

Politechnika Rzeszowska
Wydział Elektrotechniki i Informatyki

Katedra Elektrotechniki i Podstaw Informatyki

Usługi sieciowe w biznesie

Projekt: Wdrożenie aplikacji przy użyciu Kubernetes

Wykonał: Wrona Michał

IV EF-ZI

Rzeszów 2024

1. Wstęp

Tematem mojego projektu było: “Stworzenie deploymentu oraz wdrożenie aplikacji przy użyciu Gitlab CI oraz Kubernetes (k3s). Dodatkowe technologie Nginx, Varnish Cache, PostgreSQL”. Na wstępie powiem, że nie wszystko co zostało zaplanowane zostało zrealizowane (ze względów technicznych). Technologie k3s¹ (kubernetes) zastąpiłem minikube². Nie udało się uruchomić kubernetes’a na moim prywatnym serwerze ze względu na wygórowane wymagania. Projekt zrealizowałem u siebie lokalnie.

Na potrzeby mojej pracy inżynierskiej i zainteresowaniem się tą nową technologią wdrożyłem aplikację webową przy użyciu tych technologii:

- Django³
- Docker⁴
- Kubernetes⁵
- Varnish Cache⁶
- GitHub⁷
- PostgreSQL⁸
- Redis⁹
- Python¹⁰
- Linux (Ubuntu)

Aplikację, którą wdrazam na środowisko kubernetes to mój projekt inżynierski “Tablica informacyjna dla osób głuchoniemych”. Aplikacja składa się dwóch instancji frontu oraz cms. Ma ona wspomóc zrozumienie komunikatów na przykład na dworcach oraz w autobusach miejskich dla

¹ k3s (lekki kubernetes) - <https://k3s.io/>

² minikube (dystrybucja lekkiego kubernetesa) - <https://minikube.sigs.k8s.io/>

³ framework do budowania aplikacji webowych w języku Python - <https://www.djangoproject.com/>

⁴ oprogramowania służące do konteneryzacji - <https://www.docker.com/>

⁵ narzędzie służące do orkiestracji kontenerów - <https://kubernetes.io/>

⁶ oprogramowanie typu proxy służące głównie do przyspieszania stron internetowych o dużym natężeniu sieciowym - <https://varnish-cache.org/>

⁷ platforma służąca do przechowywania kodu źródłowego - <https://github.com/>

⁸ relacyjna baza danych - <https://www.postgresql.org/>

⁹ otwartoźródłowe oprogramowanie działające jako nierelacyjna baza danych przechowująca dane w strukturze klucz-wartość w pamięci operacyjnej serwera - <https://redis.io/>

¹⁰ język programowania - <https://www.python.org/>

osób nie słyszących (nie wszyscy niesłyszący umieją czytać). Na dwóch różnych domenach będzie wyświetlana inna treść, domena frontowa (dla użytkownika końcowego) posiada sklejoną film, na którym przedstawiona jest osoba pokazująca jakiś komunikat w języku migowym. Druga domena cms posiada panel administratora, w którym to administrator może ustawić komunikat, opublikować go na zadaną godzinę i dowolnie ustawić położenie na stronie takiego filmu.

2. Opis wykorzystywanych technologii

a. Kubernetes

Kubernetes jest to orkiestrator kontenerów, posiada on możliwość wdrażania w pełni skalowanych, bezpiecznych aplikacji stanowych i bez stanowych opartych na przykład na protokole HTTP. Dzięki niemu możemy uzyskać także wdrażanie bez przerwy nowych wersji aplikacji wykorzystując takie metody jak Canary Release czy Blue-green Deployment.

Canary Release jest to wzorec wdrażania bez przerwy, który polega na tym, że podczas wdrażania nowej wersji aplikacji na zadany czas na przykład 10% ruchu sieciowego jest przełączane na nową wersję aplikacji w celu sprawdzenia oraz przetestowania zmian. Jeśli wszystkie nowe funkcjonalności działają całość ruchu sieciowego jest przekazywana na nowe pody ¹¹ z nową wersją aplikacji.

Blue-green Deployment jest to metoda, w której są tworzone nowe pody z nową wersją aplikacji, developerzy mogą wykonać na nowej wersji aplikacji na przykład migracje bazodanowe, skrypty i tak dalej i dopiero wtedy cały ruch jest przepinany na nowe pody aplikacyjne.

b. Django

¹¹ to grupa jednego lub wielu kontenerów aplikacji zawierających współdzieloną przestrzeń, adres IP i informacje, jak mają być uruchamiane.

Django jest to framework napisany w języku Python, który pozwala budować szybko i bezpiecznie aplikacje webowe. Właśnie w tym frameworku została napisana moja aplikacja na pracę inżynierską.

c. Docker

Docker to platforma do konteneryzacji oprogramowania, która umożliwia izolację aplikacji w lekkich, przenośnych i jednorodnych środowiskach zwanych kontenerami. Kontenery są autonomicznymi jednostkami, które zawierają wszystko, co jest potrzebne do uruchomienia aplikacji, włączając w to kod, zależności, biblioteki i ustawienia systemowe.

d. Varnish Cache

Tutaj do opisu posłuże się już wykonanym w tym semestrze projekcie “Wykorzystanie aplikacji Varnish Cache jako odwrotnego proxy do aplikacji webowych o dużym natężeniu ruchu sieciowego”.

