

UNIwersytet w Białymstoku  
Wydział Informatyki

Mateusz WRÓBLEWSKI

**Analiza porównawcza narzędzi do  
orkiestracji kontenerów Docker**

Promotor:  
dr inż. Andrzej KUŹELEWSKI

BIAŁYSTOK 2023/2024



# Contents

<b>Wstęp</b>	<b>5</b>
<b>1 Konteneryzacja</b>	<b>9</b>
1.1 Podstawowe pojęcia konteneryzacji za pomocą Dockera . . . . .	9
1.1.1 Plik konfiguracyjny Dockerfile . . . . .	9
1.1.2 Obrazy i kontenery Dockerowe . . . . .	10
1.2 Różnica między konteneryzacją, a wirtualizacją . . . . .	11
1.3 Docker i Podman . . . . .	11
<b>2 Aplikacja</b>	<b>15</b>
2.1 Java . . . . .	15
2.2 Quarkus . . . . .	16
2.2.1 Endpointy . . . . .	17
2.2.2 Maven . . . . .	19
2.2.3 Postman . . . . .	20
2.3 PostgreSQL . . . . .	20
2.3.1 Hibernate . . . . .	21
2.3.2 pgAdmin . . . . .	21
2.4 Keycloak . . . . .	23
<b>3 Orkiestracja</b>	<b>25</b>
3.1 Docker Swarm . . . . .	26
3.2 Kubernetes . . . . .	29
3.2.1 Uruchomienie aplikacji . . . . .	30
3.3 Nomad Hashicorp . . . . .	33
<b>4 Opis przeprowadzonych eksperymentów</b>	<b>39</b>
4.1 Platforma testowa . . . . .	39
4.2 Eksperyment obciążenia procesora i użycia pamięci . . . . .	39
4.2.1 Swarmprom . . . . .	40
4.3 Eksperyment startu aplikacji . . . . .	40
4.4 Eksperyment stabilności działania . . . . .	43
<b>5 Analiza</b>	<b>45</b>
5.1 Wyniki i wnioski obciążenia procesora i użycia pamięci . . . . .	45
5.2 Wyniki i wnioski badania startu aplikacji . . . . .	49
5.3 Wyniki i wnioski stabilności działania . . . . .	50

Podsumowanie	53
Bibliografia	57

# Wstęp

Obecnie zauważalny jest trend przejściowy z dużych monolitycznych aplikacji, na aplikacje mikroserwisowe. Jak dotąd powszechnie powstawały duże i ciężkie aplikacje, lecz w powodu powstania *Dockera* i *orkiestratorów* coraz częściej zaczęto przechodzić na podział tej aplikacji na mniejsze części.

Aplikacje *monolityczne* polegają na jednoczesnym uruchomieniu wszystkich komponentów aplikacji w tym samym czasie. Co za tym idzie zmiany w jednym komponencie wymagają restartowania całości.

Z drugiej strony aplikacje *mikroserwisowe* polegają na podziale serwisu na mniejsze niezależne części, które komunikują się za pośrednictwem *API*. Pozwala to na podział obowiązków między członkami zespołu w taki sposób, żeby praca jednej osoby nie wpływała na pracę drugiej. Co prowadzi do zwiększenia elastyczności i ułatwia utrzymanie kodu aplikacji.

Popularnym narzędziem do tworzenia aplikacji mikroserwisowych jest *Docker* 1. Pozwala on odizolować i podzielić poszczególne części aplikacji, na elementy zwane *kontenerami*. Każdy *kontener* posiada wszystko co potrzebuje, aby zostać uruchomiony, więc bez znaczenia, gdzie będzie uruchomiony, będzie działał zawsze tak samo. Pozwala to programistom na wybór odnośnie używanego systemu operacyjnego. Ponadto kontenery są łatwo skalowalne i nie wymagają dużych mocy obliczeniowych w odróżnieniu do maszyn wirtualnych.

W pewnym momencie rozwoju aplikacji, gdy aplikacja staje się coraz większa. Przychodzi problem związany z zarządzaniem wszystkimi kontenerami. Z pomocą przychodzą narzędzia zwane *orkiestratorami*, za pomocą których można zarządzać całym cyklem życia *kontenerów*, czyli wdrażaniem, aktualizowaniem, skalowaniem i usuwaniem. Przykładowymi zadaniami orkiestratorów jest dynamiczne przypisanie zasobów, monitorowanie wydajności kontenerów, czy odbudowa kontenera w przypadku wystąpienia błędów. W rozdziale 3 zostały opisane wykorzystane orkiestratory.

Aspektem teoretycznym pracy jest zapoznanie się z narzędziem Docker1, jego możliwościami, komendami i sposobem działania. Następnie zaprojektowanie aplikacji2 za pomocą której będzie można porównać dane *orkiestratory* opisane w rozdziale 3.

Aspekt praktyczny składa się z własnoręcznego zaimplementowania aplikacji i jej skonteneryzowania. Następnie zainstalowania i skonfigurowania *orkiestratorów* lokalnie, a na koniec wykonanie testów, takich jak sprawdzenie obciążenia podzespołów, czasu uruchomienia, czy sprawdzenie wytrzymałości na ruch użytkowników, poprzez zalanie aplikacji dużą ilością zapytań. Teoretycznie najlepsze wyniki powinny wyjść dla Docker Swarma, ponieważ jest to wewnętrzne narzędzie Dockera do orkiestracji kontenerów. Jest również najmniej rozbudowane i ma najmniej funkcji. Wyniki dla Kubernetesa i Nomada powinny być zbliżone. Są to narzędzia, które wymagają dodatkowego oprogramowania i odpowiedniego skonfigurowania.

Table 1: Wady i zalety aplikacji Monolitycznych i mikroserwisowych

Aplikacje monolityczne	Aplikacje mikroserwisowe
Może wystąpić problem w dodaniu nowych technologii do projektu	Pozwala na stopniowe przepisywanie części na kodu na nowe technologie
Proste do implementacji	Wszystkie komponenty są tworzone oddzielnie co pozwala na lepsze ich skalowanie i możliwość osiągnięcia lepszej wydajności
Potrzeba więcej czasu na dodanie nowych funkcjonalności	Trudne testowanie wszystkich serwisów
Poszczególne elementy systemu są zależne od siebie, co powoduje pojawienie się błędu w innej części aplikacji po wprowadzeniu zmian	Potrzebna jest dodatkowa infrastruktura, która pomoże obsłużyć wszystkie komponenty aplikacji np. AWS
Mniejsza skalowalność niż w przypadku aplikacji mikroserwisowych	Serwisy są odizolowane co sprawia, że pojawienie się błędu w jednym serwisie, nie będzie mieć on wpływu na cały system.
Szybsza komunikacja między komponentami	Wydajność całej aplikacji może być niska z powodu opóźnień w komunikacji między serwisami.
Łatwe do zarządzania i monitorowania	Tworzenie aplikacji mikroserwisowej wymaga dużej wiedzy i umiejętności.

Po wstępnej analizie dostępnych narzędzi, został wybrany framework Quarkus2.2 z powodu dużego wsparcia dla Kubernetesa 3.2 i konteneryzacji. Jak i również sprawdzenia, czy rzeczywiście uzyskam lepsze dla niego lepsze wyniki w porównanie

z innymi orkiestratorami. Podczas testów startu aplikacji, czy podczas testowania ruchu w aplikacji będą wysyłane zapytania do stworzonego API i na podstawie odpowiedzi, będzie można wyciągnąć odpowiednie wnioski.

Cała praca składa się z pięciu rozdziałów. W rozdziale 1 jest przedstawiona historia konteneryzacji, jej zalety oraz wady. Następnie opisane są podstawowe elementy potrzebne do korzystania z Dockera. Na koniec jest porównanie z innym równie popularnym narzędziem do *orkiestracji*, czyli Podmanem. 2 rozdział został poświęcony informacjom o aplikacji. Co ta aplikacja ma robić i jakie technologie zostały użyte do jej wykonania. 3 rozdział będzie mówił o orkiestacji. Przedstawiony zostanie opis poszczególnych *orkiestratorów*: Docker Swarm, Kubernetes oraz Nomad. Omówione zostaną architektury systemu dla każdego orkiestratora oraz w jaki sposób serwisy się łączą. refcha:Opis eksperymentów rozdział przedstawia platformę testową oraz użyte narzędzia do testowania i sposób ich wykorzystania. Ostatni 5 rozdział przedstawia uzyskane wyniki oraz ich interpretację.





# Chapter 1

## Konteneryzacja

Za pierwsze początki konteneryzacji można uznać rok 1979. W tym roku wynaleziono operację chroot. Jest to operacja dostępna w systemach unixowych. Umożliwia ona utworzenie środowiska, które jest odizolowane od systemu plików gospodarza. Nie można odczytać folderów, ani plików, które nie należą do utworzonego środowiska. Do utworzonego procesu można przypisać zasoby, takie jak moc procesora, ilość pamięci operacyjnej, czy pojemność dysku ssd.

### 1.1 Podstawowe pojęcia konteneryzacji za pomocą Dockera

W ręcznym budowaniu odizolowanego środowiska występuje problem w przypadku wprowadzania zmian. W takim przypadku należy wszystko od początku instalować i uruchamiać. Z pomocą przychodzi Docker za pomocą, którego z łatwością uruchomimy kontener z aplikacją.

#### 1.1.1 Plik konfiguracyjny Dockerfile

Plik DockerFile jest to plik tekstowy, który zawiera listę komend do utworzenia obrazu1.1.2.

Pierwszą komendą jaka musi wystąpić w pliku jest **FROM**, która jest referencją do istniejącego obrazu. Jest to podstawa budowy całego obrazu, np. **FROM NGINX** oznacza, że obraz będzie zbudowany na podstawie obrazu serwera NGINX.

Drugą komendą jest **COPY** za pomocą, której kopuje się pliki z systemu gospodarza do odizolowanego środowiska.

Trzecią komendą jest **ENV**, która określa zmienne środowiskowe według wzoru <klucz>=<wartość>.

Ostatnimi komendami są **EXPOSE** do ustalenia portu pod jakim będzie można połączyć się z aplikacją oraz **USER** do ustanowienia nazwy użytkownika.

Przykładowy plik **Docker File**

```
FROM registry.access.redhat.com/ubi8/openjdk-17:1.16

ENV LANGUAGE='en_US:en'

COPY --chown=185 target/quarkus-app/lib/ /deployments/lib/
COPY --chown=185 target/quarkus-app/*.jar /deployments/
COPY --chown=185 target/quarkus-app/app/ /deployments/app/
COPY --chown=185 target/quarkus-app/quarkus/ /deployments/
    quarkus/

EXPOSE 8080
USER 185

ENV JAVA_OPTS="-Dquarkus.http.host=0.0.0.0 -Djava.util.logging.manager=org.jboss.logmanager.LogManager"
ENV JAVA_APP_JAR="/deployments/quarkus-run.jar"
```

### 1.1.2 Obrazy i kontenery Dockerowe

Docker pozwala na spakowanie aplikacji i uruchomienie jej w kontenerze. Operacja ta działa na każdym systemie operacyjnym umożliwiającym wirtualizację w którym zainstalowany jest Docker.

Kontener Dockerowy jest to środowisko uruchomieniowe ze wszystkimi zależnościami, jak kod programu czy biblioteki. Aby utworzyć kontener potrzebny jest obraz. Warstwy utworzone przy pomocy obrazu są niemodyfikowalne, ale jest możliwość zainstalowania na najwyższej warstwie swojej aplikacji czy utworzenie własnych zmiennych i dodanie danych. W celu efektywnego zarządzania dużą ilością kontenerów potrzebne jest narzędzie zwane *orkiestratorem* 3.

Najbardziej przydatne komendy w przypadku kontenerów to:

**docker container run** – tworzy i uruchamia kontener wykorzystując obraz,

**docker container logs** – pozwala przejrzeć logi kontenera,

**docker ps** – wyświetla listę aktualnie uruchomionych kontenerów, po dodaniu parametru -a wyświetla również zatrzymane kontenery,

**docker start** – uruchamia kontener,

**docker stop** – zatrzymuje kontener,

Wszystkie komendy można zobaczyć w dokumentacji dostępnej pod linkiem <https://docs.docker.com/reference/cli/docker/container/>.

Obraz Dockerowy jest to niezależny, uruchamialny plik używany do stworzenia kontenera. Do jego stworzenia potrzebny jest kod aplikacji, wszystkie zależności, biblioteki oraz Dockerfile 1.1.1. Służy głównie do przechowywania konfiguracji aplikacji jako szablon i można go udostępnić dla innych osób, żeby mogli go zainstalować na własnym systemie. Można także wdrożyć go na publiczne repozytorium, takie jak Docker Hub. Obrazem może być system operacyjny lub baza danych. Nie da się

danego obrazu modyfikować jak już został utworzony.  
Najbardziej przydatne komendy w przypadku obrazów to:

**docker images** – wypisuje listę dostępnych lokalnie obrazów,

**docker create** – tworzy obraz, wraz z utworzenie kontenera, ale bez uruchomienia go,

**docker run** – tworzy obraz, wraz z utworzenie kontenera i jego uruchomieniem

**docker pull** – ściąga obraz ze zdalnego repozytorium

Wszystkie komendy można zobaczyć w dokumentacji dostępnej pod linkiem <https://docs.docker.com/reference/cli/docker/image/>.

Przykładowe komendy pozwalające zbudować aplikację opisaną w rozdziale 2.

- „./mvnw package” - przed zbudowaniem obrazu, należy zbudować całą aplikację za pomocą **Mavena**,
- „**docker build -f src/main/docker/Dockerfile.jvm -t APP-NAME .**” - komenda budująca obraz przy użyciu pliku Dockerfile,
- „**docker run -i -rm -p 8080:8080 APP-NAME**” - uruchomienie kontenera na podstawie zbudowanego obrazu,

## 1.2 Różnica między konteneryzacją, a wirtualizacją

Wirtualizacja jest to proces tworzenia warstwy między systemem hosta, a maszyną wirtualną. Komputer zostaje podzielony na wiele mniejszych wirtualnych systemów, gdzie każdy z nich ma swój własny system operacyjny, zasoby i każdy z nich działa niezależnie. Oprogramowaniem, które umożliwia wykonanie wirtualizacji jest hypervisor.

W przypadku konteneryzacji, nie tworzy się osobnego systemu od początku, ale używane są zasoby i system operacyjny hosta. Do tego wykorzystywane jest specjalne oprogramowanie do konteneryzacji, jak Docker albo Podman.

