



Milestone 2

Verslag

Linux Webservices ITFactory

Naam 2 APP/AI

Academiejaar 2023-2024

Campus Geel, Kleinhoefstraat 4, BE-2440 Geel

Inhoudstafel

1	INLEIDING	5
2	UITWERKING VAN DE OPDRACHT	6
2.1	Aanmaken mappenstructuur	6
2.2	Intalleren nodige tools.....	6
2.1	Docker-compose bestand instellen	7
2.2	PostgreSQL container configureren.....	10
2.3	NodeJS container configureren	11
2.4	Apache container configureren	14
2.5	Containers op Docker hub plaatsen.....	15
2.6	Kind en kubectl installeren.....	16
2.7	Kind cluster opmaken	16
2.8	Cluster aanmaken via Vagrant	18
2.9	Eindresultaat.....	20
3	BESLUIT	21
4	BIJLAGE	22

1 INLEIDING

Ik ben Siebe Gios, momenteel een tweedejaars IT-student aan de Thomas More hogeschool, gevestigd op de campus Geel. Mijn passie voor technologie heeft me geleid naar de wereld van Applicatieontwikkeling en Kunstmatige Intelligentie (APP/AI). In het kader van mijn opleiding heb ik recentelijk deelgenomen aan een boeiende opdracht voor het vak Linux Webservices, genaamd "Milestone 2."

In deze opdracht stond het bouwen en configureren van een MEAN (apache, nodejs en postgresql) stack centraal, met behulp van Docker en Kubernetes. Ik was verantwoordelijk voor het opzetten van verschillende containers en Kubernetes cluster. De uitdaging bestond erin ervoor te zorgen dat deze containers naadloos samenwerkten om een webpagina te hosten die mijn volledige naam weergaf, deze naam moest uit de database gehaald worden door middel van een API. Ik voegde er als extra, ook nog een ingress container aan toe, om aan load-balancing te doen.

De opdracht ging echter verder dan alleen het opzetten van de MEAN-stack. Een interessant element was het gebruik van Generative AI als een hulpmiddel om mijn code te verbeteren en mijn begrip van containerisatie te vergroten. Dit gaf me nieuwe inzichten en versterkte mijn vaardigheden op het gebied van Kubernetes en containerbeheer.

In dit verslag zal ik gedetailleerd beschrijven hoe ik deze opdracht heb aangepakt, inclusief alle stappen, commando's en configuraties die ik heb gebruikt om de gewenste resultaten te bereiken. Bovendien zal ik mijn ervaringen met het gebruik van Generative AI delen en reflecteren op hoe dit mijn leerproces heeft beïnvloed. Het doel is niet alleen om mijn prestaties te presenteren, maar ook om inzicht te bieden in hoe deze technologieën kunnen worden toegepast in real-world scenario's binnen het vakgebied van IT en DevOps.

Laten we samen de reis verkennen die ik heb afgelegd om Milestone 2 te bereiken en ontdekken hoe Docker, Kubernetes en Generative AI hebben bijgedragen aan mijn groei als IT-student.

2 UITWERKING VAN DE OPDRACHT

De uitwerking van de opdracht is opgedeeld in aparte delen, om zo het overzicht te behouden. Om deze uitwerking te kunnen volgen, moet je eerst een virtuele machine aanmaken op basis van een Ubuntu image of docker op je host machine installeren. Als de installatie is afgerond, kan je aan de slag gaan met het eerste deel. Namelijk het aanmaken van de nodige mappenstructuur.

2.1 Aanmaken mappenstructuur

Voor het project maken we een nieuwe mappenstructuur, om zo onze bestanden te organiseren en beter te werk kunnen gaan. We maken een "project" map aan, waarin onze docker bestanden komen.

2.2 Intalleren nodige tools

Om de volgende stappen te kunnen volgen, moeten we eerst Docker en Docker compose installeren op onze virtuele machine. Moest je Docker for Windows gebruiken, dan moeten deze commando's niet. In de tutorial maken we gebruik van een Ubuntu machine met Docker en Docker compose. Installeer Docker inclusief Docker compose als volgt:

1. Doe **"wget -O docker.sh <https://get.docker.com/>"**. Dit commando gebruikt wget om een script met de naam docker.sh te downloaden van de opgegeven URL en slaat het op in de huidige map. De -O optie wordt gebruikt om de uitvoer op te slaan in een bestand met de naam docker.sh. Met het ls commando, zie je dat er nu een docker.sh bestand is.

```
vagrant@ubuntu2204:~$ wget -O docker.sh https://get.docker.com/
```

2. Doe **"bash docker.sh"**. Dit commando voert het gedownloade script docker.sh uit met behulp van de Bash-shell. Het docker.sh bestand zal docker intalleren en configureren op het systeem.

```
vagrant@ubuntu2204:~$ bash docker.sh
```

3. Je hebt nu succesvol Docker geïnstalleerd. Docker compose versie twee zit bij de laatste installaties bij ingebouwd.
4. Maak een account aan op <https://hub.docker.com/>.

