

Konteneryzacja — Laboratorium ASK 30.11.2023

Prowadzący:

- Adam Ples aples@ra.rockwell.com <- tutaj wysyłamy sprawozdanie
- Tymoteusz Kielan tkielan@ra.rockwell.com
- Bartosz Bątopek bbatore@ra.rockwell.com
- Karol Janik karol.janik@rockwellautomation.com

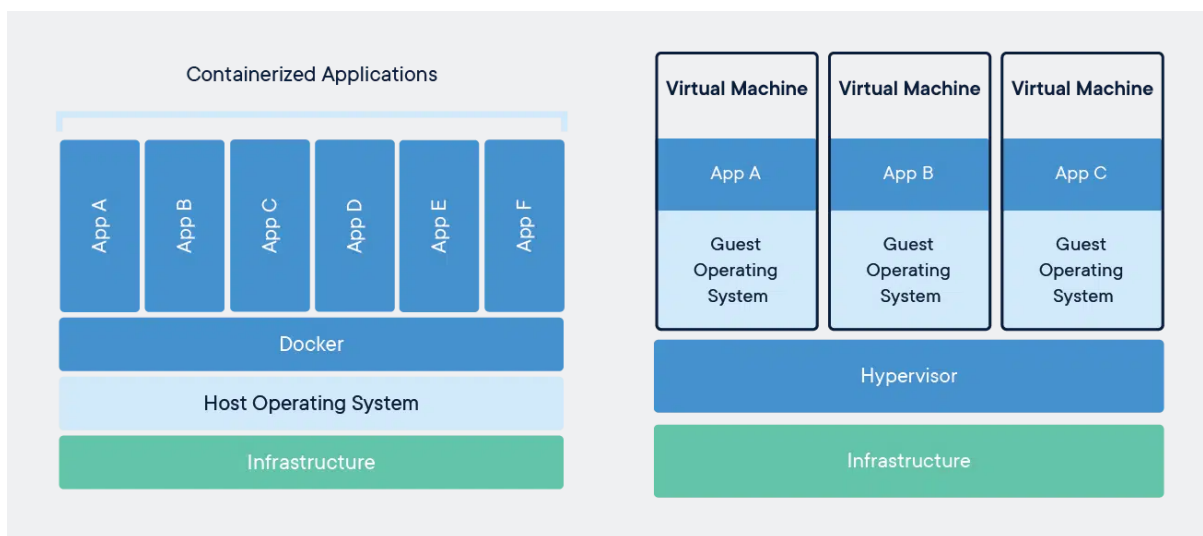
Wstęp

Konteneryzacja to coraz częściej wykorzystywana technologia, pozwalająca na uproszczenie dystrybucji oprogramowania. Umożliwia ona udostępnianie aplikacji wraz ze wszystkimi zależnościami, a także na odizolowanie jej od systemu operacyjnego hosta. W ten sposób można uniknąć problemów związanych z różnorodnością dystrybucji Linuksa: plików konfiguracyjnych, serwisów systemowych i menedżerów pakietów. Obraz kontenera zawiera wszystkie potrzebne pliki oraz konfigurację potrzebną do uruchomienia programu.

Na pierwszy rzut oka konteneryzacja przypomina wirtualizację. I rzeczywiście, konteneryzacja jest rodzajem wirtualizacji, ale na nieco innym poziomie niż wszechobecne maszyny wirtualne. W przypadku maszyn wirtualnych wirtualizacji podlega przede wszystkim procesor. Maszyna wirtualna zostaje „oszukana” przez supervisor, że ma uprzywilejowany dostęp do CPU, i dzięki temu możliwe jest uruchomienie drugiej instancji systemu operacyjnego. Może być to ten sam lub zupełnie inny system, niż host.

W przypadku kontenerów wirtualizacji podlega sam system operacyjny. W praktyce polega to na wydzieleniu „przestrzeni nazw” w zasobach zarządzanych przez jądro systemu. Zasoby te obejmują procesy, sieć, system plików, prawa użytkowników itd. Procesy działające w kontenerze dalej są kontrolowane przez to samo jądro, natomiast znajdują się w swojej własnej przestrzeni nazw, przez co są odizolowane od procesów działających w innych kontenerach i poza nimi. Dzięki temu kontenery nie wymagają dużej ilości pamięci RAM (naruszenie wynika przede wszystkim z braku możliwości współdzielenia bibliotek), a ich wydajność jest praktycznie identyczna jak w przypadku zwykłego procesu. Można powiedzieć więc, że konteneryzacja ma wiele zalet wirtualizacji, bez jej największych wad.