Przykładowy pogląd na oba przypadki przedstawiłem na rysunku 1.1

## 1.3 Docker i Podman

Podman jest narzędziem Open Source stworzonym przez firmę Red Hat. Podobnie jak Docker służy do tworzenia i zarządzania kontenerami. W tej sekcji przedstawię różnice i podobieństwa między tymi narzędziami.

Architektura Dockera jest architekturą typu klient-serwer, gdzie użytkownicy komunikują się z Docker daemonem, który jest odpowiedzialny za budowanie, uruchamianie i dystrybucję kontenerów. Komunikacja odbywa się za pomocą REST API przy pomocy Docker CLI. Podman za to nie posiada własnego deamona. Obsługa i zarządzanie kontenerami odbywa się bezpośrednio przez Podman CLI. Co przekłada się na szybkość poleceń. Obie architektury zostały przedstawione razem na rysunku

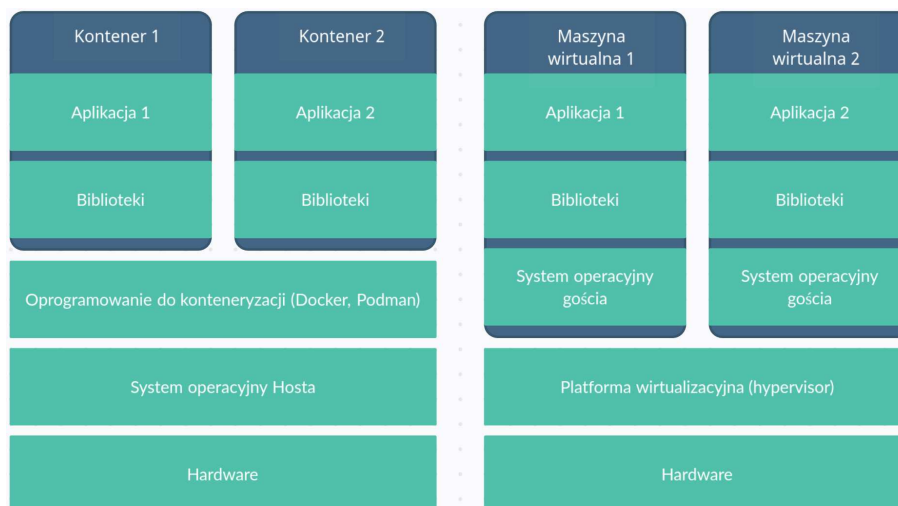


Figure 1.1: konteneryzacją i wirtualizacją

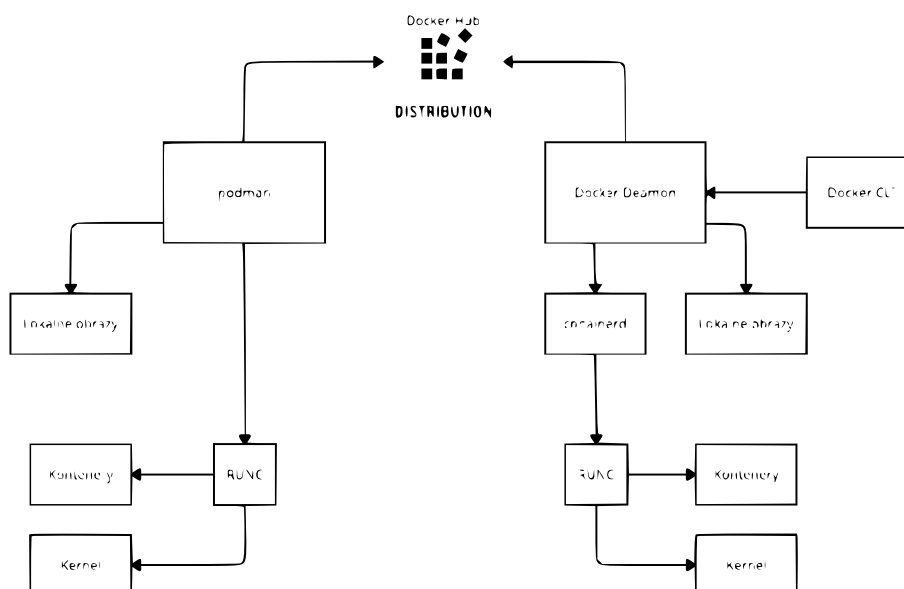


Figure 1.2: Architektura Dockera i Podmana

1.2.

Polecenia – treść i zachowanie poleceń są bardzo podobne, zwykle jedyną różnicą jest zamiana słowa kluczowego docker na podman. W poniższym przykładzie przedstawiłem dwie komendy, które tworzą kontener z systemem nginx:

- **podman run --name mynginx1 -dt -p 8080:80/tcp docker.io/nginx**
- **docker run --name mynginx1 -p 80:80 -d nginx**

Rootless containers – jest to rozwiązanie występujące w Podmanie, które pozwala na uruchamianie kontenerów jako użytkownik bez roli administratora. Odwrotna sytuacja jest w przypadku Dockera, gdzie użytkownik musi posiadać uprawnienia administratora, aby mógł zarządzać kontenerami.

Docker Desktop i Podman Desktop – są to narzędzia graficzne do Dockera i Podmana. Pozwalają przejrzeć wszystkie obrazy i kontenery działające lokalnie oraz wykonać na nich podstawowe operacje, jak usunięcie, zatrzymanie czy uruchomienie.

Networking – Docker tworzy swoją własną sieć zwaną `docker0`, która umożliwia komunikację między kontenerami. Kontenery Podmanowe dzielą sieć z hostem, czyli używają bezpośrednio interfejsu sieciowego hosta.

*Orkiestracja* – Docker posiada własny wbudowany *orkiestrator* zwany Docker Swarmem, który został szczegółowo opisany w rozdziale 3.1. Podman nie posiada własnego orkiestratora. Posiada za to większe wsparcie dla Kubernetesa<sup>3.2</sup>.

Table 1.1: Różnice między wirtualizacją, a konteneryzacją

Cecha	Konteneryzacja	Wirtualizacja
Użyte zasoby	W powodu używania systemu operacyjnego hosta potrzebują mało zasobów do działania.	Każda maszyna wirtualna korzysta z własnego systemu operacyjnego, więc do stworzenia wielu takich maszyn potrzebne jest wiele zasobów.
Czas uruchomienia	Kontenery szybciej się uruchamiają, ponieważ nie potrzebują uruchamiać całego systemu.	Maszyny wirtualne potrzebują dużo czasu do uruchomienia.
Izolacja	Kontenery używają zasoby sprzętowe hosta, więc w przypadku wystąpienia luk w zabezpieczeniach kontenera może to być wykorzystane przez osoby trzecie.	Maszyny wirtualne działają niezależnie. Błędy i luki, które mogą wystąpić w jednej maszynie wirtualnej nie mają wpływu na działanie innej maszyny wirtualnej.
Łatwość i szybkość użycia	Kontenery można łatwo i szybko tworzyć i usuać.	Maszyny wirtualne potrzebują, więcej czasu na uruchomienie oraz wymagają ich odpowiedniego zkonfigurowania.
Ilość potrzebnego miejsca	Obrazy Dockerowe są małe. Jeden obraz zajmuje jednostkę rzędu MB danych.	Maszyny wirtualne potrzebują, więcej miejsca, gdyż jedna maszyna wirtualna zajmuje jednostkę rzędu GB danych.
Kompatybilność	Kompatybilne tylko z dystrybucjami Linuxa. Narzędzia takie jak <b>Docker Desktop</b> umożliwiają działanie na <b>Windowsie</b> i <b>MacOS</b>	Maszyny wirtualne są kompatybilne ze wszystkimi systemami operacyjnymi.

# Chapter 2

## Aplikacja

Rozdział ten przedstawia narzędzia wykorzystane do stworzenia aplikacji do testowania orkiestratorów. Składa się ona z API, bazy danych i serwisu uwierzytliwiającego.

### 2.1 Java

Jest to obiektowy język programowania do tworzenia aplikacji webowych, desktopowych i mobilnych. Pozwala na to wirtualna maszyna javy (JVM), dzięki której kod może być uruchamiany na każdym systemie operacyjnym i urządzeniu, poprzez interpretację kodu.

Java jest to czysto obiektowy język, ponieważ już żeby uruchomić program trzeba napisać klasę oraz metodę **main()**. Posiada w sobie wszystkie mechanizmy stosowane w tym paradygmacie. Pierwszym jest dziedziczenie, które pozwala na przydzielenie właściwości i funkcjonalności z klasy rodzica do klasy pochodnej. Co powoduje, że nie trzeba ponownie pisać metod w klasach dziedziczących, więc jest mniej kodu jak i klasy są prostsze. Drugim jest abstrakcja, czyli zdefiniowanie własności dla obiektu. Przykładowo zdefiniowany użytkownik w aplikacji nie musi posiadać informacji o jego cechach osobowych. W celu jego logowania wystarczy login i hasło, a przy rejestracji dodatkowe dane jak imię, nazwisko czy email. Trzecim jest enkapsulacja, która pozwala na ukrywanie pól klasy przed odczytem z zewnątrz. W klasie użytkownika nie chcemy, aby ktoś niepowołany zmieniał jego dane. W Javie występuje kilka modyfikatorów dostępu, które pozwalają określić poziom dostępu. Możemy ich użyć do ochrony klas, metod i pól:

- „**public**” - używając tego specyfikatora można się odwołać do klasy (domyślne), pola, metody z dowolnego miejsca bez żadnych ograniczeń,
- „**private**” - w przypadku pól nie można się do nich odwołać poza daną klasą. Nie jest możliwe umieszczenie **private** przed deklaracją klasy, a w przypadku metod powoduje, że jest dostępna jedynie w obrębie klasy w której została zdefiniowana,
- „**protected**” - . Tutaj również nie można umieścić słowa kluczowego **protected** przed deklaracją klasy, a w przypadku metod i pól powoduje to, że będą one dostępne w klasie w której zostały one zdefiniowane oraz dla klas dziedziczących,

- „**domyślny**” - jeśli przy klasie, metodzie i polu nie użyjemy specyfikatora to dostęp będzie dostępny w obrębie pakietu,

Ostatnim paradygmatem jest polimorfizm, który pozwala dla obiektów przypisać cechy innych obiektów do siebie podobnym. Popularnym przykładem jest podział zwierząt na ich gatunki i określić ich cechy wspólne. Tworząc interfejs **zwierzę** z metodą **imię()** oraz **dźwięk()**, który następnie implementuje do klasy **pies** i **kot** można przypisać im poszczególne cechy. Przykładowo metoda **dźwięk** u psa może zwracać "hał hał", a u kota "miał miał".

Java posiada wiele bibliotek i frameworków dostępnych całkowicie za darmo, które znacząco ułatwiają tworzenie programów. Jest to zasługą dużej społeczności i sprawnej komunikacji poprzez rozwiązywanie problemów i odpowiadania na pytania techniczne na forach. Najbardziej popularnym frameworkiem jest **Spring Boot** za pomocą, którego można stworzyć własną aplikację internetową lub też Hibernate służący do mapowania obiektowo - relacyjnego.

Kolejną z zalet jest wielowątkowość, dzięki której możemy wykonywać operacje współbieżnie. Pozwala to budować aplikacje, które działają płynnie, bo nie musimy czekać, aż jedna operacja się zakończy, żeby rozpocząć inną. Przykładem może być kliknięcie przycisku w interfejsie użytkownika i jednocześnie już budowa kolejnego okienka, żeby wyświetlić odpowiedź.

Opis innych zalet i funkcji **Javy** można znaleźć w książce na pozycji [1].

## 2.2 Quarkus

Jest to framework w którym została napisana główna aplikacja. Jest to aplikacja Rest, która pobiera dane z NBP Api i zapisuje je do bazy danych. Następnie można te dane wyświetlać, przeliczyć kurs z jednej waluty na drugą, modyfikować dane oraz je usuwać.

Do uruchomienia aplikacji potrzebne jest dodatkowe narzędzie, gdyż **Framework** nie posiada metody uruchomieniowej. Można to zrobić za pomocą **Mavena** i użyciu komendy `./mvnw quarkus:dev` 2.2.2 lub budując obraz i uruchamiając aplikację używając **Dockera**.

Filary projektowego tego **frameworka** to:

### Container first

Aplikacje napisane w Quarkusie charakteryzują się tym, że są zoptymalizowane pod kątem niskiego użycia pamięci i krótkiego czasu uruchomienia. Uzyskuje to poprzez wykonanie tych operacji jeszcze w trakcie budowania aplikacji: konfigurację parsowania, skanowanie classpathu, włączanie/wyłączanie funkcji na podstawie ładowania klas itd. W odróżnieniu od tradycyjnych frameworków, gdzie są one wykonywane w trakcie działania aplikacji. Dodatkowo istotnymi cechami **frameworka** jest ograniczenie refleksji, wsparcie na **GraalVM**, pre-boot natywnego obrazu oraz generowanie podczas budowania aplikacji plików, które są potrzebne przez orkiestratory co również poprawia szybkość uruchamiania.

### Imperatywne i reaktywne programowanie

Quarkus posiada wsparcie dla różnych stylów architektonicznych takich jak:



mikroserwisy, reaktywne aplikacje, architektury zdarzeń i funkcji, jak i również można w nim tworzyć aplikacje monolityczne.

### Natywne wsparcie dla kubernetesa

Quarkus zapewnia możliwość wdrożenia aplikacji na Kubernetesa bez potrzeby posiadania wiedzy odnośnie architektury Kubernetesa. Dzięki dostępnym rozszerzeniom wymaga to tylko niewielkiej konfiguracji. W trakcie działania aplikacji mamy do dyspozycji narzędzia służące do jej debugowania oraz sprawdzanie stanu oraz metryk dzięki zastosowaniu **SmallRye Health** oraz **Micrometer**. Quarkus również zawiera rozszerzenia pozwalające programistom na użycie **ConfigMaps i Secretów** jako plików konfiguracyjnych oraz możliwość tworzenia i debugowania aplikacji w tym samym środowisku, gdzie aplikacja jest uruchomiona.

### Wygodne programowanie

Ten podpunkt opisuje elementy, które pozwalają programiście skupić się głównie na pisaniu kodzie. Jedno z udogodnień jakie oferuje **Quarkus** to **live coding**, czyli możliwość pisania programu, przy czym aplikacja się sama odświeża po wprowadzeniu zmian do kodu. **Unified config** sprawia, że cała konfiguracja aplikacji jest w jednym pliku. Kolejnym dodatkiem jest **Dev UI**, który jest ekranem, gdzie możemy zobaczyć wszystkie zależności projektu oraz logi i wykonać przypadki testowe.

