



POLITECHNIKA
LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI
I INFORMATYKI

NAZWA PRZEDMIOTU:

Programowanie aplikacji w chmurze obliczeniowej

TEMAT PROJEKTU:

*Wpływ jakości powietrza na sen mieszkańców dużego miasta i wsi
- analiza porównawcza*

Prowadzący:

mgr. D. Głuchowski

Zespół projektu:

1. Natalia Waryszak, IS 6.15
2. Agata Szysiak, IS 6.14

Lublin, 9.06.2025

1. Opis projektu

Projekt ma na celu analizę danych o śnie mieszkańców dużego miasta oraz wsi, z uwzględnieniem czynników takich jak stan jakości powietrza.

2. Cel projektu

Stworzenie w pełni funkcjonalnej aplikacji webowej, która umożliwia lekarzom przeglądanie danych pacjentów na temat snu oraz przeglądanie statystyk oraz korelacji wpływu jakości powietrza na stan snu pacjentów.

3. Zakres projektu

- Przetwarzanie danych z bazy danych oraz plików XML I JSON na temat pacjentów i ich snu oraz jakości powietrza dla miasta Krakowa oraz wsi Szarów.
- Budowa intuicyjnego interfejsu użytkownika.
- Implementacja funkcjonalności związanych z wizualizacją (wykresy) i interakcją z danymi.
- Implementacja funkcjonalności filtrowania, odświeżania i eksportowania danych.

4. Wymagania Funkcjonalne

Uwierzytelnianie i bezpieczeństwo

- Aplikacja umożliwia rejestrację i logowanie użytkowników (lekarzy) z walidacją adresu e-mail.
- Hasła są bezpiecznie haszowane przy użyciu biblioteki bcrypt.

- Proces uwierzytelniania oparty jest o tokeny JWT, a dostęp do zasobów zabezpieczony poprzez middleware.

Panel lekarza – zarządzanie danymi pacjentów

Po zalogowaniu lekarz uzyskuje dostęp do panelu, w którym może:

- Wyświetlać listę pacjentów wraz z parametrami dotyczącymi snu (czas trwania, poziom stresu, ciśnienie krwi itp.).
- Dodawać, edytować i usuwać pacjentów.
- Wyszukiwać pacjentów po imieniu, nazwisku lub wieku.
- Zarządzać pomiarami parametrów snu (dodawanie, edycja, usuwanie).
- Wyświetlać dane o jakości powietrza w Krakowie i Szarowie.
- Zobaczyć wykresy porównawcze, które zestawiają jakość powietrza z jakością snu mieszkańców Krakowa i Szarowa.

Przechowywanie danych

- Dane aplikacji są przechowywane w bazie danych NoSQL (MongoDB).
- Dodatkowo wykorzystywane są pliki XML i JSON do eksportu i importu danych.

5. Wymagania Niefunkcjonalne

Kompatybilność przeglądarek:

Aplikacja została przetestowana i działa poprawnie na Google Chrome, Mozilla Firefox oraz Opera.

Wykorzystane technologie:

React.js, Axios, Node.js, Express.js, MongoDB, Docker, Docker Compose, Prometheus, Grafana.

Konteneryzacja:

System jest skonteneryzowany za pomocą Dockera. Posiada kontenery: projekt-server, projekt-frontend, mongo:8.0.10, mongo-express:1.0.0, prom/prometheus:latest, grafana/grafana.

6. Technologie

Projekt został zrealizowany w oparciu o dwa szkieletowe frameworki – osobno dla klienta i serwera.

Frontend:

Aplikacja kliencka została zbudowana w oparciu o React.js, co umożliwia dynamiczne i responsywne wyświetlanie danych oraz płynną komunikację z serwerem poprzez REST API.

Backend:

Warstwa serwerowa oparta jest na Express.js działającym w środowisku Node.js. Odpowiada za obsługę operacji CRUD.

Baza danych:

Wykorzystano nierelacyjną bazę danych NoSQL - MongoDB.

Import danych i formaty:

System obsługuje pliki XML i JSON.

Konteneryzacja i wdrażanie:

Aplikacja działa w odizolowanych środowiskach dzięki wykorzystaniu Dockera.

Monitoring i wizualizacja:

Prometheus został użyty do zbierania danych diagnostycznych z aplikacji, a Grafana umożliwia ich wizualizację.

7. Architektura systemu, struktura projektu, opis użytych technologii i programów**Frontend:**

- React.js – biblioteka JavaScript używana do budowy dynamicznego interfejsu użytkownika w formie aplikacji SPA.
- Axios – lekka biblioteka służąca do wykonywania zapytań HTTP. Pozwala na łatwe wysyłanie żądań do serwera REST API oraz odbieranie odpowiedzi, co umożliwia sprawną komunikację między frontendem a backendem aplikacji.

