Bezpieczeństwo infrastruktury krytycznej i systemów sterowania przemysłowego IoT – Laboratorium 3 – Konteneryzacja (Docker), Terraform (Teoria)

Forma: praca w samodzielna / małe zespoły (2 - 4 osoby)

Wstęp teoretyczny

Czym jest Terraform?

Terraform to potężne narzędzie typu Infrastructure as Code (IaC) stworzone przez firmę HashiCorp. Umożliwia ono wdrażanie oraz zarządzanie infrastrukturą zarówno lokalnie, jak i w chmurze. Dzięki architekturze opartej na wtyczkach można je w łatwy sposób rozszerzać i dostosowywać do różnych środowisk.

Do czego służy Terraform?

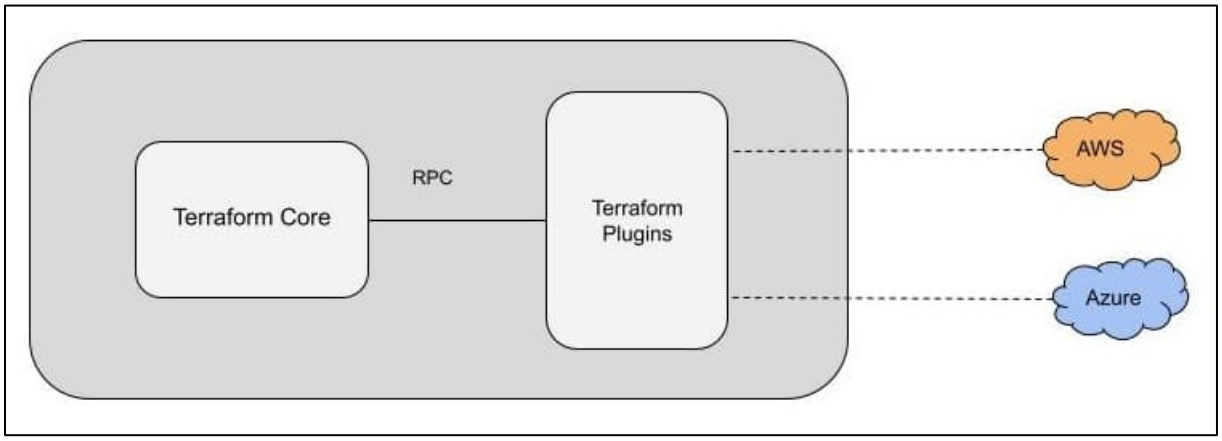
Terraform służy do łączenia się z różnymi dostawcami infrastruktury i realizowania złożonych scenariuszy zarządzania z zachowaniem zgodności pomiędzy wieloma chmurami. Jego konfiguracje można łatwo pakować, udostępniać i ponownie wykorzystywać w postaci tzw. modułów Terraform.

Jak działa Terraform?

W uproszczeniu Terraform składa się z dwóch głównych części: Terraform Core oraz wtyczek (Pluginy). Core odpowiada za zarządzanie cyklem życia infrastruktury i stanowi otwarte oprogramowanie, które uruchamia się z poziomu wiersza poleceń.

Terraform Core porównuje aktualny stan infrastruktury z pożądanym stanem zdefiniowanym w konfiguracji, po czym proponuje plan działań – dodania, modyfikacji lub usunięcia zasobów. Następnie, jeśli użytkownik zaakceptuje plan, Terraform wdraża zmiany automatycznie.

Wtyczki Terraform umożliwiają komunikację pomiędzy Terraform Core a dostawcami infrastruktury lub usług SaaS. Przykładami takich wtyczek są dostawcy (Providers) i instalatory (Provisioners). Komunikacja odbywa się za pomocą mechanizmu zdalnych wywołań procedur (RPC).



Przebieg pracy z Terraform

Typowy proces pracy z Terraform składa się z trzech etapów:

1. **Tworzenie (Write)** – definiowanie infrastruktury w języku HCL (Hashicorp Configuration Language).
2. **Przeglądanie planu (Plan)** – Terraform analizuje różnice pomiędzy stanem bieżącym a pożądanym i wyświetla plan zmian.
3. **Zastosowanie (Apply)** – użytkownik akceptuje plan, a Terraform wdraża zmiany w infrastrukturze.

W przypadku jednego autora stan Terraform przechowywany jest lokalnie. W pracy zespołowej konieczne jest przechowywanie stanu w zdalnej lokalizacji, aby każdy miał spójny widok aktualnej infrastruktury. W tym celu HashiCorp oferuje wersje Terraform Cloud oraz Terraform Enterprise.

HashiCorp Cloud Platform (HCP) Terraform to usługa w chmurze, natomiast **Terraform Enterprise** umożliwia samodzielne hostowanie tej platformy w prywatnym środowisku. Kod infrastruktury przechowywany jest w repozytorium wersji (np. GitHub), z którego Terraform pobiera aktualne wersje.

