Linux WS - Milestone 2

Documentation

Linux Web Services

Sepp Eyckmans
2CSS

2024 - 2025

Campus Geel, Kleinhoefstraat 4, BE-2440 Geel

TABLE OF CONTENTS

INTRODUCTION

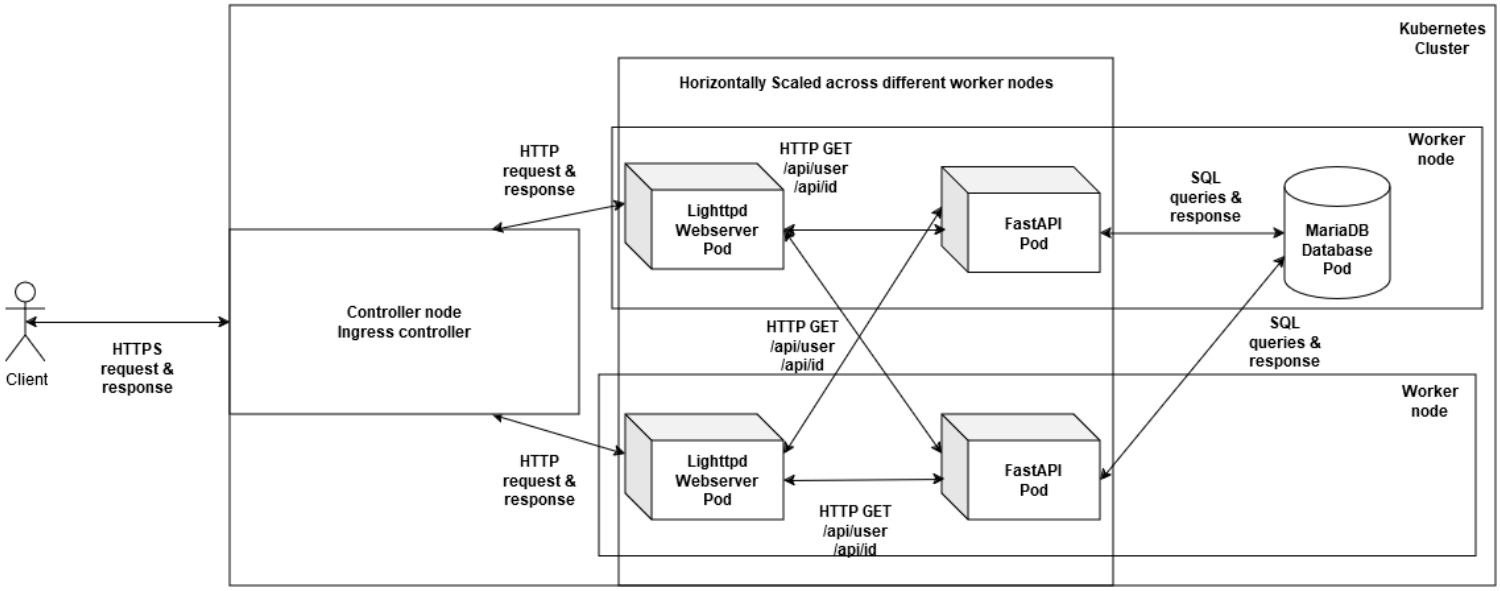
For this project, I will deploy a web application stack on a Kubernetes cluster. The primary goal is to host a dynamic web application, that retrieves data from an relational database. A webserver will serve the application's frontend, which will dynamically fetch and display data from the database via an API, along with other relevant content.

This setup will contain horizontal scaling of the API and webservers to establish redundancy and high availability. The database data will be persistent, achieved with the use of persistent volumes to ensure the durability of all database data. All traffic between the cluster and client will be secured with HTTPS (TLS encryption) to ensure data protection.

This stack will be deployed in Kubernetes. This will ensure automated scaling, load balancing, resource management, and fault tolerance, guaranteeing that the application operates efficiently and reliably at all times. For this project, we will use the Kind Kubernetes distribution. Kind is similar to standard Kubernetes, but its key advantage is that all nodes are contained within a single VM. This simplifies testing and development, making it the optimal choice for this project.

The entire cluster will be running on a Vagrant provisioned VM.

Please refer to the following diagram for the project scope :



The stack consists of the following services :

- Lighttpd : The webserver used to host the web application. It will retrieve data from API endpoints and dynamically display it on the website.
- FastAPI : The API service used for our stack. Fetches data from database and container id of itself. It provides the application logic and creates endpoints for the web server to retrieve data from.
- MariaDB : The relational database used to store application data. The API establishes a connection with the database to fetch and manage the data.
- Kubernetes : The containerization orchestration tool deploying and maintaining our pods.
- Cert-manager : Service that creates and automatically signs/renews certificates for our ingress. Ensuring data protection between client and Master controller, the entry point of all outside traffic.

Let’s start with setting up our Cloudflare DNS. This setup will prepare us for using cert-manager later in the project to generate our certificates.

1 CLOUDFLARE

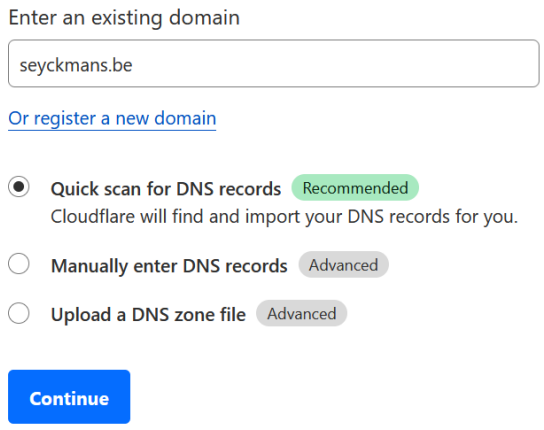
We already have a domain registered on Combell that we can use. We simply have to migrate over to Cloudflare DNS. This migration is necessary because Cloudflare supports cert-manager and facilitates DNS-01 verification, which is required for signing our certificates.

1=> Go to the Cloudflare sign-up website :

https://dash.cloudflare.com/sign-up?pt=f

2=> After account creation, you will receive a mail on the email used to create the account. Press the link in the mail and verify your account

3=> When in your account, it will ask you to enter an existing account. Use your Combell domain name.