Backend:

- Node.js + Express.js – Node.js to środowisko uruchomieniowe JavaScript po stronie serwera, które pozwala tworzyć aplikacje sieciowe. Express.js to minimalistyczny framework działający na Node.js, który ułatwia budowanie API i zarządzanie logiką aplikacji, obsługę tras, zapytań i odpowiedzi.

Baza danych:

- MongoDB – nierelacyjna baza NoSQL przechowująca dane pacjentów w formacie dokumentów JSON oraz dane rejestracji użytkowników.
- Import danych z plików JSON i XML

Konteneryzacja i uruchamianie:

- Docker – narzędzie służące do tworzenia, dystrybucji i uruchamiania aplikacji w kontenerach. Kontenery zapewniają odizolowane środowisko uruchomieniowe, które jest niezależne od systemu operacyjnego, co ułatwia przenoszenie aplikacji między różnymi maszynami i środowiskami oraz upraszcza proces wdrożenia.
Docker Compose – narzędzie do zarządzania wieloma kontenerami Docker jako jedną aplikacją, przy użyciu jednego pliku konfiguracyjnego. Umożliwia łatwe uruchamianie, konfigurowanie i komunikację między kontenerami, co upraszcza zarządzanie środowiskiem i przyspiesza wdrażanie złożonych systemów.

Monitoring i wizualizacja:

- Prometheus – narzędzie odpowiedzialne za zbieranie i łączenie danych oraz metryk z aplikacji.
- Grafana – platforma służąca do wizualizacji zebranych danych, pozwalająca na tworzenie interaktywnych pulpitów i wykresów, które wspomagają monitorowanie stanu systemu oraz analizę jego działania.

8. Opis działania aplikacji od strony użytkownika

Aplikacja pozwala na kompleksową analizę danych, umożliwiając użytkownikom przeglądanie wyników w formie tabelarycznej oraz graficznej- wykresy, a także ich filtrowanie i porównywanie pomiędzy obszarami miejskimi i wiejskimi. Dzięki temu możliwe jest wyciąganie wniosków na temat wpływu zanieczyszczeń powietrza na jakość snu mieszkańców różnych środowisk.

Serwer aplikacji został zrealizowany przy użyciu środowiska Node.js wraz z frameworkiem Express, co zapewnia szybkie i elastyczne tworzenie REST API. Frontend zbudowano z wykorzystaniem React.js, dzięki czemu użytkownicy mają dostęp do dynamicznego i intuicyjnego interfejsu.

W projekcie zastosowano nierelacyjną bazę danych MongoDB, która umożliwia przechowywanie oraz zarządzanie danymi.

Dane wykorzystywane w aplikacji pochodzą z dwóch odrębnych źródeł:

- Informacje o jakości snu mieszkańców w formacie JSON:
 - <https://www.kaggle.com/datasets/uom190346a/sleep-health-and-lifestyle-dataset>
- Dane środowiskowe dotyczące jakości powietrza
 - <https://openaq.org>

Widok strony głównej:

Lista pacjentów

Dodaj pacjenta

Szukaj pacjenta po imieniu, nazwisku lub wieku...

Grzegorz Gajda (29 lat)		
Maciej Wrona (29 lat)		
Wojciech Włodarczyk (30 lat)		
Aleksandra Kubiak (30 lat)		
Maciej Wilk (31 lat)		
Patryk Dąbrowski (33 lat)		
Sebastian Wrona (33 lat)		
Sebastian Urban (33 lat)		
Piotr Wiśniewski (33 lat)		
Agnieszka Kowalczyk (36 lat)		

Widok strony dodawania pacjenta:

Strona głównaPacjenciProfilWykresyJakość powietrzaWyloguj

Dodaj pacjenta

Imię

Nazwisko

Wiek

Dodaj

Widok strony edycji pacjenta:

Strona głównaPacjenciProfilWykresyJakość powietrzaWyloguj

Edytuj pacjenta

Grzegorz

Gajda

29

Zapisz zmiany

Widok strony z parametrami snu pacjentów:

Strona głównaPacjenciProfilWykresyJakość powietrzaWyloguj

Lista Pacjentów

Szukaj pacjenta po ID, imieniu lub nazwisku...

