Automatyzacja przy pomocy docker compose

Plik **compose.yaml**

Baza danych

```
services:
  db:
    hostname: database
    image: mysql:latest
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: Zaq12wsx
      MYSQL_DATABASE: dockerdb
      MYSQL_USER: dockerdb
      MYSQL_PASSWORD: Zaq12wsx
    volumes:
      - ~/ask-node-react-docker/db:/docker-entrypoint-
initdb.d
    networks:
      - backend
```

Backend

```
  node:
    hostname: node
    depends_on:
      - db
    build:
      context: server/
      dockerfile: Dockerfile
    restart: always
    environment:
      NODE_ENV: development
      PORT: 8080
      IP: 127.0.0.1
      DB_HOST: database
      DB_PORT: 3306
      DB_NAME: dockerdb
      DB_USER: dockerdb
```

```
      DB_PSWD: Zaq12wsx
    volumes:
      - ~/ask-node-react-docker/server/src:/app/src
    networks:
      - frontend
      - backend
```

Frontend

```
  react:
    hostname: react
    depends_on:
      - node
    build:
      context: client/
      dockerfile: Dockerfile
    restart: always
    environment:
      - DANGEROUSLY_DISABLE_HOST_CHECK=true
      - CI=true
    ports:
      - "3333:3000"
    volumes:
      - ~/ask-node-react-docker/client/src:/app/src
      - ~/ask-node-react-
docker/client/public:/app/public
    networks:
      - frontend
```

Należy dodać **DANGEROUSLY_DISABLE_HOST_CHECK** w przeciwnym wypadku otrzymamy błąd options.allowedHosts[0] should be a non-empty string. Ta zmienna środowiskowa go nie naprawia tylko sprawia, że aplikacja uruchomi się z tym błędem.

Dariusz Homa, Informatyka Techniczna, semestr IV, gr. 3

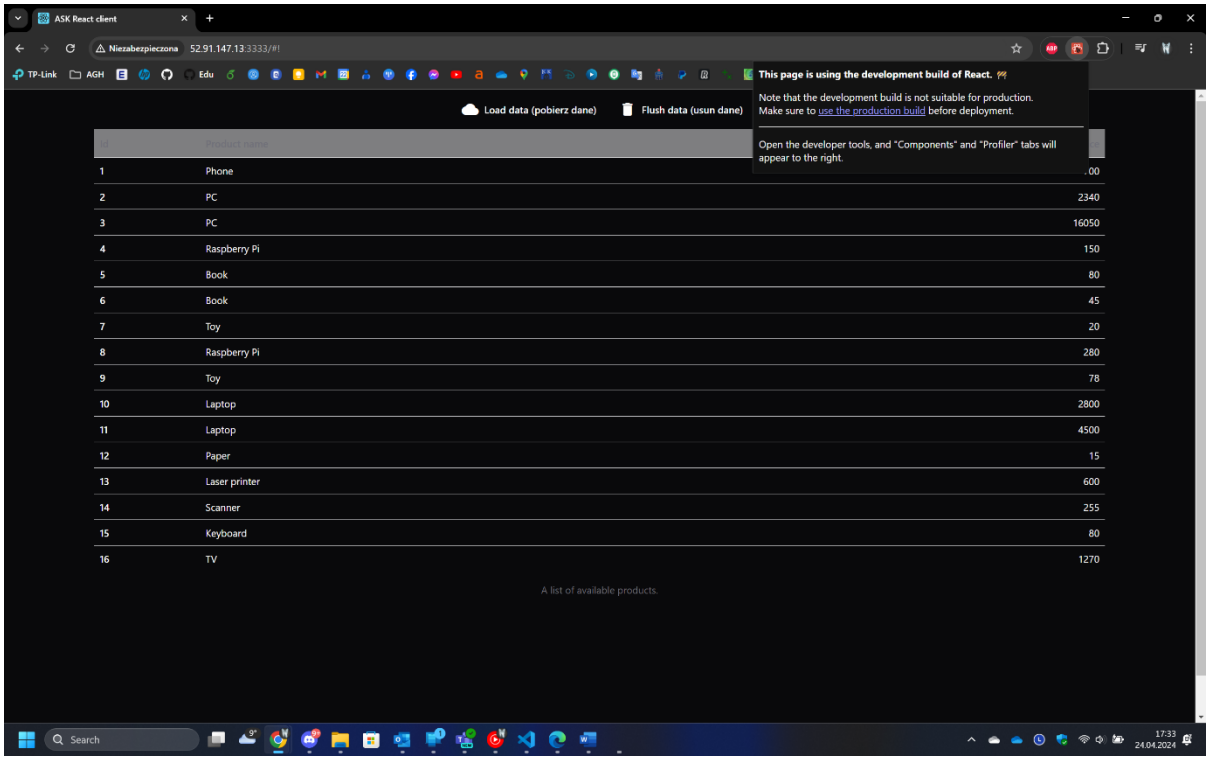
Custom networking w dokerze

Tworzymy sieci w których będą powyżej stworzone kontenery. Te dane również umieszczamy w pliku **compose.yaml**

```yaml
networks:
  backend:
    ipam:
      config:
        - subnet: 172.38.67.0/24
  frontend:
    ipam:
      config:
        - subnet: 172.58.67.0/24
```