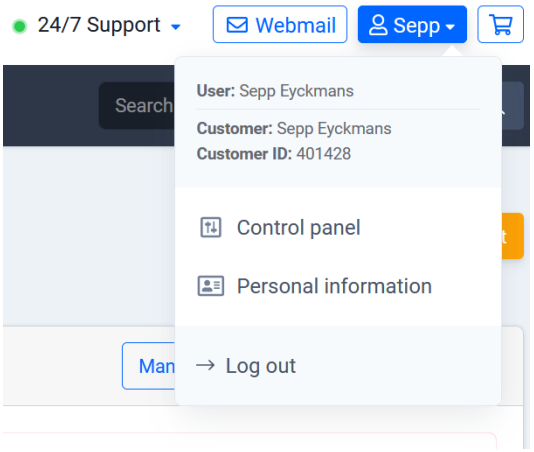
(don't use subdomains, use your main domain)

4=> It will ask to import all your domains records, allow it.
5=> It will give you 2 nameservers, remember them.

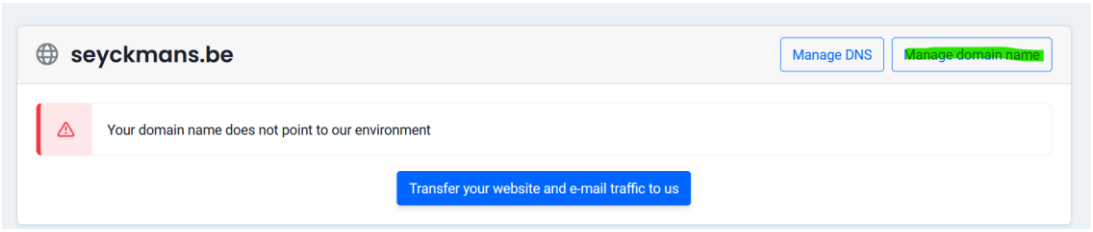
6=> Go to Combell website using the following link :

https://my.combell.com

7 = > Login and go to the account control panel

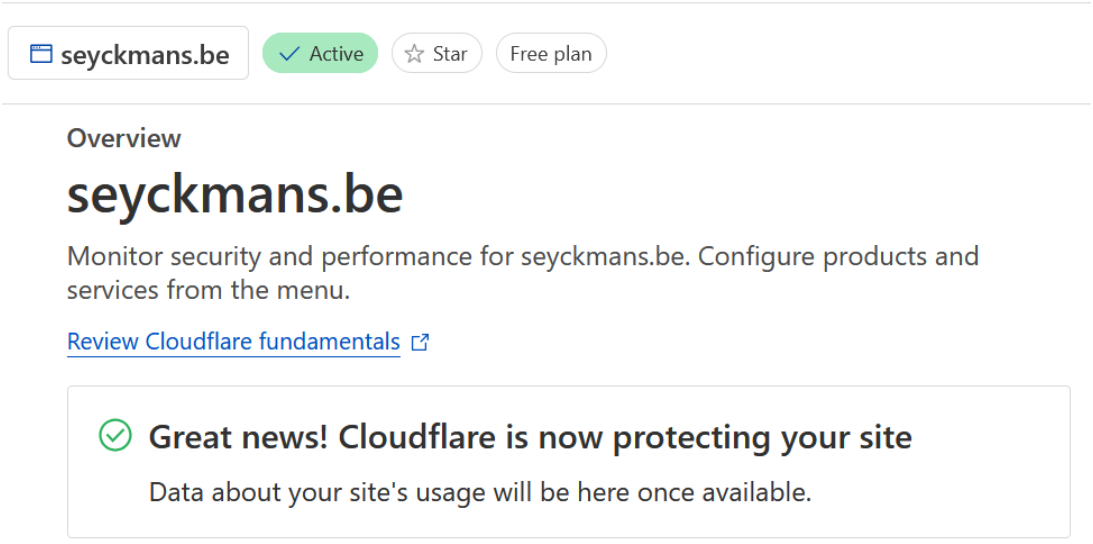


8 = > Go to “Manage Domain name” on your domain of choice

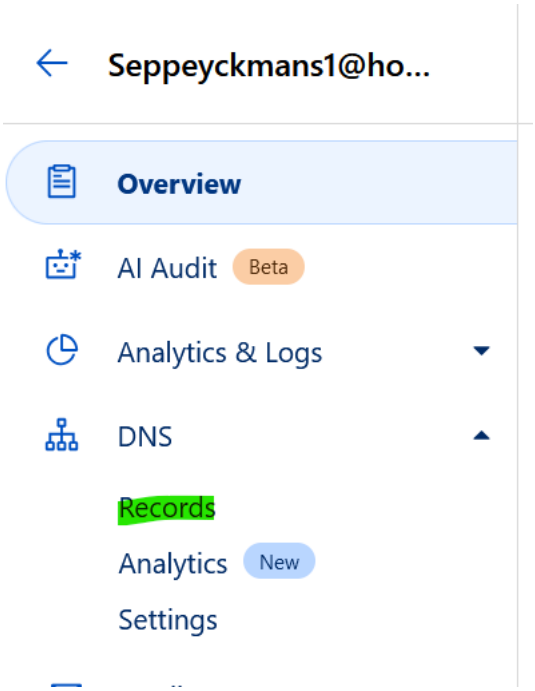


9 = > Go to “Name servers” and change them with the Cloudflare nameservers.

It will take an hour or 2 to fully migrate to Cloudflare. So leave this be and come back later to this step. You know its done migrating when on Cloudflare you see the following on login.



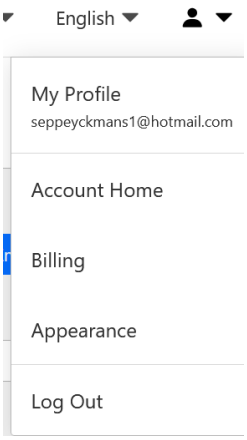
10=> Back on Cloudflare. On your domain, go to “records”



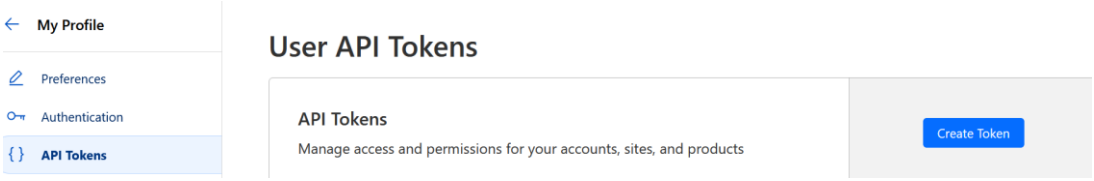
11=> You will see a list of all your records. Create a new A record of choice to use for our project. (So pick a fitting name.)