Pierwszym narzędziem umożliwiającym konteneryzację był mechanizm `jail` zaimplementowany w systemie operacyjnym FreeBSD już w 1999 roku. Niedługo później podobne funkcje zaczęto dodawać do jądra Linuksa. W trakcie dzisiejszego laboratorium skupimy się na konkretnym narzędziu wykorzystującym te możliwości – Dockerze. Jako system hosta wykorzystamy Ubuntu 22.04. W folderze `lab_docker` na pulpicie znajduje się obraz maszyny wirtualnej UbuntuDocker, którą należy zaimportować do VMWare Workstation lub VirtualBox i uruchomić. Będziemy korzystać z użytkownika `student`,



Rysunek 1: Schemat prezentujący różnice między kontenerami, a maszynami wirtualnymi

którego hasło to również `student`.

Uwaga: w dalszej części instrukcji występują polecenia, które należy wykonać w systemie hosta (Ubuntu), poprzedzone znakiem `$`, np.:

```
$ sudo apt update
```

... oraz polecenie do wykonania w kontenerze, poprzedzone znakiem `#`, np.:

```
# dnf update
```

Zadania z wykonania których raport powinien znaleźć się w sprawozdaniu zostały wyróżnione pochyłą czcionką.

Instalacja

Dockera zainstalujemy korzystając z pakietów dostępnych w repozytoriach Ubuntu:

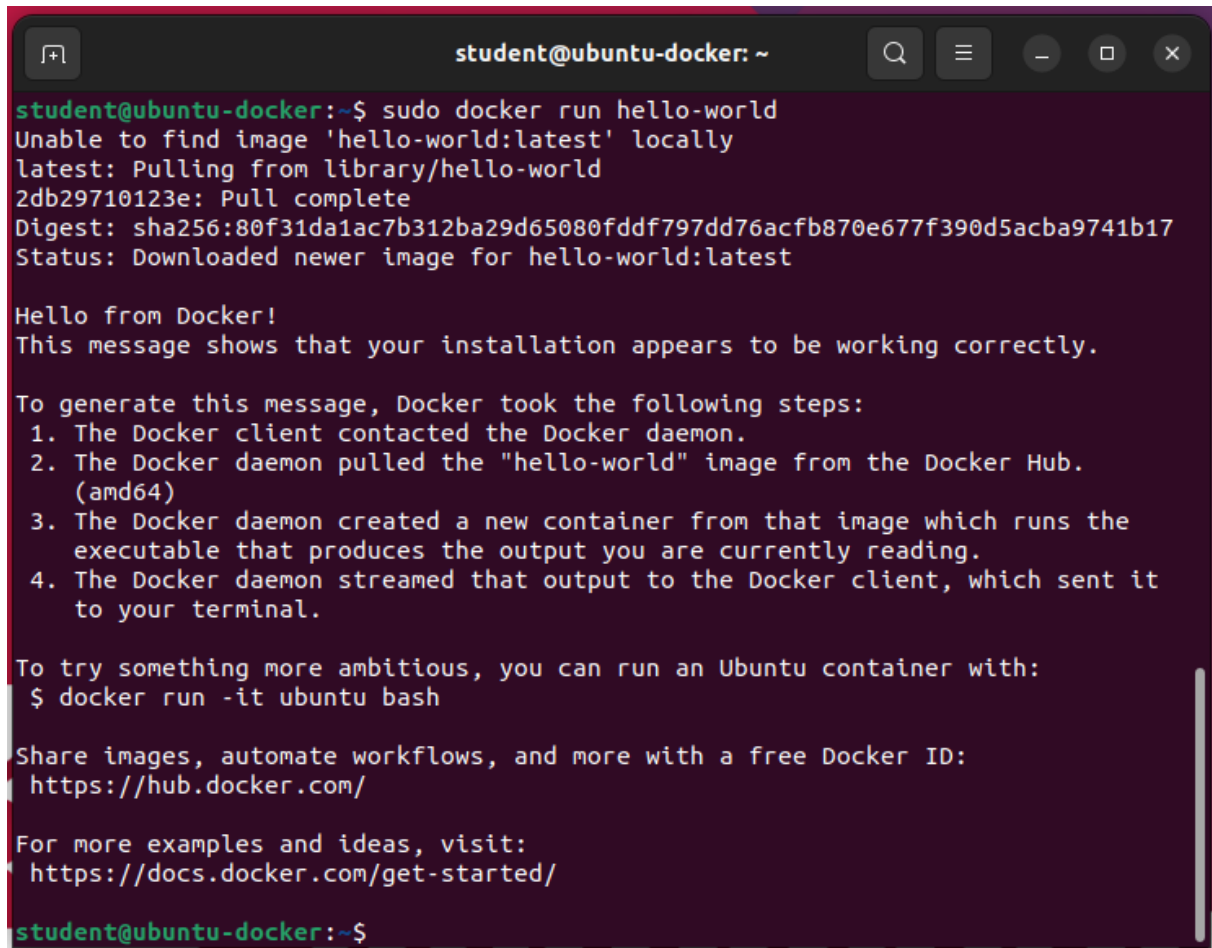
```
$ sudo apt update
$ sudo apt install docker.io
```