Szczegółowy opis tych zagadnień i kod na, którym się wzorowałem można znaleźć w książce na pozycji [2].

## 2.2.1 Endpointy

Endpoint obliczania kursu: **GET localhost:8080/currency/currencyExchangeValue**  
Body:

```
{
  "amount": 1,
  "myCurrency": "USD",
  "targetCurrency": "PLN"
}
```

Zwraca informacje o wartości waluty po wymianie z USD na PLN w formacie TEXT PLAIN. W przypadku błędnego podania kodu waluty lub błędnego typu danych wywoływany jest kod błędu 500.

Endpoint logowania: **POST localhost:8080/login**  
Body:

```
{
  "username": "admin",
  "password": "admin"
}
```

Zwraca token w formacie TEXT PLAIN, który następnie można użyć przy innych zapytaniach.

Endpoint rejestracji: **POST localhost:8080/register**

Body:

```
{
  "username": "matiu",
  "password": "pass",
  "firstName": "mat",
  "lastName": "wro",
  "email": "mat@wro.pl"
}
```

Endpoint rejestruje nowego użytkownika i przy użyciu wartości username i password można się zalogować.

Endpoint pobrania kursu z bazy danych w formacie JSON lub XML **GET localhost:8080/currency/getCurrency/table/table/value/value**

**Header: Accept** – „możliwość ustawienia typu zwracanego na application/json lub application/xml”;

**Table** – „tabela z której pobierzemy kurs np A, B lub C”;

**Value** – „numer waluty, którą pobierzemy z tabeli”;

Przykładowa odpowiedź:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
  <currencyReponseXML>
    <code>AOA</code>
    <exchangeRate>0.0048</exchangeRate>
    <nameOfCurrency>kwanza (Angola)</nameOfCurrency>
  </currencyReponseXML>
```

```
{
  "code": "AOA",
  "nameOfCurrency": "kwanza (Angola)",
  "exchangeRate": 0.0048
}
```

Endpoint pobrania kursów z bazy danych **GET localhost:8080/currency/getCurrencies/table/table**

**Header: Accept** – „możliwość ustawienia typu zwracanego na application/json lub application/xml”;

**Table** – „tabela z której pobierzemy kurs np A, B lub C”;

Endpoint pobrania danych z NBP Api i aktualizacja bazy danych:  
**POST localhost:8080/currency/postCurrency/table/table**

**Table** – „tabela z której pobierzemy kurs np A, B lub C”

**Auth** – „Bearer token z odpowiednią rolą. W przeciwnym wypadku zostanie zwrócony kod błędu 401 w przypadku braku tokenu uwierzytelnienia lub kod błędu 403 w przypadku braku uprawnień. ”

Endpoint usuwania waluty: **DELETE localhost:8080/currency/deleteCurrency/code**

**Code** – „Code - skrót nazwy waluty np. USD”

**Auth** – „Bearer token z odpowiednią rolą”

Endpoint aktualizacji waluty: **PUT localhost:8080/currency/putCurrency/table/table**

**Table** – „tabela z której pobierzemy kurs np A, B lub C ”;

**Code** – „skrót nazwy waluty np. USD”

**Auth** – „Bearer token z odpowiednią rolą”

W przypadku nieobsługiwanych **endpointów** zwracany jest kod błędu 404.

## 2.2.2 Maven

Celem tego narzędzia to nie tylko budowanie aplikacji, lecz jest to narzędzie do nadzorowania całego projektu. W odróżnieniu do narzędzi takich jak Ant, który służy do kompilacji, pakowania, testowania i dystrybucji, Maven również pozwala na reportowanie, generowanie stron internetowych oraz ułatwia komunikację między członkami zespołu. Szerzej to zagadnienie zostało opisane w pozycji [3].

Główną zasadą Mavena jest **Convention over configuration**. W **Javie** kod programu ma z góry ustaloną ścieżkę, pliki konfiguracyjne i testowe również. W ten sposób **Maven** może bez problemu odnaleźć te pliki i wykorzystując je może zbudować wykonywalny plik JAR (Java ARchive).

Cała konfiguracja Mavena jest zapisana w pliku pom.xml, który wykona wszystkie operacje automatycznie i przejdzie przez cały cykl budowania projektu. Najprostszy szablon takiego pliku może wyglądać następująco:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.acme</groupId>
  <artifactId>code-with-quarkus</artifactId>
  <version>1.0.0-SNAPSHOT</version>
</project>
```

Funkcje Mavena, które zostały wykorzystane w trakcie tworzenia aplikacji to:

- „**Lifecycle**” - są to kroki, które definiują proces budowania i dystrybucji projektu. Standardowymi funkcjami są: **validate** - sprawdza czy w projekcie nie ma błędów i czy wszystkie informacje są dostępne, **compile** - kompiluje

kod źródłowy projektu, **test** - uruchamia napisane testy jednostkowe, **package** - pakuje skompilowany kod do wykonywalnego pliku JAR, **verify** - uruchamia i sprawdza poprawność testów integracyjnych, **install** - instaluje paczkę w lokalnym repozytorium, która można wykorzystać w innym projekcie, **deploy** - kopiuje paczkę z lokalnego repozytorium i przesyła na zdalne repozytorium,

- „**Plugins**” - dzielą się na 2 grupy: **build plugins** - są wykonywane podczas budowania aplikacji i powinny być zdefiniowane pomiędzy tagami <build/> w pliku POM oraz **Reporting plugins** - są uruchamiane podczas generowania strony i powinny być zdefiniowane w tagu <reporting w pliku pom. Kilka najbardziej przydanych pluginów to: **clean** - czyści pliki po zbudowaniu, **compiler** - kompiluje pliki źródłowe, **install** - tworzy i instaluje plik wykonywalny w lokalnym repozytorium,
- „**Dependencies**” - opisuje zależności pobrane ze zdalnego repozytorium Mavena. Zależności wstawia się między taki **dependencies** w pliku pom,
- „**Repositories**” - przedstawia ścieżkę do repozytorium lokalnego i link do repozytorium zdalnego.

Alternatywnym narzędziem jest **Gradle**. Posiada on takie same funkcje co Maven, więc wybór zależy od użytkownika.

### 2.2.3 Postman

**Postman** jest narzędziem do testowania, budowania i modyfikacji API. Posiada możliwości testowania różnych zapytań http: GET, POST, PUT, DELETE i zapisać je w celu przyszłego wykorzystania. Jest dostępna wersja zdalna tego narzędzia i lokalna. Oto kilka z jego możliwości:

**Obsługa metod HTTP** – można wysyłać żądania na dany adres url przy użyciu wybranego typu zapytania,

**Dane wejściowe i wyjściowe** – można ustawić headery np. Accept: application/json lub application/xml, który definiuje jaki typ danych chcemy zwrócić, można ustawić **body** zapytania czyli dane przesyłane w trakcie requesta,

**Uwierzytelnianie** – można ustawić parametry zapytania, sposób autoryzacji np. używając **Bearer Tokenu** lub OAuth 2,

**Organizacja testów** – Postman zapewnia możliwość rejestracji i logowania, co daje możliwość zapisywania testów i ich udostępniania.

## 2.3 PostgreSQL

Jest to system zarządzania relacyjnymi bazami danych (RDBMS). Został stworzony w 1986 jako część projektu **POSTGRES** w Uniwersytecie Kalifornijskim. Od tego czasu jest projektem **open-source** obiektowo relacyjnej bazy danych używającej języka SQL połączonego z dodatkami pozwalającymi zapisywać dane.

### 2.3.1 Hibernate

Do komunikacji z bazą danych użyłem frameworka **Hibernate**. Jest to narzędzie umożliwiające mapowanie obiektowo-relacyjne. W języku obiekowym takim jak **Java**, gdzie wszystko jest klasami i obiektami. Możemy przełożyć daną klasę na tabelę w bazie danych. Można to uzyskać za pomocą adnotacji dostępnych w JPA. Również **Hibernate** ma zaimplementowaną tę bibliotekę. Aby użyć tego mechanizmu wystarczy, że dodamy adnotację **@Entity** przed nazwą klasy i oznaczymy klucz główny za pomocą **@Id**.

```
@Entity
public class Currency {

    @Id
    private String code;
    private String nameOfCurrency;
    private double exchangeRate;
}
```

Należy też ustalić adres bazy danych i dane logowania użytkownika. Taką konfigurację należy zapisać w pliku **application.properties** według następującego wzoru.

```
quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/postgres
quarkus.datasource.password = admin
quarkus.datasource.username = postgres
quarkus.hibernate-orm.database.generation=update
```

Pierwsza linijka odpowiada za adres bazy danych, port oraz nazwę. Druga określa hasło, a trzecia nazwę użytkownika. Ostatnia linijka jest odpowiedzialna za sposób tworzenia bazy danych po każdym uruchomieniu aplikacji.

### 2.3.2 pgAdmin

Jest to narzędzie **open-source** wspomagające pracę w bazą danych **PostgreSQL**. PgAdmin jest to graficzny interfejs użytkownika, który można uruchomić lokalnie w przeglądarce internetowej standardowo na porcie 5050. Po wejściu należy się zalogować i można połączyć się z bazą danych. Posiada on szereg funkcji takich jak:

- możliwość przejrzenia bazy danych,
- możliwość wykonywania zapytań SQL,
- możliwość modyfikacji tabel
- możliwość zaimportowania danych z tabeli w różnych formatach, np. Excel, CSV jak i również możliwość stworzenia i wyeksportowania kodu SQL tworzącego tabele i zapisujące do niej dane,
- możliwość śledzenia używanych zasobów w czasie rzeczywistym przez baze danych,

- możliwe jest również stworzenia diagramu ERD

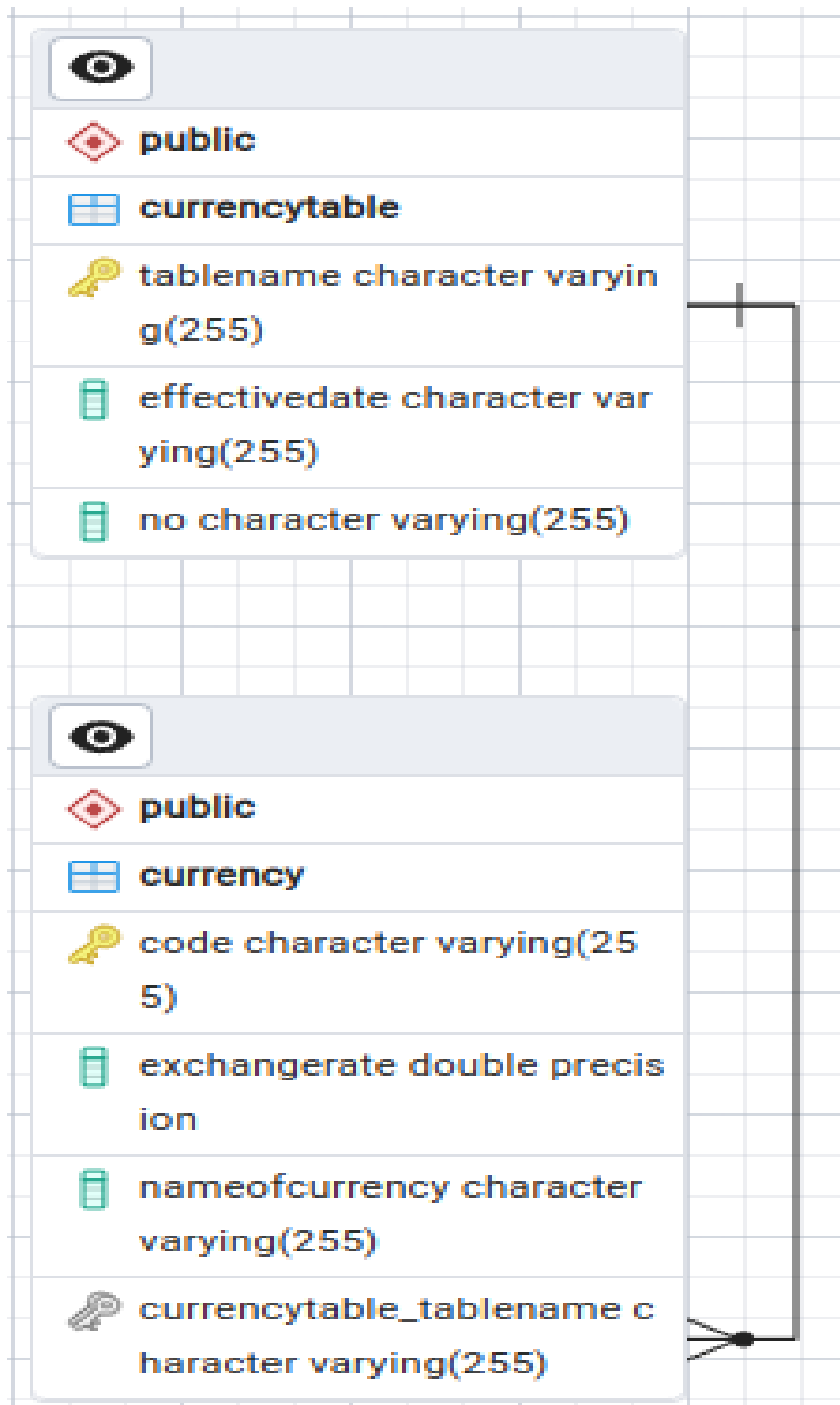


Figure 2.1: ERD bazy danych

## 2.4 Keycloak

Jest to narzędzie open-source służące do uwierzytelniania użytkowników i dostępem do zasobów. Ułatwia czynności związane z bezpieczeństwem użytkowników. Nie trzeba zajmować się przechowywaniem danych użytkowników.

Po zalogowaniu do KeyCloaka nie trzeba powtarzać tej czynności, co oznacza, że nie trzeba korzystać z wielu aplikacji, bo wszystko dostępne jest w jednym miejscu.

KeyCloak umożliwia logowanie przy użyciu sociali takich jak facebook, Github czy Google. Wystarczy dodać taką opcję w konsoli admina. Nie ma potrzeby pisania kodu.

Kolejną możliwością jest możliwość przechowywania danych w zewnętrznej bazie danych, połączenie z serwerami **LDAP** i połączenie z **Active Directory**.

Keycloak posiada konsolę admina, gdzie można przejrzeć wszystkie dostępne możliwości tego narzędzia.

Keycloak posiada również wsparcie dla standardowych protokołów komunikacyjnych, takich jak: OpenID Connect, OAuth 2.0, SAML.