I picked “m2”. Creating the full domain name “m2.seyckmans.be”. Pick your own name of choice.

12=> Go to “My Profile”



13 => Go to “API token” and generate a token.



- Use the “Edit zone DNS” template.
- Set permissions to : Zone, SSL and Certificates, Edit
- Set another permissions to : Zone, DNS, Edit
- Set zone resources to : include, all zones
- Go down to “Continue to summary”
- Press “Create token”

The token it will create and show is important in creating our trusted certificates. Save this token for later use in this project!

By the time we reach the cert-manager step, our DNS records will have propagated. With our Cloudflare DNS setup complete, we can now move on to provisioning our Vagrant VM.

2 VAGRANT

1=> Go to your project folder and execute the command :

Vagrant Init

```
\Linux_Web_Services\Linux_milestone2> vagrant init
```

2=> Open the Vagrant file.
3=> Replace the content with the following script :

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

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

  config.vm.box = "generic/ubuntu2204"
  config.vm.hostname = "m2-kubeserver-sepp-eyckmans"

  config.vm.network "private_network", ip: "192.168.56.50"

  # Share milestone 2 project subdirectory (data) with Linux VM (vagrant)
  config.vm.synced_folder "./data", "/vagrant"

  config.vm.provider "virtualbox" do |vb|
    vb.name = "m2-vm-se"
    vb.memory = 4096
    vb.cpus = 2
  end

  # Packages & tools to install after VM provisioning
  config.vm.provision "shell", inline: <<-SHELL
    # Install Docker, Docker Compose and MariaDB-client
    apt update -y && apt install apt-transport-https ca-certificates curl -y
    curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
    echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu $(. /etc/os-release && echo "$VERSION_CODENAME") stable" |
    sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
    apt update -y && apt install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin mariadb-client golang -y
    usermod -aG docker vagrant
    newgrp docker

    newgrp docker

    # Install kind
    [ $(uname -m) = x86_64 ] && curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.24.0/kind-linux-amd64
    chmod +x ./kind
    mv ./kind /usr/local/bin/kind
    snap install go --classic
    echo 'export PATH=$PATH:/snap/bin' >> ~/.profile
    source ~/.profile
    go install sigs.k8s.io/kind@v0.24.0

    # Install kubectl
    curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.31/deb/Release.key | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
    echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.31/deb/ /' |
    sudo tee /etc/apt/sources.list.d/kubernetes.list > /dev/null
    apt update -y && apt install kubectl -y

    # Get HELM repo and install HELM
    curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash

    # Restart VM
    shutdown -r now
  SHELL
end
```

Vagrant will provision the VM and execute the script after VM creation. This script will install Docker, MariaDB Client, Kind, Kubectl and HELM.

Kubectl is the CLI tool to connect to our cluster and create our stack in Kubernetes.

HELM is the Kubernetes package manager. This will be used to install cert-manager on our cluster.

4=> Save and close the file.
5=> In your project folder, execute the command :

Vagrant up

This will start provisioning the VM just like in our last milestone project. However, this time it will take approximately 10 - 15 minutes to complete. Once you have control over your command prompt, the VM provisioning is finished and ready for use.

6=> To verify if the VM is done provisioning, try to SSH to the VM :

Vagrant ssh

If you see a Linux CMD with your custom hostname, then the Vagrant step of the project is finished!

The next part of the project is creating the services in our stack. The first step is creating our different config files for our webserver and creating our API application files. Then we can bundle them into different Docker Dockerfiles to create our images. These images together with our MariaDB image will then be used in our Kubernetes cluster creation.

3 BUILDING THE WEB APP STACK

To create our stack services, we have to create a Dockerfile, Lighttpd configuration file and an index page for our Lighttpd webserver. Lighttpd is an lightweight and easy to setup webserver similar to Apache in the previous milestone project.

For our FastAPI, we have to also create a Dockerfile and the Application Python file itself. FastAPI is a lightweight, easy-to-set-up API, offering functionality similar to Node.js.

MariaDB is our trusty database service that we also used in our previous project. We don't have to create custom files or an Dockerfile for it. We can simply pull the official MariaDB image and build on top of this in our Kubernetes.

Lets first start with creating our Lighttpd files.

3.1 Lighttpd

1=> Now in the VM. Go to the Vagrant shared folder directory (/vagrant). This will be the working directory.

2=> Create an empty Dockerfile using this command :

Touch Dockerfile-web

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant$ touch Dockerfile-web
```

3=> Open the file and write the following code :

```
1   FROM rtsp/lighttpd
2
3   WORKDIR /vagrant
4   # replaces default page with self-made index page
5   RUN rm -f /var/www/html/index.html && rm -f /etc/lighttpd/lighttpd.conf
6   COPY ./index.html /var/www/html/
7   RUN chmod -R 755 /var/www/html
8   # replace file with self-made configuration file
9   COPY ./lighttpd.conf /etc/lighttpd/
10
11  # Run webserver in foreground
12  ENTRYPOINT ["lighttpd", "-D", "-f", "/etc/lighttpd/lighttpd.conf"]
13  EXPOSE 80
```

This will use the Lighttpd image and replace the default configuration and index file with the custom files we will create after this step.

Then it will run the webserver in the foreground to prevent it from instantly becoming idle and shutting down.

4=> Save and close the file.