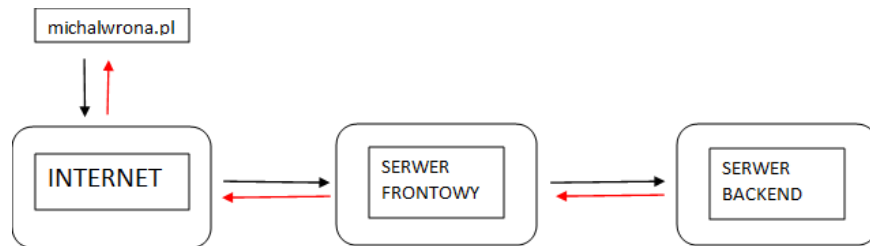
1. Co to Varnish Cache?

Varnish Cache jest to serwer proxy odwrotnego buforowania używany głównie jako akcelerator HTTP dla dynamicznych witryn internetowych o dużym natężeniu ruchu oraz dużej zawartości treści jak i interfejsów API. Varnish przechowuje kontent po stronie serwera w swojej pamięci RAM.

Varnish to oprogramowanie open-source. Pierwsza wersja programu została wydana w 2006 roku.

2. Jak działa większość stron internetowych?

Wyobraźmy sobie, że mamy aplikację składającą się z backendu napisanego w Django REST Framework oraz z frontendu napisanego w React. Gdy wchodzimy na stronę internetową zapytanie musi najpierw dotrzeć do serwera frontowego. Serwer frontowy może, ale nie musi odpytać niejawnie (po stronie serwera) backend o jakieś dane z bazy danych.



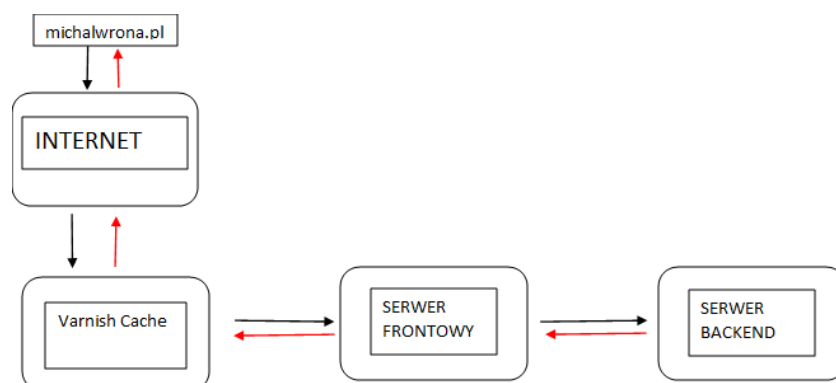
Powyżej jest przedstawiony uproszczony typowy schemat jak działa większość teraźniejszych stron internetowych. Wpisujemy adres, klient w ułamku sekundy zobaczy swoje treści pobrane z serwera frontowego a także dane z bazy danych.

Skomplikujmy trochę sprawę i wyobraźmy sobie, że serwer backendowy jest bardzo wolny, bo ma do przetworzenia skomplikowane dane i zajmuje mu to 5 sekund zanim wystawi odpowiedź dla serwera frontowego.

W ten sposób mamy małe kłopoty, ponieważ wchodząc na stronę internetową klient dostanie swoją odpowiedź w minimum 5 sekund a dodając jeszcze content HTML, CSS'y, JS'y oraz pliki mediowe wartość tego czasu się zwiększa. Z pomocą przychodzi tutaj Varnish Cache!

3. Zastosowanie Varnish'a.

Jednym z rozwiązań problemu powolnego wczytywania stron jest "postawienie" reverse proxy (Varnish Cache) przed serwerem frontowym. Uproszona architektura takiego systemu poniżej.



Teraz zanim zapytanie trafi na serwer frontowy musi przejść przez Varnish'a. Pierwsze zapytanie dotrze do serwera frontowego a serwer frontowy odpyta serwer backendowy o dane. Odpowiedź wraca spowrotem do Varnish'a, na tym etapie następuje hasowanie i

“zapamiętanie” nagłówków, plików cookie oraz ciała całej odpowiedzi przez Varnish’a. Następne zapytania zostaną obsłużone tylko przez Varnish’a i zapytania nawet nie dotrą do serwera frontowego. Proces ten jest nazywany cachowaniem.

Istnieje wiele parametrów, którymi możemy zmodyfikować cachowanie Varnish’a. Poniżej przedstawię parę najprostszych do zrozumienia.