W stworzonej aplikacji łączę się z **Keycloakiem** za pomocą następujących zapytań **HTTP**.

```
@RegisterRestClient(baseUri = "http://localhost:8180/")
public interface KeyCloakService {

    @POST
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    @Path("realms/master/protocol/openid-connect/token")
    Response getToken(MultivaluedMap<String, String>
        formData);

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Path("admin/realms/master/users")
    String register(@HeaderParam("Authorization") String
        authorizationHeader, String json);
}
```

Pierwsze polecenie w powyższym kodzie pobiera **token** przy prawidłowym zalogowaniu. Ten **token** później jest potrzebny, aby wykonać akcje dostępne tylko dla zalogowanych użytkowników i posiadających odpowiednią rolę w systemie. W celu zalogowania trzeba wysłać polecenie **POST** na dany adres, który składa się z nazwy **localhost** oraz portu, czyli **http://localhost:8180/** oraz kolejnych wartości, które są specyficzne dla tego narzędzia i są one ustawione domyślnie, czyli **realms/master/protocol/openid-connect/token**. Funkcja przyjmuje jeden argument, którymi są dane logowania opisane w sekcji 2.2.1 przy **endpointcie** logowania.

Drugie polecenie służy do stworzenia użytkownika. Podobnie jak poprzednio część podstawowa adresu jest taka sama, ale pozostałym argumentom się zmieniają na **admin/realms/master/users** i kolejny raz wysyłamy zapytanie typu **POST**. Tutaj mamy dwa argumenty, gdzie pierwszym jest token logowania użytkownika z rolą admina, żeby mógł utworzyć użytkownika. W kodzie programu są zapisane te dane, więc użytkownik nie musi ich podawać. Drugim elementem są dane do

stworzenia nowego użytkownika. Sposób na wykonanie zapytania znajduje się w sekcji 2.2.1 przy **endpointcie** rejestracji.



# Chapter 3

## Orkiestracja

Kontenery dostarczają możliwość tworzenia oprogramowania dla rozwiązań chmurowych i centrów danych. Dzięki nim aplikacja w każdym systemie zachowuje się tak samo, umożliwiając szybkie i efektywne ich wdrażanie. Przy czym, w trakcie rozwoju aplikacji i jej stopniowemu zwiększaniu się przychodzi potrzeba kontroli nad cyklem życia projektu i automatyzacji pewnych procesów. Pozwalają także uzyskać stałe i nieprzerwane działanie kontenerów, monitorowanie stanu systemu, oraz umożliwiając tego samo regenerowanie się. Takimi narzędziami są właśnie orkiestratory.

Ważnymi pojęciami odnośnie orkiestracji są:

**Continuous Integration (CI)** odnosi się do integrowania, budowania i testowania kodu w środowisku developerskim. Programiści muszą często integrować kod w zdalnych repozytoriach (np. **Github**). Od wielkości projektu zależy jak często to ma następować. Jednak samo dzielenie kodu nie wystarczy, należy go też sprawdzić, wykonać testy. Wykonywane jest to przez **pipeline**, które dają odpowiednie sygnały czy wystąpiły jakieś błędy czy wszystko przebiegło pomyślnie. **Continuous Integration** powinno być uruchamiane po każdym **commitcie** albo **pushu**.

Kolejnym krokiem jest **Continuous Delivery (CD)**. Jest to mechanizm, który występuje wtedy, kiedy ktoś zaaplauduje zmianę na zdalnym repozytorium i przejdzie pozytywnie testy. Wtedy użytkownik może kliknąć przycisk i nowy kod zostanie automatycznie zaimportowany na środowisko produkcyjne.

Jest jeszcze **Continuous Deployment (CDP)**, przy którym nie potrzeba osoby, która potwierdzi wdrożenie. Cały proces od **commita** do wdrożenia na produkcję odbywa się automatycznie.

W **Docker Swarmie** można stworzyć wielu klastrów, gdzie jeden może służyć jako środowisko testowe, a reszta jako środowisko produkcyjne. W przedstawionym dalej przypadku zostanie stworzony tylko jeden klaster w celach testowych. Więcej o tym jak podzielić klastry i ustawić w nich środowiska testowe i produkcyjne można przeczytać w rozdziale 5 z książki na pozycji [4].

W tym rozdziale zostały przedstawione najpopularniejsze z nich oraz w jaki sposób zostały skonfigurowane do testów opisanych w rozdziale 4.

Jeszcze przed rozpoczęciem konfiguracji, został stworzony kontener dla **Keycloak**, działa niezależnie do każdego przedstawionego orkiestratora w tym rozdziale. Jest budowany jako zwykły kontener na **Dockerze** za pomocą komendy:

```
docker service create --name pgadmin -p 5050:80
--network="NBP-APP" -e PGADMIN_DEFAULT_EMAIL="admin@admin.
com"
-e PGADMIN_DEFAULT_PASSWORD="admin" dpage/pgadmin4
```

## 3.1 Docker Swarm

Przydatne komendy:

- „**docker swarm init**” - inicjalizacja Docker Swarm w klastrze,
- „**docker service**” - służy do kontroli serwisu(kontenera) w klastrze. Dostępne parametry to: *create* do stworzenia serwisu, *ls* do zobaczenia wszystkich serwisów, *inspect* do sprawdzenia szczegółowych informacji o serwisie, *scale* do zwiększenia liczby replik serwisu
- „**docker stack**” - służy do stworzenia lub usunięcia jednego lub wielu serwisów zdefiniowanych z pliku *yaml*,
- „**docker network create**” - służy do utworzenia sieci, która służy do komunikacji między serwisami. W Docker Swarmie dodatkowo należy dodać wartość – **driver overlay**.

### Utworzenie smarma:

Na początku należy utworzyć klaster **Docker Swarma** przy użyciu **Advertise address**, którym jest adres **IP**. To **IP** jest potrzebne kiedy klient będzie chciał dołączyć do serwera, więc powinno się wybrać taki adres, aby każdy mógł się z nim połączyć. W systemie **Ubuntu** zostało użyte polecenie **ifconfig** i pobrany adres sieci.

```
docker swarm init --advertise-address <ip_addr>
```

### Przykładowa odpowiedź:

```
Swarm initialized: current node (jbdzcfqf1er2t96u7z11743xk)
is now a manager.
```

To add a worker to this swarm, run the following command:

```
docker swarm join --token SWMTKN-1-6
diawyi338mqi75ma5vupxhtezugl3t6xwpaijn2a5uj957922-3
k6gfgxj98qjq0mpnelnbpnj2 192.168.65.9:2377
```

To add a manager to this swarm, run 'docker swarm join--token manager' and follow the instructions.

Można także przejrzeć listę użytkowników połączonych z danym **Nodem** za pomocą komendy **docker node ls** i wtedy odpowiedź może wyglądać w następujący sposób:

ID	AVAILABILITY	MANAGER	STATUS	HOSTNAME	ENGINE	VERSION	STATUS
feszonzqwmnbapcv9a76wxq122	*	docker	desktop	Ready			
	Active	Leader		25.0.3			

Utworzenie networku z opcją **overlay**. Pozwala ona na komunikację kontenerów utworzonych na różnych klastrach orkiestratora. Jeśli nie utworzymy sieci to utworzone serwisy połączą się z **ingressem**. Rekomendowane jest utworzenie własnej sieci dla każdej grupy aplikacji, które mają pracować razem.

```
docker network create --driver overlay NBP-APP
```

**Najpierw należy stworzyć serwis bazy danych:**

```
version: '3.8'

services:
  postgresql:
    image: postgres
    deploy:
      replicas: 1
      placement:
        constraints:
          - node.role == manager
      restart_policy:
        condition: any
    ports:
      - "5432:5432"
    networks:
      - NBP-APP
    environment:
      - POSTGRES_PASSWORD=admin
      - PGDATA=/var/lib/postgresql/data/pgdata
    volumes:
      - pg_data:/var/lib/postgresql/data/pgdata

networks:
  NBP-APP:
    external: true

volumes:
  pg_data:
    driver: local
```

W powyższym kodzie został ustawiony obraz bazodanowy postgres, liczba replik na 1, port na 5432. W **environment** zostało ustawione hasło oraz miejsce magazynowania danych bazodanowych. Połączone również zostało z wcześniej utworzoną siecią NBP-APP, co spowodowało, że połączenie z bazą danych wewnątrz danej sieci będzie można uzyskać ustawiając adres na **host.docker.internal**.

```
quarkus.datasource.jdbc.url=jdbc:postgresql://host.docker.internal:5432/postgres
```

```
quarkus.datasource.password = admin
quarkus.datasource.username = postgres
```

Następnie tworzona jest aplikacja za pomocą następującego kodu:

```
version: '3.8'

services:
  nbp-app:
    image: matiuw/nbp-app:latest
    deploy:
      replicas: 1
      placement:
        constraints:
          - node.role == manager
      restart_policy:
        condition: any
    ports:
      - "8080:8080"
    networks:
      - NBP-APP

networks:
  NBP-APP:
    external: true
```

W powyższym przykładzie został stworzony serwis wykorzystujący utworzony obraz aplikacji przesłanej na zdalne repozytorium **Docker Hub**. Aplikacja jest dostępna na porcie 8080 i dołączona do sieci NBP-APP.

Utworzona architektura jest dostępna na obrazku 3.1.

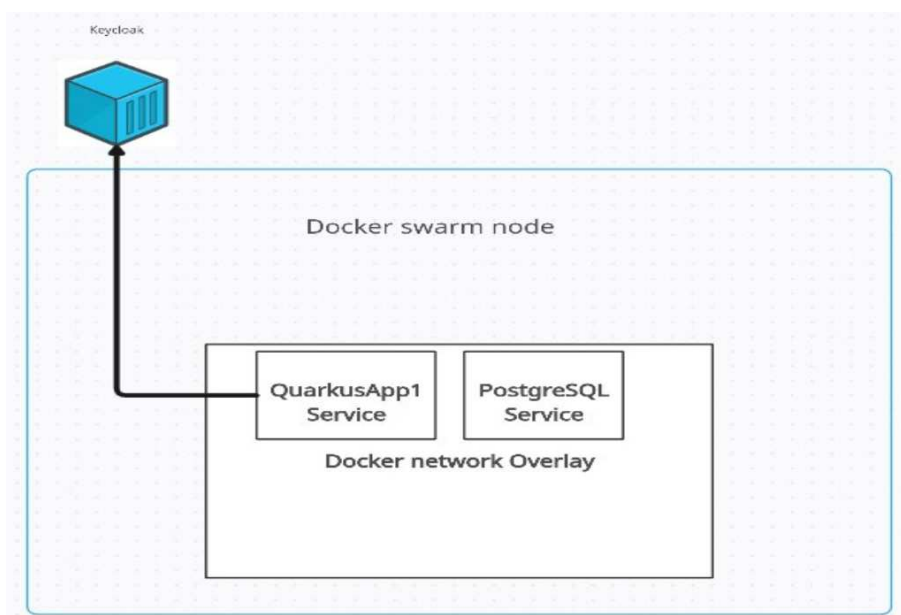


Figure 3.1: Architektura aplikacji w Docker Swarmie

Szczegółowy sposób uruchomienia został opisany w rozdziale 3 w pozycji [4].

## 3.2 Kubernetes

Architektura Kubernetesa składa się z przynajmniej jednego **master node** połączonego z **worker nodes**. Każdy z **worker nodów** posiada proces o nazwie **cubelet**, który pozwala innym **nodom** komunikować się między sobą oraz przeprowadzać zadania na nich. Na przykład uruchamiać aplikacje. Na każdym **worker nodzie** uruchomione są aplikacje zamknięte w kontenerach. Na **master nodzie** uruchomione procesy, które są niezbędne do prawidłowego działania klastra. Jednym z takich procesów jest **Api Server**, który jest punktem wejścia dla **klastra Kubernetesa**. Jest to proces do którego będą się komunikować takie elementy jak **kubernetes dashboard** i narzędzie linii poleceń. Kolejnym procesem jest **Controller manager**, którego zadaniem jest kontrola nad tym co się dzieje wewnątrz klastra. Jego zadaniem jest na przykład odbudowa **poda**, gdy wystąpi jakiś błąd lub jego restart. Trzecim procesem jest **Scheduler**, którego zadaniem jest obserwowanie API serwera w poszukiwaniu nowego zadania i przydzielenie odpowiedniego **worker noda**. Czwartym procesem jest **etcd**, który przetrzymuje stan **klastra Kubernetesa**. Zatem posiada wszystkie dane odnośnie każdego noda i każdego kontenera. Dane zapisane w tym procesie wykorzystywane są w przypadku zrobienia **Backupu**. Komponent umożliwiający **worker i master nodom** porozumiewanie się między sobą jest **virtual network**.

Do uruchomienia Kubernetesa lokalnie na komputerze został wykorzystany **Minikube**. Zostało stworzone przez Google w 2016 i obecnie jest jednym z popularniejszych sposobów na uruchomienie klastra **Kubernetesa** lokalnie. Uruchomienie jest proste, gdyż wymaga jedynie użycia komendy **minikube start** i cała konfiguracja wykona się automatycznie. Zostanie wybrany odpowiedni **driver** (domyślnie **Docker**) oraz przypisane odpowiednie zasoby komputera.

Dodatkowo jest możliwość włączenia graficznego interfejsu za pomocą komendy **minikube dashboard**. Pozwala on wdrażać skonteneryzowane aplikacje do klastra **Kubernetesa**. Rozwiązywać problemy odnośnie aplikacji. Zarządzać zasobami klastra. Pobierać informacje o podach i aplikacjach uruchomionych w klastrze. Tworzyć i modyfikować indywidualne elementy **Kubernetesa**.

Do **Minikube** dołączona też jest aplikacja służąca do komunikacji między API Kubernetesa, a systemem sterowania za pomocą linii poleceń. Zaczyna się polecenia od nazwy **kubectl**. Pozwala wykonać operacje za pomocą komend. Poniżej kilka z nich, które są wykorzystywane do wdrożenia aplikacji:

- Node - jest to fizyczna lub wirtualna maszyna
- Pod - najmniejsza jednostka w **Kubernetesie**, jest to abstrakcja nad kontenerem. Jego zadaniem jest uruchomienie środowiska albo powłoki nad kontenerem. Zazwyczaj w jednym **podzie** jest jeden kontener. W specyficznych przypadkach można uruchomić więcej kontenerów w ramach **poda**, lecz nie jest to zalecane. Każdy **pod** posiada swój własny adres IP i możliwa jest komunikacja między **podami** za pomocą tego adresu.
- Service - jest to statyczny lub dynamiczny adres IP, który może być dołączony do **poda**. Jednocześnie pod i service działają niezależnie, więc jeśli coś się stanie z **podem** to service wciąż będzie działał.
- Ingress - zamiast łączenia się przy pomocy adresu IP, można wykorzystać ten element w celu nadania nazwy dla **servisu**. Przykładowo zamiast łączyć się

z adresem `http://192.168.0.1:8080` można ustawić to na `https://nbp-app:8080` z nazwą domeny i szyfrowanym połączeniem. Zapytania w ten sposób będą najpierw trafiać do ingressa i następnie do serwisu.