5=> Create an empty Lighttpd config file using this command :

Touch lighttpd.conf

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant$ touch lighttpd.conf
```

6=> Open the file and write the following code :

```
server.document-root        = "/var/www/html"
server.port                 = 80
server.name                 = "localhost"

index-file.names            = ( "index.php", "index.html" )
static-file.exclude-extensions = ( ".php", ".pl", ".fcgi" )
```

This configuration file tells the webserver to listen to request on port 80 and look for the application files in the document root "/var/www/html/". Their it will search for files named "index" and display it.

7=> Save and close the file.

8=> Create an empty index.html file with the following command :

Touch index.html

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant$ touch index.html
```

9=> Open the file and write the following code :

```
<!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>
    <h1>Container ID : <span id="id">Loading...</span></h1>

    <script>
      // fetch user from user API endpoint
      fetch("https://m2.seyckmans.be/api/user")
        .then((res) => res.json())
        .then((data) => {
        // get username
        const user = data.name;
        // display username
        document.getElementById("user").innerText = user;
        });

      // fetch container id from id API endpoint
      fetch("https://m2.seyckmans.be/api/id")
        .then((res) => res.json())
        .then((data) => {
          // get container id
          const id = data.container_id
          // display container id
          document.getElementById("id").innerText = id;
        });
    </script>
  </body>
</html>
```

The website will fetch for data from the API endpoints and use that data to dynamically display it on the page. Replace the fetch URLs with your own domain name.

10=> Save and close the file.

3.2 FastAPI

1=> Still in our work directory (/vagrant), create an empty Dockerfile :

Touch Dockerfile

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant$ touch Dockerfile
```

2=> Open the file and write the following code :

```
1  FROM python:3.9
2
3  WORKDIR /app
4  RUN pip install --no-cache-dir --upgrade "fastapi[all]" mariadb
5  COPY ./app /app
6
7  # Run the app on port 3000
8  ENTRYPOINT ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "3000", "--reload"]
9  EXPOSE 3000
```

This will use the Python image to install FastAPI and the MariaDB package. MariaDB package is needed to create the code needed for the connection to the database.
With the FastAPI package is the tool Uvicorn included. Uvicorn is a fast and lightweight Asynchronous Server Gateway Interface (ASGI) for Python. This tool is vital in serving asynchronous web applications built by FastAPI to the webserver.

The Dockerfile will place our custom made Python application files on the API and then run Uvicorn in the foreground. It will constantly listen on port 3000 for any incoming GET requests made by the webserver.

3=> Save and close the file.

4=> In you working directory, create a directory called “app”.

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant$ mkdir app
```

5=> Go inside this directory and create an empty Python script :

Touch main.py

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant/app$ touch main.py
```

6=> Open this file and write the following code :

```python
from fastapi import FastAPI, HTTPException
from fastapi.middleware.cors import CORSMiddleware
import mariadb, time, os, socket

# Create a FastAPI instance
app = FastAPI()

# Add CORS middleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"], # Allow from any domain
    allow_credentials=True, # Allow cookies and tokens
    allow_methods=["GET"], # Only allow GET requests
    allow_headers=["*"], # allow any header type
)

# Global database connection pool
pool = None

# Database credentials
user = os.getenv('MYSQL_USER')
password = os.getenv('MYSQL_PASSWORD')

# Database connection
@app.on_event("startup")
def database_connection():
    # Attemps to create a connection with database every 30 seconds 10 times
    global pool
    retries = 10
    for attempt in range(retries):
        try:
            pool = mariadb.ConnectionPool(user=user, password=password, host="app-m2-se-service-db", port=3306, database="m2-db-se", pool_name="DB", pool_size=5)
            print("Database connection established")
            return
        except Exception as e:
            print(f"Database connection failed: {e}. Retrying {attempt + 1}/{retries}")
            time.sleep(30)
    raise Exception("Database not reachable after retries")

# Disconnects database connection when API is turned off.
@app.on_event("shutdown")
def shutdown():
    if pool:
        pool.close()
        print("Database connection closed")
```

```python
# Disconnects database connection when API is turned off.
@app.on_event("shutdown")
def shutdown():
    if pool:
        pool.close()
        print("Database connection closed")

# API queries database for name
def query_name(query, parameters=None):
    if not pool:
        raise HTTPException(status_code=500, detail="Database connection failed")

    cursor = None
    result = None
    try:
        conn = pool.get_connection()
        cursor = conn.cursor(dictionary=True)
        cursor.execute(query, parameters)
        if query.strip().upper().startswith("SELECT"):
            # Fetch database results
            result = cursor.fetchall()
        conn.commit()
    except mariadb.Error as e:
        print("Error executing SQL query:", e)
        raise HTTPException(status_code=500, detail=str(e))
    finally:
        if cursor:
            cursor.close()
        if conn:
            conn.close()
    return result
```

```python
# API route to fetch name
@app.get("/api/user")
async def get_name():
    try:
        query = "SELECT Name FROM users"
        result = query_name(query)
        if not result:
            raise HTTPException(status_code=404, detail="No user found")
        return {"name": result[0]["Name"]}
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

# API route to fetch API container id
@app.get("/api/id")
async def get_id():
    try:
        return {"container_id": socket.gethostname()}
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

This script will create the API endpoints for our webserver to reach. It will also create a connection to the database and query the name stored. The result will be available on the `/api/user` endpoint.

The username and password of the database is stored as secrets in Kubernetes that will be put in environment variables. This script will use these environment variables to create the connection to the database. It prevents the credentials from being in plaintext in this script.

This script will also fetch the container ID of the current API and its result will be available on the `/api/id` endpoint.

CORS is an security middleware that allows us to deny requests if it doesn't meet our requirements. We configured it so any domain, HTTP(S) header type and any type of credentials are allowed and can reach the site. But we only allow GET requests to be send. Any other type of request will be blocked.

7 => Save and close the file.

3.3 Docker Hub

Now that all the configuration and Dockerfiles are created, we just have to create these images and push these to Docker Hub.

We will put these images on Docker Hub because Kubernetes doesn't allow the use of local images. This is because the cluster is separated into different nodes. For pods to work with images locally, each images needs to be on all nodes constantly. This not only takes up a lot of space, but is also hard to manage and maintain. To prevent this, Kubernetes doesn't allow this.

For these same reasons, we will push our images to our Docker account. When we deploy our stack on Kubernetes, it will pull the images from Docker Hub.