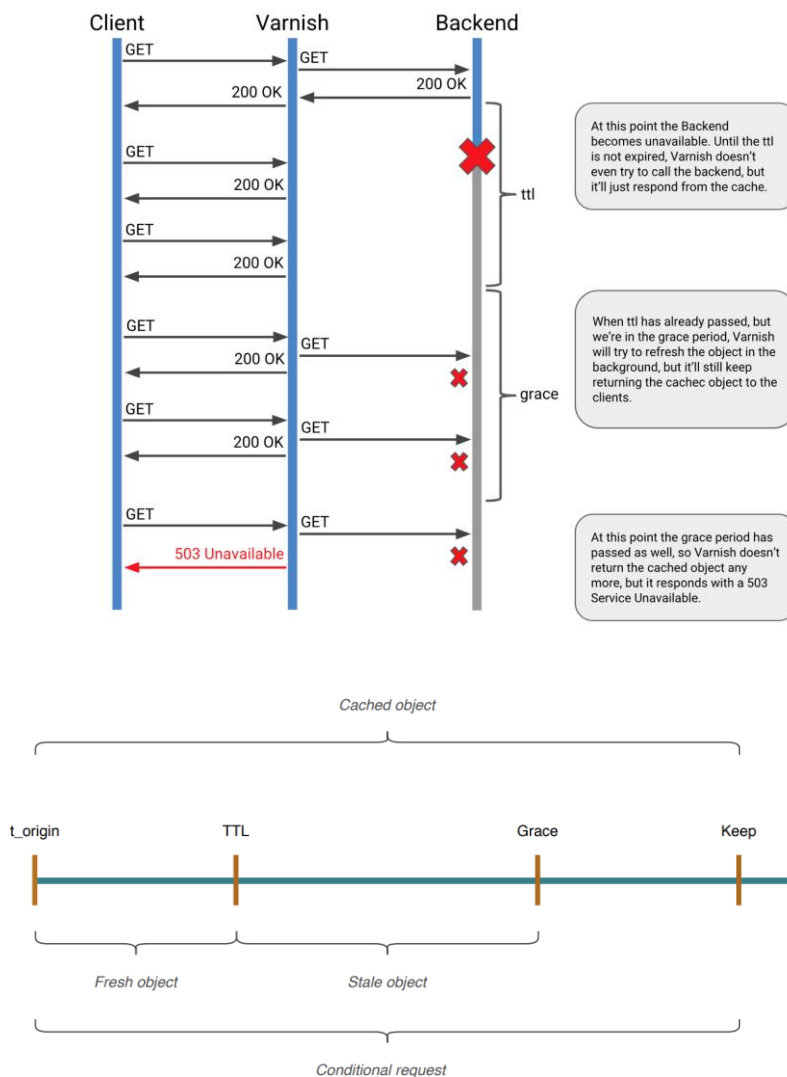
A) TTL (Time To Live) - parametr określający, ile czasu obiekt, który cachował varnish ma przechowywać.

Przykład: TTL jest ustawiony na 60 min. Varnish zacachował obiekt o godzinie 12:30 np. `<h1>Test Varnisha</h1>`, klienci wchodząc na taką stronę od godziny 12:30 do 13:30 zobaczą content `<h1>Test Varnisha</h1>` zobaczą go nawet wtedy, kiedy content na serwerze został zmodyfikowany np. Na `<h1>Test Nginx</h1>`. Dopiero po 13:30 Varnish wykona zapytanie do serwera frontowego, ponieważ tak zacachowanego obiektu skończył się czas ttl.

B) Grace Time (w wolnym tłumaczeniu “czas łask”) - parametr określający, czas życia obiektu, kiedy backend (serwer frontowy) nie odpowiada.

Przykład: TTL jest ustawiony na 1h grace time jest ustawiony na 24h.

Obiekt został zacachowany o godzinie 13:00. O 13:45 serwer frontowy miał awarię i przestał odpowiadać. Logicznie według TTL o godzinie 14:00 obiekt powinien zostać zaktualizowany. Varnish o 14:00 próbuje pobrać nowy obiekt z backend lecz mu się to nie udaje, tutaj wkracza parametr grace time. Dzięki niemu zyskamy działającą stronę o jeszcze 24h! Poniżej obrazek obrazujący czas życia obiektu.



e. PostgreSQL

PostgreSQL, często nazywany "Postgres", to otwartoźródłowy system zarządzania bazami danych relacyjnych (RDBMS). Został stworzony w celu dostarczenia wydajnego, rozbudowanego i zgodnego z normami rozwiązania do przechowywania i zarządzania danymi.

f. GitHub

GitHub to platforma internetowa, która umożliwia zarządzanie projektem, śledzenie zmian w kodzie źródłowym oraz współpracę programistyczną.

g. Redis

Redis to otwartoźródłowy, zaawansowany system zarządzania danymi, znany przede wszystkim jako in-memory key-value store.