Odśwież listę

ID	Imię	Nazwisko	Płeć	Wiek	Zawód	Czas snu (h)	Jakość snu	Aktywność fizyczna	Poziom stresu	Kategoria BMI	Ciepłota krwi	HR	Lokalizacja
10	Grzegorz	Gajda	Male	29	Doctor	7.8	7	75	6	Normal	120/80	70	village
16	Maciej	Wrona	Male	29	Doctor	6	6	30	8	Normal	120/80	70	city
25	Wojciech	Włodarczyk	Male	30	Doctor	7.8	7	75	6	Normal	120/80	70	village
31	Aleksandra	Kubiak	Female	30	Nurse	6.4	5	35	7	Normal Weight	130/86	78	city
39	Maciej	Wilk	Male	31	Doctor	7.6	7	75	6	Normal	120/80	70	village
71	Patryk	Dąbrowski	Male	33	Doctor	6.1	6	30	8	Normal	125/80	72	city

Widok strony z wyszukiwaniem pacjentów:

Lista Pacjentów

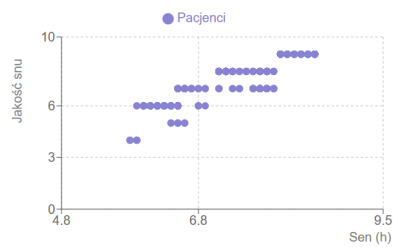
grz

Odśwież listę

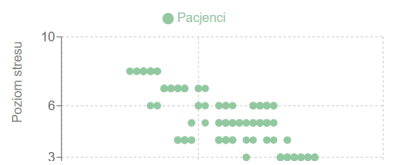
ID	Imię	Nazwisko	Płeć	Wiek	Zawód	Czas snu (h)	Jakość snu	Aktywność fizyczna	Poziom stresu	Kategoria BMI	Ciepłota krwi	HR	Lokalizacja
10	Grzegorz	Gajda	Male	29	Doctor	7.8	7	75	6	Normal	120/80	70	village
169	Grzegorz	Lis	Male	41	Lawyer	7.1	7	55	6	Overweight	125/82	72	village
62	Grzegorz	Gajda	Male	32	Doctor	6	6	30	8	Normal	125/80	72	city
142	Grzegorz	Jaworski	Male	38	Lawyer	7.1	8	60	5	Normal	130/85	68	village
180	Grzegorz	Zając	Male	42	Lawyer	7.8	8	90	5	Normal	130/85	70	village
23	Grzegorz	Jaworski	Male	30	Doctor	7.7	7	75	6	Normal	120/80	70	village
138	Grzegorz	Dąbrowski	Male	38	Lawyer	7.1	8	60	5	Normal	130/85	68	village
230	Grzegorz	Mazur	Male	44	Salesperson	6.3	6	45	7	Overweight	130/85	72	city
18	Grzegorz	Wiśniewski	Male	29	Doctor	6	6	30	8	Normal	120/80	70	village

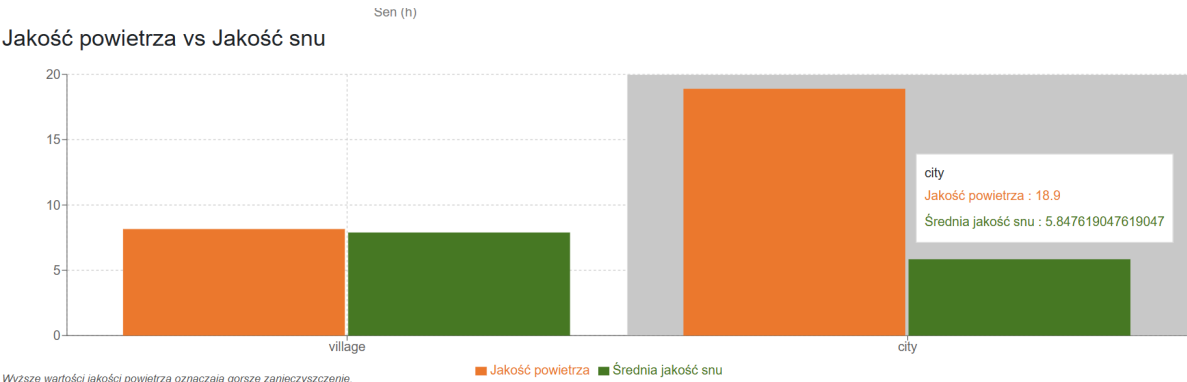
Widok strony z analizą porównawczą/ wykresami:

Sen vs Jakość snu



Sen vs Poziom stresu





Widok strony z danymi o jakości powietrza:

[Strona główna](#) [Pacjenci](#) [Profil](#) [Wykresy](#) [Jakość powietrza](#) [Wyloguj](#)

Dane jakości powietrza

Wies Miasto

Location ID

Location Name

Parameter

Value

Unit

Dodaj pomiar

Data	Lokalizacja	Parametr	Wartość	Jednostka	Akcje
3.06.2025, 15:20:50	Kraków, Aleja Krasińskiego	bc	0.4	µg/m³	Edytuj Usuń
12.05.2025, 04:00:00	Kraków, Aleja Krasińskiego	bc	0.44	µg/m³	Edytuj Usuń
12.05.2025, 05:00:00	Kraków, Aleja Krasińskiego	bc	0.33	µg/m³	Edytuj Usuń

