Projekt - Technologie Chmurowe

Artur Jakubowski

Czerwiec 2024

Spis treści

1 Opis projektu

Projekt polega na stworzeniu Roadmapy kierunku Informatyka Praktyczna na Uniwersytecie Gdańskim oraz przedstawienie danych dotyczących kierunku takich jak:

- **Opis kierunku**
- **Listy semestrów**
- **Listy przedmiotów na danych semestr i ich opis**
- **Umiejętności nabytych po każdym z przedmiotów**
- **Punkty ECTS, wykładowcy prowadzący, języki wykładowe, itd.**

Wszsytkie te informacje mają zostać docelowo przedstawione w jak najbardziej atrakcyjny wizualnie sposób dla końcowego odbiorcy, w celu zachęcenia większej liczby osób do ukończenia naszego kierunku. Docelowo strona ma zostać wystawiona na stronę Uniwersytetu.

2 Opis architektury

2.1 Skrócony opis architektury

Architektura aplikacji składa się z front-endu oraz back-endu, które są wdrożone w środowisku Kubernetes. Front-end to aplikacja React, a back-end to aplikacja Express, obsługująca logikę biznesową oraz API.

2.2 Opis architektury Frontend

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: react-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: react
  template:
    metadata:
      labels:
        app: react
        version: latest
    spec:
      containers:
      - name: react
        image: loppiko/cloud-project:react
        imagePullPolicy: Always
        ports:
        - containerPort: 3000
        env:
        - name: REACT_APP_API_URL
          value: http://localhost:30001
```

Jest ona opakowana w kontener Docker i wdrożona jako Deployment w Kubernetes. Deployment zapewnia skalowalność oraz zarządzanie cyklem życia kontenerów. Aplikacja zawiera zmienną środowiskową, która definiuje komunikację z Backendem (na potrzeby testów została ona ustwiona na 'localhost:30001') oraz także port na którym będzie działała sama aplikacja wewnątrz kontenera.

2.3 Opis architektury Backend

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: express-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: express
  template:
    metadata:
      labels:
        app: express
        version: latest
    spec:
      containers:
      - name: express
        image: loppiko/cloud-project:express
        imagePullPolicy: Always
        ports:
        - containerPort: 3001
        env:
        - name: MONGO_URL
          value: mongodb://mongo-service:27017/mydatabase
```

Aplikacja Express także została stworzona z wykorzystaniem własnego obrazu Docker oraz Deploymnet w Kubernetes. Plik ten także definiuje port na jakim będzie działała aplikacja, jak i ustawia zmienną środowiskową jak i port na którym działała baza danych.

2.4 Opis architektury bazy danych

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongo-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /data/mongo
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongo-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongo-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mongo
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
      - name: mongo
        image: loppiko/cloud-project:mongo
        ports:
        - containerPort: 27017
        volumeMounts:
        - mountPath: /data/db
          name: mongo-storage
      volumes:
      - name: mongo-storage
        persistentVolumeClaim:
```

```
49          claimName: mongo-pvc
```

Aplikacja mongo wykorzystuje, wykorzystuje mechanizmy Persistent Volume (PV) i Persistent Volume Claim (PVC) w Kubernetes, aby zapewnić trwałość danych i ich integralność w przypadku awarii.

3 Opis infrastruktury

3.1 Przedstawienie plików obrazów dla kontenerów

3.1.1 Plik obrazu Frontend

```
1  FROM node:latest
2
3  WORKDIR /app
4
5  COPY frontend/ .
6
7  RUN npm install
8
9  ENV REACT_APP_API_URL=http://express-app:3001
10
11 CMD [ "npm", "start" ]
```

Plik obrazu Frontendu kopiuje wszytskie dane aplikacji a także instaluje niezbędne zależności. Po wykonaniu tego zadania uruchamia wersję deweloperską aplikacji.

3.1.2 Plik obrazu Backend

```
1  FROM node:latest
2
3  WORKDIR /app
4
5  COPY backend/ .
6
7  RUN npm install
8
9  CMD [ "node", "databaseCommunication.js" ]
```

Plik obrazu dla apliakcji express, także kopiuje pliki jak i konfiugurację serwera, a następnie instaluj zależności i uruchmia aplikację.

3.1.3 Plik obrazu Bazy danych

```
FROM node:latest

WORKDIR /app

COPY backend/ .

RUN npm install

CMD [ "node", "databaseCommunication.js" ]
```

3.2 Zasoby Infrastrukturalne

3.2.1 Sieci

Kubernetes zarządza siecią kontenerów za pomocą wbudowanego modelu sieciowego, który zapewnia każdemu podowi unikalny adres IP. Komunikacja między podami jest realizowana za pomocą usług Kubernetes (Services), które definiują stałe punkty końcowe dostępne dla innych komponentów systemu.

3.2.2 Pamięć Masowa

Kubernetes oferuje różne mechanizmy zarządzania pamięcią masową, takie jak Persistent Volumes (PV) oraz Persistent Volume Claims (PVC), które umożliwiają trwałe przechowywanie danych niezależnie od cyklu życia kontenerów.

4 Opis komponentów aplikacji

Aplikacja składa się z trzech głównych warstw:

- Frontend-u
- Backend-u
- Bazy danych

4.1 Opis części Frontend

Część Frontend-owa zajmuje się wizualną stroną działania aplikacji, w niej napisane są wszystkie pliki stylów jak i komponenty i kontenery, które użytkownik widzi na stronie końcowej. Warstwa ta została napisana z wykorzystaniem React oraz różnych bilbiotek do poprawy samego wyglądu strony, jak i do polepszenia jej funkcjonalności oraz końcowego wrażenia użytkownika. Frontend ma także za zadnia komunikować się z backend-em.

4.2 Opis części Backend

Cześć Backend-owa ma za zadanie serwować na różnych enpointach poszczególne dane, które następnie pobierze frontend i je wyświetli. Warstwa ta łączy się z mongo i przekazuje dane do frontendu, w taki sposób aby było możliwe ograniczenie danych jakie są dostarczane do warstwy frontendowej. Został on napisany w express.js.

4.3 Opis bazy danych

Ostatnia warstwa to baza danych, którą jest MongoDB. Dostep do bazy danych ma tylko i wyłącznie backend, a baza danych przechowuje informacje na wolumenie.

5 Konfiguracja i zarządzanie

Każda warstwa posiada swój osobny plik konfiguracyjny.

5.1 Konfiguracja Frontend