Uruchamiamy komendą: docker compose up

Aplikacja poprawnie się uruchamia w trybie developerskim



Produkcyjny Dockerfile apki klienckiej

Nowy plik **Dockerfile**

```
# Stage 1
FROM node:latest as build

WORKDIR /app

COPY package*.json ./
COPY *.config.js ./
COPY *.json ./

RUN npm install

COPY . .

RUN npm run build

# Stage 2
FROM nginx:latest

COPY ./nginx.conf /etc/nginx/nginx.conf

COPY --from=build /app/build /usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Budujemy obraz aplikacji komendą: docker build -f Dockerfile.production -t nginx-react:0.1 .

Produkcyjna wersja compose.yaml

Nowy plik **compose.production.yaml**

```
services:
  react:
    image: nginx-react:0.1
    environment:
      NGINX_ENVSUBST_OUTPUT_DIR:
"/etc/nginx/nginx.conf"
    volumes:
    ports:
      - "80:80"
```

Zmiany w pliku **nginx.conf**

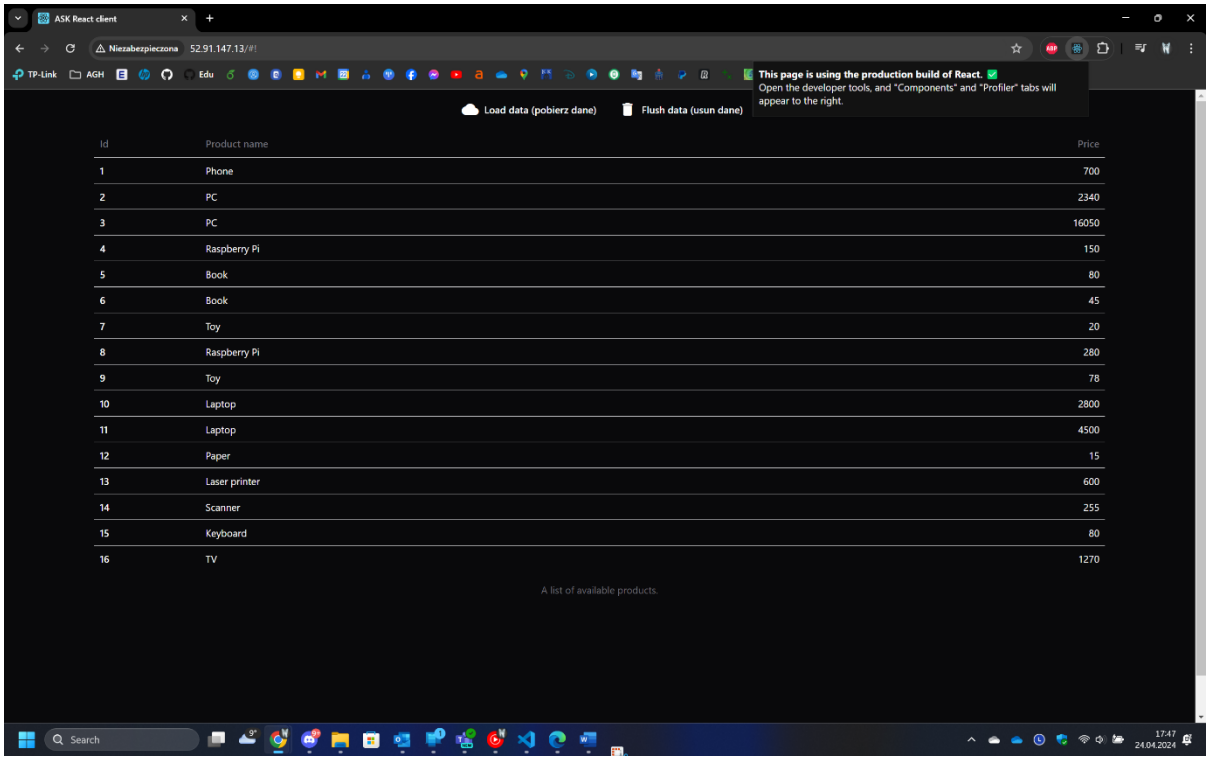
```
upstream backend {
        server node:8080;
}
```

```
# Listen on port 80
        listen 80;
```

Uruchamiamy komendą: docker compose -f compose.yaml -f compose.prod.yaml up

Dariusz Homa, Informatyka Techniczna, semestr IV, gr. 3

Aplikacja działa poprawnie w wersji production



Pytania:

1. Środowisko dev działa na podpiętych voluminach, więc w łatwy sposób możemy wprowadzać zmiany w plikach i na bieżąco sprawdzać czy program działa. Środowisko prod korzysta z wcześniej stworzonego obrazu więc nie możemy tak łatwo wprowadzać zmian.
2. Aplikacja klienta uruchamiana jest z obrazu w którym użyty jest **nginx** oraz zmieniamy port na jakim aplikacja działa.
3. Tworzymy kilka plików **.yaml** aby w łatwy sposób przełączać się między konfiguracjami. Przy uruchamianiu **docker compose** w parametrach dodajemy -f i wybieramy jaką konfigurację chcemy użyć. Jest to wygodne i szybkie rozwiązanie.
4. Nadpisujemy plik **compose.yaml** wartościami z pliku **compose.production.yaml**