Podstawowe elementy Terraform
Terraform CLI to narzędzie wiersza poleceń, dostępne jako pojedynczy plik wykonywalny. Umożliwia inicjalizację środowiska, walidację konfiguracji, planowanie i wdrażanie infrastruktury, a także jej usuwanie.

https://github.com/hashicorp/terraform

```
~  terraform --help
Usage: terraform [global options] <subcommand> [args]

The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.

Main commands:
  init          Prepare your working directory for other commands
  validate      Check whether the configuration is valid
  plan          Show changes required by the current configuration
  apply         Create or update infrastructure
  destroy       Destroy previously-created infrastructure

All other commands:
  console       Try Terraform expressions at an interactive command prompt
  fmt           Reformat your configuration in the standard style
  force-unlock  Release a stuck lock on the current workspace
  get           Install or upgrade remote Terraform modules
  graph         Generate a Graphviz graph of the steps in an operation
  import        Associate existing infrastructure with a Terraform resource
  login         Obtain and save credentials for a remote host
  logout        Remove locally-stored credentials for a remote host
  output        Show output values from your root module
  providers     Show the providers required for this configuration
  refresh       Update the state to match remote systems
  show          Show the current state or a saved plan
  state         Advanced state management
  taint         Mark a resource instance as not fully functional
  test          Experimental support for module integration testing
  untaint       Remove the 'tainted' state from a resource instance
  version       Show the current Terraform version
  workspace     Workspace management

Global options (use these before the subcommand, if any):
  -chdir=DIR    Switch to a different working directory before executing the
                given subcommand.
  -help         Show this help output, or the help for a specified subcommand.
  -version      An alias for the "version" subcommand.
```

Język Terraform
Infrastruktura w Terraform opisywana jest w języku HCL - deklaratywnym języku stworzonym specjalnie do zarządzania infrastrukturą. Zasoby (Resources) to podstawowe elementy konfiguracji - mogą to być maszyny wirtualne, bazy danych, kontenery, sieci VPC czy zasobniki S3.

Przykład definicji zasobu VPC w AWS:

```
resource "aws_vpc" "default_vpc" {
  cidr_block = "172.31.0.0/16"
  tags = {
    Name = "example_vpc"
  }
}
```

Dostawcy Terraform (Providers)
Aby Terraform mógł zarządzać infrastrukturą, musi łączyć się z jej dostawcą. Dostawcy publikują interfejsy API, które umożliwiają operacje tworzenia, modyfikowania i usuwania zasobów. Terraform korzysta z tych interfejsów za pośrednictwem dostawców, którzy są udostępniani jako wtyczki w rejestrze Terraform.

Przykład konfiguracji dostawcy AWS:

```
provider "aws" {
  version = "~> 3.0"
  region  = "us-east-1"
}
```

Provisioners

Provisioners to wtyczki służące do konfigurowania infrastruktury po jej wdrożeniu, np. kopiowania plików lub uruchamiania dodatkowych skryptów konfiguracyjnych (Chef, Puppet). HashiCorp zaleca jednak ich użycie tylko w ostateczności, ponieważ zwiększają one złożoność i trudność w przewidywaniu zachowania środowiska.

https://spacelift.io/blog/terraform-provisioners

Moduły Terraform
Moduły pozwalają grupować powiązane zasoby w jedną całość, co ułatwia ponowne wykorzystanie konfiguracji. Każda konfiguracja Terraform posiada co najmniej jeden moduł główny (root module), który może wywoływać inne moduły podrzędne. Gotowe moduły dostępne są w publicznym rejestrze Terraform.

https://registry.terraform.io/browse/modules

Stan Terraform
Terraform zapisuje informacje o stanie infrastruktury w pliku terraform.tfstate. Zawiera on dane o zasobach, ich relacjach i aktualnych wartościach. Dzięki temu Terraform może planować kolejne zmiany w sposób przewidywalny. Stan może być przechowywany lokalnie lub zdalnie (np. w chmurze).

https://spacelift.io/blog/terraform-architecture

Przykładowe zastosowania Terraform

1. **Szybkie tworzenie i usuwanie środowisk** – Terraform umożliwia definiowanie całych środowisk (np. testowego, produkcyjnego) jako kodu, co pozwala szybko je tworzyć, kopiować i usuwać.
2. **Praca z wieloma chmurami** – Terraform jest niezależny od dostawcy chmury. Umożliwia jednoczesne wdrażanie aplikacji w AWS, Azure i GCP.
3. **Wdrażanie aplikacji wielowarstwowych (N-Tier)** – Terraform dba o kolejność wdrożenia zależnych komponentów, np. najpierw baza danych, potem serwery aplikacji.
4. **Integracja z platformami PaaS** – Terraform może współpracować z Kubernetes czy innymi narzędziami opartymi o API, zarządzając całą infrastrukturą i konfiguracją z jednego miejsca.

Terraform a inne narzędzia
OpenTofu to otwarty fork Terraform, powstały po zmianie licencji HashiCorp. Jest w pełni otwartym projektem wspieranym przez społeczność i organizacje z branży. Jego zaletami są brak ograniczeń licencyjnych, dynamiczna społeczność, szybkie wsparcie oraz gwarancja pozostania projektem open-source.

Terraform vs Chef/Puppet - Terraform służy do tworzenia infrastruktury, natomiast Chef i Puppet do jej konfiguracji po wdrożeniu. Oba typy narzędzi mogą działać równolegle.

Terraform vs CloudFormation/ARM Templates – narzędzia te są dedykowane konkretnym chmurom (AWS, Azure), podczas gdy Terraform jest niezależny i wspiera wiele środowisk.

Terraform vs Kubernetes - Terraform może zarządzać samą infrastrukturą Kubernetes, a także tworzyć klastry poprzez dedykowany Provider.

Terraform vs skrypty własne - rozwiązania oparte na niestandardowych skryptach często trudno skalować i utrzymywać. Terraform wprowadza standaryzację, automatyzację i lepszą kontrolę zmian.