```
const serverConfig = {
    "server-url": process.env.REACT_APP_API_URL,
    "endpoints": {
        "footer": "/footer",
        "header": "/header",
        "mainSite": "/mainSite",
        "subjects": "/subjects"
    }
}

module.exports = serverConfig;
```

Konfiguracja Frontend zawiera adres serwera backend, który jest zapisany w zmiennej środowiskowej oraz obiekt, który reprezentuje lokalizację danych enpointów na serwerze backend.

5.2 Konfiguracja Backend

```
const config = {
    "client-url": process.env.MONGO_URL,
    "database": "mydatabase",
    "port": 3001
}

module.exports = config;
```

Konfiguracja Backend ma za zadanie określić dokładną lokalizację serweru bazo-danowego oraz określić port wewnętrzny na jakim będzie działała aplikacja w kontkretnym podzie.

5.3 Konfiguracja bazy danych

```
#!/bin/sh

# Wait for MongoDB to be ready
echo "Waiting for MongoDB to be ready..."
until mongosh --eval "print(\"waited for connection\")"; do
    echo "MongoDB not ready, sleeping..."
    sleep 2
done

echo "MongoDB is ready, starting import..."

# Import JSON files
for file in /docker-entrypoint-initdb.d/*.json; do
    collection=$(basename "$file" .json)
    mongoimport --host localhost --db mydatabase
        --collection "$collection" --file "$file"
done

echo "Import finished."
```

Skrypt ten ma za zadanie utowrzyć kolekcję z gotowych plików .json, które reprezetują bazę danych i utowrzyć z nich kolekcje, które następnie zoostaną wczytane do bazy. Plik ten jest automatycznie wykonywany podaczas startu kontenera. Sama pliki bazodanowe, są podzielowne na 4 osobne pliki, które docelowo w mongo będą wystawiane jako kolekcje. Każdą z tych kolekcji wystawia express na konkretnym endpoincie.

6 Zarządzanie błędami

W momencie wystąpienia błędów poszczególne pody są restartowane i przywracane do domyślnego stanu aby przywrócić ich prawidłowe działanie. Możliwa liczba restartów podów została ograniczona do 4. Zarządznie błędami jest także kontrolowane w samym kodzie aplikacji i w zależności od typu błędu aplikacj aróżnie na niego zareaguje.

7 Skalowalność

Skalowalność jest zapewniona przez samą warstwę Kubernetes, a jej dokładnie zachowaine może zostać zmodyfikowane poprzez zamianę liczby replik

8 Wymagania dotyczące zasobów

Tutaj przedstwaione są wymagania dotyczące poszczególnych serwisów:

8.1 Wymagania Frontend

```yaml
resources:
    requests:
        memory: "256Mi"
        cpu: "500m"
    limits:
        memory: "512Mi"
        cpu: "1"
```

8.2 Wymagania Backend

```yaml
resources:
    requests:
        memory: "512Mi"
        cpu: "500m"
    limits:
        memory: "1Gi"
        cpu: "1"
```

8.3 Wymagania Bazy danych

```yaml
requests:
    memory: "512Mi"
    cpu: "800m"
limits:
    memory: "1.5Gi"
    cpu: "1"
```

Wymagania dotyczące zasobów są elastyczne i mogą być dostosowywane w zależności od rzeczywistego obciążenia i wydajności aplikacji. Poniższe przykłady przedstawiają minimalne wymagania, które mogą być skalowane w górę, aby sprostać rosnącym potrzebom:

- React Frontend: 0.5 vCPU, 256 MB RAM, 1 GB Dysku
- Express Backend: 0.5 vCPU, 512 MB RAM, 1 GB Dysku
- MongoDB: 0.8 vCPU, 512MB MB RAM, 1.5GB Dysku

9 Architektura sieciowa

9.1 Architektura sieciowa Frontend

```yaml
apiVersion: v1
kind: Service
metadata:
  name: react-service
spec:
  type: NodePort
  selector:
    app: react
  ports:
    - protocol: TCP
      port: 3000
      targetPort: 3000
      nodePort: 30000
```

9.2 Architektura sieciowa Backend

```yaml
apiVersion: v1
kind: Service
metadata:
  name: express-service
spec:
  type: NodePort
  selector:
    app: express
  ports:
    - protocol: TCP
      port: 3001
      targetPort: 3001
      nodePort: 30001
```

9.3 Architektura sieciowa bazy danych

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mongo-service
spec:
  selector:
    app: mongo
  ports:
    - protocol: TCP
      port: 27017
      targetPort: 27017
```