5.

Doe een "docker login". Log hier in met je account dat je net hebt aangemaakt. We doen dit om de rate limiting bij het downloaden van images van Docker Hub te voorkomen. Apache2 container configureren

```
vagrant@ubuntu2204:~$ docker login
```

2.1 Docker-compose bestand instellen

Nu we succesvol Docker geïnstalleerd hebben op onze machine, kunnen we aan de slag gaan met een docker-compose.yml bestand aan te maken. We gebruiken Docker Compose voor het uitvoeren en instellen van multi-container Docker-applicaties.

Hiervoor maken we een docker-compose.yml bestand voor aan. Dit bestand wordt gebruikt om services, netwerken, volumes en andere configuratieopties voor de applicatie in te stellen. Als we dit bestand delen met andere developers, dan kunnen zij dit bestand uitvoeren om zo dezelfde configuratie te bekomen als jijzelf. Laten we ons docker-compose.yml bestand aanmaken, waarin we uiteindelijk vier verschillende containers zullen draaien:

1. Maak een nieuwe map aan voor het project.

```
vagrant@ubuntu2204:~/MEANStack$ mkdir project
```

2. Maak een nieuwe docker-compose.yml bestand aan in de project map "**nano docker-compose.yml**".

```
vagrant@ubuntu2204:~/MEANStack/project$ nano docker-compose.yml
```

3. Neem de volgende instellingen over in het bestand:

```
GNU nano 6.2
version: '3'

services:
  apache:
    container_name: apache-sg
    build:
      context: ./apache
    ports:
      - "80:80"
    depends_on:
      - nodejs

  nodejs:
    container_name: nodejs-sg
    build:
      context: ./nodejs
    ports:
      - "3000:3000"
    depends_on:
      - postgresql
    networks:
      - app-network

  postgresql:
    container_name: postgresql-sg
    build:
      context: ./postgresql
    environment:
      POSTGRES_USER: exampleuser
      POSTGRES_PASSWORD: examplepassword
      POSTGRES_DB: exampledb
    networks:
      - app-network

networks:
  app-network:
```

Laten we even de tijd nemen om de bepaalde bouwstenen van het docker-compose.yml bestand te bekijken en beter te begrijpen:

- Version: "3.9"
 - o Dit geeft de versie van Docker Compose aan waarmee dit bestand compatibel is.
- Services
 - o Definieert de services/containers die je wilt uitvoeren als onderdeel van de applicatie.
- apache:
 - o Dit is de naam van de eerste service. Deze service zal een container draaien met apache2 geïnstalleerd.

- Ports: -"3000:3000"
 - De poortkoppeling tussen de host en de container. De poort 3000 wordt doorgegeven naar poort 3000 van de container. We zullen dus kunnen connecteren met onze api via <http://192.168.56.5:3000>
 - Build: `./nodejs`
 - Geeft aan waar de Docker-image voor de service moet worden gebouwd. We maken hier later een Dockerfile aan op het aangegeven pad.
 - Depends on
 - Start pas op nadat de postgresql container is opgestart.
- nodejs:
 - Dit is de naam van de tweede service. Deze service zal een container draaien met apache2 geïnstalleerd.
 - Ports: -"80:80"
 - De poortkoppeling tussen de host en de container. De poort 80 wordt doorgegeven naar poort 80 van de container. We zullen dus kunnen connecteren met onze webpagina via <http://192.168.56.5>
 - Build: `./apache`
 - Geeft aan waar de Docker-image voor de service moet worden gebouwd. We maken hier later een Dockerfile aan op het aangegeven pad.
 - Depends on
 - Start pas op nadat de nodejs container is opgestart.
- postgresql
 - Dit is de naam van de derde service. Deze container zal postgresql geïnstalleerd hebben.
 - Build: `./postgresql`
 - Geeft aan waar de Dockerfile met de configuratie voor de sql container staat.
 - Environment
 - Op deze plek stellen we omgevingsvariabelen in voor de postgresql container. Het root wachtwoord en de naam van de database worden hier bepaald.
- Networks
 - Er is een netwerk aangemaakt zodat postgresql en nodejs met elkaar kunnen comminceren via hun service naam en moeten ze niet opengezet worden.

2.2 PostgreSQL container configureren

Nu onze docker-compose.yml in orde is voor de basis configuratie van de drie containers. Kunnen we aan de slag gaan met het aanmaken van de dockerfiles. Beginnen doe we bij de gemakkelijkste configuratie, namelijk de PostgreSQL container. Om de SQL container aan te maken, volgen we volgende stappen:

1. Maak een nieuwe map postgresql.

```
vagrant@ubuntu2204:~/MEANStack/project$ mkdir postgresql
```

2. Navigeer jezelf naar de postgresql map met het cd commando. Maak vervolgens een Dockerfile aan met **"nano Dockerfile"**. Nano is een teksteditor, waarmee je een leeg bestand aanmaakt.
3. Zet de volgende code in het bestand en sla het op met de sneltoets Ctrl + X:

```
GNU nano 6.2
FROM postgres:latest

COPY init.sql /docker-entrypoint-initdb.d/
```