Następnie sprawdzimy, czy demon dockera działa:

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

... i dodamy naszego użytkownika do grupy docker, aby móc używać interfejsu tekstowego bez `sudo`:

```
$ sudo usermod -aG docker student
```

A screenshot of a terminal window titled 'student@ubuntu-docker: ~'. The terminal shows the command 'sudo docker run hello-world' being executed. The output indicates that the 'hello-world:latest' image was pulled from the Docker Hub and a container was created and run successfully. The container outputs a 'Hello from Docker!' message and a list of steps taken by Docker. The terminal also shows some additional information about Docker and links to Docker Hub and documentation. The prompt 'student@ubuntu-docker:~\$' is visible at the bottom.

```
student@ubuntu-docker:~$ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:80f31da1ac7b312ba29d65080fddf797dd76acfb870e677f390d5acba9741b17
Status: Downloaded newer image for hello-world:latest

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

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (amd64)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/

student@ubuntu-docker:~$
```

Rysunek 2: Poprawnie zainstalowany Docker

Po wszystkim zrestartujemy maszynę:

```
$ reboot
```

Zadanie: Zainstaluj dockera w systemie Ubuntu na maszynie wirtualnej.

Pierwsze kroki

Interfejs tekstowy

Interfejs tekstowy dockera jest dostępny za pośrednictwem programu o mało ekscytującej nazwie „docker”. Funkcjonalność jest dostępna za pośrednictwem komend zorganizowanych w grupy. Składnia

wywołania jest następująca:

```
$ docker [opcje] [grupa] komenda
```

Przykładowo listę aktualnie działających kontenerów możemy wyświetlić za pomocą polecenia:

```
$ docker container list
```

Niektóre komendy posiadają wygodniejsze aliasy, np. powyższa:

```
$ docker ps
```

Aby uzyskać pomoc dotyczącą konkretnej komendy lub grupy komend należy użyć przełącznika `--help`, np.:

```
$ docker container --help
```

```
$ docker container list --help
```

Zadanie: Sprawdź jakie opcje są dostępne dla komendy `docker ps`.

Podstawowe polecenia do zarządzania kontenerami

Podczas instalacji stworzyliśmy już pierwszy kontener poleceniem:

```
$ docker run hello-world
```

„hello-world” jest tutaj nazwą obrazu, o czym w dalszej części laboratorium. Mimo to polecenie `docker ps` zwraca pustą listę. Jest tak dlatego, że kontener, który utworzyliśmy zakończył swoje działanie i aby go wyświetlić należy użyć przełącznika `--all`:

```
$ docker ps --all
```

Zwróć uwagę na status kontenera: „Exited”. W nawiasie podano kod wyjścia programu.

Kontenery, które nie zostały wprost nazwane, mają przypadkowe nazwy przydzielone przez silnik dockera, takie jak „busy_beaver” albo „eternal_elon”, a także heksadecymalny identyfikator. Dla własnej wygody możemy wybrać bardziej przyjazną nazwę tworząc nowy kontener:

```
$ docker run --name <nazwa kontenera> <nazwa obrazu>
```

Lub zmienić jego nazwę później:

```
$ docker container rename <stara nazwa lub id> <nowa nazwa>
```

Zadanie: Wyświetl i zmień nazwę kontenera `hello-world`.

Możliwe jest również ponowne wystartowanie kontenera:

```
$ docker start <nazwa kontenera>
```

(Analogicznie zatrzymanie: `docker stop <nazwa kontenera>`)

Co ciekawe nie spowoduje to ponownego wyświetlenia komunikatu. Aby go zobaczyć, konieczne jest podłączenie się do standardowego wejścia/wyjścia procesu działającego w kontenerze. Polecenie `docker run` domyślnie podłącza się do nowoutworzonego kontenera, natomiast `docker start` nie.

Aby utworzyć kontener bez podłączania się wykorzystywany jest przełącznik `--detach`:

```
$ docker run --detach <nazwa obrazu>
```

Aby wystartować kontener i jednocześnie się do niego podłączyć użyjemy `--attach`:

```
$ docker container start --attach <nazwa kontenera>
```

Do podłączenia się do kontenera działającego w tle wykorzystamy polecenie:

```
$ docker container attach <nazwa kontenera>
```

Aby się odłączyć używamy kombinacji klawiszy `Ctrl+P`, a następnie `Ctrl+Q`.

W przypadku procesów interaktywnych, takich jak powłoka BASH konieczne jest użycie opcji `--interactive` oraz `--tty`, albo krócej `-it`.

Zadanie: Stwórz nowy kontener w trybie interaktywnym bez podłączania się do niego, korzystając z obrazu `bash`. Następnie podłącz się do niego komendą `attach`. Odłącz się za pomocą `Ctrl+P`, `Ctrl+Q`. Zatrzymaj kontener komendą `container stop`.