1=> Login to your Docker Hub account on the VM itself with the following command :

Docker login -u <username>

<enter password>

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant$ docker login -u seppeyckmans1@hotmail.com
Password:
WARNING! Your password will be stored unencrypted in /home/vagrant/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credential-stores

Login Succeeded
```

If you don't have an Docker Hub account, create one using the following link :

https://hub.docker.com/

2=> Build your FastAPI Docker image using the following command :

Docker build -t <your-DockerHub-username>/<image-name>:latest .

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant$ docker build -t seyckmans/fastapi-m2-se:latest .
```

(execute command in same directory as the Dockerfile!)

3=> Do the same for Lighttpd :

docker build -t <your-DockerHub-username/<image-name>:latest -f ./Dockerfile-web .

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant$ docker build -t seyckmans/lighttpd-m2-se:latest -f ./Dockerfile-web .
```

(execute command in same directory as the Dockerfile!)

4=> Push the FastAPI image to your Docker Hub :

Docker push <your-DockerHub-username/<image-name>:latest

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant$ docker push seyckmans/fastapi-m2-se:latest
```

5=> Same for Lighttpd :

Docker push <your-DockerHub-username/<image-name>:latest

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant$ docker push seyckmans/lighttpd-m2-se:latest
```

When both images are successfully pushed, We have completed building our web application stack! Now we can proceed with creating our cluster and deploying our stack onto it.

4 KUBERNETES

Creating the cluster and deploying our stack involves several important steps. First, let's focus on setting up the cluster itself before proceeding to create the resources required for deployment.

4.1 Cluster

1=> In you working directory, create a YAML file called “kindconfig.yaml” :

Touch kindconfig.yaml

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant$ touch kindconfig.yaml
```

2=> Open this file and write the following code :

```yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4 # Version of the kind config
name: m2-cluster-se
nodes:
- role: control-plane # Creates 1 controller/master node
  kubeadmConfigPatches: # Controller node configuration
  # Enable ingress support
  - |
    kind: InitConfiguration
    nodeRegistration:
      kubeletExtraArgs:
        node-labels: "ingress-ready=true"
  extraPortMappings: # Port mapping for port 80, 443 and 30306 (db external port)
  - containerPort: 80
    hostPort: 80
    protocol: TCP
  - containerPort: 443
    hostPort: 443
    protocol: TCP
  - containerPort: 30306
    hostPort: 30306
    protocol: TCP
- role: worker # Creates 2 worker nodes
- role: worker

```

This will create our cluster named “m2-cluster-se”, you can pick different name. Inside the cluster, it will create the following :

- 1 Controller/control plane node, this node manages all activities within the cluster.
- 2 worker nodes, these nodes will contain our web application stack.

Ingress is enabled on the controller node, allowing external access to the cluster. We also configure specific ports for traffic from outside the cluster to reach our stack: 80 for HTTP, 443 for HTTPS, and 30306 for the external database connection.

3=> Save and close the file.

4=> Create the cluster with the following command :

Kind create cluster –config=kindconfig.yaml

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant$ kind create cluster -config=kindconfig.yaml
```

(execute command inside the directory containing the YAML file!)

This process will take approximately 5 to 10 minutes, so it's perfectly fine if it takes some time. Once the cluster creation is complete, the first step in deploying our stack will be finished! Now, let's continue by creating all the necessary resources to deploy our application stack.

4=> Create an directory called “deployment” to store all resource YAML files.

Mkdir deployment

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant$ mkdir deployment
```

5=> Go inside this directory

4.2 Namespace

The first resource we are going to create is the Namespace. This is the isolated environment were our stack will be deployed and exist.

1=> create the following file and write the following code inside it :

Touch namespace.yaml

```
sepp-eyckmans:/vagrant/deployment$ touch namespace.yaml
```

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: app-m2-se-ns # Creates Namespace
```

2=> Save and close the file.

4.3 Secret

This resource will store all sensitive credentials used in our stack. This contains our MariaDB login and the API token used during the certificate generation step.

1=> create the following file and write the following code inside it :

Touch secret.yaml

```
sepp-eyckmans:/vagrant/deployment$ touch secret.yaml
```

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: app-m2-se-secret-logindb
  namespace: app-m2-se-ns
type: Opaque
data:
  root_password: cm9vdA==  # Base64 encoded value of root password
  username: c2VwcA== # Base64 encoded value of database user password
  password: ZXlja21hbnM=  # Base64 encoded value of database username

---

apiVersion: v1
kind: Secret
metadata:
  name: app-m2-se-secret-cloudflare-api-token
  namespace: cert-manager # API key stored in cert-manager namespace
type: Opaque
data:
  api-token: SWlkRV94UU13WmZTNTdWVzJSU01hSzhLRmVkUnRUbDVSMjFkc0l4cg== # Base64 encoded value of Cloudflare API token
```

For extra security, all sensitive credentials here are encoded in Base64 instead of plaintext. This approach helps maintain the integrity of the security measures in place. Otherwise it won't serve a point doing this in the first place.

Use the following command to encode each sensitive credential and Cloudflare API token :

echo -n "<value_to_encode>" | base64

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant/deployment$ echo -n "sepp" | base64
c2VwcA==
```

Replace mine with your encoded credentials in the resource file.

2=> Save and close the file.

4.4 PVC

This is the PersistentVolumeClaim (PVC) resource. Due to complexity and synchronization limitations, we can't horizontally scale the database like we do with the API and webserver. However, to ensure redundancy, we implement persistence for the database data. This way, if the database fails, the data remains accessible on the VM itself. When the database pod automatically restarts through Kubernetes, the data is preserved and still available.

1=> create the following file and write the following code inside it :

Touch pvc.yaml

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant/deployment$ touch pvc.yaml
```

```yaml
---

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: app-m2-se-pvc-db
  namespace: app-m2-se-ns
spec:
  accessModes:
    - ReadWriteOnce # Volume can be accessed by one node at a time
  resources:
    requests:
      storage: 5Gi # 5 GB Storage size
  storageClassName: standard # Persistent volume is a local volume storage device
```

The PVC will make space on the local VM storage and use 5GB of it for the database data. Only 1 node can reach the persistent volume at a time. We will bind the volume to the container in the deployment file.

2=> Save and close the file.