- ConfigMap - jest to zewnętrzna konfiguracja dla aplikacji. Przydaje się w sytuacji, gdyby zaszła potrzeba zmiany nazwy aplikacji to wtedy trzeba zmienić to w aplikacji, zbudować ją, wysłać na repozytorium i z powrotem zbudować podą. Za pomocą configmapy możemy pominąć wszystkie te kroki. Configmap również może przetrzymywać dane logowania do bazy danych, lecz dane te są dostępne i nie są zakodowane.
- Secret - działa podobnie do configmapy, ale służy do przechowywania danych wrażliwych. Dane wewnątrz są zakodowane za pomocą algorytmu base 64. Dane te nie są całkowicie zabezpieczone, ponieważ każdy kto posiada dostęp do API może zobaczyć te dane i je zmodyfikować. Również osoby, które mają zezwolenia do tworzenia podów mają dostęp do sekretu. Aby w pełni zabezpieczyć takie dane należy zainstalować aplikacje zewnętrzne. Innym zastosowaniem sekretów może być trzymanie wewnątrz nich zmiennych środowiskowych i parametrów aplikacji.
- Volume - służy do przechowywania danych na przykład z bazy danych na fizycznym nośniku. Przydaje się w przypadku wystąpienia awarii dla podą posiadającego bazę danych lub coś innego posiadającego swój stan. Miejsce przechowywania może być lokalnie dysk ssd na komputerze lub zdalnie, poza klastrem Kubernetesa.
- Deployment - w czasie tworzenia i działania aplikacji występują błędy powodujące przerwanie działania aplikacji. Kubernetes oferuje mechanizm auto regeneracji podą, lecz czasie w którym to następuje użytkownik, nie ma dostępu do aplikacji. Z pomocą przychodzi deplyoment, który tworzy repliki podów i natychmiast zastępuje niedziałającego podą na takiego wolnego od błędów. Pozwala on na ustalenie liczby replik podów. Jest to kolejna warstwa abstrakcji nad podami. Co za tym idzie powinno się pracować nad deploymentem, a nie nad podami. Nie można zastosować deplyomentu w przypadku baz danych, ponieważ mają swój stan, czyli dane. Trzeba ustalić, który pod ma obecnie uprawnienia do dostępu do tych danych. Potrzebny jest mechanizm pozwalający zarządzać dostępem do miejsca przechowywania danych w celu uniknięcia niespójności danych.
- StatefulSet - używane jest do aplikacji jak bazy danych: MySQL, Elastic Search, mongoDB i innych aplikacji posiadających swój stan. Podobnie jak deployment jego zadaniem jest replikacja podów oraz ich skalowanie, ale również zapewnia synchronizację dostępu do bazy danych.

Informacje o elementach Kubernetesa można znaleźć przykładowo w książce na pozycji [5].

### 3.2.1 Uruchomienie aplikacji

Do stworzenia każdego elementu Kubernetesa zostały wykorzystane pliki konfiguracyjne, które składają się z następujących 3 części. Początek każdego pliku zawiera informacje

o wersji i odpowiedzi, jakiego elementu kubernetesa dotyczy: poda, service itp. Pierwsza część zawiera informacje o metadanych, druga część informuje o specyfikacji, czyli przykładowo o nazwie i obrazie. Ostatnia część mówi o statusie, którego dane pobierane są z etcd.

Stworzenie bazy danych za pomocą komendy **kubectl apply -f postgresPod.yml**.

```
apiVersion: v1
kind: Pod
metadata:
  name: pg-pod
  labels:
    name: postgres
spec:
  containers:
  - name: postgres
    image: postgres:12.3
    imagePullPolicy: IfNotPresent
    ports:
    - name: pg-port
      containerPort: 5432
    env:
    - name: POSTGRES_PASSWORD
      value: admin
    - name: PGDATA
      value: /data/k8s
    volumeMounts:
    - name: pg-vol
      mountPath: /data
    securityContext:
      runAsUser: 0
      runAsGroup: 0
  volumes:
  - name: pg-vol
    hostPath:
      path: /data
      # path: /Users/<>/minikube/pgdata
  restartPolicy: Never
  securityContext:
    runAsUser: 1000
    runAsGroup: 1000
    fsGroup: 1000
```

Kolejnym krokiem jest stworzenie Deploymentu dla aplikacji za pomocą komendy **kubectl apply -f quarkusDeployment.yml**. Komunikacja między **podami** w ramach tego samego klastra odbywa się wykorzystując ich adresy **IP**. Można to zrobić za pomocą komendy **kubectl describe pod nazwa-poda**. I następnie należy zaktualizować **application.properties** w aplikacji o nowy adres dla bazy danych.

plik quarkusDeployment.yml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nbp-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nbp-app
  template:
    metadata:
      labels:
        app: nbp-app
    spec:
      containers:
      - name: nbp-app
        image: matiuw/nbp-appkubernetes:latest

```

Następnie trzeba utworzyć serwis.

```

apiVersion: v1
kind: Service
metadata:
  name: nbp-app
spec:
  selector:
    app: nbp-app
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
  type: LoadBalancer

```

Ostatnim krokiem jest wystawienie aplikacji na dostęp zewnętrzny. Można to zrobić na 2 sposoby. Z użyciem **NodePortu** oraz **LoadBalancer**a. W tym przypadku został wykorzystany drugi sposób, ponieważ przypisuje on każdemu **serwisowi** własny adres **IP**. **LoadBalancer**a używa się poprzez wpisanie na nowej konsoli komendy **minikube tunnel**. Utworzy się ścieżka między hostem, a serwisem CIDR klastra używając IP klastra jako bramę. Tunnel wystawia zewnętrzne IP bezpośrednio do jakiegokolwiek programu na systemie operacyjnym hosta.

Przykład odpowiedzi dla komendy **kubectl get svc** przed użyciem **tunnelu**.

```

nbp-app          LoadBalancer    10.106.154.68    <pending>
8080:32735/TCP   50s

```

Oraz po użyciu **tunnelu**.

```

nbp-app          LoadBalancer    10.106.154.68    127.0.0.1
8080:32735/TCP   10m

```

Obraz całości można zobaczyć na obrazku 3.2



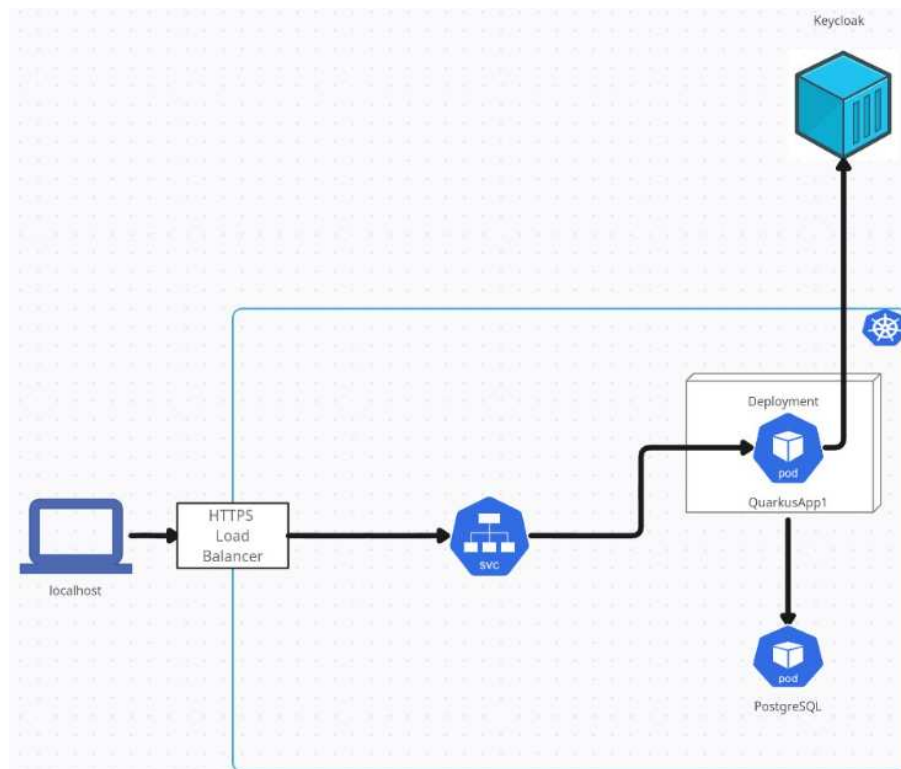


Figure 3.2: Architektura aplikacji w Kubernetesie

### 3.3 Nomad Hashicorp

Jest to mniej popularny orkiestrator od dwóch poprzednich. Został stworzony przez firmę HashiCorp. Charakteryzuje się on:

- **„Inteligentne zarządzanie zasobami”** - Nomad optymalizuje zasoby przypisane do klastra poprzez efektywne obciążenia między nodami w klastrze używając procesu zwanego **bin packing**,
- **„Samo regeneracja”** - stałe sprawdzanie i kontrola nad aktywnymi kontenerami i podjęcie odpowiednich działań w przypadku wystąpienia błędów,
- **„Ciągłe działanie aplikacji”** - wsparcie dla wielu strategii aktualizacji oprogramowania, np. textbfrolling updates, blue/green deployments, canary deployments,
- **„Różne typy obciążeń”** - elastyczność Nomada pochodzi z **task driverów** pozwalając na orkiestrację Dockera i innych kontenerów, jak i również dla wykonywalnych plików Jar, wirtualnych maszyn QEMU, czystych poleceń **exec driver**, oraz daje możliwość dla użytkowników stworzenia ich własnych driverów,
- **„Wsparcie dla wielu systemów operacyjnych”** - Nomad uruchamia się jako typ binarny i pozwala orkiestrować aplikacje jednocześnie na macOS, Windowsie i Linuxie,
- **„Zunifikowany przepływ pracy”** - bez względu na to czy zadaniem jest zarządzanie kontenerami, aplikacjami Java maszynami wirtualnymi to w Nomadzie sposób zarządzania nimi jest jednakowy,

- „**Deklaratywna specyfikacja zadania**” - użytkownik podaje co chce zrobić, a Nomad zajmuje się resztą.

Konfiguracja Nomada składa się z:

- „**agent**” - jest to proces Nomada działający na serwerze albo jako tryb klienta,
- „**client**” - jest odpowiedzialny za uruchomienie, które są do niego przypisane. Sprawdza czy jakieś zadania muszą być przydzielone. Kiedy agent jest uruchomiony, client staje się nodem,
- „**server**” - jego zadaniem jest zarządzanie wszystkimi jobsami i clientami. Serwer monitoruje stan zadań i sprawdza, czy działają prawidłowo. Decyduje również jakie zadania na jakich węzłach klienckich powinny zostać umieszczone, uwzględniając zasoby i wymagania tych zadań. Między serwerami następuje wymiana danych, dzięki czemu jeśli jeden z serwerów przestanie działać to inny będzie mógł go zastąpić. Powoduje to, że klaster jest cały czas dostępny,
- „**dev agent**” - jest to tryb, który przydaje się podczas tworzenia i testowania aplikacji, ponieważ można szybko i łatwo uruchamiać aplikacje bez skomplikowanej konfiguracji. Agent działa jednocześnie jako klient i serwer, co ułatwia testowanie. Agent nie zapisuje swojego stanu, więc po restarcie dane zostaną utracone. Co również może być zaletą, bo nie trzeba usuwać pozostałości i można wszystko od początku przetestować.

Najważniejsze operacje Nomada to:

- „**task**” - jest to podstawowym elementem w Nomadzie. Są wykonywane przez **task drivers** takie jak docker albo exec,
- „**group**” - jest to zbiór tasków, które są uruchamiane na tym samym clientcie,
- „**job**” - główna jednostka zarządzania w Nomadzie, za pomocą niej definiuje się i zarządza się aplikacjami. Zawiera wszystkie informacje potrzebne do ich uruchomienia. Job określa w jaki sposób aplikacja ma być uruchamiana. Job składa się z jednego lub więcej tasków,
- „**job specification**” - jest to plik w którym jest określony schemat działania dla jobów. Określa typ dla jobów, zadania i potrzebne zasoby sprzętowe oraz zawiera informacje dotyczące, który client ma uruchomić danego joba,
- „**allocation**” - określa specyfikacje dla jobów. Opisuje typ joba, jego zadania i zasoby potrzebne do uruchomienia.

Dokładny opis tych elementów Nomada można znaleźć w dokumentacji technicznej dostępnej pod adresem [6].

Uruchomienie klastra odbywa się za pomocą komendy:

```
sudo nomad agent -dev -bind 0.0.0.0 -network-interface='{{
  GetDefaultInterfaces | attr "name" }}' export NOMAD_ADDR=
http://localhost:4646
```

Następnie jest uruchamiany **consul ??** za pomocą komendy:

```
consul agent -dev -bind=0.0.0.0 -advertise=<
  your_advertise_address> -log-level=INFO
```

Gdzie **advertise** jest to adres IP komputera. Po tym można już zacząć tworzyć bazę danych oraz uruchomić aplikację. Jako adres po którym będzie można się komunikować z nimi będzie adres IP komputera. Najpierw kod uruchamiający postgresa za pomocą konydy **nomad job run -hcl1 nbpapppostgres.nomad.hcl:**

```
job "postgres" {
  datacenters = ["dc1"]
  type = "service"

  group "postgres" {
    count = 1

    task "postgres" {
      driver = "docker"
      config {
        image = "postgres"
        ports = ["db", "http"]
      }
      env {
        POSTGRES_USER="postgres"
        POSTGRES_PASSWORD="admin"
      }

      logs {
        max_files      = 5
        max_file_size = 15
      }

      resources {
        cpu = 1000
        memory = 1024
      }
      service {
        name = "postgres"
        tags = ["postgres for vault"]
        port = "db"

        check {
          name      = "alive"
          type      = "tcp"
          interval = "10s"
          timeout  = "2s"
        }
      }
    }
  }
}
```