Konteneryzacja

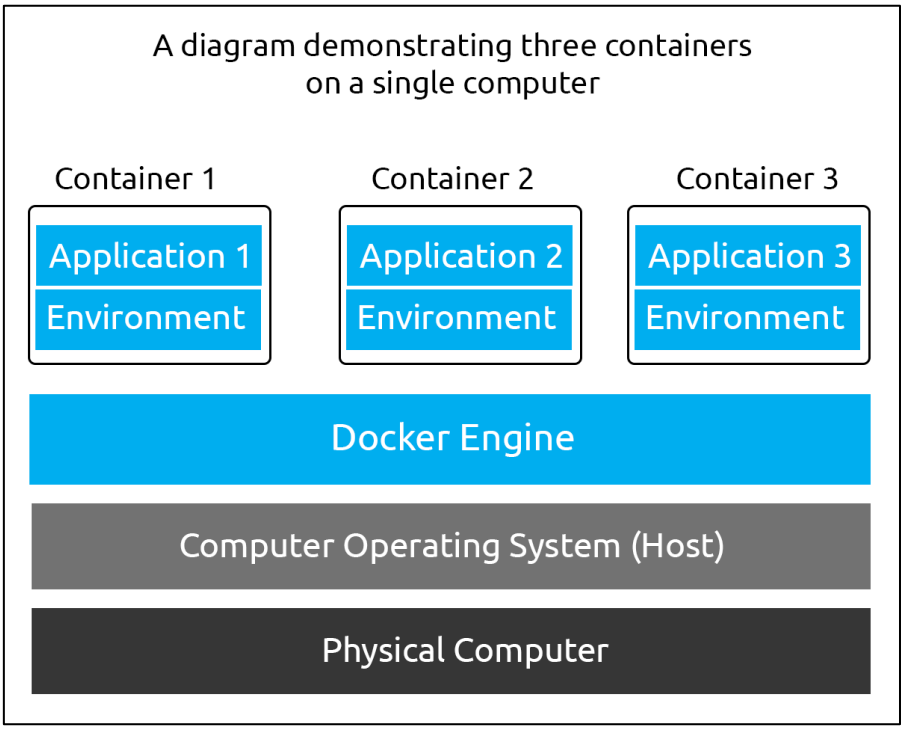
W informatyce konteneryzacja to proces pakowania aplikacji wraz ze wszystkimi niezbędnymi zasobami (takimi jak biblioteki i pakiety) w jeden pakiet zwany kontenerem. Takie połączenie aplikacji i jej zależności w całość sprawia, że programy stają się znacznie bardziej przenośne i łatwiejsze do uruchomienia.

Współczesne aplikacje są często złożone i zwykle wymagają, aby przed ich uruchomieniem na urządzeniu zainstalowane były odpowiednie frameworki i biblioteki. Zależności te mogą:
- być trudne do zainstalowania w zależności od środowiska, w którym działa aplikacja (niektóre systemy operacyjne mogą ich w ogóle nie obsługiwać!),
- utrudniać programistom diagnozowanie i odtwarzanie błędów, ponieważ problem może dotyczyć środowiska aplikacji, a nie samej aplikacji,
- wchodzić ze sobą w konflikt. Na przykład posiadanie kilku wersji Pythona do uruchamiania różnych aplikacji bywa uciążliwe, a jedna aplikacja może działać poprawnie tylko z określoną wersją Pythona, a z inną już nie.

Platformy konteneryzacyjne eliminują te problemy, pakując wszystkie zależności razem z aplikacją i „izolując" jej środowisko (należy jednak podkreślić, że nie chodzi tu o pełną izolację bezpieczeństwa w sensie systemowym).

Jeśli urządzenie obsługuje silnik konteneryzacji, użytkownik może uruchomić aplikację i będzie ona zachowywać się w ten sam sposób niezależnie od miejsca uruchomienia.

Na diagramie przedstawiono przykład trzech kontenerów działających na jednym komputerze.



W tym przykładzie widać, że trzy aplikacje wraz ze swoimi środowiskami (czyli zależnościami) są zapakowane oddzielnie i nie komunikują się bezpośrednio ze sprzętem fizycznym, lecz z silnikiem konteneryzacji – w tym przypadku jest to Docker. Warto również wiedzieć, że platformy konteneryzacyjne wykorzystują funkcję jądra systemu operacyjnego zwaną „namespace" (przestrzenią nazw). Dzięki niej procesy mogą korzystać z zasobów systemu, ale bez możliwości bezpośredniego oddziaływania na inne procesy.

Izolacja zapewniana przez przestrzenie nazw daje dodatkową warstwę bezpieczeństwa — jeśli aplikacja działająca w kontenerze zostanie przejęta, inne kontenery zazwyczaj pozostają nienaruszone (o ile nie współdzielą tej samej przestrzeni nazw).