Kontenery usuwamy za pomocą polecenia (muszą być najpierw zatrzymane):

```
$ docker container rm <nazwa kontenera>
```

Aby szybko pozbyć się wszystkich zatrzymanych kontenerów możemy również użyć:

```
$ docker container prune
```

Zadanie: Usuń wszystkie kontenery z systemu.

Obrazy

Do tworzenia kontenerów wykorzystywane są tzw. obrazy. Obraz zawiera system plików dla kontenera oraz jego konfigurację. Obrazy dostępne są w publicznych repozytoriach, podobnie jak pakiety systemu

operacyjnego. Na potrzeby laboratorium do maszyny wirtualnej zostały już poprane niektóre potrzebne obrazy. Można je wyświetlić poleceniem:

```
$ docker image list
```

Jak widać obrazy różnią się znacząco rozmiarem, od kilobajtów do setek mega- i gigabajtów. Obrazy są pobierane automatycznie przy tworzeniu kontenerów, ale możliwe jest również ręczne ściągnięcie obrazu za pomocą komendy `pull`

```
$ docker image pull <nazwa obrazu>
```

Nazwa obrazu ma format: `<nazwa repozytorium>[:tag]`, przy czym tag jest opcjonalny i odnosi się zwykle do wersji obrazu. Domyślnym tagiem jest `latest`.

Publiczne repozytorium obrazów można znaleźć np. pod adresem <https://hub.docker.com/>.

Czym właściwie jest kontener

Aby zbadać właściwości kontenerów użyjemy obrazu bazującego na systemie operacyjnym. Celowo skorzystamy z innego systemu, niż nasz host: z popularnej dystrybucji Linuksa Fedora.

Kontener uruchomimy w trybie interaktywnym, to jest z dostępem do powłoki:

```
$ docker container run --interactive --tty --name fedora fedora
```

Na początek sprawdzimy jakiego jądra używamy i porównajmy je z systemem hosta:

```
$ uname -a
```

```
# uname -a
```

Jak widać w obu przypadkach wynik jest niemalże identyczny. Co więcej, nawet w kontenerze powstałym z obrazu Fedory jądro identyfikuje się jako Ubuntu. Jest tak, ponieważ jest to dokładnie to samo jądro.

Jako jaki użytkownik jesteśmy zalogowani? Polecenie:

```
# id
```

... powie nam, że jako `root`. Czy oznacza to, że docker daje nam nieograniczony dostęp do systemu operacyjnego? Na szczęście nie, ten `root` jest innym rootem, istniejącym tylko wewnątrz kontenera. Podglądnijmy plik `/etc/passwd` w kontenerze i na hoście:

```
# cat /etc/passwd
```

Okazuje się, że w kontenerze istnieje zupełnie osobna lista użytkowników. W rzeczy samej, cały system plików nie pochodzi z Ubuntu, ale z obrazu ściągniętego z Docker huba. Zawiera on kopie plików konfiguracyjnych, podstawowych programów, takich jak powłoka BASH, a także menedżer pakietów z systemu Fedora.

Dowiedzmy się teraz jakie procesy działają wewnątrz kontenera. Aby to zrobić posłużymy się poleceniem:

```
# ps aux
```

Jego wywołanie zwraca błąd, ponieważ nawet to podstawowe narzędzie nie jest częścią obrazu. Konieczne jest doinstalowanie pakietu za pomocą menadżera pakietów systemu Fedora, a więc dnf (a nie apt, jak w Ubuntu). Aby dowiedzieć się jaki pakiet zawiera program `ps` użyjemy więc polecenia:

```
# dnf provides '*bin/ps'
# dnf install procps-ng
```

Po wywołaniu polecenia okazuje się, że w kontenerze działają tylko dwa programy: powłoka `BASH` oraz samo polecenie `ps`. Dodatkowo powłoka działa z PIDem 1, który w pełnym systemie Linux jest zarezerwowany dla programu `init` (w nowoczesnych dystrubucjach zwykle `systemd`), czyli pierwszego procesu uruchamianego przez jądro. Zadaniem tego programu jest uruchomienie i nadzorowanie wszystkich usług działających w systemie. Kontener natomiast nie posiada zwykle skryptów startowych ani żadnych usług. Pełne uruchomienie demona takiego jak `systemd` jest wręcz niemożliwe w kontenerze.