4.5 Service

The Service resource is responsible for assigning each application pod an internal IP, enabling communication and collaboration within the namespace. It also facilitates load balancing, ensuring smooth traffic distribution across horizontally scaled pods. Additionally, the Service resource allows pods to connect to the Ingress resource, making them accessible from outside the cluster.

1=> create the following file and write the following code inside it :

Touch service.yaml

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant/deployment$ touch service.yaml
```

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: app-m2-se-service-web # light webserver service
  namespace: app-m2-se-ns
spec:
  type: ClusterIP
  selector:
    app: app-m2-se-light
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80

---

apiVersion: v1
kind: Service
metadata:
  name: app-m2-se-service-api # fast api service
  namespace: app-m2-se-ns
spec:
  type: ClusterIP
  selector:
    app: app-m2-se-api
  ports:
    - protocol: TCP
      port: 3000
      targetPort: 3000

---
```

```yaml
---

apiVersion: v1
kind: Service
metadata:
  name: app-m2-se-service-db # db service
  namespace: app-m2-se-ns
spec:
  type: NodePort # opens a port on the node to directly access the db pod
  selector:
    app: app-m2-se-db
  ports:
    - protocol: TCP
      port: 3306
      targetPort: 3306
      nodePort: 30306 # port on node

```

Every application pod is connected to its own service. If the pod is horizontally scaled, they both fall under 1 service, load balancing traffic between them.

- The webserver container and service will listen on port 80.
- The API container and service will listen on port 3000.
- The database container and service on port 3036 and a NodePort is exposed on port 30306 of the worker node.

The database, on top of getting an internal IP, also is assigned an NodePort. Port 30306 on the worker node is exposed and directly mapped to the database's internal port, enabling access from outside the cluster. The database is secured through a combination of HTTPS-encrypted traffic and credential-based authentication, ensuring that the exposed port cannot be exploited by malicious actors.

2=> Save and close the file.

4.6 Ingress

The Ingress resource is responsible for establishing a connection between all services and the Ingress Controller, making the associated pods accessible from outside the cluster.

The Ingress Controller is a vital component responsible for processing Ingress resources and routing external HTTP and HTTPS traffic to the appropriate services within the cluster. It acts as a reverse proxy, handling load balancing, SSL/TLS encryption, SSL/TLS termination, and routing for the exposed services. The Ingress Controller is a pod running on the Controller node that serves as the entry point for all traffic coming from outside the cluster.

The Ingress Controller components is not pre-installed by default! We must first install this component before we can actually use the Ingress resource.

1=> Install the NGINX Ingress Controller using the following command :

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

When you have control over your command prompt, we can start creating the Ingress resource.

2=> create the following file and write the following code inside it :

Touch ingress.yaml

```yaml
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-m2-se-ingress
  namespace: app-m2-se-ns
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"  # Ensures HTTP is redirected to HTTPS
spec:
  tls: # TLS/HTTPS encryption
  - hosts:
    - m2.seyckmans.be
    secretName: app-m2-se-secret-tls # TLS certificate stored in secret resource
  rules:
    - host: m2.seyckmans.be # Webserver ingress
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: app-m2-se-service-web
                port:
                  number: 80
    - host: m2.seyckmans.be # API ingress
      http:
        paths:
          - path: /api
            pathType: Prefix
            backend:
              service:
                name: app-m2-se-service-api
                port:
                  number: 3000

```

Ingress assigns the following routing rules :

- Traffic directed to *m2-seyckmans.be/* is redirected to the webserver service on port 80
- Traffic directed to *m2-seyckmans.be/api* is redirected to the API service on port 3000
- All HTTP traffic is redirected to HTTPS automatically to ensure secure communication at all times.
- All traffic under the *m2-seyckmans.be* domain is securely encrypted with HTTPS TLS encryption and is covered by the TLS certificate.

The TLS certificate is securely stored and encoded inside a Secret resource.

3=> Save and close the file.

4.7 Deployment

The Deployment resource creates the application stack pods and connects them to the previously created namespace, services, persistent volumes, and secret volumes.

It also manages underlying resources that enable automatic pod restarts, as well as updates to the pods when changes are made to the stack or the Docker images in Docker Hub. The automated management, updating, and maintenance of our stack are key features of Kubernetes, showcasing its efficiency and why we use it for our project.

1=> create the following file and write the following code inside it :

Touch deployment.yaml

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant/deployment$ touch deployment.yaml
```

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-m2-se-deployment-light
  namespace: app-m2-se-ns
spec:
  replicas: 2 # Horizontal scaling of webserver pod
  selector:
    matchLabels:
      app: app-m2-se-light
  template:
    metadata:
      labels:
        app: app-m2-se-light
    spec:
      containers:
        # Webserver container
        - name: lighttpd-webserver-se
          image: seyckmans/lighttpd-m2-se:latest
          ports:
            - containerPort: 80
              name: http # port 80 has the alias 'http'


---

apiVersion: apps/v1
kind: Deployment
```

```yaml
---

apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-m2-se-deployment-api
  namespace: app-m2-se-ns
spec:
  replicas: 2 # Horizontal scaling of API pod
  selector:
    matchLabels:
      app: app-m2-se-api
  template:
    metadata:
      labels:
        app: app-m2-se-api
    spec:
      containers:
        # FastAPI container
        - name: fastapi-se
          image: seyckmans/fastapi-m2-se:latest
          ports:
            - containerPort: 3000
          env:
            - name: PYTHONUNBUFFERED
              value: "1"
            - name: MYSQL_USER # Username value fetched from secret resource
              valueFrom:
                secretKeyRef:
                  name: app-m2-se-secret-logindb
                  key: username
            - name: MYSQL_PASSWORD # Username password value fetched from secret resource
              valueFrom:
                secretKeyRef:
                  name: app-m2-se-secret-logindb # The name of the secret
                  key: password # The key to use from the secret

---

apiVersion: apps/v1
kind: Deployment
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-m2-se-deployment-db
  namespace: app-m2-se-ns
spec:
  selector:
    matchLabels:
      app: app-m2-se-db
  template:
    metadata:
      labels:
        app: app-m2-se-db
    spec:
      containers:
        # MariaDB container
        - name: db-se
          image: mariadb:latest
          ports:
            - containerPort: 3306
          env:
            - name: MYSQL_ROOT_PASSWORD # Root password value fetched from secret resource
              valueFrom:
                secretKeyRef:
                  name: app-m2-se-secret-logindb
                  key: root_password
            - name: MYSQL_USER # Username value fetched from secret resource
              valueFrom:
                secretKeyRef:
                  name: app-m2-se-secret-logindb
                  key: username
            - name: MYSQL_PASSWORD # Username password value fetched from secret resource
              valueFrom:
                secretKeyRef:
                  name: app-m2-se-secret-logindb
                  key: password
            - name: MYSQL_DATABASE
              value: m2-db-se
          volumeMounts:
            - name: db-data
              mountPath: /var/lib/mysql # Persistent volume for db
      volumes:
          - name: db-data
            persistentVolumeClaim:
              claimName: app-m2-se-pvc-db # Links to PVC db