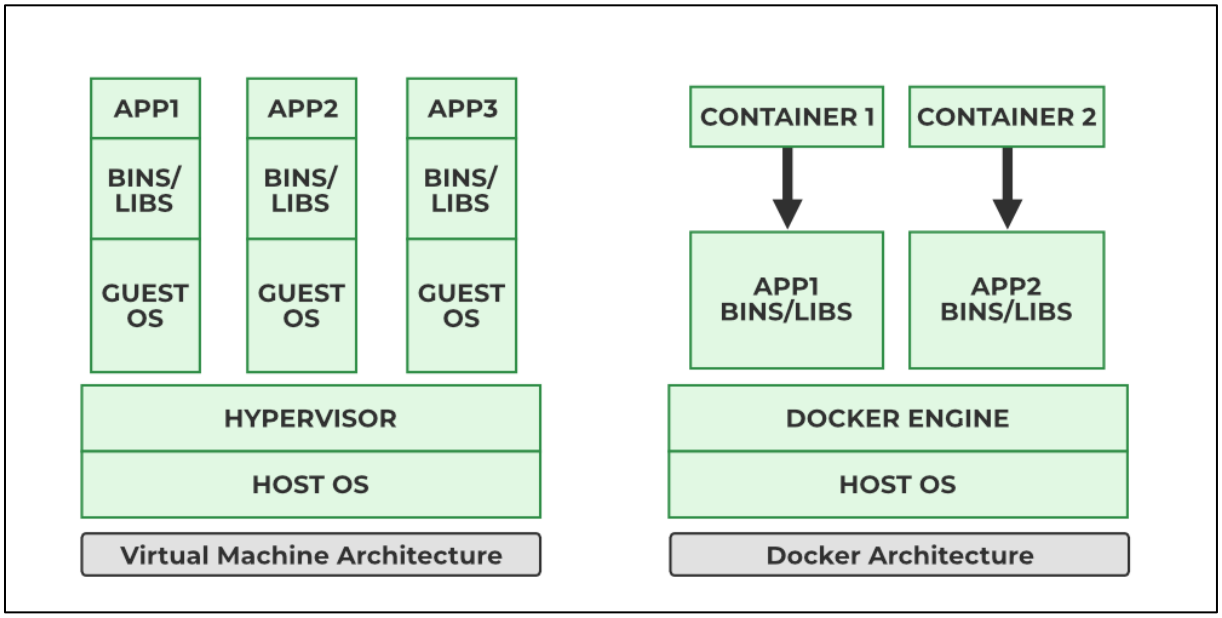
Alternatywą dla kontenerów są maszyny wirtualne, które wymagają instalacji całego systemu operacyjnego, aby uruchomić aplikację, co wiąże się z dużym zużyciem miejsca na dysku oraz zasobów takich jak procesor i pamięć RAM.

Dodatkowo można przeczytać: https://www.geeksforgeeks.org/devops/what-is-containerisation/

Docker

Docker to platforma wirtualizacji na poziomie systemu operacyjnego (konteneryzacji), która umożliwia aplikacjom współdzielenie jądra systemu operacyjnego hosta zamiast uruchamiania osobnego systemu gościa jak w tradycyjnej wirtualizacji, takie podejście sprawia, że kontenery Docker są lekkie, szybkie i przenośne, a jednocześnie pozostają od siebie odizolowane. Docker jest napisany w języku Go. Wspiera instalacje na Windows, macOS i Linux (Docker Engine działa natywnie w systemie Linux). Rozwiązuje problem „works on my machine", zapewniając identyczne działanie kodu w różnych środowiskach. W przeciwieństwie do VMware (wirtualizacja na poziomie sprzętu) Docker działa na poziomie systemu operacyjnego.

Przed Dockerem wdrażanie aplikacji w różnych środowiskach bywało trudne, ponieważ zależności, konfiguracje i różnice między systemami operacyjnymi powodowały kłopotliwe sytuacje „tu działa, tam nie".



Wykorzystanie dockera polega na ustandaryzowaniu środowiska uruchomieniowego poprzez spakowanie wszystkiego (aplikacji i zależności) w kontenery. Zapewnia to przenośność (uruchamianie na komputerze lokalnym, w chmurze i na serwerach on-prem), spójność (to samo zachowanie w developmentcie, testach i produkcji), lekkość (brak pełnego systemu operacyjnego na aplikację, kontenery współdzielą jądro hosta), skalowalność (idealne dla mikroserwisów oraz orkiestratorów takich jak Kubernetes i Docker Swarm), efektywność (start w sekundy, mniejsze zużycie zasobów).

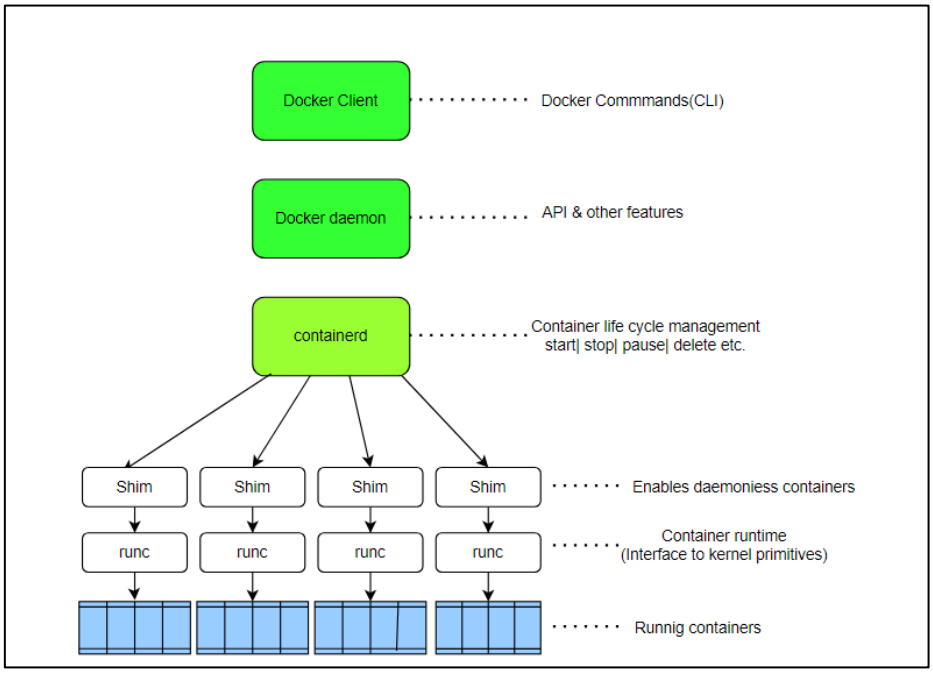
Składniki Dockera obejmują między innymi:

- Docker Engine, którego podstawowym elementem jest docker daemon odpowiedzialny za tworzenie i zarządzanie kontenerami
- Obraz Dockera (Docker Image), czyli szablon tylko do odczytu używany do tworzenia kontenerów, zawierający kod aplikacji i zależności.
- Docker Hub, chmurowe repozytorium służące do wyszukiwania i udostępniania obrazów kontenerów
- Dockerfile, plik opisujący kroki szybkiego zbudowania obrazu
- Docker Registry, system przechowywania i dystrybucji obrazów, w którym można trzymać obrazy w trybie publicznym lub prywatnym.