4. Maak ook het init.sql bestand aan met **"nano init.sql"** met volgende inhoud:

```
GNU nano 6.2
CREATE TABLE person (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255)
);
INSERT INTO person (name) VALUES ('Siebe Gios');
```

Laten we vervolgens even de code in het bestand bespreken:

- FROM postgres:latest
 - o Specificeerd de basisimage die gebruikt wordt. In dit geval wordt de officiële postgresql image gebruikt. Latest zegt dat het de laatst beschikbaar postgresql image wil gebruiken.
- COPY ...
 - o We kopieëren onze configuratie van onze database naar de juiste locatie, zodat de configuratie wordt uitgevoerd in de container.

2.3 NodeJS container configureren

Na de PostgreSQL container gaan we nu de NodeJS container configureren. Doe dit aan de hand van volgende stappen:

1. Maak een nieuwe map met nano nodejs.

```
vagrant@ubuntu2204:~/MEANStack/project$ mkdir nodejs
```

2. Navigeer jezelf naar de node map met het cd commando. Maak vervolgens een app.js bestand aan met **"nano app.js"**.

3. Zet de volgende lijnen code in de app.js:

```
GNU nano 6.2
const express = require('express');
const { Pool } = require('pg');
const cors = require('cors');
const os = require('os'); // Import the os module

const app = express();
const port = 3000;

// CORS setup
app.use(cors());

// PostgreSQL configuration
const pool = new Pool({
  user: 'exampleuser',
  host: 'postgresql',
  database: 'exampledb',
  password: 'examplepassword',
  port: 5432,
});

// API endpoints
app.get('/user', async (req, res) => {
  try {
    const client = await pool.connect();
    const result = await client.query('SELECT name FROM person ORDER BY id DESC LIMIT 1');
    const name = result.rows[0].name;
    client.release();
    res.json({ name });
  } catch (err) {
    console.error('Error fetching data', err);
    res.status(500).json({ error: 'Internal Server Error' });
  }
});

// New API endpoint to get the container ID
app.get('/container_id', (req, res) => {
  const containerId = os.hostname(); // Get the container ID using the os module
  res.json({ container_id: containerId });
});

app.listen(port, () => {
  console.log(`App running on port ${port}`);
});
```

Laten we even deze code bekijken:

- `const express = require('express')`
 - o Importeer de Express-module, een webframework voor Node.js.
- `const { Pool } = require('pg');`
 - o Importeer de Pool-class van het pg-pakket, dat wordt gebruikt voor het beheren van PostgreSQL-databaseverbindingen.
- `const cors = require('cors');`

- Importeer het CORS-pakket, dat Cross-Origin Resource Sharing (CORS) mogelijk maakt voor het omgaan met verzoeken van verschillende domeinen.
- `const os = require('os');`
 - Import the os module: Importeer de os-module, die toegang geeft tot enkele besturingssysteemfunctionaliteiten, zoals het verkrijgen van de hostnaam.
- `const app = express();`
 - Creëer een Express-applicatie.
- `const port = 3000;`
 - Wijs de poort 3000 toe aan de variabele 'port'.
- `app.use(cors());`
 - Voeg het CORS-middleware toe aan de Express-applicatie om cross-origin verzoeken mogelijk te maken.
- `Const pool`
 - Postgresql configuratie
- `App.get()`
 - Zijn onze 2 API eindpunten. Één voor de username uit de database te tonen en één voor de container id te tonen. Bij de eerste maakt hij een connectie met de database en haalt hij het eerste veld eruit. Daarna toont hij deze in json formaat. De tweede voert de hostname() methode uit om de container id te verkrijgen en toont die ook in json formaat.
- `App.listen`
 - Toont in de log op welke poort de applicatie runt.

4. Maak nu een dockerfile aan met nano Dockerfile met volgende inhoud:

```
GNU nano 6.2
FROM node:latest

WORKDIR /app

COPY package*.json ./
RUN npm install
RUN npm install cors

COPY . .

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

Deze code gebruikt de laatste versie van de officiële nodejs image op de docker hub.

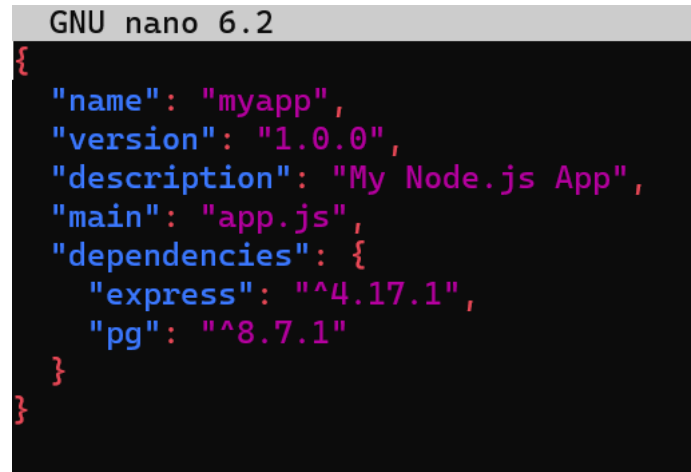
Hij zet de /app map als locatie om commando's uit te voeren en bestanden naar over te zetten.

Met de copy gaat hij de package.json en app.js erin kopiëren.

Hij doet een npm install om alle pakketten te installeren en voegt het cors pakket toe, omdat we deze nodig hebben om naar onze API te kunnen linken van onze eigen frontend.

CMD zegt welk commando er moet zijn om de applicatie uit te voeren.

5. Maak ook een package.json aan met nano package.json met volgende inhoud:



```
GNU nano 6.2
{
  "name": "myapp",
  "version": "1.0.0",
  "description": "My Node.js App",
  "main": "app.js",
  "dependencies": {
    "express": "^4.17.1",
    "pg": "^8.7.1"
  }
}
```

Laten we even de code bekijken:

- "name": "myapp"
 - o Dit is de naam van je Node.js-applicatie. Het moet uniek zijn binnen het npm-ecosysteem.
- "version": "1.0.0"
 - o Dit geeft de versie van je Node.js-applicatie aan. Het volgt het semantische versiebeheer (SemVer) formaat, bestaande uit drie numerieke delen: major, minor en patch.
- "description": "My Node.js App"
 - o Een korte beschrijving van je Node.js-applicatie.
- "main": "app.js"
 - o Dit geeft aan welk bestand het hoofdingangspunt is van je applicatie. Als iemand anders je module importeert, zal Node.js dit bestand als standaard gebruiken.
- "dependencies"
 - o Dit is een object dat de afhankelijkheden van je project definieert. Afhankelijkheden zijn externe pakketten (bibliotheken) die je applicatie nodig heeft om te kunnen werken. Hier zijn twee voorbeelden:
- "express": "^4.17.1"
 - o Hiermee wordt aangegeven dat je applicatie afhankelijk is van Express.js, een webframework voor Node.js. De versie "^4.17.1" betekent dat het project compatibel is met versie 4.17.1 van Express.js en alle hogere patch-versies (zoals 4.17.2, 4.17.3, enz.), maar niet met versie 5 of hoger.
- "pg": "^8.7.1"

- Dit geeft aan dat je applicatie afhankelijk is van de pg-module, die wordt gebruikt voor het werken met PostgreSQL-databases. De versie "^8.7.1" betekent dat het project compatibel is met versie 8.7.1 van pg en alle hogere patch-versies, maar niet met versie 9 of hoger.

2.4 Apache container configureren

Na de NodeJS container gaan we nu de Apache container configureren. Deze container zal gemaakt worden op basis van laatste HTTPd image. Doe dit aan de hand van volgende stappen:

6. Maak een nieuwe apache map met mkdir apache.

```
vagrant@ubuntu2204:~/MEANStack/project$ mkdir apache
```

7. Navigeer jezelf naar de Apache map met het cd commando. Maak vervolgens een Dockerfile aan met **"nano Dockerfile"**.

```
GNU nano 6.2
FROM httpd:latest

COPY ./ /usr/local/apache2/htdocs/
```

Laten we de code even bespreken:

- FROM httpd:latest
 - Dit geeft aan dat de Docker-image wordt gebaseerd op het officiële Apache HTTP Server-image met de tag latest. httpd is de officiële naam van de Apache HTTP Server Docker-image. Het gebruik van latest betekent dat de nieuwste beschikbare versie van het image wordt gebruikt.
- COPY ./ /usr/local/apache2/htdocs/
 - Dit kopieert alle bestanden en mappen van de huidige context (lokaal bestandssysteem) naar de map /usr/local/apache2/htdocs/ binnen de container. In het geval van Apache HTTP Server is dit de map waar de standaarddocumentroot zich bevindt, wat betekent dat alle gekopieerde bestanden beschikbaar zullen zijn om te worden bediend door de Apache-webserver.

- Maak ook een index.html bestand aan met **"nano index.html"** met volgende inhoud:

```

GNU nano 6.2 index.ht
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Milestone 2</title>
  </head>
  <body>
    <h1><span id="user">Loading...</span> has reached milestone 2!</h1>
    <h2><span id="containerId">Loading...</span></h2>

    <script>
      // fetch user from API
      fetch("http://milestone2.local/user")
        .then((res) => res.json())
        .then((data) => {
          // get user name
          const user = data.name;
          // display user name
          document.getElementById("user").innerText = user;
        });

      // fetch container ID from API
      fetch("http://milestone2.local/container_id")
        .then((res) => res.json())
        .then((data) => {
          // get container ID
          const containerId = data.container_id;
          // display container ID
          document.getElementById("containerId").innerText = `Container ID: ${containerId}`;
        });
    </script>
  </body>
</html>

```

Laten we ook even de inhoud van dit bestand even bespreken:

- Het bevat een simpele javascript code om van beide API's de inhoud te gaan halen in deze in te laten in de juiste titel.

2.5 Containers op Docker hub plaatsen

Nu onze Dockerfiles voor het aanmaken van de containers en de docker-compose.yml in orde zijn, kunnen we onze containers gaan builden en op de Docker hub gooien.

- Doe een docker compose build, om al de images te maken aan de hand van de dockerfiles en docker-compose.