```

=> Lighttpd : 2 horizontally scaled Lighttpd webservers are made, each on a separate worker node. These use the Lighttpd image we created and pushed to DockerHub. Port 80 is exposed internally within the container.

=> FastAPI : 2 horizontally scaled FastAPI servers are created, each on a separate worker node. They use the FastAPI image we created and pushed to DockerHub. Port 3000 is exposed internally within the container. FastAPI environment variables are set to facilitate container setup, including a database login retrieved from a secret volume for database connection in the Python application file.

=> MariaDB : A single MariaDB instance is deployed on a worker node using the official MariaDB Docker image. Port 3306 is exposed internally within the container. Database login credentials, including the root password and default user, are retrieved from a secret volume. Environment variables are set to create the default user, root password, and database during deployment. A persistent volume is referenced from a PVC resource and bound to the MariaDB container, ensuring data inside the */var/lib/mysql* directory is stored.

2=> Save and close the file.

All resources required for deploying our stack have been successfully created! The final step before deployment is to create both our certificate generation resource and certificate signing resource for HTTPS encryption.

5 TLS ENCRYPTION

TLS encryption with certificates is handled differently in this project compared to the previous one. We will utilize the cert-manager tool, which deploys an ACME server pod within the Kubernetes cluster, available across all namespaces. With cert-manager, we can generate a certificate and submit a certificate signing request (CSR) to the ACME server. The ACME server then communicates with the Let's Encrypt Certificate Authority (CA) to have the certificate signed. Once signed, the certificate is encoded and stored in a Secret volume (similar to our database login credentials). This secret can then be referenced in our Ingress resource for use in securing our web application traffic.

Cert-manager is not pre-installed on the cluster by default. So we will use HELM to install a HELM chart for deploying the cert-manager component in our cluster.

1=> Set the cluster context to our cluster. Otherwise HELM won't know the cluster to install it to :

kubectl config use-context kind-m2-cluster-se

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant/deployment$ kubectl config use-context kind-m2-cluster-se
```

2=> Add the cert-manager repo to our cluster :

helm repo add jetstack https://charts.jetstack.io

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant/deployment$ helm repo add jetstack https://charts.jetstack.io
```

3=> Update HELM with its new repo :

helm repo update

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant/deployment$ helm repo update
```

4=> Install cert-manager from the newly added repo :

helm install cert-manager jetstack/cert-manager --namespace cert-manager --create-namespace --version v1.13.2 --set installCRDs=true

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant/deployment$ helm install cert-manager jetstack/cert-manager --namespace cert-manager --c
reate-namespace --version v1.13.2 --set installCRDs=true
```

Installing Cert-manager will take a couple of minutes. When you have control over your command prompt again, Cert-manager is done installing.

Next, we need to create two key resources:

- certificate-issuer.yaml : This resource configures the ACME server, making it available to all namespaces within the cluster. It defines the issuer responsible for signing the certificates.

- certificate.yaml : This resource generates a certificate request and submits it to the ACME server. Once the request is signed, the resulting certificate is stored as a Secret resource, ready for use in our applications.

5.1 Cluster Issuer

1=> Create the following file and write the following code inside it :

Touch cluster-issuer.yaml

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant/deployment$ touch cluster-issuer.yaml
```

```yaml
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: app-m2-se-certificate-issuer
spec:
  acme: # Sends certificate to Let's Encrypt ACME server
    server: https://acme-v02.api.letsencrypt.org/directory  # ACME server
    email: seppeyckmans1@gmail.com # email used to register certificate
    privateKeySecretRef:
      name: letsencrypt-account-key
    solvers:
    - dns01: # Challenge issuer to verify domain ownership
        cloudflare:
          apiTokenSecretRef: # DNS API token stored in secret resource
            name: app-m2-se-secret-cloudflare-api-token
            key: api-token
```

This resource will create the ACME server pod in the cluster. It defines the issuer responsible for signing the certificates. It will use our Cloudflare DNS for DNS-01 verification to ensure our domain is valid.

2=> Save and close the file.

5.2 Certificate

1=> Create the following file and write the following code inside it :

Touch certificate.yaml

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant/deployment$ touch certificate.yaml
```

```yaml
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: app-m2-se-certificate
  namespace: app-m2-se-ns
spec:
  secretName: app-m2-se-secret-tls # Saves created & signed certificate in this secret resource
  issuerRef:
    name: app-m2-se-certificate-issuer # Reference to issuer resource (ACME server) for certificate signing
    kind: ClusterIssuer
  dnsNames:
  - m2.seyckmans.be # domain name to create certificate for
```

It will generate resource and send it to our ACME server. The trusted certificate will be stored in the secret resource.

2=> Save and close the file.

With all preparations complete, it's time to deploy our stack!