Docker Engine

To podstawowy komponent umożliwiający działanie kontenerów w systemie, korzysta z architektury klient-serwer i odpowiada za budowanie, uruchamianie oraz zarządzanie kontenerami Docker.

- Demon Docker (dockerd) działa w tle, nasłuchuje żądań API i zarządza obiektami takimi jak obrazy, kontenery, sieci i wolumeny.
- Klient Docker (docker CLI) komunikuje się z demonem przez REST API i dostarcza środowisko wykonawcze, w którym obrazy są uruchamiane jako żywe kontenery.



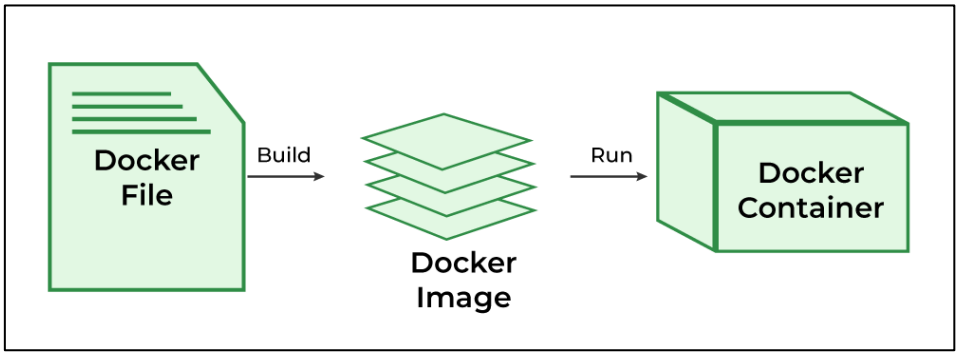
Bez Docker Engine nie można budować obrazów ani uruchamiać kontenerów.

- Klient wysyła polecenia Dockera (docker build, docker run itd.),
- demon je odbiera i wykonuje operacje na kontenerach,
- REST API stanowi interfejs umożliwiający tę komunikację.

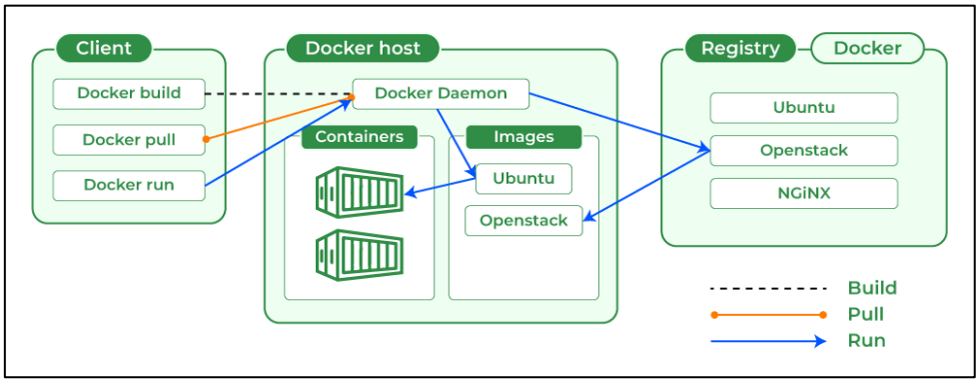
W skrócie: Docker Engine to środowisko uruchomieniowe, które umożliwia konteneryzację, łącząc klienta Docker z demonem w celu efektywnego budowania i zarządzania kontenerami.

Dockerfile

Wykorzystuje język DSL (Domain Specific Language) i zawiera instrukcje generowania obrazu Dockera, definiuje kroki potrzebne do szybkiego wytworzenia obrazu, a podczas tworzenia aplikacji należy przygotować Dockerfile w odpowiedniej kolejności, ponieważ demon Dockera wykonuje instrukcje od góry do dołu, Dockerfile jest „kodem źródłowym" obrazu. Jest to dokument tekstowy zawierający niezbędne polecenia, których wykonanie pomaga złożyć obraz Dockera, obraz Dockera powstaje z Dockerfile. „Dockerfile") i



Architektura i działanie Dockera opierają się na modelu klient-serwer: klient Docker rozmawia z demonem Docker, który pomaga budować, uruchamiać i dystrybuować kontenery; klient może działać na tym samym systemie co demon lub łączyć się z nim zdalnie; klient i demon komunikują się przez REST API za pośrednictwem gniazda UNIX lub sieci.



Docker CLI

To interfejs wiersza poleceń do pracy z Dockerem, typowe komendy to docker run, docker build, docker pull.

Docker REST API

To interfejs HTTP używany przez CLI i inne narzędzia, który umożliwia komunikację z demonem Dockera.

Docker Daemon

Obsługuje obrazy, kontenery, sieci i wolumeny, stanowi główną usługę zarządzającą obiektami Dockera.

High-Level Runtime

Zarządza operacjami cyklu życia kontenerów, takimi jak tworzenie, uruchamianie, zatrzymywanie i usuwanie kontenerów.

Obraz Dockera

To plik złożony z wielu warstw zawierający instrukcje budowy i uruchomienia kontenera, działa jak wykonywalny pakiet obejmujący wszystko, czego potrzeba do uruchomienia aplikacji kod, środowisko uruchomieniowe, biblioteki, zmienne środowiskowe i konfiguracje.