Widoki prezentujące działanie dodatkowych serwisów do zarządzania i monitorowania

Mongo ExpressDatabase: Projekt

Viewing Database: Projekt

Collections

Collection Name

+ Create collection

View

Export

[JSON]

Import

measurements

Del

View

Export

[JSON]

Import

pacjenci

Del

View

Export

[JSON]

Import

users

Del

Database Stats

Collections (incl. system.namespaces)	3
Data Size	152 KB
Storage Size	49.2 KB
Avg Obj Size #	403 Bytes
Objects #	378
Indexes #	6
Index Size	73.7 KB

Mongo Express

PrometheusQueryAlertsStatus > Target health

Select scrape poolFilter by target healthFilter by endpoint or labels

nodejs_server

1 / 1 up

Endpoint

Labels

Last scrape

State

http://server:5000/metrics

instance="server:5000"

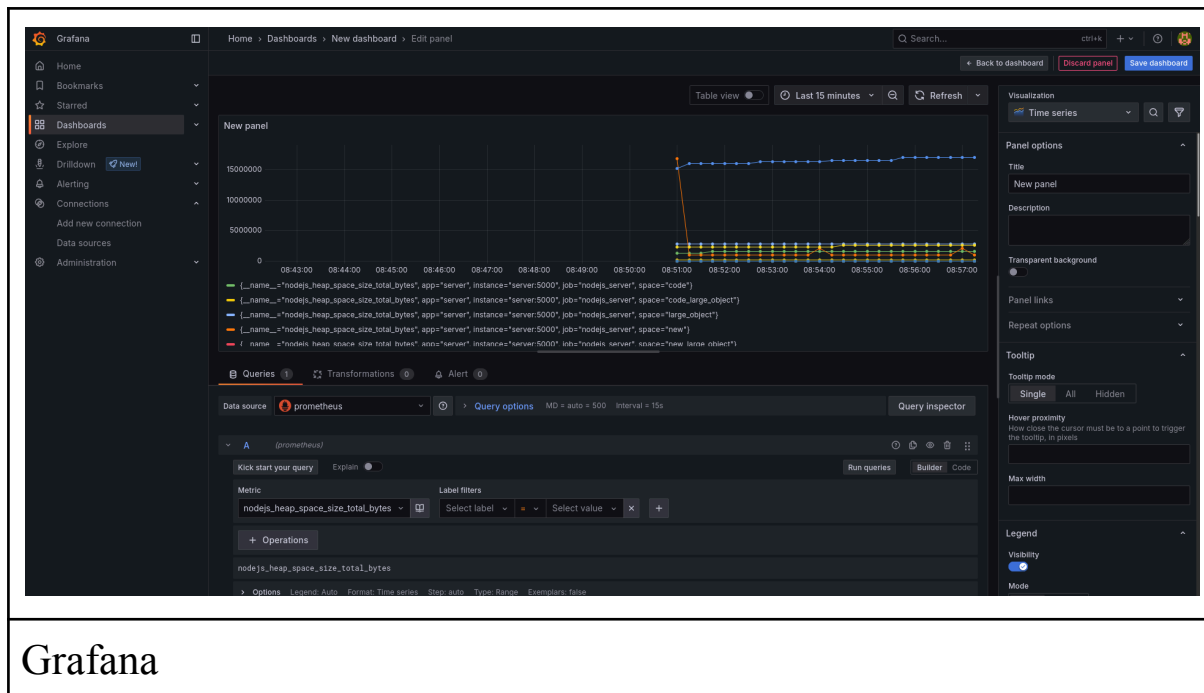
job="nodejs_server"

3.664s ago

3.6ms

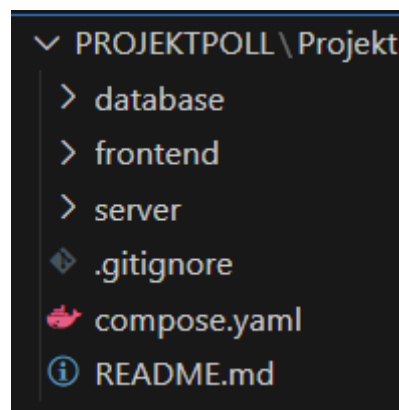
Up

Prometheus



Struktura projektu:

Projekt składa się z 3 folderów: database, frontend, server. Dodatkowo w głównym katalogu projektu znajduje się plik `compose.yaml`, który pełni funkcję centralnego punktu konfiguracji dla całego środowiska w oddzielnych kontenerach Dockera. Dzięki niemu możliwe jest jednoczesne uruchomienie wszystkich komponentów aplikacji. W katalogu frontend i server znajdują się pliki `Dockerfile`, które definiują sposób budowy obrazu kontenerów.