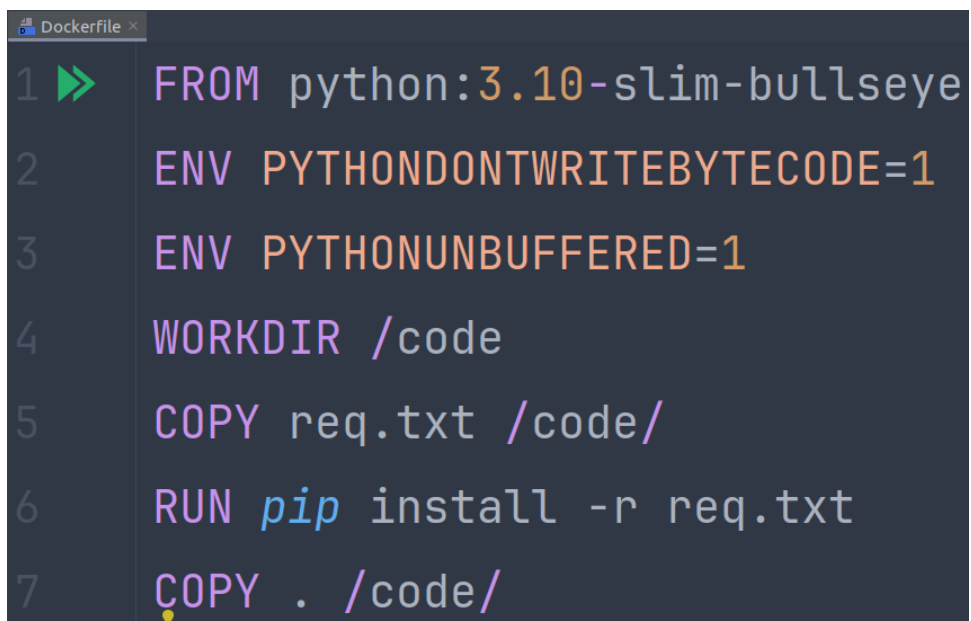
h. Python

Python to wysokopoziomowy, ogólnego przeznaczenia język programowania, który zdobył popularność ze względu na czytelność składni, prostotę użycia i wszechstronność.

3. Realizacja projektu

a. Utworzenie pliku Dockerfile

Jest to plik, który posiada instrukcję o budowania obrazu aplikacji.



```
1 FROM python:3.10-slim-bullseye
2 ENV PYTHONDONTWRITEBYTECODE=1
3 ENV PYTHONUNBUFFERED=1
4 WORKDIR /code
5 COPY req.txt /code/
6 RUN pip install -r req.txt
7 COPY . /code/
```

Pierwsza linia kodu definiuje obraz bazowy w tym przypadku jest to python 3.10. Druga oraz trzecia linijka kodu są to potrzebne konfiguracje do działania języka python. W czwartej linii ustawiany jest katalog roboczy, piąta linia to kopiowanie listy zależności do instalacji. Szósta linijka to instalowanie zależności za pomocą narzędzia pip, siódma linia to kopiowanie całego kodu lokalnego do tworzonego obrazu aplikacji.

b. Instalacja minikube (kubernetes)

```
michal@michal-wrona:~$ curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
michal@michal-wrona:~$ sudo install minikube-linux-amd64 /usr/local/bin/minikube
michal@michal-wrona:~$ minikube start
🐹 minikube v1.32.0 on Ubuntu 22.04
🌟 Using the docker driver based on existing profile
👉 Starting control plane node minikube in cluster minikube
🔄 Pulling base image ...
🔄 Updating the running docker "minikube" container ...
🔄 Preparing Kubernetes v1.28.3 on Docker 24.0.7 ...
🔍 Verifying Kubernetes components...
   📌 Using image docker.io/kubernetes/metrics-scraper:v1.0.8
   📌 Using image docker.io/kubernetes/dashboard:v2.7.0
   📌 Using image gcr.io/k8s-minikube/minikube-ingress-dns:0.0.2
   📌 Using image gcr.io/k8s-minikube/storage-provisioner:v5
   📌 Using image registry.k8s.io/ingress-nginx/kube-webhook-certgen:v20231011-8b53cabe0
   📌 Using image registry.k8s.io/ingress-nginx/kube-webhook-certgen:v20231011-8b53cabe0
   📌 Using image registry.k8s.io/ingress-nginx/controller:v1.9.4
🔍 Verifying ingress addon...
💡 Some dashboard features require the metrics-server addon. To enable all features please run:

    minikube addons enable metrics-server

🌟 Enabled addons: default-storageclass, storage-provisioner, ingress-dns, dashboard, ingress
🏁 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

c. Utworzenie pliku konfiguracyjnego docker-compose.yml