Co ciekawe proces działający w kontenerze jest również widoczny „na zewnątrz”. Aby to zweryfikować uruchomimy wewnątrz kontenera polecenie `more`, np.

```
# more /etc/fedora-release
```

Sprawdźmy jego PID wewnątrz kontenera, używając komendy `container exec`, która uruchamia dowolny program wewnątrz kontenera. `pidof more` zwróci PID procesu o podanej nazwie:

```
$ docker container exec fedora pidof more
```

Następnie możemy użyć `ps` w Ubuntu, aby znaleźć interesujący nas proces:

```
$ ps aux | grep "more /etc/fedora-release"
```

Jak widać ten sam proces jest widoczny z innym PIDem. Co więcej, możliwe jest wpływanie na niego, np. przez wysłanie sygnału:

```
$ sudo kill -SIGINT <pid z ubuntu>
```

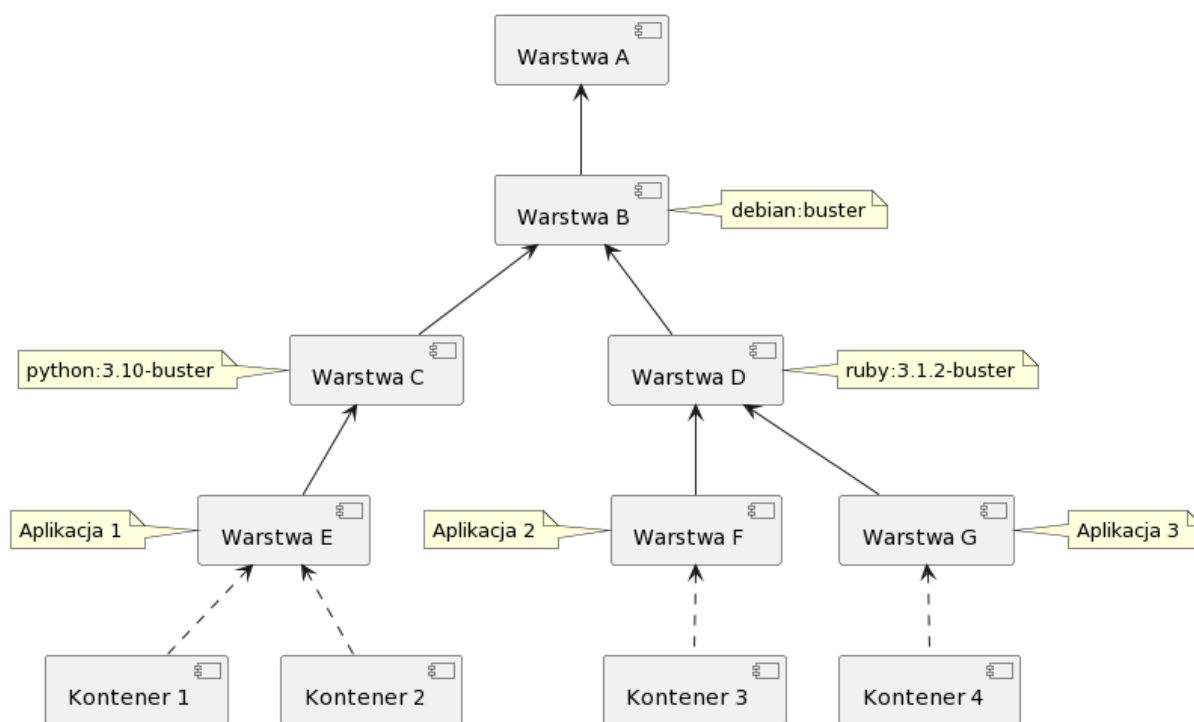
Polecenie `kill` wysyła sygnał do procesu o podanym PIDzie. `SIGINT` (interrupt) jest odpowiednikiem

wciśnięcia kombinacji `Ctrl+C` z klawiatury.

Warstwy i sterownik overlay

Obrazy są *niezmienne* (immutable), co oznacza, że nie można do nich dodać lub usunąć z nich plików. Możliwe jest natomiast utworzenie nowego obrazu, bazującego na istniejącym. Różnice między obrazami zapisywane są w tzw. warstwach. Każda warstwa posiada unikalny identyfikator. Taka architektura pozwala na współdzielenie obrazów między kontenerami (a także warstw między obrazami), co znacznie redukuje wymagania dockera jeśli chodzi o przestrzeń na dysku. Za to zachowanie jest odpowiedzialny sterownik `overlay` (konkretnie `overlay2`).

Schematycznie pokazano to na diagramie:



Rysunek 3: Współdzielenie warstw w Dockerze

Niezmienność obrazu nie powoduje, że system plików w kontenerze jest „tylko do odczytu”. Docker wykorzystuje tutaj technikę kopiowania przy zapisie (copy-on-write). W momencie zapisu do pliku jest on kopiowany do nowej warstwy, tzw. roboczej, i tam modyfikowany. Przy odczycie brana jest pod uwagę ta warstwa, gdzie plik ostatnio się zmienił.

Każdy kontener posiada warstwę roboczą. Jej zawartość jest tracona po jego usunięciu. Rozmiar warstwy roboczej można wyświetlić używając przełącznika `--size` dla komendy `ps`.