9. Konteneryzacja

Projekt składa się z sześciu kontenerów Docker (Fig. 9.1), które tworzą kompletne środowisko aplikacyjne i deweloperskie. Trzy z nich stanowią rdzeń aplikacji, a pozostałe trzy to narzędzia wspierające.

Główne kontenery aplikacji:

- ***projekt-server (serwer)***

Kontener z aplikacją backendową, zbudowany na podstawie dedykowanego pliku Dockerfile. Odpowiada za logikę biznesową, przetwarzanie danych i komunikację z bazą danych

- ***projekt-frontend (klient)***

Kontener z aplikacją frontendową, również oparty na własnym Dockerfile. Dostarcza interfejs użytkownika, który komunikuje się z serwerem.

- ***mongo:8.0.10 (baza danych)***

Kontener z bazą danych MongoDB, który służy do przechowywania danych aplikacji. Użycie konkretnej wersji (8.0.10) zapewnia spójność środowiska.

Kontenery pomocnicze i monitorujące:

- ***mongo-express:1.0.0***

Narzędzie administracyjne z interfejsem webowym do zarządzania bazą danych MongoDB. Ułatwia przeglądanie, edytowanie i usuwanie danych, co jest szczególnie przydatne podczas rozwoju i testowania.

- ***prom/prometheus:latest***

Kontener z systemem monitorowania i alarmowania Prometheus. Zbierane są w nim metryki dotyczące działania aplikacji (np. zużycie zasobów, czasy odpowiedzi), co pozwala na obserwację jej stanu.

- ***grafana/grafana***

Platforma do analityki i wizualizacji danych. Współpracuje z Prometheusem, umożliwiając tworzenie interaktywnych pulpitów nawigacyjnych (dashboardów) do graficznego przedstawienia zebranych metryk.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
13b63ac9f45c	grafana/grafana	"/run.sh"	About a minute ago	Up About a minute	0.0.0.0:3001->3000/tcp, [::]:3001->3000/tcp	grafana
f0e2ed741a7b	projekt-frontend	"docker-entrypoint.s..."	About a minute ago	Up About a minute	0.0.0.0:3000->3000/tcp, [::]:3000->3000/tcp	projekt-frontend-1
4963df540ec0	prom/prometheus:latest	"/bin/prometheus --c..."	About a minute ago	Up About a minute	0.0.0.0:9090->9090/tcp, [::]:9090->9090/tcp	prometheus
13ea924dffe	mongo-express:1.0.0	"/sbin/tini -- /dock..."	About a minute ago	Up About a minute	127.0.0.1:8081->8081/tcp	projekt-mongo-express-1
43ae855e8776	projekt-server	"docker-entrypoint.s..."	About a minute ago	Up About a minute	0.0.0.0:5000->5000/tcp, [::]:5000->5000/tcp	projekt-server-1
7d6ad2bbe589	mongo:8.0.10	"docker-entrypoint.s..."	About a minute ago	Up About a minute (healthy)	0.0.0.0:27018->27017/tcp, [::]:27018->27017/tcp	mongodb

Fig. 9.1. Kontenery aplikacji

Opis plików Dockerfile w projekcie

W projekcie wykorzystywane są dwa pliki Dockerfile: jeden dla serwera (Fig. 9.2), a drugi dla klienta (Fig. 9.3). Oba bazują na lekkim obrazie node:22.12.0-alpine, co pozwala na efektywne uruchamianie aplikacji Node.js w środowisku kontenerowym.

Dockerfile serwera (Fig. 9.2):

- Ustawia wersję Node.js przez argument `NODE_VERSION`.
- Używa obrazu bazowego `node:${NODE_VERSION}-alpine`.
- Ustawia katalog roboczy na `/usr/src/app`.
- Instalacja zależności odbywa się z wykorzystaniem mountów dla plików `package.json` i `package-lock.json` oraz cache dla `.npm`, co przyspiesza budowanie obrazu.
- Przełącza użytkownika na `node` dla zwiększenia bezpieczeństwa.
- Kopiuje wszystkie pliki serwera do obrazu (oprócz tych niepotrzebnych w `.dockerignore`)
- Wystawia port 5000.
- Uruchamia aplikację poleceniem `npm start`.
- Całość obrazu składa się z 7 warstw.


```

# syntax=docker/dockerfile:1

ARG NODE_VERSION=22.12.0

FROM node:${NODE_VERSION}-alpine

WORKDIR /usr/src/app

RUN --mount=type=bind,source=package.json,target=package.json \
    --mount=type=bind,source=package-lock.json,target=package-lock.json \
    --mount=type=cache,target=/root/.npm \
    npm ci --omit=dev

USER node

COPY . .

EXPOSE 5000

CMD [ "npm", "start" ]