```
vagrant@ubuntu2204:~/MEANStack/project$ docker compose build
```

- Bekijk de images die gemaakt zijn met docker images.

```
vagrant@ubuntu2204:~/MEANStack/project$ docker images
```

- Doe een docker tag image naam username/repository:tag voor elke image die je gemaakt hebt en pas de gegevens aan.

```
vagrant@ubuntu2204:~/MEANStack/project$ docker tag project-apache sibsiewibsie/apache:1
```

4. Push de images naar de docker hub met docker push username/repository:tag.

```
vagrant@ubuntu2204:~/MEANStack/project$ docker push sibsiewibsie/apache:1
```

2.6 Kind en kubectl installeren

Nu alles op zijn plek staat en op de Docker hub staan, kunnen we een kubernetes cluster gaan maken. Hier hebben we enkele tools voor nodig, namelijk kubectl en kind. Deze installeren we als volgt:

1. Vul volgende commando's in om kubectl te installeren:

```
sudo apt-get update
sudo apt-get install -y ca-certificates curl
curl -fsSL https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo gpg --
dearmor -o /etc/apt/keys/kubernetes-archive-keyring.gpg
echo "deb [signed-by=/etc/apt/keys/kubernetes-archive-keyring.gpg]
https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee
/etc/apt/sources.list.d/kubernetes.list
sudo apt-get update
sudo apt-get install -y kubectl
```

2. Installeer kind als volgt:

```
curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.20.0/kind-linux-arm64
chmod +x ./kind
sudo mv ./kind /usr/local/bin/kind
```

2.7 Kind cluster opmaken

Om een kind cluster te gaan maken van onze docker files en docker hub repositories hebben we nog enkele zaken nodig. We gaan als volgt te werk:

1. Installer kompose met volgende commando's:

```
curl -L
https://github.com/kubernetes/kompose/releases/download/v1.31.2/kompose-
linux-arm64 -o kompose
chmod +x kompose
sudo mv ./kompose /usr/local/bin/kompose
```

2. Converteer nu je docker-compose.yml bestand naar aparte service en deployment yml bestanden om een cluster te kunnen maken:

```
vagrant@ubuntu2204:~/MEANStack/project$ kompose convert -f docker-compose.yaml
```

3. Pas je deployment.yaml bestanden aan en pas de image door te linken naar je images op de docker hub als volgt:

```
spec:
  containers:
    - image: sibsiewibsie/juisteapache:1
      name: apache-sg
      ports:
```

4. Maak een kindconfig bestand aan met nano kindconfig en zet er volgende inhoud in:

```
GNU nano 6.2
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
    kubeadmConfigPatches:
      - |
        kind: InitConfiguration
        nodeRegistration:
          kubeletExtraArgs:
            node-labels: "ingress-ready=true"
    extraPortMappings:
      - containerPort: 80
        hostPort: 80
        protocol: TCP
      - containerPort: 443
        hostPort: 443
        protocol: TCP
  - role: worker
  - role: worker
```

Deze configuratie maakt 2 workers aan om aan load-balancing te gaan doen.

5. Maak 2 ingress.yaml bestanden aan met nano ingress.yaml en ingress2.yaml.

```
GNU nano 6.2
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: apache-ingress
  namespace: default
spec:
  rules:
    - host: apache.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: apache
                port:
                  number: 80
```

```
GNU nano 6.2
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: milestone2-ingress
  namespace: default
spec:
  rules:
    - host: milestone2.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: nodejs
                port:
                  number: 3000
```

6. Maak een cluster met kind create cluster --config=kindconfig.

```
vagrant@ubuntu2204:~/MEANStack/project$ kind create cluster --config=kindconfig
```

7. Doe dan volgend commando om ingress in je cluster te krijgen.

```
vagrant@ubuntu2204:~/MEANStack/project$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/deploy/static/provider/kind/deploy.
```

8. Doe een kubectl apply -f apache-service.yml. Voor elk service, deployment en ingress.yml die je gemaakt hebt.

```
vagrant@ubuntu2204:~/MEANStack/project$ kubectl apply -f apache-deployment.yml
```

9. Met volgend commando zie je alle pods die runnen en zie je of het werkt of niet.

```
vagrant@ubuntu2204:~/MEANStack/project$ kubectl get pod --all-namespaces
```


10. Open notepad als administrator en ga naar je etc/hosts bestand en voeg de volgende 2 toe.