Zadanie: Utwórz trzy kontenery korzystając z obrazu `ubuntu:latest`. Zanotuj o ile zwiększa się wykorzystanie dysku po utworzeniu każdego z nich. Można do tego wykorzystać polecenie `df -BM /`. Zaktualizuj system w jednym z kontenerów poleceniami `apt update` i `apt upgrade`. Ponownie sprawdź wykorzystanie dysku w systemie hosta i porównaj z danymi zwracanymi przez `docker ps --size`.

Zadanie: Pobierz obraz `mariadb:10.8` (bazujący na obrazie `ubuntu`) używając polecenia `docker pull <nazwa obrazu>`. Zanotuj o ile zwiększa się wykorzystanie dysku po wykoaniu polecenia. Porównaj wynik z danymi zwracanymi przez `docker image list`.

Konteneryzacja aplikacji

W kolejnej części laboratorium skupimy się na uruchomieniu w kontenerze przykładowej aplikacji. Kod aplikacji korzystającej z frameworka Ruby on Rails znajduje się w katalogu `~/lab_docker/example_app`. Aby ją uruchomić konieczny jest interpreter języka Ruby, oraz odpowiednie biblioteki. Ani jedno, ani drugie nie jest zainstalowane w naszym systemie.

Utworzenie obrazu aplikacji

Pracę zaczniemy od zbudowania obrazu aplikacji. Do definiowania obrazów zwykle wykorzystuje się pliki `Dockerfile`, które opisują m. in. z jakiego obrazu bazowego będziemy korzystać, jakie pakiety należy zainstalować oraz jakie pliki skopiować do obrazu.

Zadanie: utwórz w katalogu `~/lab_docker/example_app` plik o nazwie `Dockerfile` i o następującej zawartości:

```
FROM ruby:3.1.2

COPY . /opt/example_app
WORKDIR /opt/example_app/

RUN gem install rails bundler
RUN bundle install

CMD ["/bin/bash", "start.sh"]

EXPOSE 3000/tcp
```

Zadanie: Korzystając z dokumentacji pod adresem <https://docs.docker.com/engine/reference/builder/> wytłumacz znaczenie poszczególnych instrukcji w pliku.

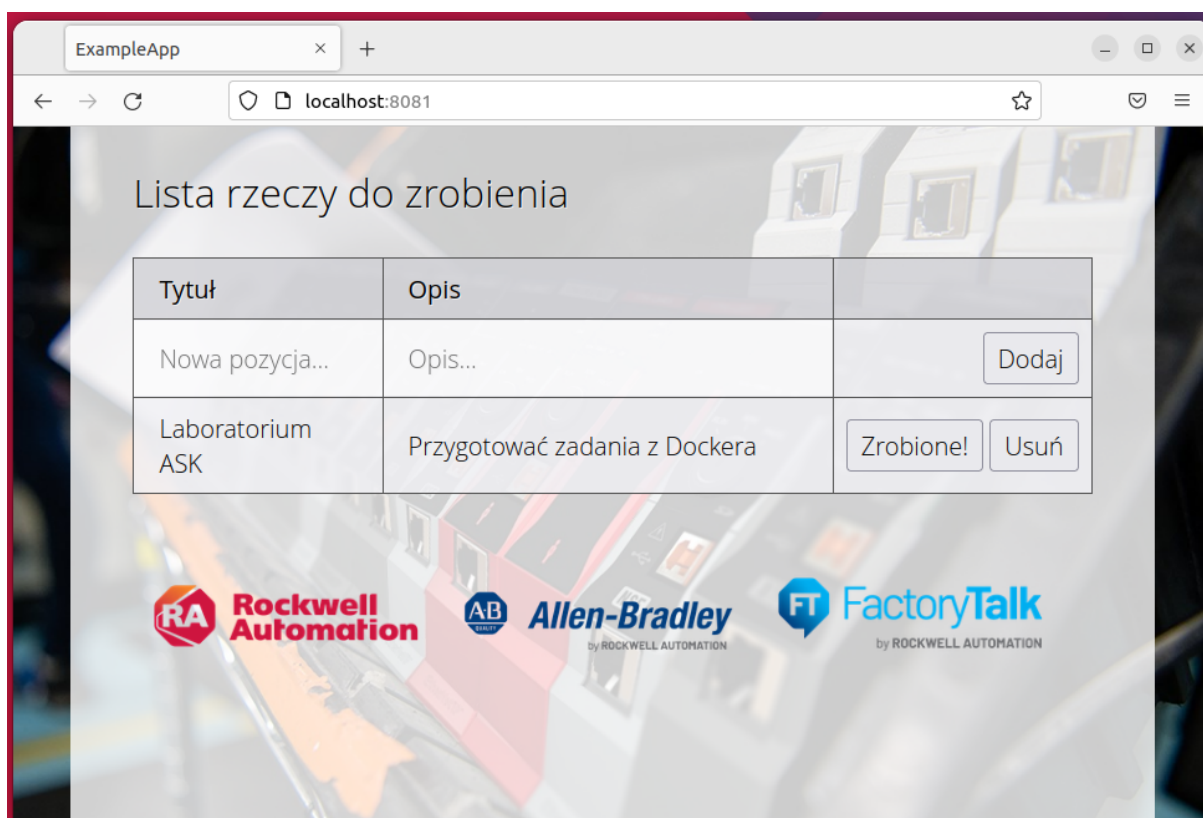
Aby zbudować obraz wydajemy polecenie:

```
$ docker build . --tag example_app:1
```