```

version: "3.8"

services:
  front:
    build: .
    volumes:
      - ./code
    ports:
      - "8080:8080"
    command: python manage.py runserver 0.0.0.0:8080
    container_name: infoboard_front
    env_file:
      - envs/local/web/front/env.front
    depends_on:
      - db_default
      - db_public
      - es
      - redis

  cms:
    build: .
    volumes:
      - ./code
    ports:
      - "8001:8001"
    command: python manage.py runserver 0.0.0.0:8001
    container_name: infoboard_cms
    env_file:
      - envs/local/web/cms/env.cms
    depends_on:
      - db_default
      - db_public
      - front
      - es
      - redis

  db_default:
    image: postgres:15.1-bullseye
    volumes:
      - postgres_data_default:/var/lib/postgresql/data/
    ports:
      - "5432:5432"
    container_name: infoboard_db_default
    environment:
      PGTZ: 'Europe/Warsaw'
      TZ: 'Europe/Warsaw'
    env_file:
      - envs/local/db/env.default

  db_public:
    image: postgres:15.1-bullseye
    volumes:
      - postgres_data_public:/var/lib/postgresql/data/
    ports:
      - "5433:5432"
    container_name: infoboard_db_public
    environment:
      PGTZ: 'Europe/Warsaw'
      TZ: 'Europe/Warsaw'
    env_file:
      - envs/local/db/env.public

  es:
    image: docker.elastic.co/elasticsearch/elasticsearch:7.5.2
    environment:
      - node.name=es
      - cluster.name=es-docker
      - discovery.type=single-node
      - bootstrap.memory_lock=true
      - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
    ulimits:
      memlock:
        soft: -1
        hard: -1
    volumes:
      - es_data:/usr/share/elasticsearch/data
    ports:
      - "9200:9200"