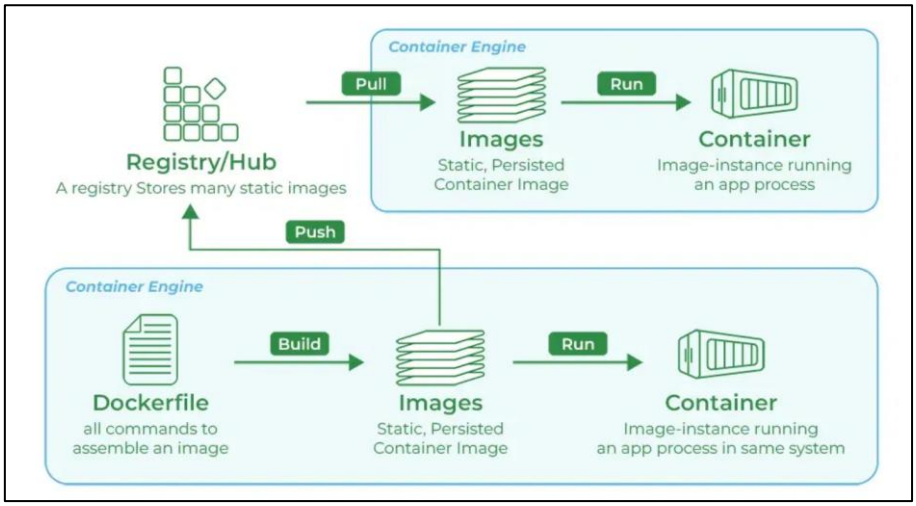
Działanie wygląda tak:

- obraz definiuje sposób tworzenia kontenera i określa, które komponenty oprogramowania będą uruchamiane oraz jak są konfigurowane
- po uruchomieniu obrazu powstaje kontener.

Relacja do kontenerów:

Obraz Dockera to plan (statyczny, tylko do odczytu), a kontener Dockera to działająca instancja tego planu (dynamiczna, wykonywalna). Kontener Dockera to lekka, uruchamialna instancja obrazu Dockera, pakuje kod aplikacji wraz ze wszystkimi zależnościami i działa w odizolowanym środowisku, kontenery pozwalają uruchamiać aplikacje szybko i spójnie w różnych środowiskach, czy to na laptopie dewelopera, serwerach testowych, czy w produkcji. Kontener powstaje w chwili uruchomienia obrazu, działa jako odizolowany proces na maszynie hosta, ale współdzieli jądro systemu operacyjnego hosta, a wiele kontenerów może działać na tym samym systemie bez wzajemnego zakłócania się. Przykład: jeśli istnieje obraz systemu Ubuntu z serwerem NGINX, to po uruchomieniu tego obrazu poleceniem docker run zostanie utworzony kontener, a serwer NGINX będzie działał na Ubuntu. Relacja do obrazów jest taka sama jak wcześniej: obraz to plan (statyczny, tylko do odczytu), kontener to żywa instancja tego planu (dynamiczna, wykonywalna).

Docker Hub to usługa repozytorium i chmurowa usługa, w której użytkownicy publikują obrazy kontenerów Docker i skąd mogą je pobierać w dowolnym momencie i miejscu przez internet.



Ułatwia znajdowanie i ponowne użycie obrazów, oferuje możliwość publikowania obrazów w rejestrze publicznym lub prywatnym, gdzie można je przechowywać i współdzielić. Głównie korzystają z niego zespoły DevOps. Jest to narzędzie otwartoźródłowe i dostępne bezpłatnie dla wszystkich systemów operacyjnych, działa jak magazyn, w którym przechowujemy obrazy i z którego je pobieramy, gdy są potrzebne, aby wysyłać/pobierać obrazy z Docker Hub, trzeba mieć podstawową znajomość Dockera

Polecenia Dockera: wprowadzenie podstawowych komend uczyniło Dockera potężnym oprogramowaniem usprawniającym zarządzanie kontenerami, zapewniającym płynne przepływy pracy w tworzeniu i wdrażaniu; najczęściej używane to:

- Docker Run - służy do uruchamiania kontenerów z obrazów z podaniem opcji wykonania i poleceń
- Docker Pull - pobiera obrazy kontenerów z rejestru, takiego jak Docker Hub, na maszynę lokalną
- Docker ps - wyświetla uruchomione kontenery wraz z ważnymi informacjami, takimi jak identyfikator kontenera, użyty obraz i status
- Docker Stop - zatrzymuje działające kontenery, z eleganckim zamknięciem procesów wewnątrz –
- Docker Start - ponownie uruchamia zatrzymane kontenery, wznawiając ich działanie ze stanu poprzedniego Docker Login - loguje do rejestru Dockera, umożliwiając dostęp do repozytoriów prywatnych.

Docker Engine to oprogramowanie, które hostuje kontenery; jest aplikacją w architekturze klient-serwer i składa się z trzech głównych komponentów:

- Serwer - odpowiedzialny za tworzenie i zarządzanie obrazami, kontenerami, sieciami i wolumenami Dockera, nazywany procesem demona
- REST API - określa, jak aplikacje mogą wchodzić w interakcje z serwerem i instruować go, co robić
- Klient - interfejs wiersza poleceń Dockera (CLI), który pozwala nam pracować z Dockerem za pomocą poleceń docker.

Docker występuje w dwóch edycjach:

- Community Edition (CE) - bezpłatna, otwartoźródłowa, używana przez osoby indywidualne, zespoły deweloperskie i współtwórców open source
- Enterprise Edition (EE) - płatna, z rozszerzeniami bezpieczeństwa, certyfikowanymi wtyczkami/obrazami i wsparciem klasy enterprise.