Spowoduje to zbudowanie obrazu i nadanie mu nazwy „example_app” z tagiem „1”. Aby uruchomić aplikację wydamy polecenie:

```
$ docker run --name app1 -itd --publish 8081:3000 example_app:1
```

W tym momencie aplikacja powinna już działać. Opcja `--publish` powoduje opublikowanie portu, czyli przekierowanie go na dowolnie wybrany port hosta. W tym przypadku port 3000 (domyślny dla Ruby on Rails) przekierowano na 8081. Dzięki temu jesteśmy w stanie otworzyć w przeglądarce adres <http://localhost:8081> bez sprawdzania adresu IP kontenera:



Rysunek 4: Aplikacja uruchomiona w dockerze i wyświetlona w przeglądarce

Zadanie: Uruchom przykładową aplikację i przetestuj jej działanie w przeglądarce internetowej.

Uruchomienie bazy danych – wolumeny

Instancja aplikacji, którą utworzyliśmy w poprzednim ćwiczeniu korzysta z plikowej bazy danych SQLite. W kolejnym kroku uruchomimy serwer popularnej bazy danych MariaDB (dawniej MySQL). Przy tej okazji zaznajomimy się z funkcją *wolumenów*.

Wolumeny są używane do przechowywania danych, które są często nadpisywane przez kontenery. W takim zastosowaniu wydajność sterownika `overlay` jest niewystarczająca. Dodatkowo wolumeny pozwalają na współdzielenie danych między wieloma kontenerami oraz na zachowanie danych po usunięciu kontenera. Ich typowym zastosowaniem jest m. in. przechowywanie plików baz danych.

Wolumen utworzymy poleceniem:

```
$ docker volume create database_volume
```

Następnie wystartujemy kontener bazy danych kierując się instrukcjami podanymi pod adresem <https://mariadb.com/kb/en/installing-and-using-mariadb-via-docker/>. Bazę skonfigurujemy tak, aby użytkownik root mógł się połączyć tylko lokalnie i wygenerujemy dla niego przypadkowe hasło. Zamontujemy również utworzony wcześniej wolumen w ścieżce `/var/lib/mysql`:

```
$ docker run --detach --name database \
  --volume database_volume:/var/lib/mysql \
  --env MARIADB_ROOT_HOST=localhost \
  --env MARIADB_RANDOM_ROOT_PASSWORD=yes \
  mariadb:10.8
```

Hasło zostanie wypisane przez serwer bazy danych na standardowe wyjście. Aby je poznać użyjemy komendy `container logs`, która wyświetla przechwycone wyjście z kontenera:

```
$ docker container logs database 2>&1 | grep 'ROOT PASSWORD'
```

Teraz musimy połączyć się z serwerem, aby utworzyć użytkownika i bazę danych dla naszej aplikacji. Ponieważ użytkownik root może logować się tylko lokalnie (przez UNIX domain socket), klienta musimy uruchomić wewnątrz kontenera. Posłużymy się do tego komendą `container exec`:

```
$ docker container exec -it database mariadb -p
```

Po przekopiowaniu wygenerowanego hasła powinniśmy uzyskać dostęp do bazy danych. Wykonujemy kolejno polecenia SQL:

1. Tworzymy nowego użytkownika:

```
MariaDB [(none)]> create user 'user1'@'%' identified by 'password1';
```

2. Tworzymy bazę danych: MariaDB [(none)]> create database app1;

3. Nadajemy użytkownikowi uprawnienia:

```
MariaDB [(none)]> grant all privileges on app1.* to 'user1'@'%';
```

4. Rozłączamy się z serwerem:

```
MariaDB [(none)]> exit
```

Sprawdźmy jaki adres IP został przypisany do kontenera, komendą `container inspect`:

```
$ docker container inspect database | grep IPAddress
```

Zadanie: Uruchom kontener bazy danych MariaDB opisaną powyżej metodą.

Przekazywanie parametrów do kontenera

Aby przekazać parametry przy uruchamianiu kontenera wykorzystuje się zmienne środowiskowe. Służy do tego opcja `--env <nazwa>=<wartosc>` komendy `container run`, która musi być powtórzona dla każdej zmiennej.