```

restart {
  attempts = 10
  interval = "5m"
  delay = "25s"
  mode = "delay"
}
network {
  mbits = 10
  port "db" {
    static = 5432
  }
  port "http" {
    to = 8080
  }
}
}
update {
  max_parallel = 1
  min_healthy_time = "5s"
  healthy_deadline = "5m"
  auto_revert = false
  canary = 0
}
}

```

Następnie kod uruchamiający aplikację **nomad job run -hcl1 nbpAppQuarkus.nomad.hcl**:

```

job "nbpapp" {
  datacenters = ["dc1"]
  type        = "service"

  group "web" {
    count = 1

    network {
      port "quarkus"
      {
        to = 8080
      }
    }

    task "service" {
      driver = "docker"

      config {
        image           = "matiuw/nbp-appnomad:latest"
        network_mode    = "host"
        ports           = ["quarkus"]
      }
    }
  }
}

```

}  
}  
}

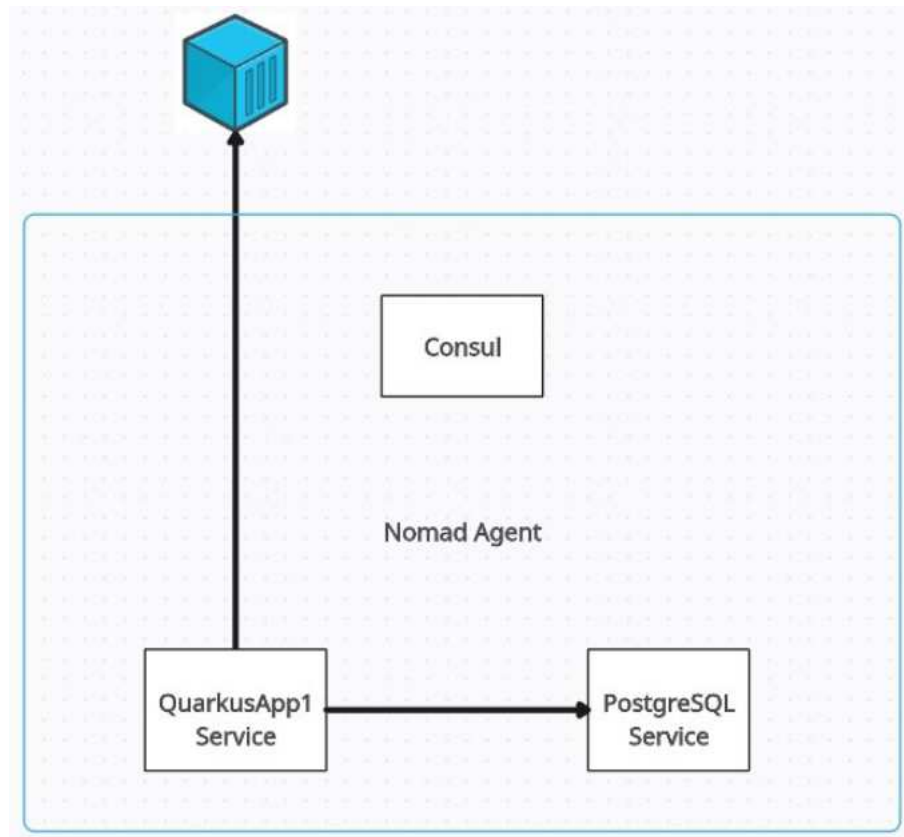


Figure 3.3: Architektura aplikacji w Nomadzie

Jest to narzędzie stworzone przez firmę HashiCorp. Link do dokumentacji dostępny jest na pozycji [7]. Jest to serwis



## Chapter 4

# Opis przeprowadzonych eksperymentów

W tym rozdziale został przedstawiony opis przeprowadzonych eksperymentów. Jakie narzędzia zostały użyte oraz opis przeprowadzonych czynności.

### 4.1 Platforma testowa

Stanowiskiem do testowania jest komputer stacjonarny o następujących parametrach.

- „**Procesor**” - AMD Ryzen 5 5600X,
- „**Pamięć ram**” - Crucial 32GB (2x16GB) 3600MHz CL16 Ballistix Black,
- „**Dysk SSD**” - GOODRAM 256GB 2,5" SATA SSD CX400,
- „**System operacyjny**” - Ubuntu 22.04.4 LTS,

W przypadku innych konfiguracji wyniki będą się różniły.

### 4.2 Eksperyment obciążenia procesora i użycia pamięci

W tym eksperymencie zostały przeprowadzone testy użycia najważniejszych parametrów w komputerze, czyli pamięci operacyjnej oraz procesora. Dane w przypadku Docker Swarma zostały pozyskane przy pomocy narzędzia Swarmprom. W przypadku **Kubernetesa** został użyty addon, który można zainstalować za pomocą komendy **minikube addons enable metrics-server**, a w przypadku **Nomada** nie trzeba było nic dodatkowo instalować.

Przyjętą jednostką użycia pamięci zostały MB. W **Kubernetes** i **Nomadzie** nie przedstawiały wyników w tej jednostce, więc zaszła potrzeba dopasowania tego. Tą jednostką są MiB, czyli mebibity. Jest to jednostka binarna. Przykładowo 1MiB posiada mnożnik  $2^{20} = 1024^2$ , gdzie 1MB ma mnożnik  $10^6$ . Zatem powinno się przeliczyć to za pomocą następującego wzoru:  $1MiB = 1024^2B = 1048576B = 1.048576MB$ .

W przypadku użycia procesora trudno porównać uzyskane wyniki z powodu różnych jednostek pomiarowych. **Swarmprom** użycie procesora przedstawia w procentach. Kubernetes z drugiej strony przedstawia wyniki w wirtualnych CPU, gdzie jedna jednostka CPU oznacza jeden rdzeń CPU. Jeśli się nie sprecyzuje ilość przeznaczonych zasobów to dla jednego **node** przypisane jest 1m co jest równe 0.001 CPU.

Dane w **Nomadzie** są podawane w Mhz co znacząco utrudnia porównanie wyników. Żeby zamienić wyniki z procent na Mhz trzeba pobrać wyniki aktualnego użycia rdzenia, co jest trudne do wykonania.

### 4.2.1 Swarmprom

Jest to zbiór narzędzi do monitorowania użytych zasobów przez serwisy w **Docker Swarmie**. W jego skład wchodzi:

- „**Prometheus**” - narzędzie **open source** służące do monitorowania, gromadzenia i powiadamiania o danych z klastra **Docker Swarm**,
- „**Grafana**” - służy do prezentowania danych otrzymanych z **Prometheusa** i na ich podstawie tworzenie wykresów i tabel,
- „**cAdvisor**” - jest to narzędzie **open source** stworzone przez **Google** do monitorowania kontenerów. Zbiera, segreguje i eksportuje dane odnośnie kontenerów takie jak CPU i użycie pamięci, system plików i statystyki sieci,
- „**alertmanager**” - służy do odbierania i przetwarzania alertów z takich narzędzi jak **Prometheus**. Odpowiednio przetwarza te dane usuwając duplikaty, grupuje dane i przekazuje bezpośrednio do miejsca docelowego,
- „**unsee**” - narzędzie do wizualizacji i zarządzania alertami w systemach monitorowania,
- „**node-exporter**” - służy do zbierania metryk systemu i przekazywania ich w formie w którym może być zrozumiały dla **Prometheusa**.

Indywidualne instalacje poszczególnych elementów zostały opisane w rozdziale 10 w książce na pozycji [4].

## 4.3 Eksperyment startu aplikacji

Czas budowania jest istotnym elementem porównawczym między orkiestratorami. Podczas pracy wiele razy tworzy się i usuwa serwisy.

Celem testu jest sprawdzenie czasu jaki potrzebuje dany orkiestrator od momentu rozpoczęcia budowania aplikacji, aż do uzyskania prawidłowej odpowiedzi z zapytania HTTP. Kod testujący czas startu serwisu w Docker Swarmie:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```



```

#include <unistd.h>
#include <stdint.h>
#include <sys/time.h>
#define BUFFER\_SIZE 128\\newline

int containsExchangeValue(const char *buffer) \\{
    return strstr(buffer, "3.95") != NULL;
}

long long timeInMilliseconds(void) \\{
    struct timeval tv;

    gettimeofday(&tv, NULL);
    return (((long long)tv.tv_sec)*1000)+(tv.tv_usec/1000);
}

void writeToFile(double execution_time) {
    FILE *file = fopen("wyniki.txt", "a");
    if (file == NULL) {
        printf("Błąd podczas otwierania pliku.\\n");
        return;
    }

    fprintf(file, "%.4f\\n", execution_time);
    fclose(file);
}

void startProgram(char *buffer, FILE *curl_output) {

    long long start_time, end_time;
    double execution_time;

    start_time = timeInMilliseconds();

    FILE *fp = popen("docker service create --name NBP-APP --p 8080:8080 --network=NBP-APP matiuw/nbp-app:latest",
        "r");
    pclose(fp);

    do {
        curl_output = popen("curl -X GET -H \"Content-Type: application/json\" -d '{\\\"amount\\\": 1, \\\"myCurrency\\\": \\\"USD\\\", \\\"targetCurrency\\\": \\\"PLN\\\"}' http://localhost:8080/currency/currencyExchangeValue", "r");

        fgets(buffer, BUFFER_SIZE, curl_output);
    } while (1);
}

```

```

    } while (!containsExchangeValue(buffer));

    pclose(curl_output);

    memset(buffer, 0, sizeof(buffer1));

    end_time = timeInMilliseconds();
    execution_time = (double)(end_time - start_time) / 1000;
    printf("Czas wykonania: %.4f sekund.\n", execution_time)
        ;

    writeToFile(execution_time);

    fp = popen("docker service rm NBP-APP", "r");
    pclose(fp);
}

int main() {

    char buffer[BUFFER_SIZE];
    FILE *curl_output;

    for(int i = 0; i < 1; i++) {
        startProgram(buffer, curl_output);
        sleep(15);
    }

    return 0;
}

```

Powyższy kod składa się z następujących kroków. W funkcji **main** jest tworzony **buffer** oraz pętla, która odpowiada za ilość przeprowadzonych testów. Między testami jest przerwa wykonana za pomocą funkcji **sleep(x)**. Na początku funkcji **startProgram()** następuje pobranie czasu początkowego oraz wykonanie komendy utworzenia serwisu w **Docker Swarmie** (w przypadku Kubernetesa jest tworzony deployment, a w przypadku Nomada jest tworzony nowy **job**). W pętli do while wykonywane jest polecenia GET i wynik zapisywany jest do buffora za pomocą funkcji **fgets**. Pętla się kończy w przypadku pobrania prawidłowej odpowiedzi, która jest sprawdzana w funkcji **containsExchangeValue()**. Po zakończeniu pętli czyszczony jest **buffer** oraz ponownie pobierany jest czas zakończenia działania. Za pomocą tych dwóch czasów obliczana jest różnica, która jest wynikiem działania programu i jest on zapisywany do pliku. Na koniec usuwany jest serwis, żeby móc wszystkie kroki powtórzyć od początku. Średni czas uruchomienia na podstawie 100 takich uruchomień można znaleźć w sekcji 5.2.

## 4.4 Eksperyment stabilności działania

Eksperyment stabilności działania został przeprowadzony przy użyciu narzędzia **Gatling**. Służy do sprawdzenia czy w aplikacji pojawiają się błędy w przypadku dużej liczby użytkowników próbujących się połączyć w tym samym czasie. Dzięki, temu część błędów można jeszcze wykryć przed wypuszczeniem kodu na produkcję.

Jest to narzędzie napisane w **Scala** 2.12.10 i potrzebuje do działania również **Jawę** 1.8, gdyż jest to język, który korzysta z maszyny wirtualnej **Jawy** jako platformy uruchomieniowej.

Utworzony kod programu:

```
class MySimulation extends Simulation {

  //protocol
  val httpProtocol = http
    .baseUrl("http://localhost:8080")

  //scenario
  val scn1 = scenario("Przelicz kurs").exec(
    http("Przelicz kurs req")
      .get("/currency/getCurrency/table/A/value/1")
      .header("content-type", "application/json")
      .asJson
  //      .body(StringBody(
  //        """
  //          |{
  //          |    "amount": 1,
  //          |    "myCurrency": "USD",
  //          |    "targetCurrency": "PLN"
  //          |}
  //          |""".stripMargin)).asJson
    .check(
      status is 200,
      jsonPath("$.code") is "BGN"
    )
  )