```

Fig. 9.2. Dockerfile serwera

Dockerfile klienta (Fig.9.3):

- Struktura bardzo podobna do serwerowego, również bazuje na obrazie node:\${NODE_VERSION}-alpine.
- Dodatkowo przed przełączeniem użytkownika na node wykonuje polecenie `chown -R node:node /usr/src/app`, zapewniając odpowiednie prawa dostępu do katalogu roboczego.
- Instalacja zależności oraz kopiowanie plików przebiega analogicznie.
- Wystawia port 3000.
- Komenda startowa to również `npm start`.
- Dockerfile klienta składa się z 8 warstw (o jedną więcej niż serwerowy, ze względu na dodatkowe polecenie `chown`).

```
# syntax=docker/dockerfile:1

ARG NODE_VERSION=22.12.0

FROM node:${NODE_VERSION}-alpine

WORKDIR /usr/src/app

RUN chown -R node:node /usr/src/app

USER node

RUN --mount=type=bind,source=package.json,target=package.json \
    --mount=type=bind,source=package-lock.json,target=package-lock.json \
    --mount=type=cache,target=/root/.npm \
    npm ci --omit=dev

COPY . .

EXPOSE 3000

CMD [ "npm", "start" ]
```

Fig. 9.3. Dockerfile klienta

Struktura pliku Compose (Fig. 9.10)

Sekcja Services:

Definiuje sześć serwisów aplikacji, z których każdy jest oznaczony jako "Run Service":

- **mongo** - kontener bazy danych MongoDB

Sekcja mongo w pliku Compose (Fig.9.4) uruchamia kontener z obrazem mongo:8.0.10, ustawia nazwę mongodb, mapuje port 27017, definiuje użytkownika i bazę startową, montuje dwa wolumeny (na dane i skrypty

inicjalizujące), przypisuje do sieci sleep_app oraz posiada healthcheck sprawdzający dostępność bazy co 10 sekund

```
mongo:
  image: mongo:8.0.10
  container_name: mongodb
  restart: always
  ports:
    - "27018:27017"
  environment:
    MONGO_INITDB_ROOT_USERNAME: admin
    MONGO_INITDB_ROOT_PASSWORD: admin
    MONGO_INITDB_DATABASE: Projekt
  volumes:
    - ./mongo_data:/data/db
    - ./database:/docker-entrypoint-initdb.d:ro
  networks:
    - sleep_app
  healthcheck:
    test: ["CMD", "mongosh", "--eval", "db.adminCommand('ping')"]
    interval: 10s
    timeout: 5s
    retries: 5
```

Fig. 9.4. Konfiguracja serwisu mongo

- **server** - serwer aplikacji (projekt-server)

Serwis server (Fig. 9.5) budowany jest z katalogu ./server na podstawie pliku Dockerfile, wystawia port 5000, korzysta z pliku .env, montuje dwa wolumeny (./server:/app i /app/node_modules), działa w sieci sleep_app i uruchamia się po uzyskaniu gotowości przez serwis mongo (depends_on z condition: service_healthy)

```

server:
  build:
    context: ./server
    dockerfile: Dockerfile
  ports:
    - "5000:5000"
  env_file:
    - server/.env
  volumes:
    - ./server:/app
    - /app/node_modules
  networks:
    - sleep_app
  depends_on:
    mongo:
      condition: service_healthy

```

Fig. 9.5. Konfiguracja serwisu serwera

- **frontend** - aplikacja kliencka (projekt-frontend)

Serwis frontend (Fig. 9.6) budowany jest z katalogu ./frontend, wystawia port 3000, montuje dwa wolumeny (./frontend:/app, /app/node_modules), działa w sieci sleep_app i uruchamia się po starcie serwisu server

```

frontend:
  build:
    context: ./frontend
    dockerfile: Dockerfile
  ports:
    - "3000:3000"

  volumes:
    - ./frontend:/app
    - /app/node_modules
  networks:
    - sleep_app
  depends_on:
    - server

```

Fig. 9.6. Konfiguracja serwisu frontendu

- ***mongo-express*** - interfejs administracyjny dla MongoDB

Serwis mongo-express (Fig.9.7) używa obrazu mongo-express:1.0.0 z polityką restartu always. Konfiguruje połączenie z bazą danych mongo oraz dane logowania do własnego interfejsu. Wystawia port 8081, ograniczając dostęp do hosta lokalnego (127.0.0.1), działa w sieci sleep_app i uruchamia się po starcie serwisu mongo.

```
mongo-express:
  image: mongo-express:1.0.0
  restart: always
  environment:
    ME_CONFIG_MONGODB_URL: "mongodb://admin:admin@mongo:27017/Projekt?authSource=admin"
    ME_CONFIG_BASICAUTH_USERNAME: admin
    ME_CONFIG_BASICAUTH_PASSWORD: docker
  ports:
    - "127.0.0.1:8081:8081"
  networks:
    - sleep_app
  depends_on:
    - mongo
```

Fig. 9.7. Konfiguracja serwisu mongo-express.