Materiały dodatkowe:

- https://www.docker.com/play-with-docker/
- https://www.docker.com/trust/security/
- https://www.docker.com/101-tutorial/
- https://docs.docker.com/build/building/base-images/
- https://www.tutorialspoint.com/docker/index.htm
- https://www.youtube.com/watch?v=3c-iBn73dDE&t=7026s
- https://www.youtube.com/watch?v=pg19Z8LL06w
- https://developer.hashicorp.com/terraform/tutorials/docker-get-started
- https://spacelift.io/blog/terraform-tutorial
- https://www.youtube.com/watch?v=SLB_c_ayRMo&t=1251s
- https://www.youtube.com/watch?v=7xngnjfIlK4&t=56s
- https://www.youtube.com/watch?v=l5k1ai_GBDE
- https://www.youtube.com/watch?v=MnUtHSpcdLQ
- https://www.youtube.com/watch?v=FnHfmDeoQo8
- https://www.youtube.com/watch?v=0oTuH_xY3mw&t=506s
- https://www.youtube.com/watch?v=dTqxNc1MVLE
- https://docker.courselabs.co/

Przykłady

Ubuntu/Debian

Docker commands: https://www.geeksforgeeks.org/devops/docker-instruction-commands/

Zainstaluj zależności i repo:

```
sudo apt-get update
sudo apt-get install -y docker
```

Test:

Wszystkie polecenia docker należy wykonywać jako użytkownik uprzywilejowany więc, używamy sudo / użytkownik root

```
docker version
sudo docker run --rm hello-world
```

Pobierz i uruchom NGINX

```
sudo docker pull nginx:stable
sudo docker run -d --name moj-nginx -p 8080:80 nginx:stable
```

Wejdź w przeglądarce: http://localhost:8080 powinna być strona Welcome to nginx!.

Podgląd i logi:

```
sudo docker ps
sudo docker logs -f moj-nginx
```

Zatrzymanie i usunięcie:

```
sudo docker stop moj-nginx
sudo docker rm moj-nginx
```

Uruchom prosty kontener interaktywny (Ubuntu + bash)

```
docker run -it --rm ubuntu:22.04 bash
# w środku:
apt update && apt install -y curl
exit
```

Pierwszy Dockerfile

Utwórz folder projektu:

```
mkdir -p moj-app && cd moj-app
```

Stwórz plik app.py:

```
# app.py
from http.server import BaseHTTPRequestHandler, HTTPServer
class H(BaseHTTPRequestHandler):
  def do_GET(self):
    self.send_response(200); self.end_headers()
    self.wfile.write(b"Hello from Docker!")
HTTPServer(("", 8000), H).serve_forever()
```

Stwórz plik Dockerfile:

Nano/vim/vi dockerfile

```
FROM python:3.12-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY app.py .
EXPOSE 8000
CMD ["python", "app.py"]
```

Utwórz plik requierements.txt:

```
requests
```

Zbuduj obraz i uruchom:

```
sudo docker build -t hello-docker:1.0 .
sudo docker run -d --name hello -p 8000:8000 hello-docker:1.0
Wejdź w przeglądarce: http://localhost:8000 → „Hello from Docker!”.
```

Aktualizacja obrazu po zmianie kodu:

```
sudo docker stop hello && docker rm hello
sudo docker build -t hello-docker:1.1 .
sudo docker run -d --name hello -p 8000:8000 hello-docker:1.1
```

Wolumeny - NGINX serwujący lokalny folder

Stwórz katalog z plikiem:

```
mkdir -p www && echo "Moja strona" > www/index.html
```

Uruchom z wolumenem:

```
sudo docker run -d --name web -p 8080:80 -v $(pwd)/www:/usr/share/nginx/html:ro nginx:stable
```

Odśwież http://localhost:8080 → zobaczysz Moja strona.

Minimalny serwer HTTP w Pythonie (pojedynczy plik)

Cel: zbudować obraz z jednym plikiem i sprawdzić logi/exec.

Struktura:

```
mini-http/
 ├─ Dockerfile
 └─ app.py
```

app.py

```
from http.server import BaseHTTPRequestHandler, HTTPServer
class H(BaseHTTPRequestHandler):
   def do_GET(self):
      self.send_response(200); self.end_headers()
      self.wfile.write(b"Hello from minimal Python HTTP!")
HTTPServer(("", 8000), H).serve_forever()
```

Dockerfile

```
FROM python:3.12-slim
WORKDIR /app
COPY app.py .
EXPOSE 8000
CMD ["python", "app.py"]
```

Budowa i uruchomienie:

```
cd mini-http
sudo docker build -t mini-http:1.0 .
sudo docker run -d --name mini -p 8000:8000 mini-http:1.0
```

Interakcja:

```
curl http://localhost:8000        # sprawdzenie odpowiedzi
sudo docker logs -f mini            # podgląd logów
sudo docker exec -it mini /bin/sh       # wejście do kontenera
exit                        # wyjście z powłoki
sudo docker stop mini && docker rm mini
```

Node.js (Express) + .dockerignore (mniejsze obrazy)

Struktura:

```
node-api/
 ├─ Dockerfile
 ├─ .dockerignore
 ├─ package.json
 └─ index.js
```

package.json

```
{
  "name": "node-api",
  "version": "1.0.0",
  "main": "index.js",
  "type": "module",
  "scripts": { "start": "node index.js" },
  "dependencies": { "express": "^4.19.2" }
}
```

index.js

```
import express from "express";
const app = express();
app.get("/", (req, res) => res.send("Hello from Node API!"));
app.listen(3000, () => console.log("API on :3000"));
```