Zadanie: Wyświetl zmienne środowiskowe ustawione w kontenerze bazy danych za pomocą programu `env`. Zwróć uwagę na wartości zmiennych przekazanych przy tworzeniu kontenera.

Wreszcie możemy uruchomić przykładową aplikację, podając jej jako zmienną `DATABASE_URL` parametry połączenia z serwerem bazy danych:

```
$ docker run --name app2 -d -p 8082:3000 \
  --env DATABASE_URL=mysql2://user1:password1@<ip bazy danych>/app1 \
  example_app:1
```

Zadanie: Uruchom dwie instancje przykładowej aplikacji na różnych portach, ale korzystając z tej samej bazy danych. Sprawdź ich działanie.

Docker Compose

Docker compose to dodatkowy „plugin”, który pozwala w dużej mierze zautomatyzować uruchamianie aplikacji wymagających wielu kontenerów. Zainstalujemy go poleceniem:

```
$ sudo apt install docker-compose
```

Przed rozpoczęciem pracy wyłączymy wszystkie obecnie działające kontenery komendą `container stop <nazwa kontenera>`. Następnie utworzymy w katalogu `~/lab_docker/` plik o nazwie `docker-compose.yml` i o następującej treści:

```
version: '3.8'
services:
  database:
    image: mariadb:10.8
    volumes:
      - database_volume:/var/lib/mysql
```

```
app1:
  image: example_app:1
  depends_on:
    - database
  environment:
    DATABASE_URL: 'mysql2://user1:password1@database/app1'
  ports:
    - '80:3000'
volumes:
  database_volume:
    external: true
    name: database_volume
```

W pliku `docker-compose.yml` znajduje się opis kontenerów do utworzenia wraz z wszystkimi parametrami, które wcześniej przekazywaliśmy ręcznie do komendy `run`. Dodatkowo nie musimy posługiwać się już adresami IP, a zamiast tego używamy nazw kontenerów.

Aby uruchomić wszystkie kontenery wystarczy wydać w katalogu z plikiem polecenie:

```
$ docker-compose up
```

Nazwy utworzonych kontenerów otrzymają przedrostek `lab_docker`, który jest wybierany na podstawie nazwy aktualnego katalogu. Wciśnięcie kombinacji `Ctrl+C` spowoduje zatrzymanie wszystkich kontenerów.

Zadanie: Uruchom przykładową aplikację korzystając z polecenia `docker-compose`, jak opisano powyżej. Sprawdź listę uruchomionych i zatrzymanych kontenerów. Sprawdź działanie aplikacji w przeglądarce.

Część nieobowiązkowa: czy w kontenerze można uruchomić aplikację graficzną?

Innymi słowy, czy w dockerze można zagrać w DOOMa? Nie jest to typowe zastosowanie, ale nie ma powodu, aby nie dało się tego osiągnąć.

Inaczej niż w przypadku maszyny wirtualnej, kontener nie posiada emulowanej karty graficznej. Na szczęście protokół X11 używany do renderowania aplikacji graficznych* jest protokołem sieciowym. Pozwala to na uruchomienie programu na innym komputerze, niż terminal przy którym znajduje się użytkownik. Tę funkcję można wykorzystać aby uruchomić program poprzez połączenie SSH.

Na początek wystartujmy Ubuntu w kontenerze:

```
$ docker run -it --name doom ubuntu
```

Następnie wewnątrz kontenera zainstalujemy serwer SSH i silnik gry:

```
# apt update
# apt install openssh-server prboom-plus
```

Utwórzmy katalog dla kluczy SSH:

```
# mkdir ~/.ssh
```

Wynenerujemy klucz i skopiujmy go do kontenera używając komendy `copy`:

```
$ ssh-keygen
$ docker container cp ~/.ssh/id_rsa.pub doom:/root/.ssh/authorized_keys
```

Następnie trzeba nieco poprawić uprawnienia pliku `.ssh/authorized_keys`:

```
# chown root ~/.ssh/authorized_keys
# chgrp root ~/.ssh/authorized_keys
```

Lekkiej poprawy wymaga też domyślna konfiguracja serwera:

```
# echo "X11UseLocalhost no" >> /etc/ssh/sshd_config
```

Uruchamiamy serwer:

```
# service ssh start
```

Ostatecznie możemy skopiować plik WAD (teraz już za pomocą `scp`) i uruchomić grę:

```
$ scp ~/lab_docker/game root@<ip kontenera>:DOOM1.WAD
$ ssh -Y root@<ip kontenera> prboom-plus -iwad DOOM1.WAD
```

Zadanie dodatkowe: KILLS: 100% ITEMS: 100% SECRET: 100%



Rysunek 5: Ilustracja poglądowa