  //setup
  setUp(scn1.inject(rampUsers(6000).during(30))
    ).protocols(httpProtocol)
}
```

W powyższym kodzie została utworzona klasa rozszerzająca klasę **Simulation**. Kod programu składa się z 3 części. Pierwsza z nich określa główną część adresu **http**. W drugiej zdefiniowana jest metoda **http** oraz argumenty **linku**, ciało metody oraz przewidywany rezultat polecenia. W ostatniej zostały zdefiniowane informacje na temat tego ile ma się wykonać zapytań i w jakim czasie.

Po wykonaniu testów generowany jest raport w postaci **strony www**.

# Chapter 5

## Analiza

### 5.1 Wyniki i wnioski obciążenia procesora i użycia pamięci

Wyniki dla użycia procesora zostały przedstawione na rysunku 5.1. Jednostką wspólną zostały MHz. W przypadku Kubernetesa i Nomada wyniki 5 instancji są wielokrotnością wyniku dla 1 instancji, a w przypadku Docker Swarma dla jednej instancji wynik wyszedł 3,22 MHz, a dla 5 instancji 13,8 MHz. Wyniki zostały pobrane, gdy orkiestratory były bez obciążenia.

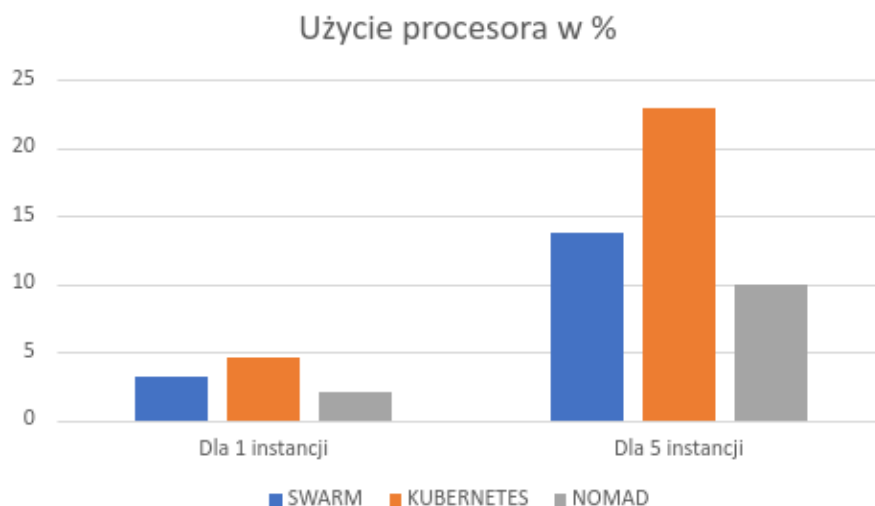


Figure 5.1: Wyniki użycia procesora dla aplikacji w orkiestratorach

Porównanie użycia pamięci i obciążenia procesora między orkiestratorami jest trudne, ponieważ wszystkie mają mechanizmy zarządzania zasobami. W ten sposób ilość przydzielonych zasobów może się różnić w czasie. Przy uruchomieniu aplikacji orkiestrator przydziela pewną ilość tych zasobów i po wykonaniu zapytania do aplikacji ta ilość może się zwiększyć dynamicznie.

Kontrola nad tym jest bardzo ważna. W systemach na Linuxie, jeśli jądro systemu wykryje, że nie ma wystarczająco pamięci, aby uruchomić ważne funkcje

systemu, to wyrzuci błąd **Out of Memory Exception** i zacznie zabijać mniej ważne procesy, aby zwolnić tę pamięć. Również aplikacje uruchomienie w Dockerze, a w ostateczność może przez przypadek usunąć ważny proces i wyłączyć system.

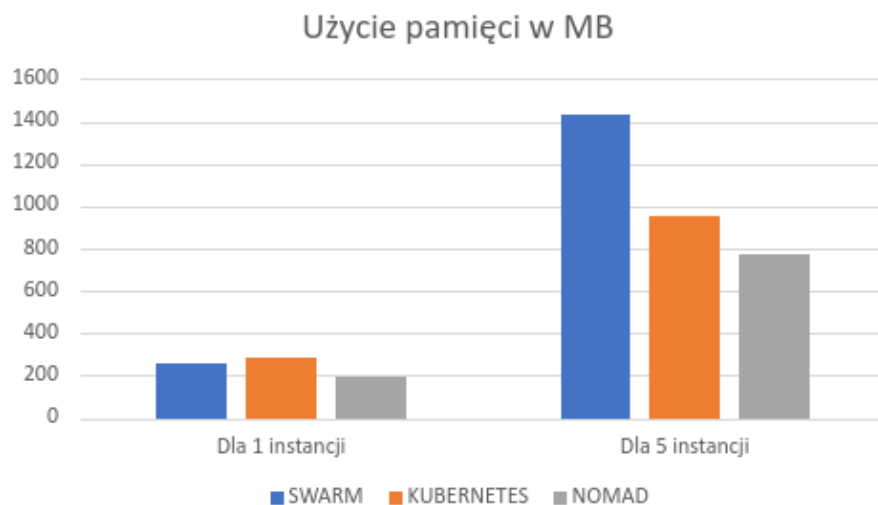


Figure 5.2: Wyniki użycia pamięci dla aplikacji w orkiestratorach

Otrzymane wyniki użycia pamięci zostały ukazane na rysunku 5.2. Użycie pamięci w przypadku jednej repliki aplikacji nieznacznie się różnią. Najlepszy wynik uzyskał Nomad, a najgorszy Kubernetes. Różnice się uwidaczniają w przypadku stworzenia nowych replik, gdzie najgorzej poradził sobie Docker Swarm, a wyniki dla Kubernetesa i Nomada nieznacznie się poprawiły w stosunku do wyników dla jednej repliki. Gdyby wymnożyć wynik dla 1 repliki razy 5 to wynik wyszedłby większy niż dla uruchomienia jednocześnie tych replik. Spowodowane jest to odpowiednim zarządzaniem zasobami. W dalszej części tego rozdziału zostały opisane techniki jakie stosuje każdy z orkiestratorów do rozdzielania zasobów i ich efektywnego wykorzystania.

**Quality of Service classes** - jest używane w przypadku **Kubernetesa** do klasyfikowania podów, które są uruchomione i alokowanie każdego poda w klasę QoS. Powoduje to, że pody są różnie obsługiwane w zależności od tej klasyfikacji. Wynik tej klasyfikacji jest oparty na requestach kontenerów oraz czy został przekroczony limit zasobów. Klasy QoS są używane przez Kubernetesa do decydowania, które pody należy usunąć z węzła w którym występuje **Node Pressure**. Jest to proces w którym Kubelet usuwa pody, aby zwolnić używane przez nie zasoby.

W Kubernetesie są trzy rodzaje QoS. Kiedy Node będzie używał zasoby poza limitem w pierwszej kolejności uruchomi się **BestEffort**, następnie **Burstrable** oraz **Guaranteed**.

„**Guaranteed**” - pody w tym trybie mają z góry ustalony limit zasobów i są najmniej narażone na usunięcie. Mają gwarancję ich nie usuwania, aż do momentu przekroczenia limitów lub w przypadku, gdy nie będzie już podów o niższym priorytecie, które mogą zostać usunięte. Nie mogą uzyskać zasobów poza ustalonym w ich specyfikacji.

Kryteria jakie muszą spełniać pody, aby zaliczały się do tej kategorii:

- zapytanie CPU musi być równe limitowi CPU każdego kontenera,

- każdy kontener musi mieć limit CPU i żądania CPU,
- zapytanie pamięci musi być równe limitowi pamięci każdego kontenera,
- każdy kontener musi mieć limit pamięci i żądania pamięci.

Przykład:

```
resources :
  requests :
    memory: "128Mi"
    cpu: "250m"
  limits :
    memory: "128Mi"
    cpu: "250m"
```

„**Burstable**” - pody w tym trybie mają ustalone minimalne gwarantowane zasoby bazujące na requestach, ale nie mają ustalonego górnego limitu. Zatem, jeśli limit nie jest sprecyzowany to domyślnie zasobami noda, więc odpowiedzialność za przydzielanie zasobów i zabierania ich spoczywa na nodzie i jego limicie. W przypadku, gdy node będzie chciał usunąć podę to będzie to mógł zrobić dopiero po wykonaniu **BestEffort**, ponieważ dany pod, który nie ma limitu może zabrać tyle zasobów ile będzie chciał.

Kryteria jakie muszą spełniać pody, aby zaliczały się do tej kategorii:

- pod nie spełnia kryteriów, które charakteryzują klasę QoS **Guaranteed**,
- przynajmniej jeden kontener w podzie musi mieć limit pamięci albo CPU lub memory request albo cpu request,

Przykład:

```
resources :
  requests :
    memory: 100Mi
    cpu: 200m
```

„**BestEffort**” - pody w tej klasie mają dostęp do zasobów noda, które nie są przydzielone i używane przez inne pody. Jeśli wziąć na przykład procesor ryzen 5 5600x, który ma 12 wątków i 4 wątki są już przydzielone do podów. To pod w tej klasie może użyć pozostałych wolnych zasobów w nodzie. Pody w tej kategorii są najbardziej narażone na usunięcie w przypadku braku zasobów.

Pody w tej kategorii nie spełniają kryteriów tak jak w dwóch poprzednich kategoriach. Jeśli żaden kontener w podzie nie ma limitu pamięci lub limitu CPU, oraz nie ma zdefiniowanej pamięci oraz CPU na request to klasyfikuje się jako BestEffort.

Dodatkowo kryteria, które zawsze muszą być zachowane bez względu na posiadaną klasę:

- jak kontener przywłaszczy sobie zasobów poza ustalony limit to zostanie usunięty i zrestartowany przez kubelet bez wpływu na inne kontenery,

- jeśli kontener przekroczy dane requesta i node na którym działa ten kontener ma problemy z zasobami to taki kontener jest kandydatem do usunięcia. Jeśli taka sytuacja się wydarze to wszystkie kontenery w podzie będą zatrzymane i zostaną stworzone zastępcze pody w innym nodzie,
- zasoby requesta w podzie są równe sumie zasobów requestów kontnerów w tym podzie. Co za tym idzie limit zasobów poda jest równy sumie zasobów tworzących go kontenerów,
- kube-scheduler nie rozważa klas QoS podczas wyboru poda do usunięcia.

Nomad nie posiada tak zaawansowanego systemu zarządzania zasobami, lecz także posiada kilka następujących procesów. Przy stwrzeniu nowego joba i przy jego zmianie lub błędzie Noda tworzony jest proces zwany **evaluation**, który jest przetwarzany przez evaluation brokera. Służy on do zarządzania kolejką z oczekujących **evaluation**. Odpowiednio zarządzają ich kolejnością i zapewnia, że wykonywany jest tylko jeden w tym samym czasie. Następnie **scheduling workers** uruchamiają odpowiedni scheduler na podstawie joba. Zadaniem **schedulera** jest wygenerowanie planu alokacji i analiza aktualnego stanu zasobów oraz potrzeby nodów. Za pomocą algorytmu takiego jak **bin packing** oraz reguł **anti-affinity** do optymalizacji zasobów. Ostatnim etapem zajmuje się **plan queue**, który podejmuje ostateczną decyzję.

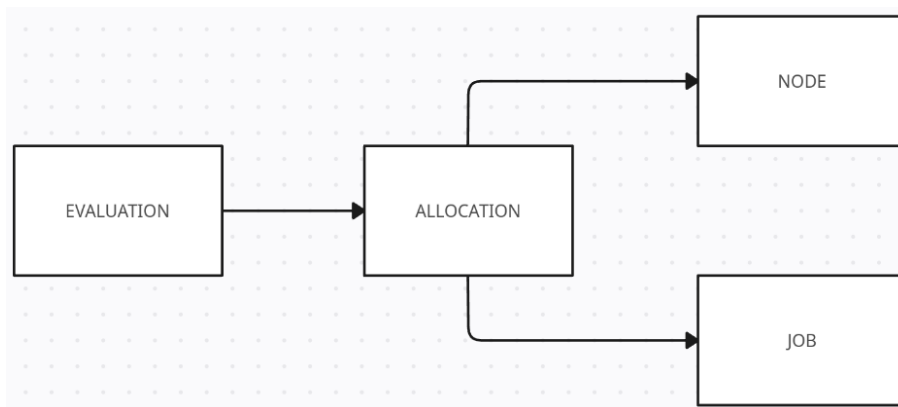


Figure 5.3: Alokacja zasobów Nomad

W nomadzie również można ustalić liczbę przedzielonych zasobów dla joba oraz ich maxymalna ilość.

```

resources {
  cpu      = 100
  memory   = 256
  memory_max = 512
}
  
```

Docker Swarm nie posiada algorytmu dla zarządzania zasobami serwisów, ale możliwe jest przypisanie wartości maksymalnych i minialnych. Co nie jest wystarczające, aby uzyskać takie wyniki jakie posiadają wcześniej opisane orkiestratory.