```
192.168.56.5 apache.local
192.168.56.5 milestone2.local
```


2.8 Cluster aanmaken via Vagrant

Nu onze cluster volledig werkt, kunnen we hem automatisch gaan aanmaken met Vagrant aan de hand van een script en een vagrantfile. Hiermee kan elke developer mijn MEAN-stack opstarten en gebruiken, zoals hij bij mij werkt. We gaan volgt te werk:

1. Maak een nieuwe folder op je schijf aan.

 Milestone2

2. Maak een vagrantfile aan in deze map.

 Vagrantfile

3. Zet er volgende inhoud in:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

#This sets up the Vagrant configuration. The number "2" represents the configuration version and is not recommended to be changed unless you know what you're doing.
Vagrant.configure("2") do |config|

  #Specifies the base box for the virtual machine. In this case, it's using the "generic/ubuntu2204" box, which is an Ubuntu 22.04 LTS image.
  config.vm.box = "generic/ubuntu2204"

  #Configures the VirtualBox provider settings. It sets the VM name, memory (RAM) to 4096 MB, number of CPUs to 2, and enables the GUI.
  config.vm.provider "virtualbox" do |v|
    v.name = "Milestone2-Vagrant"
    v.memory = 4096
    v.cpus = 2
    v.gui = true
  end


  #Specifies that provisioning will be done using a shell script located at "script.sh".
  config.vm.provision "shell", path: "script.sh"

  #Creates a port forwarding rule, mapping port 80 on the guest machine to port 8080 on the host machine.
  config.vm.network "forwarded_port", guest: 80, host: 8080

  #Creates a private network with a static IP address "192.168.56.5" for host-only access.
  config.vm.network "private_network", type: "static", ip: "192.168.56.5"

  #Syncs the current folder (where the Vagrantfile is located) with the "/vagrant" folder inside the virtual machine.
  config.vm.synced_folder ".", "/vagrant"
end
```

4. Maak nu ook een script.sh aan in dezelfde map.

 script

5. Zet er volgende inhoud in:

```
#!/bin/bash

# Install Docker
wget -O docker.sh https://get.docker.com/
bash docker.sh
sudo usermod -aG docker vagrant
newgrp docker

REGISTRY_URL="https://index.docker.io/v1/"
USERNAME="sibsiwibsie"
PASSWORD="Siebe2004."

docker login $REGISTRY_URL -u $USERNAME -p $PASSWORD

# Install kind
curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.20.0/kind-linux-amd64
chmod +x ./kind
sudo mv ./kind /usr/local/bin/kind

# Install kubectl
sudo apt-get update
sudo apt-get install -y ca-certificates curl
curl -fsSL https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-archive-keyring.gpg
echo "deb [signed-by=/etc/apt/keyrings/kubernetes-archive-keyring.gpg] https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee /etc/apt/sources.list.d/kubernetes.list
sudo apt-get update
sudo apt-get install -y kubectl

# Install Git
sudo apt-get install -y git

# Clone your Git repository
git clone https://github.com/GiosSiebe/MeanStack.git

cd MeanStack

# Create Kubernetes cluster
kind create cluster --config=kindconfig.yml

# Apply Ingress-NGINX deployment
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/deploy/static/provider/kind/deploy.yaml

# Pause to allow Ingress-NGINX to start
sleep 120

# Apply other Kubernetes resources
kubectl apply -f apache-service.yaml
kubectl apply -f apache-deployment.yaml
kubectl apply -f nodejs-service.yaml
kubectl apply -f nodejs-deployment.yaml
kubectl apply -f postgresql-service.yaml
kubectl apply -f postgresql-deployment.yaml

# Pause to allow other resources to start
sleep 360

# Apply Ingress resources
kubectl apply -f ingress.yaml
kubectl apply -f ingress2.yaml

# Print success message
echo "Kubernetes setup completed!"
```

6. Open in je map in een terminal.



Open in Terminal

7. Doe een vagrant up en zie hoe de machine gebouwdt wordt.

8. Krijg je op het einde het bericht "Kubernetes setup completed", dan is je machine volledig gebouwd. Als je nu naar apache.local gaat, zou je terug de pagina moeten zien.

2.9 Eindresultaat

Met trots kan ik melden dat de implementatie van onze MEAN-stack en load balancing met behulp van ingress succesvol is afgerond. Dit project, als onderdeel van Milestone 2 voor het vak Linux Webservices, heeft een fascinerende reis vertegenwoordigd, waarbij ik een breed scala aan vaardigheden en kennis heb toegepast en verworven.

In volgende Youtube video's kan je het bewijs vinden van volgende zaken:

Demo Kubernetes en mappenstructuur

<https://www.youtube.com/watch?v=6LXhsXnwldA>

3 BESLUIT

Het afronden van Milestone 2 voor het vak Linux Webservices is voor mij een leerzame en verrijkende ervaring geweest. Het heeft me niet alleen geholpen om mijn kennis van Docker en clustering te verdiepen, maar heeft me ook kennis laten maken met het waardevolle gebruik van generative AI als hulpmiddel in het veld van IT.