.dockerignore

```
node_modules
.git
.DS_Store
*.log
```

Dockerfile

```
FROM node:20-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

Budowa/uruchomienie:

```
cd node-api
sudo docker build -t node-api:1.0 .
sudo docker run -d --name api -p 3000:3000 node-api:1.0
```

Interakcja (hot-inspect):

```
sudo docker logs -f api
sudo docker exec -it api sh
cat /etc/os-release
exit
```

Healthcheck + zmienne środowiskowe + ENTRYPOINT vs CMD

Struktura:

```
health-app/
 ├─ Dockerfile
 └─ app.py
```

app.py

```
import os
from http.server import BaseHTTPRequestHandler, HTTPServer
class H(BaseHTTPRequestHandler):
  def do_GET(self):
    msg = os.getenv("MESSAGE", "Hello")
    if self.path == "/healthz":
      self.send_response(200); self.end_headers(); self.wfile.write(b"ok"); return
    self.send_response(200); self.end_headers(); self.wfile.write(msg.encode())
HTTPServer(("", int(os.getenv("PORT","8080"))), H).serve_forever()
```

Dockerfile

```
FROM python:3.12-slim
WORKDIR /app
COPY app.py .
ENV PORT=8080 MESSAGE="Hello from HEALTH!"
EXPOSE 8080
HEALTHCHECK --interval=10s --timeout=2s --retries=3 CMD python -c "import urllib.request;
urllib.request.urlopen('http://localhost:8080/healthz')"
ENTRYPOINT ["python", "app.py"]
# Gdy dodasz CMD, nadpisujesz domyślne parametry dla ENTRYPOINT
# CMD ["--some", "default", "args"]
```

Uruchomienie i sprawdzenie stanu:

```
sudo docker build -t health-app:1.0 .
sudo docker run -d --name h1 -p 8082:8080 -e MESSAGE="Custom!" health-app:1.0
sudo docker ps --format "table {{.Names}}\t{{.Status}}\t{{.Ports}}"
curl http://localhost:8082/
```

NGINX z własnym configiem (kopiowanie plików + wolumen named)

Cel: własna konfiguracja + trwały wolumen na logi.

Struktura:

```
nginx-custom/
 ├─ Dockerfile
 └─ nginx.conf
```

nginx.conf

```
events {}
http {
  server {
    listen 80;
    location / { return 200 "Hello from custom NGINX!\n"; }
  }
}
```

Dockerfile

```
FROM nginx:stable-alpine
COPY nginx.conf /etc/nginx/nginx.conf
EXPOSE 80
```

Budowa/uruchomienie z wolumenem:

```
docker build -t nginx-custom:1.0 .
docker volume create nginx_logs
docker run -d --name nxc -p 8083:80 -v nginx_logs:/var/log/nginx nginx-custom:1.0
curl http://localhost:8083/
docker exec -it nxc sh -c 'ls -l /var/log/nginx'
```

Pushing do prywatnego rejestru (lokalny registry)

Cel: Tagowanie i push/pull bez Docker Huba.

Uruchom registry:

```
docker run -d -p 5000:5000 --name registry registry:2
```

Tag i push swojego obrazu:

```
docker tag mini-http:1.0 localhost:5000/mini-http:1.0
docker push localhost:5000/mini-http:1.0
```

Pull i uruchom z rejestru:

```
docker rmi mini-http:1.0
docker run -d --name from-reg -p 8001:8000 localhost:5000/mini-http:1.0
```

Interakcja z działającymi kontenerami

Wejście do kontenera:

```
docker exec -it NAZWA sh       # lub bash, jeśli jest
```

Logi:

```
docker logs NAZWA
docker logs -f NAZWA          # follow
docker logs --since=10m NAZWA   # ostatnie 10 minut
```

Procesy i zasoby:

```
docker top NAZWA
docker stats                  # live CPU/MEM/NET/IO
```

Sieci i IP:

```
docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' NAZWA
docker network ls
docker network inspect bridge
```

Kopiowanie plików:

```
docker cp NAZWA:/ścieżka/w/kontenerze ./lokalnie
docker cp ./lokalny_plik NAZWA:/tmp/
```

Sygnały i zatrzymanie:

```
docker stop NAZWA             # SIGTERM + timeout → SIGKILL
docker kill NAZWA             # natychmiast (SIGKILL)
docker kill --signal=HUP NAZWA   # wysłanie konkretnego sygnału
```

Zadania praktyczne

Zadanie 1 – Docker Tryhackme
Przerób oba pokoje TryHackMe: **Intro to Containerisation** i **Intro to Docker**.

- https://tryhackme.com/room/introtocontainerisation
- https://tryhackme.com/room/introtodockerk8pdqk

Zadanie 2 – Docker Online

Zgodnie z instrukcją: https://www.docker.com/101-tutorial/

Uruchom zdalnie: https://labs.play-with-docker.com/ lub lokalnie workshop 101-tutorial i wykonaj po kolei zadania.

Dodatkowe laby dla chętnych:

- https://kodekloud.com/free-labs/docker/
- https://killercoda.com/docker
- https://killercoda.com/cloudnc

Zadanie 3 – Docker Security

Na podstawie walktrough (https://github.com/IngoKl/THM-Hamlet/blob/main/solution/official-walkthrough.md) wykonaj pokój tryhackme Hamlet: https://tryhackme.com/room/hamlet.

Materiały dodatkowe do zadania:

- https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html
- https://book.hacktricks.wiki/en/linux-hardening/privilege-escalation/docker-security/index.html
- https://book.hacktricks.wiki/en/network-services-pentesting/2375-pentesting-docker.html
- https://exploit-notes.hdks.org/exploit/container/docker/docker-escape/