```
resources :  
  limits :  
    cpus : '0.50'  
    memory : '512M'  
  reservations :  
    cpus : '0.25'  
    memory : '256M'
```

## 5.2 Wyniki i wnioski badania startu aplikacji

Wyniki dla czasu startu aplikacji w każdym z orkiestratorów można zobaczyć na rysunku 5.4. W przypadku jednej repliki wyniki dla Docker Swarma i Kubernetesa wyszły bardzo podobne (4,77s i 4,87s), a w przypadku Nomada wynik wyszedł trochę lepszy (3,59s). Sytuacja się odwróciła w przypadku czasu uruchomienia 5 aplikacji jednocześnie. Największy wzrost czasu przypadł Docker Swarmowi, bo aż 10,97s. Spowodowane jest to brakiem zaawansowanych mechanizmów skalowania oraz słabsza optymalizacją zasobów. Najlepszy wynik uzyskał Kubernetes (5,154). Co jest wynikiem niewiele większym niż w przypadku jednej repliki. Możliwe jest to dzięki efektywnemu mechanizmowi zarządzania zasobami **QoS** opisanym w sekcji 5.1. Jak i również dzięki mechanizmowi wspomagającego tworzenie wiele aplikacji. Kubernetes posiada dwa takie mechanizmy, które nazywają się **anti-affinity** i **pod affinity**, które są zbiorem reguł pomagających określić, gdzie dany pod ma zostać umieszczony. Jest to coś podobnego do parametrów **nodeSelector**, ale oferuje większą elastyczność i funkcjonalność. Nomad uzyskał wynik 5,31s. Jest to nieznacznie gorszy wynik niż w przypadku Kubernetesa. Taki wynik mógł uzyskać dzięki wykorzystaniu mechanizmów zarządzania zasobami takimi jak: **bin packing** oraz **memory max**.

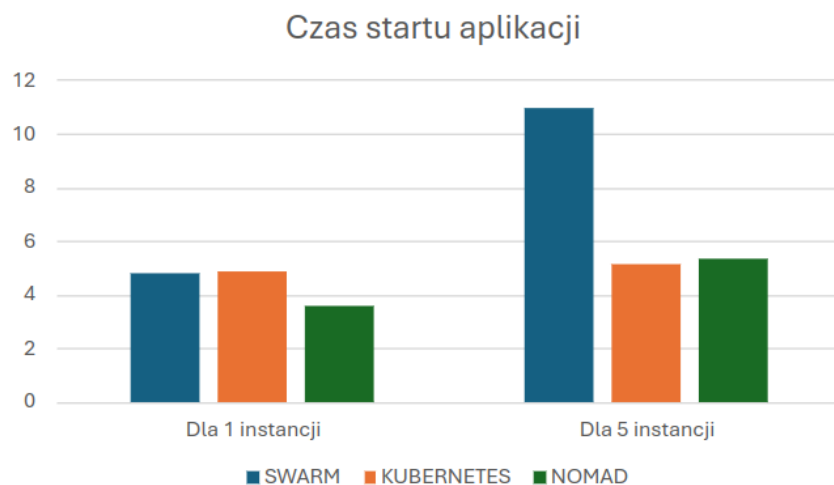


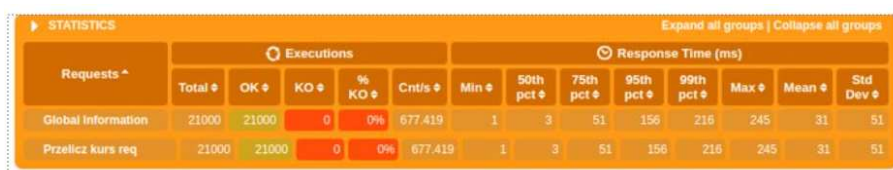
Figure 5.4: Czas startu aplikacji

## 5.3 Wyniki i wnioski stabilności działania

Test został przeprowadzony lokalnie. Wykorzystując **Keycloak** wewnątrz **Dockera** przez co można było pominąć sytuacje chwilowego obciążenia tego narzędzia. Użytkownicy wysyłali zapytania do aplikacji, która potem pobierała dane z lokalnie zainstalowanej bazy danych i zwracała te dane. Po czym powinno się otrzymać prawidłowe dane oraz status.

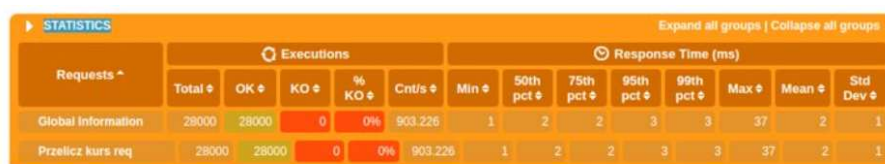
Test polegał na wysłaniu określonej liczby zapytań w określonym czasie. W tym przypadku zostało wysłanych tyle zapytań, aż do momentu pojawienia się błędów.

Najgorszy wynik uzyskał **Docker Swarm**, gdyż uzyskał jedynie 6000 pozytywnych **requestów** w czasie 30 sekund. Kubernetes uzyskał 21000 pozytywnych **requestów**, a najlepszy wynik uzyskał **Nomad**, bo aż 28000.



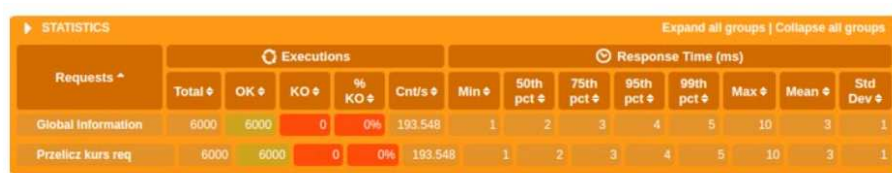
Requests ^	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Crit/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	21000	21000	0	0%	677.419	1	3	51	156	216	245	31	51
Przelicz kurs req	21000	21000	0	0%	677.419	1	3	51	156	216	245	31	51

Figure 5.5: Wynik działania narzędzia Gatling dla Kubernetesa



Requests ^	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Crit/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	28000	28000	0	0%	903.226	1	2	2	3	3	37	2	1
Przelicz kurs req	28000	28000	0	0%	903.226	1	2	2	3	3	37	2	1

Figure 5.6: Wynik działania narzędzia Gatling dla Nomada



Requests ^	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Crit/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	6000	6000	0	0%	193.548	1	2	3	4	5	10	3	1
Przelicz kurs req	6000	6000	0	0%	193.548	1	2	3	4	5	10	3	1

Figure 5.7: Wynik działania narzędzia Gatling dla Docker Swarma

Obserwując użyte zasoby, można wywnioskować, że ilość udanych wysłanych zapytań http ma swój limit w momencie przekroczenia przypisanej ilości pamięci ram. Na załączonym obrazku 5.8 można zaobserwować skoki użycia procesora ponad limit 100MHz, lecz ilość użytej pamięci ram nie przekracza ustawionej granicy 300MiB. W momencie przekroczenia tej wartości job jest restartowany i kolejne requesty są bez odpowiedzi. Alokowanie pamięci nie jest dokładnie, ponieważ w przypadku aktualnego użycia pamięci na poziomie 150MiB/300MiB job pobierał ponad przy teście, a gdy aktualnie miał w użyciu ponad 250MiB (co mu wystarczało do obsłużenia zapytań) test przebiegał pomyślnie.

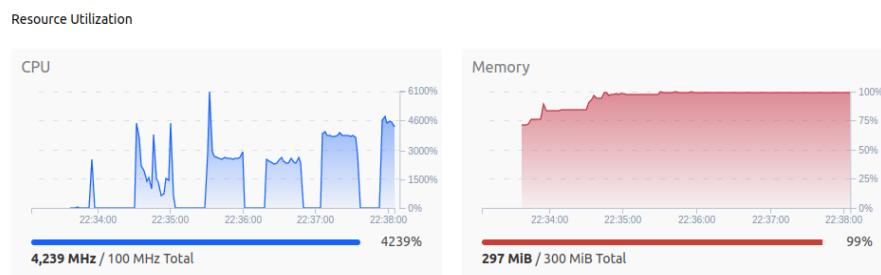


Figure 5.8: Użyte zasoby podczas testu Gatling



# Podsumowanie

Motywacją podjęcia tematu analizy orkiestratorów było poznanie tych narzędzi z powodu coraz bardziej zwiększającego się zainteresowania tymi narzędziami w środowiskach komercyjnych. Są to narzędzia wspomagające pracę nad dużymi projektami, które znacznie ułatwiają organizację pracy nad nimi. Również programiści zaczęli przechodzić z tworzenia dużych monolitycznych aplikacji na aplikacje mikroserwisowe. Spowodowało to potrzebę wprowadzenia nowych rozwiązań pod ten typ. Szczegółowe różnice zostały opisane we wstępie w tabeli 1. W indywidualnych zadaniach programistycznych użycie ich mija się z celem, gdyż nie osiągają one takiej wielkości. Czas poświęcony na konfigurację tych narzędzi będzie dużo większy, a korzyści z tego będą znikome. Podział aplikacji daje wiele korzyści w przypadku podziału zadań w zespole i łatwości w ustaleniu przyczyny błędów z powodu tworzenia aplikacji, gdzie jeden błąd nie ma tak dużego wpływu na resztę. Orkiestratory również wspomagają zarządzanie tymi aplikacjami poprzez mechanizmy takie jak automatyczna regeneracja. W obecnych czasach poznanie tych narzędzi jest argumentem do wejścia do nowoczesnych i dużych projektów.

Celem pracy było zapoznanie się z narzędziami do orkiestracji kontenerów Docker. Poprzez przejście literatury oraz dostępnych materiałów online. W pierwszej kolejności trzeba było sobie przypomnieć sposób działania Dockera i jego możliwości oraz komendy. Kolejnym krokiem było stworzenia aplikacji do testowania orkiestratorów oraz skonteneryzowania jej. Utworzona aplikacja opisana w rozdziale 2 ma za zadanie pobranie danych z <https://api.nbp.pl/>. Zapisać te dane do bazy danych i na ich podstawie wykonać operacje CRUD. Dodatkową możliwością jest kalkulator do przeliczenia jednej waluty na drugą przy pomocy kursów walut. Aplikacja została wykonana przy użyciu frameworka Quarkus. Pozwoliło to stworzyć aplikację już od samego początku przystosowaną do działania w kontenerze. Utworzony szablon zawierał instrukcję i plik za pomocą, którego został stworzony obraz. Aplikacja dodatkowo zawierała uwierztelnianie użytkownika za pomocą Keycloak. Tak utworzona aplikację następnie została uruchomiona na każdym z orkiestratorów. Sposób uruchomienia został opisany w rozdziale 3. Architektury w poszczególnych orkiestratorach zostały własnoręcznie opracowane i wdrożone lokalnie. Ostatnim etapem było przygotowanie testów do porównania orkiestratorów. Do przeprowadzenia testu sprawdzającego użyte zasoby zostały użyte narzędzia wbudowane oraz dockerswarm w przypadku Docker swarma. Test sprawdzający szybkość uruchomienia aplikacja, aż do momentu uzyskania prawidłowej odpowiedzi został napisany własnoręcznie w języku c. Ostatnim testem był test obciążenia wykonany przy użyciu narzędzia Gatling. Został napisany skrypt, którego zadaniem było wysyłanie określonej ilości zapytań w określonym czasie do aplikacji.

Osiągnięte wyniki częściowo pokrywały się z przewidywanymi jeszcze przed rozpoczęciem testowania. Najgorsze wyniki w przypadku jednej instancji uzyskał Kubernetes. Spowodowane jest to przez samo rozbudowanie orkiestratora. Posiada on wiele funkcji, które są zoptymalizowane pod działanie przy wielu replikach i kontenerach. Dodatkowo wymaga zainstalowania i uruchomienia klastra w kontenerze, co powoduje dodatkowy narzut na wydajność. Drugie w kolejności wyniki dla jednej repliki uzyskał Docker Swarm z powodu tego, że jest to narzędzie wbudowane do Dockera. Ma również najprostsza budowę i małą ilość funkcji. Najlepsze wyniki osiągnął Nomad dla jednej repliki. W przypadku testów dla wielu replik najlepsze wyniki uzyskał Kubernetes. Można było zauważyć działające mechanizmy, które zoptymalizowały wykonywane operacje. Za pomocą mechanizmów takim jak QoS, którego celem jest inteligentne zarządzanie zasobami. Pozwoliło mu to w teście na użycie zasobów uzyskać wynik, który nie jest wielokrotnością wyniku dla jednej instancji. W kolejnym teście na czas startu instancji uzyskał on wynik dla 5 instancji prawie identyczny jak w przypadku jednej instancji anty-affinity i affinity pozwalającymi określić miejsce umieszczenia podów w nodzie. Najgorzej skalował się Docker Swarm, gdyż nie posiada w sobie zaimplementowanych zaawansowanych mechanizmów kontroli nad zasobami. Istnieje tylko możliwość podania na sztywno tych danych. Nomad wypadł gorzej od Kubernetesa, ale lepiej od Docker Swarma. Można było zauważyć pewne mechanizmy jak bin-packing, które wpłynęły na ilość użytych zasobów przy zwiększeniu liczby instancji.

Docker Swarm mimo, że nie posiada zaawansowanych mechanizmów to idealnie sprawdzi się w mniej skomplikowanych wdrożeniach i w zespołach mniej doświadczonych osób, gdyż jest prosty w obsłudze oraz jest zintegrowany z Dockerem. Jest to również narzędzie bezpłatne, więc można od niego rozpocząć zaznajamianie się z tematyką orkiestracji.

Zaawansowane funkcje Kubernetesa idealnie sprawdzą się przy dużych i złożonych projektach, które wymagają jak największej wydajności i kontroli oraz wysokiej skalowalności. Co pokazują uzyskane wyniki testów. Wykorzystany w pracy klastery Kubernetesa Minikube jest rozwiązaniem, które powinno się używać wyłącznie do testów lokalnie. Nie posiada on wszystkich funkcji dostępnych w Kubernetesie. Korzystanie z Kubernetesa w chmurze obciążone jest kosztami za zasoby obliczeniowe, pamięć, sieć oraz dodatkowe usługi jak w AWS albo Azure.

Nomad może mieć zastosowanie tam, gdzie wymagana jest lekka architektura i efektywne zarządzanie zasobami. Warto go rozważyć w organizacjach, które mają ograniczony budżet i zasoby. Nomad podobnie jak Kubernetes ma opcję używania go bezpłatnie oraz możliwość płatnego dostępu w celu używania go w chmurze. Jest także możliwość wykupienia wersji Enterprise oprogramowania, która zawiera funkcje niedostępne w zwykłej wersji.

Ostatecznie udało się stworzyć aplikację i ją skonteneryzować. Skonfigurować orkiestratory i wdrożyć aplikację do każdego z nich. Aplikacja działała tak jak było to oczekiwane. Wdrożenie aplikacji na orkiestratory było takie samo. Z tą różnicą, że każdy z nich składa się z innych elementów. Wyniki dla obciążenia procesora nie pokazywały różnic z powodu, że różne narzędzia wskazywały różne jednostki. Przez co nie można było wyciągnąć żadnych wniosków. Wyniki dla pozostałych testów wyszły poprawne. Można było wyciągnąć z nich wnioski i zauważyć różnice

w zastosowanych rozwiązaniach i użytych mechanizmów.

W dalszej perspektywie można rozważyć użycie jednego narzędzia do sprawdzenia obciążenia podzespołów. Przykładem takiego narzędzia może być Prometheus w połączeniu z Graphaną.

Można również bardziej rozbudować architekturę aplikacji. Dodając do niej kolejne serwisy, np. część kliencką aplikacji. Pozwalającą użytkownikowi używać aplikacji przy pomocy interfejsu graficznego w przeglądarce. W ten sposób można stworzyć architekturę zbliżoną do tych, które tworzy się w komercyjnych projektach.

Kolejną opcją może być analiza kosztów w przypadku używania orkiestratorów w chmurze.





# Bibliography

- [1] Cay S. Horstmann. Java. In *Podstawy. Wydanie XI*, page 768, 2020.
- [2] Alex Soto Bueno and Jason Porter. Quarkus cookbook. In *Quarkus Cookbook Kubernetes-Optimized Java Solutions*, page 394, 2020.
- [3] Jason Van Zyl's. Maven. In *Maven The Definitive Guide*, page 470, 2009.
- [4] Viktor Farcic. The devops 2.1 toolkit: Docker swarm. In *The DevOps 2.1 Toolkit*, page 414, 2017.
- [5] Nigel Poulton. The kubernetes book. In *The Kubernetes Book*, page 235, 2021.
- [6] HashiCorp Developer. [https://developer.hashicorp.com/nomad/tutorials/get-started/gs-overview?product\\_intent=nomad](https://developer.hashicorp.com/nomad/tutorials/get-started/gs-overview?product_intent=nomad). [dostęp: 2013-03-08].
- [7] HashiCorp Developer. <https://developer.hashicorp.com/consul/docs/intro>. [dostęp: 2013-03-08].

# List of Figures

1.1	konteneryzacją i wirtualizacją . . . . .	12
1.2	Architektura Dockera i Podmana . . . . .	12
2.1	ERD bazy danych . . . . .	22
3.1	Architektura aplikacji w Docker Swarmie . . . . .	28
3.2	Architektura aplikacji w Kubernetesie . . . . .	33
3.3	Architektura aplikacji w Nomadzie . . . . .	37
5.1	Wyniki użycia procesora dla aplikacji w orkiestratorach . . . . .	45
5.2	Wyniki użycia pamięci dla aplikacji w orkiestratorach . . . . .	46
5.3	Alokacja zasobów Nomad . . . . .	48
5.4	Czas startu aplikacji . . . . .	49
5.5	Wynik działania narzędzia Gatling dla Kubernetesa . . . . .	50
5.6	Wynik działania narzędzia Gatling dla Nomada . . . . .	50
5.7	Wynik działania narzędzia Gatling dla Docker Swarma . . . . .	50
5.8	Użyte zasoby podczas testu Gatling . . . . .	51

# List of Tables

1	Wady i zalety aplikacji Monolitycznych i mikroserwisowych . . . . .	6
1.1	Różnice między wirtualizacją, a konteneryzacją . . . . .	14