- ***prometheus*** - system monitorowania

Serwis prometheus (Fig. 9.8) używa obrazu prom/prometheus:latest, wystawia port 9090, montuje plik konfiguracyjny prometheus.yml, działa w sieci sleep_app i uruchamia się po serwisie server.

```

prometheus:
  image: prom/prometheus:latest
  container_name: prometheus
  ports:
    - "9090:9090"
  volumes:
    - ./prometheus.yml:/etc/prometheus/prometheus.yml
  networks:
    - sleep_app
  depends_on:
    - server

```

Fig. 9.8. Konfiguracja serwisu prometheus

- **grafana** - platforma wizualizacji danych

Serwis grafana (Fig. 9.9) używa obrazu grafana/grafana, wystawia port 3000, zapisuje dane w wolumenie grafana_data, działa w sieci sleep_app, uruchamia się po serwisie prometheus i ma politykę restartu unless-stopped.

```

grafana:
  image: grafana/grafana
  container_name: grafana
  restart: unless-stopped
  ports:
    - "3001:3000"
  volumes:
    - grafana_data:/var/lib/grafana
  networks:
    - sleep_app
  depends_on:
    - prometheus

```

Fig. 9.9. Konfiguracja serwisu grafana


Sekcja Volumes:

Definiuje dwa nazwane wolumeny do przechowywania danych:

- *mongo_data* - wolumen dla trwałego przechowywania danych MongoDB
- *grafana_data* - wolumen dla konfiguracji i danych Grafana

Sekcja Networks:

Konfiguruje niestandardową sieć o nazwie *sleep_app* z sterownikiem *bridge*. Sieć typu bridge umożliwia komunikację między kontenerami w ramach izolowanego środowiska sieciowego.



```
compose.yaml > ...
  ▸ Run All Services
1  services:
  ▸ Run Service
2  >  mongo: ...
  ▸ Run Service
22 >  server: ...
  ▸ Run Service
38 >  frontend: ...
  ▸ Run Service
52 >  mongo-express: ...
  ▸ Run Service
65 >  prometheus: ...
  ▸ Run Service
76 >  grafana: ...
88
89  volumes:
90    mongo_data:
91    grafana_data:
92
93  networks:
94    sleep_app:
95      driver: bridge
96
```