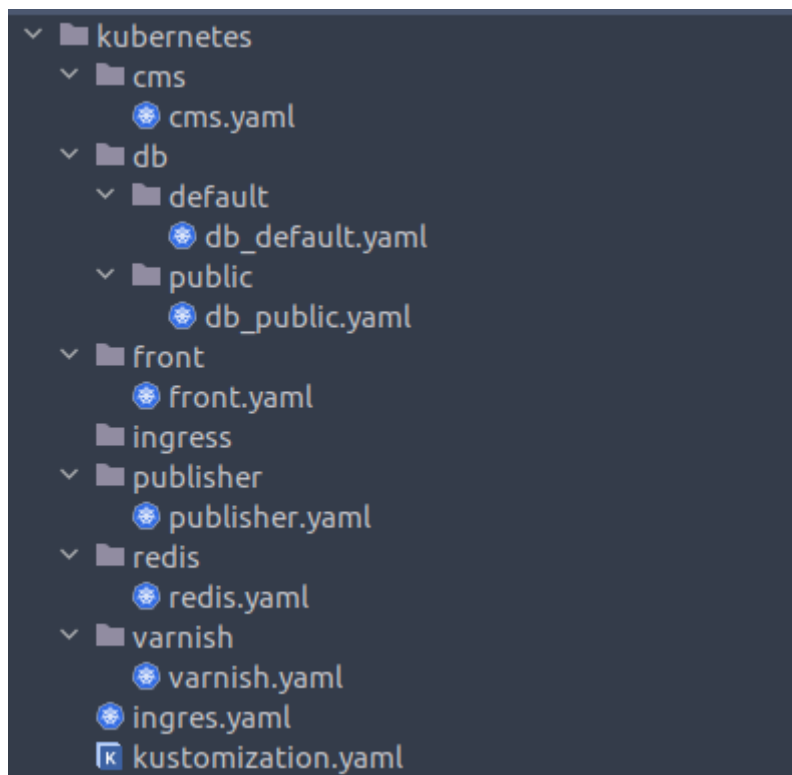
  redis:
    image: redis:alpine

volumes:
  postgres_data_default:
  postgres_data_public:
  es_data:

```

Plik `docker-compose.yaml` definiuje serwisy (aplikacje) ich konfigurację, gdzie mają być przechowywane dane, klucze oraz wolumeny dla danych. Dzięki temu rozwiązaniu możemy jedną komendą `docker-compose up` uruchomić wszystkie kontenery, które pracują w jednej wirtualnej sieci dockera. Kontenery mogą komunikować się między sobą w celu wymiany danych.

d. Stworzenie struktury plików do wdrożenia na kubernetes



- Konfiguracja plików dla wdrożenia aplikacji cms.yaml oraz front.yaml , publisher.yaml oraz redis.yaml.
 - W plikach tych znajdują się stworzone konfiguracje Deploymentu, Servisu oraz ConfigMapy. W ConfigMapie określamy zmienne środowiskowe na przykład hasła do bazy danych. W sekcji Deployment określamy ilość działających kontenerów w podzie jaką komendą server w tym przypadku django ma być uruchamiany oraz z jakiego obrazu ma być korzystany w tej sekcji możemy także określić ilość replik podów. Repliki mogą pomóc w aplikacjach o dużym natężeniu sieciowym wtedy ruch sieciowy jest rozkładany pomiędzy różnymi instancjami aplikacji za pomocą odpowiedniego algorytmu na przykład karuzelowego (round-robin).
- Konfiguracja plików wdrożeniowych baz danych db_default.yaml oraz db_public.yaml.
 - Zawierają definicję ConfigMap, PersistentVolume, PersistentVolumeClaim, Deployment i Service.
 - ConfigMap (db-default-cm):

- Utworzenie ConfigMap o nazwie "db-default-cm" zawierającej dane konfiguracyjne dla aplikacji PostgreSQL, takie jak nazwa bazy danych, nazwa użytkownika i hasło.
- PersistentVolume (db-default-persistent-volume):
 - Utworzenie PersistentVolume o nazwie "db-default-persistent-volume" dla przechowywania danych bazy PostgreSQL.
 - Typ lokalny ("type: local"), o pojemności 8GB, dostępny w trybie ReadWriteMany.
 - Wykorzystanie hostPath do mapowania na ścieżkę "/data/db" na hoście.
- PersistentVolumeClaim (db-default-persistent-volume-claim):
 - Utworzenie PersistentVolumeClaim o nazwie "db-default-persistent-volume-claim" do żądania zasobów przechowywania o pojemności 8GB z wykorzystaniem wcześniej zdefiniowanego PersistentVolume.
- Deployment (db-default):
 - Utworzenie Deployment o nazwie "db-default" dla aplikacji PostgreSQL.
 - Jeden replika kontenera, obraz "postgres", port 5432.
 - Konfiguracja kontenera przy użyciu danych z ConfigMap (db-default-cm).
 - Utworzenie volumeMount i volume dla przechowywania danych na PersistentVolumeClaim.
- Service (db-default):
 - Utworzenie usługi o nazwie "db-default" typu NodePort, dostarczającej dostęp do kontenera PostgreSQL na porcie 5432.
 - Powiązanie usługi z Deployment za pomocą odpowiednich etykiet.

- Konfiguracja pliku varnish.yaml
 - Jest to bardzo podobna konfiguracja jak cms.yaml lecz tutaj mamy także plik konfiguracyjny default.vcl. Jest to plik konfiguracyjny, który mówi Varnish Cache o tym jak ma przyjmować ruch sieciowy i gdzie go kierować a także ma zasady cachowania contentu dla klienta końcowego.

- default.vcl

```
default.vcl: |
vcl 4.1;

backend cms {
    .host = "cms-service";
    .port = "8080";
}

backend front {
    .host = "front-service";
    .port = "8081";
}

sub vcl_recv {
    if (req.http.host == "infoboard-kubernetes.michalwrona.local.pl") {
        set req.backend_hint = front;
    } else if (req.http.host == "cms.infoboard-kubernetes.michalwrona.local.pl") {
        set req.backend_hint = cms;
        return (pass);
    }
}
```

- Konfiguracja pliku ingress.yaml
 - Ten plik odpowiada głównie za to, aby otworzyć dostęp do ruchu sieciowego spoza klastra kubernetes na localhoście.

e. Dodanie hostów

Aby mieć dostęp do aplikacji wystawionej za pośrednictwem kubernetes musimy ustawić sobie hosty w systemie. Aby to zrobić należy najpierw dowiedzieć się na jakim zewnętrznym IP działa nasz ingress-controller wpisując komendę `kubectl get ingress`.

```
(venv) michal@michal-wrona:~/workspaces/InfoBoard$ kubectl get ingress
NAME      CLASS  HOSTS      ADDRESS      PORTS      AGE
ingress   nginx  *          192.168.49.2  80         2d17h
(venv) michal@michal-wrona:~/workspaces/InfoBoard$
```

Jak widać nasz ingress-controler działa na IP 192.168.49.2.
Wchodzimy następnie do katalogu ~/etc i edytujemy plik hosts.

```
michal@michal-wrona:/$ cd etc
michal@michal-wrona:/etc$ sudo nano hosts
```

Wpisujemy adres naszego ingress-controlera i domenę, na której będziemy mieć wystawioną naszą aplikację.

```
GNU nano 6.2 hosts
127.0.0.1 localhost
127.0.1.1 michal-wrona
192.168.49.2 cms.infoboard-kubernetes.michalwrona.local.pl
192.168.49.2 infoboard-kubernetes.michalwrona.local.pl
# The following lines are desirable for IPv6 capable hosts
::1 ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Po zapisie wchodzą na podaną domenę powinniśmy widzieć błąd 404.

f. Uruchomienie zdefiniowanych deploymentów na kubernetesie.

W pliku kustomization.yaml definiuje się ścieżki do wszystkich plików konfiguracyjnych, aby ułatwić sobie sprawę i jedną komendą uruchomić wszystkie deploymenty.