Een van de meest boeiende aspecten van deze opdracht was het creëren van een aangepaste Apache2-container en het koppelen ervan aan een PostgreSQL-container via een NodeJS API. Hierbij moest ik ervoor zorgen dat mijn webpagina dynamisch reageerde op wijzigingen in de database, en dat de gegevens persistent waren, zelfs na het verwijderen en opnieuw starten van de containers. Deze hands-on ervaring heeft mijn begrip van containerisatie en webserverconfiguratie aanzienlijk verbeterd.

Het gebruik van generative AI heeft een nieuwe dimensie aan mijn leerproces toegevoegd. Het stelde me in staat om mijn code te verfijnen en optimaliseren op manieren die ik misschien niet zelf had bedacht. Het benadrukte het potentieel van AI als een waardevol hulpmiddel in de wereld van IT.

Als ik terugkijk op deze opdracht, realiseer ik me dat de IT-wereld voortdurend evolueert en dat het cruciaal is om op de hoogte te blijven van nieuwe technologieën en benaderingen. Docker, containers en generative AI zijn slechts enkele voorbeelden van de innovatieve tools die ons kunnen helpen om complexe IT-uitdagingen aan te gaan.

Kortom, Milestone 2 heeft me niet alleen technische vaardigheden bijgebracht, maar ook het belang van experimenteren, leren en het omarmen van nieuwe technologieën in de snel veranderende wereld van IT benadrukt. Ik kijk uit naar verdere uitdagingen en leermogelijkheden binnen mijn studie en mijn toekomstige carrière in de IT.

Met dank aan deze opdracht en de begeleiding van mijn docenten, ben ik beter voorbereid om de complexe IT-vraagstukken van morgen aan te pakken en blijf ik gemotiveerd om mijn kennis en vaardigheden verder te ontwikkelen. De reis gaat verder, en ik ben vastbesloten om deze te blijven verkennen en te groeien als een IT-professional.

4 BIJLAGE

Hieronder vind je ook de definitieve code van elk bestand terug.

Docker-compose.yml

```
version: '3'
services:
  apache:
    container_name: apache-sg
    build:
      context: ./apache
    ports:
      - "80:80"
    depends_on:
      - nodejs
  nodejs:
    container_name: nodejs-sg
    build:
      context: ./nodejs
    ports:
      - "3000:3000"
    depends_on:
      - postgresql
    networks:
      - app-network
  postgresql:
    container_name: postgresql-sg
    build:
      context: ./postgresql
    environment:
      POSTGRES_USER: exampleuser
      POSTGRES_PASSWORD: examplepassword
      POSTGRES_DB: exampledb
    networks:
      - app-network
networks:
  app-network:
```

Dockerfile PostgreSQL

```
FROM mysql:latest
USER root
RUN chmod 755 /var/lib/mysql
```

Init.sql

```
CREATE TABLE person (  
  
    id SERIAL PRIMARY KEY,  
  
    name VARCHAR(255)  
  
);  
  
INSERT INTO person (name) VALUES ('Siebe Gios');
```

App.js NodeJS

```
const express = require('express');  
  
const { Pool } = require('pg');  
  
const cors = require('cors');  
  
const os = require('os'); // Import the os module  
  
const app = express();  
  
const port = 3000;  
  
// CORS setup  
  
app.use(cors());  
  
// PostgreSQL configuration  
  
const pool = new Pool({  
  
    user: 'exampleuser',  
  
    host: 'postgresql',  
  
    database: 'exampledb',  
  
    password: 'examplepassword',  
  
    port: 5432,  
  
});
```

```
// API endpoints

app.get('/user', async (req, res) => {

  try {

    const client = await pool.connect();

    const result = await client.query('SELECT name FROM person ORDER BY id DESC LIMIT 1');

    const name = result.rows[0].name;

    client.release();

    res.json({ name });

  } catch (err) {

    console.error('Error fetching data', err);

    res.status(500).json({ error: 'Internal Server Error' });

  }

});

// New API endpoint to get the container ID

app.get('/container_id', (req, res) => {

  const containerId = os.hostname(); // Get the container ID using the os module

  res.json({ container_id: containerId });

});

app.listen(port, () => {

  console.log(`App running on port ${port}`);});
```

Dockerfile NodeJS

```
FROM node:latest

WORKDIR /app
```

```

COPY package*.json ./

RUN npm install

RUN npm install cors

COPY . .

CMD ["node", "app.js"]

```

Package.json NodeJS

```

{
  "name": "myapp",
  "version": "1.0.0",
  "description": "My Node.js App",
  "main": "app.js",
  "dependencies": {
    "express": "^4.17.1",
    "pg": "^8.7.1"
  }
}

```

Dockerfile apache

```

FROM httpd:latest

COPY ./ /usr/local/apache2/htdocs/

```

Kindconfig

```

kind: Cluster

apiVersion: kind.x-k8s.io/v1alpha4

nodes:
- role: control-plane

  kubeadmConfigPatches:
  - |
    kind: InitConfiguration
    nodeRegistration:
      kubeletExtraArgs:
        node-labels: "ingress-ready=true"

  extraPortMappings:
  - containerPort: 80
    hostPort: 80

