te nazwy powinny wskazywać na adres IP kontenera.

Jak widać w pliku, powinieneś użyć tutaj adresu IP Twojego mostka `docker0` lub adresu interfejsu `ens3` lub `eth0`, ponieważ `localhost` i `127.0.0.1` to adresy nierutowalne z kontenera. Adres, z którego korzystamy tutaj, jest zazwyczaj domyślnym dla `docker0`, dlatego prawdopodobnie będzie on właściwy i w Twoim przypadku.

Gdy już to zapisaliśmy, musimy uruchomić kontener, korzystając z poniższej konfiguracji:

```
$ docker container run --rm -d -p 9090:9090 \
    -v /tmp/prometheus/prometheus.yaml:/etc/prometheus.yaml \
    prom/prometheus --config.file=/etc/prometheus.yaml
```

To powinno uruchomić kontener i zamontować przygotowany plik konfiguracyjny w kontenerze, dzięki czemu będzie on widział potrzebne ustawienia do monitorowania naszego interfejsu Dockera. Jeśli uruchomi się to poprawnie, powinieneś mieć możliwość otwarcia swojej przeglądarki i skierowania się do portu 9090 na serwerze Dockera. W tym momencie powinieneś zobaczyć okno systemu Prometheus, podobne do zaprezentowanego na rysunku 6.2.



Rysunek 6.2. Graf zdarzeń w systemie Prometheus (przykładowy)

Na rysunku widać, że wybraliśmy tutaj jedną z metryk, `engine_daemon_events_total`, i wykreśliliśmy ją w krótkim okresie. W łatwy sposób możesz wybrać dowolną inną metrykę, korzystając z listy rozwijanej. Jeśli poeksperymentujesz z systemem Prometheus, pozwoli Ci to również zdefiniować ostrzeżenia i reguły dla ostrzeżeń w oparciu o te metryki. Nie musisz ograniczać się do monitorowania serwera `dockerd`. Możesz też udostępnić systemowi Prometheus metryki swojej aplikacji. Jeśli jesteś zaintrygowany i zechcesz przyjrzeć się czemuś bardziej zaawansowanemu, możesz zwrócić uwagę na DockProm (*https://github.com/stefanprodan/dockprom*), który wykorzystuje do przygotowywania miłych dla oka paneli Grafanę i również odpytuje o metryki Twego kontenera, takie jak te w interfejsie `/stats` Docker API.

Dalsze eksperymenty

Zdobyłeś już wszystkie podstawowe informacje potrzebne do tego, by móc zacząć uruchamiać kontenery. Warto prawdopodobnie pobrać jeden lub kilka kontenerów z rejestru Docker Hub i samodzielnie poeksperymentować, by utrwalić sobie poznane przed chwilą polecenia. Można z Dockerem wykonać wiele innych czynności, między innymi:

- kopiowanie plików do kontenera i z kontenera za pomocą `docker container cp`,
- zapisywanie obrazu w archiwum TAR za pomocą `docker image save`,
- wczytywanie obrazu z archiwum TAR za pomocą `docker image import`.

Docker ma duży zakres możliwości, który będzie z czasem się rozszerzał. W każdym nowym wydaniu dodawane są kolejne funkcje. W dalszej części książki omówimy dużo bardziej szczegółowo wiele innych poleceń i mechanizmów, przy czym warto pamiętać, że Docker zawiera bardzo wiele mechanizmów.

Podsumowanie

W kolejnym rozdziale zanurzymy się w trochę bardziej techniczne szczegóły dotyczące działania Dockera i tego, w jaki sposób możesz wykorzystać tę wiedzę przy debugowaniu Twojej skonteneryzowanej aplikacji.

Dowiedz się, jak budować nowoczesne, niezawodne systemy rozproszone o wysokiej dostępności.

Mihai Todor, TLCP

Docker radykalnie zmienił proces wdrażania oprogramowania. Obrazy i kontenery Dockera ułatwiają zarządzanie zależnościami, co uprościło testowanie, wdrażanie i skalowanie aplikacji. Technologia ta intensywnie się rozwija, wciąż zmieniają się dostępne narzędzia i zalecane praktyki. To wszystko sprawia, że dogłębne zrozumienie działania współczesnego Dockera nie jest trywialnym zadaniem.

To gruntownie zaktualizowane i uzupełnione wydanie praktycznego przewodnika po wdrażaniu i testowaniu kontenerów Dockera. Przedstawia proces przygotowania pakietu aplikacji ze wszystkimi ich zależnościami, a także jego testowania, wdrażania, skalowania i utrzymywania w środowiskach produkcyjnych. Zawiera omówienie Docker Compose i trybu Docker Swarm, opis zagadnień związanych z Kubernetes, jak również przykłady optymalizacji obrazów Dockera. W tym wydaniu zaprezentowano ponadto najlepsze praktyki i narzędzie BuildKit, opisano wsparcie obrazów wieloarchitekturowych, kontenerów rootless i uwzględniono wiele innych ważnych informacji.

Ucz się z bezcennych praktycznych lekcji zebranych podczas wdrażania Dockera na dużą skalę!

Liz Rice, Isovalent

Ta książka wykracza poza pierwsze fascynacje Dockerem i przygotuje Cię do rzeczywistych wyzwań!

Kelsey Hightower, Google Cloud Platform

W książce między innymi:

- integracja Dockera i kontenerów linuksowych z usługami chmurowymi i Kubernetes
- zarządzanie kontenerami linuksowymi z poziomu wiersza poleceń
- tworzenie i stosowanie obrazów OCI
- sprawne wdrażanie aplikacji w środowiskach produkcyjnych
- wdrażanie kontenerów linuksowych w publicznych i prywatnych chmurach

Sean Kane specjalizuje się w tworzeniu kontenerowych aplikacji produkcyjnych. Jest też autorem nowatorskich rozwiązań dotyczących kontenerów.

Karl Matthias zajmował wysokie stanowiska w kilku czołowych firmach technologicznych. Jest entuzjastą systemów rozproszonych, skalowalnych magazynów danych i zautomatyzowanej infrastruktury.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-0371-5

9 788328 903715

Cena: 87,00 zł