Fig. 9.10. Struktura pliku compose.yaml

Wizualizacja działania kontenerów

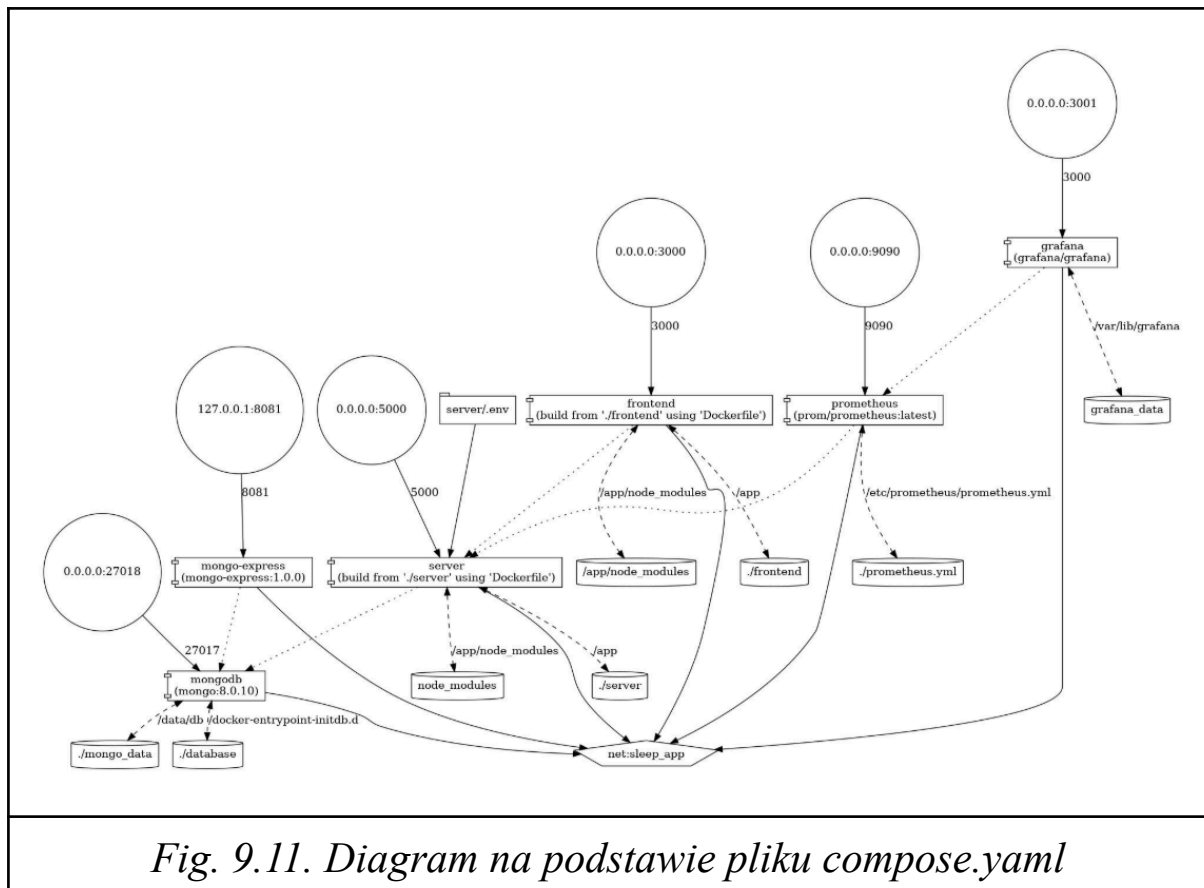


Fig. 9.11. Diagram na podstawie pliku compose.yaml

Powyższy diagram przedstawia architekturę i komunikację kontenerów w projekcie. Każdy kontener realizuje odrębną funkcję: serwer, frontend, baza danych MongoDB, narzędzie administracyjne mongo-express oraz systemy monitorowania Prometheus i Grafana. Dla uproszczenia wszystkie serwisy są połączone jedną wspólną siecią Docker (`sleep_app`). W środowisku produkcyjnym można rozważyć rozdzielenie tej sieci dla większego bezpieczeństwa i izolacji. Wymiana danych i trwałość zapewniane są przez wolumeny, a porty usług są mapowane na hosta, umożliwiając dostęp do aplikacji i narzędzi monitorujących z zewnątrz. Taka architektura gwarantuje izolację, niezależność środowisk oraz łatwe zarządzanie całością systemu przy pomocy Docker Compose.

Testowanie

Aplikacja została przetestowana pod kątem kompatybilności i poprawności działania na popularnych przeglądarkach: Google Chrome, Mozilla Firefox oraz Opera. Testy obejmowały zarówno interfejs użytkownika, jak i komunikację z backendem oraz poprawność operacji CRUD na bazie danych.

Wnioski

Konteneryzacja z wykorzystaniem Docker i Docker Compose okazała się kluczową zaletą projektu, znacząco ułatwiając wdrożenie oraz zarządzanie całym środowiskiem aplikacyjnym. Możliwość uruchomienia wszystkich komponentów za pomocą jednej komendy zapewniła spójność, wyeliminowała problemy z zależnościami i pozwoliła na szybkie testowanie oraz odtwarzanie środowiska.

Największe wyzwania pojawiły się podczas pracy z bazą danych MongoDB w kontenerze. Główne napotkane trudności to:

- Problem z transakcjami: Domyślna konfiguracja kontenera MongoDB uruchamia bazę w trybie standalone, który nie obsługuje transakcji wielodokumentowych. Wymagają one środowiska typu replica set. W rezultacie, każda próba wykonania operacji w transakcji kończyła się błędem, co uniemożliwiało atomowe operacje na danych. Problem rozwiązano poprzez usunięcie logiki transakcyjnej z kodu aplikacji Node.js i oparcie się na operacjach na pojedynczych dokumentach.
- Inicjalizacja danych: Wypełnienie bazy danych (seeding) wymagało przygotowania dedykowanych skryptów, a ich poprawne uruchomienie w cyklu życia kontenera bywało problematyczne. Błędy dotyczyły zarówno kolejności startu usług, jak i poprawnej inicjalizacji danych w wolumenach.

- Niezgodność identyfikatorów: Wystąpił problem z formatem identyfikatorów między backendem a frontendem. MongoDB domyślnie używa pola `_id` typu `ObjectId`, podczas gdy aplikacja kliencka oczekiwała pola `id` w formie tekstowej (`string`). Rozwiązaniem było dodanie do modelu Mongoose pola `id` typu `String` oraz implementacja logiki generującej jego wartość dla nowo tworzonych rekordów, co ustandaryzowało strukturę danych.

Jak na ten moment, efekt działania aplikacji jest zadowalający i spełnia swoje zadanie. Patrząc w przyszłość, warto rozważyć dalsze usprawnienia procesu konteneryzacji, aby był on maksymalnie zgodny z dobrymi praktykami CI/CD oraz zapewniał wysoki poziom bezpieczeństwa.