```

```

    protocol: TCP

  - containerPort: 443

    hostPort: 443

    protocol: TCP

  - role: worker

  - role: worker

```

Ingress.yml

```

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

  name: apache-ingress

  namespace: default

spec:

  rules:

    - host: apache.local

      http:

        paths:

          - path: /

            pathType: Prefix

            backend:

              service:

                name: apache

                port:

                  number: 80

```

ingress2.yml

```

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

  name: milestone2-ingress

  namespace: default

spec:

  rules:

    - host: milestone2.local

```



```

http:

  paths:

    - path: /

    pathType: Prefix

  backend:

    service:

      name: nodejs

    port:

      number: 3000

```

vagrantfile

```

# -*- mode: ruby -*-

# vi: set ft=ruby :

#This sets up the Vagrant configuration. The number "2" represents the configuration version and is not
recommended to be changed unless you know what you're doing.

Vagrant.configure("2") do |config|

  #Specifies the base box for the virtual machine. In this case, it's using the "generic/ubuntu2204" box, which is
  an Ubuntu 22.04 LTS image.

  config.vm.box = "generic/ubuntu2204"

  #Configures the VirtualBox provider settings. It sets the VM name, memory (RAM) to 4096 MB, number of CPUs
  to 2, and enables the GUI.

  config.vm.provider "virtualbox" do |v|

    v.name = "Milestone2-Vagrant"

    v.memory = 4096

    v.cpus = 2

    v.gui = true

  end

  #Specifies that provisioning will be done using a shell script located at "script.sh".

  config.vm.provision "shell", path: "script.sh"

  #Creates a port forwarding rule, mapping port 80 on the guest machine to port 8080 on the host machine.

  config.vm.network "forwarded_port", guest: 80, host: 8080

```

```
#Creates a private network with a static IP address "192.168.56.5" for host-only access.

config.vm.network "private_network", type: "static", ip: "192.168.56.5"

#Syncs the current folder (where the Vagrantfile is located) with the "/vagrant" folder inside the virtual machine.

config.vm.synced_folder ".", "/vagrant"

end
```

script.sh

```
#!/bin/bash

# Install Docker

wget -O docker.sh https://get.docker.com/

bash docker.sh

sudo usermod -aG docker vagrant

newgrp docker

REGISTRY_URL="https://index.docker.io/v1/"

USERNAME="sibsiewibsie"

PASSWORD="Siebe2004."

docker login $REGISTRY_URL -u $USERNAME -p $PASSWORD

# Install kind

curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.20.0/kind-linux-amd64

chmod +x ./kind

sudo mv ./kind /usr/local/bin/kind

# Install kubectl

sudo apt-get update

sudo apt-get install -y ca-certificates curl

curl -fsSL https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo gpg --dearmor -o
/etc/apt/keyrings/kubernetes-archive-keyring.gpg

echo "deb [signed-by=/etc/apt/keyrings/kubernetes-archive-keyring.gpg] https://apt.kubernetes.io/ kubernetes-
xenial main" | sudo tee /etc/apt/sources.list.d/kubernetes.list

sudo apt-get update

sudo apt-get install -y kubectl
```

```
# Install Git

sudo apt-get install -y git


# Clone your Git repository

git clone https://github.com/GiosSiebe/MeanStack.git


cd MeanStack


# Create Kubernetes cluster

kind create cluster --config=kindconfig.yml


# Apply Ingress-NGINX deployment

kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/deploy/static/provider/kind/deploy.yaml


# Pause to allow Ingress-NGINX to start

sleep 120


# Apply other Kubernetes resources

kubectl apply -f apache-service.yaml

kubectl apply -f apache-deployment.yaml

kubectl apply -f nodejs-service.yaml

kubectl apply -f nodejs-deployment.yaml

kubectl apply -f postgresql-service.yaml

kubectl apply -f postgresql-deployment.yaml


# Pause to allow other resources to start

sleep 360


# Apply Ingress resources

kubectl apply -f ingress.yml

kubectl apply -f ingress2.yml


# Print success message

echo "Kubernetes setup completed!"
```

index.html

```
<html lang="en">

<head>

  <meta charset="UTF-8" />

  <meta http-equiv="X-UA-Compatible" content="IE=edge" />

  <meta name="viewport" content="width=device-width, initial-scale=1.0" />

  <title>Milestone 2</title>

</head>

<body>

  <h1><span id="user">Loading...</span> has reached milestone 2!</h1>

  <h2><span id="containerId">Loading...</span></h2>


  <script>

    // fetch user from API

    fetch("http://milestone2.local/user")

      .then((res) => res.json())

      .then((data) => {

        // get user name

        const user = data.name;

        // display user name

        document.getElementById("user").innerText = user;

      });


    // fetch container ID from API

    fetch("http://milestone2.local/container_id")

      .then((res) => res.json())

      .then((data) => {

        // get container ID

        const containerId = data.container_id;

        // display container ID

        document.getElementById("containerId").innerText = `Container ID: ${containerId}`;

      });

  </script>

</body>

</html>
```