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
  - cms/cms.yaml
  - front/front.yaml
  - ingres.yaml
  - redis/redis.yaml
  - db/default/db_default.yaml
  - db/public/db_public.yaml
  - varnish/varnish.yaml
  - publisher/publisher.yaml
```

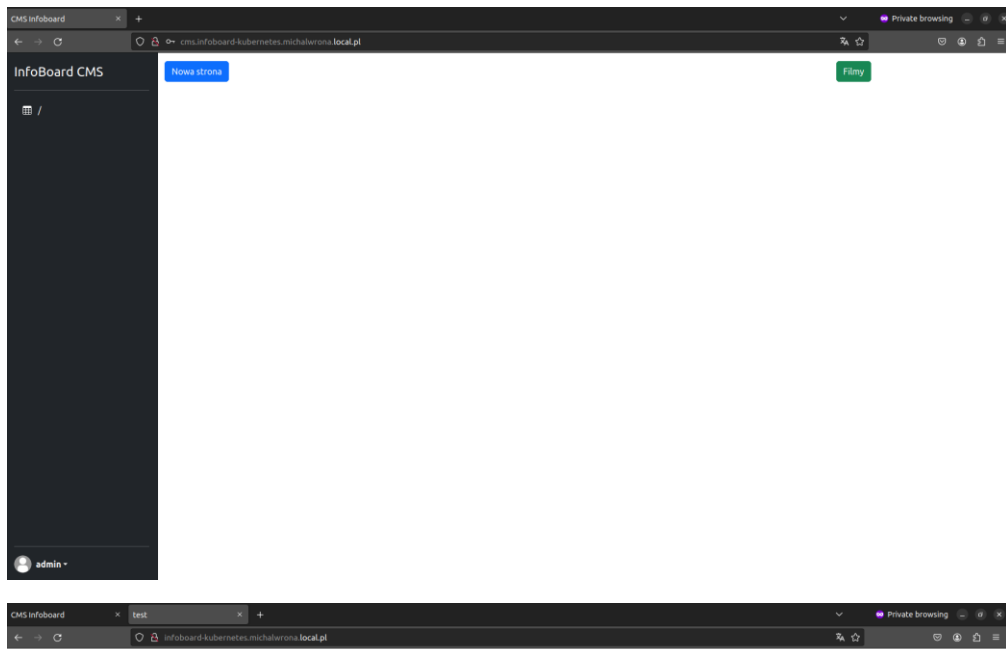
Aby uruchomić wdrożenie należy napisać komendę `kubectl apply -k kubernetes/`

```
(venv) michał@michał-wrona:~/workspaces/InfoBoard$ kubectl apply -k kubernetes/
```

Następnie wpisujemy komendę `kubectl get pods` aby pobrać aktualne pody.

```
(venv) michał@michał-wrona:~/workspaces/InfoBoard$ kubectl apply -k kubernetes/
configmap/cms-config unchanged
configmap/db-default-cm unchanged
configmap/db-public-cm unchanged
configmap/front-config unchanged
configmap/publisher-config unchanged
configmap/varnish-configmap unchanged
service/cms-service unchanged
service/db-default unchanged
service/db-public unchanged
service/front-service unchanged
service/redis unchanged
service/varnish unchanged
persistentvolume/db-default-persistent-volume unchanged
persistentvolume/db-public-persistent-volume unchanged
persistentvolumeclaim/db-default-persistent-volume-claim unchanged
persistentvolumeclaim/db-public-persistent-volume-claim unchanged
deployment.apps/cms unchanged
deployment.apps/db-default unchanged
deployment.apps/db-public unchanged
deployment.apps/front unchanged
deployment.apps/publisher unchanged
deployment.apps/redis unchanged
deployment.apps/varnish unchanged
ingress.networking.k8s.io/ingress unchanged
(venv) michał@michał-wrona:~/workspaces/InfoBoard$ kubectl get pods
NAME                                READY   STATUS              RESTARTS   AGE
cms-65d7769566-ntrhv                1/1     Running             1 (71m ago) 3h37m
db-default-6f48bcd4f-lm9m6          1/1     Running             3 (71m ago) 43h
db-public-5d648589cf-989wx          1/1     Running             3 (71m ago) 43h
front-959576c8-jbppn                1/1     Running             1 (71m ago) 3h37m
publisher-787fb6ddc7-nkcbk          0/1     CrashLoopBackOff    35 (3m56s ago) 3h37m
redis-5d6594f4bf-bhf8x              1/1     Running             3 (71m ago) 43h
varnish-56d4cc87f8-qnjdh            1/1     Running             2 (70m ago) 3h40m
```

Jak widać wszystko przeszło pomyślnie nasze pody się uruchomiły oprócz jednego, ale to wynika już ze specyfikacji aplikacji, po zrobieniu migracji bazodanowych możemy wejść na nasze wcześniej wprowadzone domeny i zobaczyć działającą aplikację.



4. Realizacja pipeline'ów na GitHub

Potoki to bardzo przydatne narzędzie do automatyzacji wdrażania, testowania kodu aplikacji. W swoim projekcie umieściłem dwa zadania:

1. Sprawdzanie poprawności kodu

Poprawność pisanego kodu sprawdzam takimi narzędziami jak black oraz isort. Black w lokalnym środowisku formatuje nasz kod, aby był zgodny z PEP 8¹², isort natomiast sprawdzam nam poprawność importów w kodzie python.

Na obrazku poniżej przedstawiona jest konfiguracja pipeline'u do testowania poprawności kodu:

```
8 jobs:
9   linter:
10     runs-on: ubuntu-latest
11     steps:
12       - uses: actions/checkout@v3
13       - name: Set up Python 3.10
14         uses: actions/setup-python@v3
15         with:
16           python-version: "3.10"
17       - name: Install dependencies
18         run: |
19           python -m pip install --upgrade pip
20           if [ -f req.txt ]; then pip install -r req.txt; fi
21       - name: Lint with black
22         run: |
23           python -m black --check --verbose .
24       - name: Lint with isort
25         run: |
26           python -m isort --check-only .
```

W tym kroku korzystam z bazowego obrazu Python 3.10. Następnie instaluję potrzebne zależności w tym właśnie black oraz isort, następuje testowanie kodu jeśli wystąpi błąd dostajemy maila pipeline nie przechodzi.

2. Budowanie obrazu i wypychanie go na zdalny rejestr.

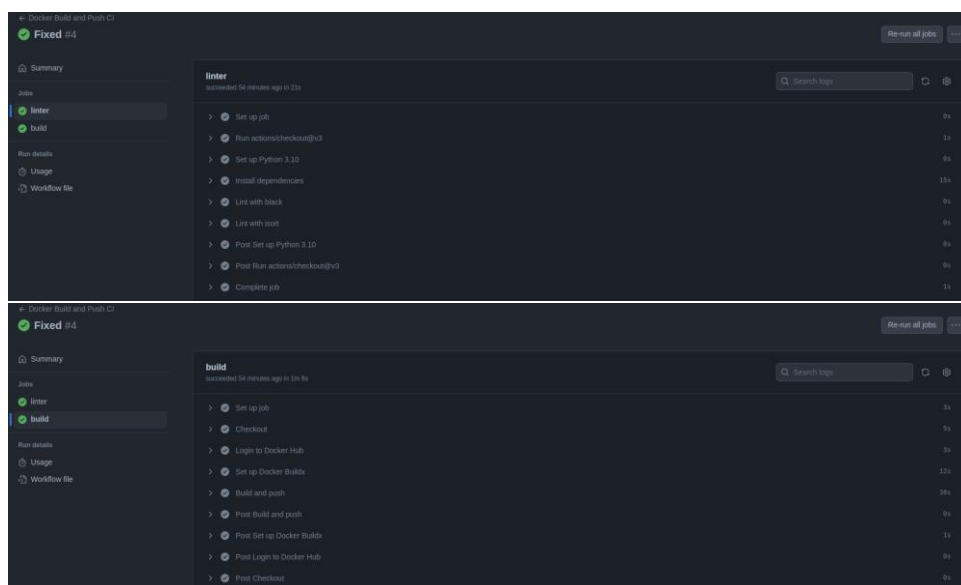
Aby wdrażać aplikacje na kubernetes potrzebujemy mieć obraz aplikacji w jakimś zdalnym rejestrze. Ja użyłem rejestru [dockerhub](https://hub.docker.com/).

¹² zbiór zasad i praktyk pisania czystego kodu w języku Python

Wcześniej stworzyłem plik Dockerfile to właśnie on definiuje nam co ma się znaleźć w takim obrazie. Poniżej przedstawiona jest konfiguracja pipeline budującego i pushującego obraz do zdalnego rejestru.

```
28     build:
29       runs-on: ubuntu-latest
30       steps:
31         -
32           name: Checkout
33           uses: actions/checkout@v4
34         -
35           name: Login to Docker Hub
36           uses: docker/login-action@v3
37           with:
38             username: ${ secrets.DOCKERHUB_USERNAME }
39             password: ${ secrets.DOCKERHUB_TOKEN }
40         -
41           name: Set up Docker Buildx
42           uses: docker/setup-buildx-action@v3
43         -
44           name: Build and push
45           uses: docker/build-push-action@v5
46           with:
47             context: .
48             file: ./Dockerfile
49             push: true
50             tags: ${ secrets.DOCKERHUB_USERNAME }/infoboard:latest
```

Po pushu na branch master do zdalnego repozytorium na GitHubie uruchamiany jest pipeline sprawdzający poprawność kodu a następny krok to budowanie obrazu i pushowanie.



5. Podsumowanie

Wdrożyłem aplikację przy użyciu Kubernetes oraz skonfigurowałem potrzebne pipeline. Niestety mój prywatny serwer nie sprostał wymaganiom sprzętowym kubernetes'a więc wszystko zostało zrealizowane lokalnie. Podczas pracy dużo się nauczyłem o konteneryzacji i orkiestracji kontenerów. Nauczyłem się także tworzyć proste pipeline, które wspomagają pracę programistów. Kod źródłowy został umieszczony w repozytorium [GitHub'a](#).