6 DEPLOYING APPLICATION STACK

1=> Inside our resource directory, execute the following command to deploy the stack :

kubectl apply -f namespace.yaml -f cluster-issuer.yaml -f secret.yaml -f pvc.yaml -f service.yaml -f ingress.yaml -f deployment.yaml -f certificate.yaml

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant/deployment$ kubectl apply -f namespace.yaml -f cluster-issuer.yaml -f secret.yaml -f pvc
.yaml -f service.yaml -f ingress.yaml -f deployment.yaml -f certificate.yaml
```

It is crucial that our resources are deployed in a specific order to avoid potential issues, as some resources depend on others to function properly.

The resources will be deployed in the following sequence:

- Namespace.yaml
- Cluster-issuer.yaml
- Secret.yaml
- Pvc.yaml
- Service.yaml
- Ingress.yaml
- Deployment.yaml
- Certificate.yaml

It will around 5 minutes for our stack to be up and running. Use the following command to verify if the stack is ready :

Kubectl get pod -n app-m2-se-ns

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant/deployment$ kubectl get pod -n app-m2-se-ns
NAME                                          READY   STATUS    RESTARTS   AGE
app-m2-se-deployment-api-d9cb649bd-hwx7p      1/1     Running   0          4h20m
app-m2-se-deployment-api-d9cb649bd-twlmt      1/1     Running   0          4h20m
app-m2-se-deployment-db-cb8486ccd-64qw2       1/1     Running   0          4h20m
app-m2-se-deployment-light-666589979f-jslzk   1/1     Running   0          4h20m
app-m2-se-deployment-light-666589979f-zbdw6   1/1     Running   0          4h20m
```

Before we begin celebrating and start browsing to our web application, we first have to manually create our database data for our stack to properly function!

6.1 MariaDB

1=> Use to following command to connect to our database using the MariaDB Client :

mysql -h 192.168.56.50 -P 30306 --protocol=tcp -u root -p

<enter root password chosen>

```
vagrant@m2-kubeserver-sepp-eyckmans:/vagrant/deployment$ mysql -h 192.168.56.50 -P 30306 --protocol=tcp -u root -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 14
Server version: 11.6.2-MariaDB-ubu2404 mariadb.org binary distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

(If you get an connection error, it means the database is still initializing.
Wait a minute and retry login again.)

We want to try connection to our database using the VMs IP address and the external database port. We want to login here as root with the root password.

2=> Enter the following sets of commands into the database :

USE m2-db-se;
=> Enter the “m2-db-se” database.
GRANT SELECT ON `m2-db-se`.* TO 'sepp'@'%';
=> Change the permissions of your database user to read only.
CREATE TABLE users (Name varchar(255));
=> Create the “users” table with the “Name” column.
INSERT INTO users VALUES ('Sepp Eyckmans');
=> Create a record on the “users” table called “Sepp Eyckmans”.

```
MariaDB [(none)]> USE m2-db-se;
Database changed
MariaDB [m2-db-se]> GRANT SELECT ON `m2-db-se`.* TO 'sepp'@'%';
MariaDB [m2-db-se]> CREATE TABLE users (Name varchar(255));
Database changed
MariaDB [m2-db-se]> INSERT INTO users VALUES ('Sepp Eyckmans');
```

We have now successfully build a web application stack on Kubernetes!

7 RESULT

Sepp Eyckmans has reached milestone 2!

Container ID : app-m2-se-deployment-api-d9cb649bd-hwx7p

Sepp Eyckmans has reached milestone 2!

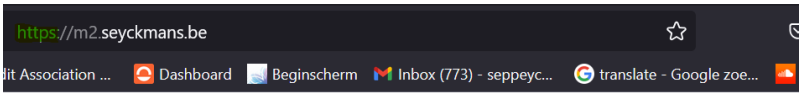
Container ID : app-m2-se-deployment-api-d9cb649bd-twlmt

(Load balancing between API's)

Sepp has reached milestone 2!

Container ID : app-m2-se-deployment-api-d9cb649bd-hwx7p

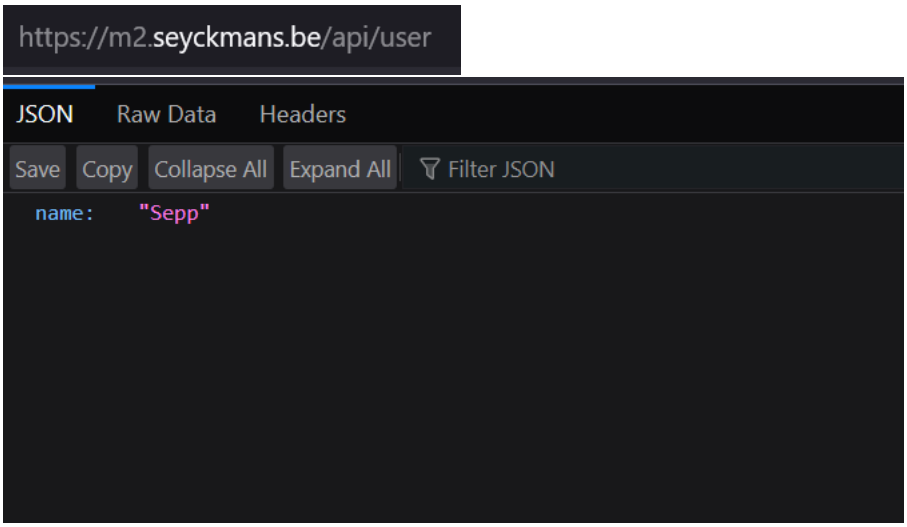
(Change to database name automatically updates after refresh)

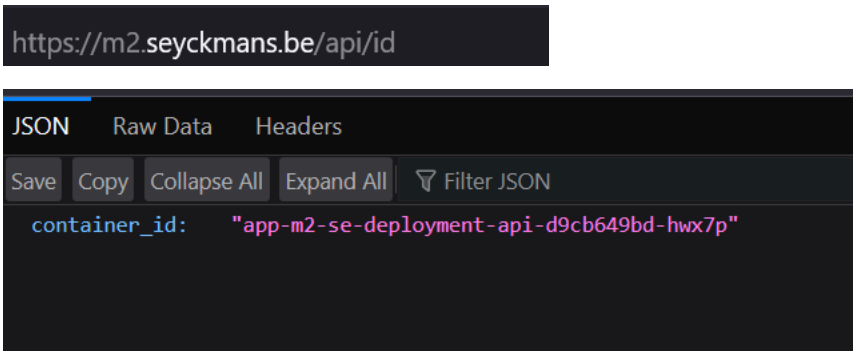


d milestone 2!

ıpp-m2-se-deployment-api-d9cb649bd-hwx7p

(HTTPS on all public web pages)





(API with name data and container ID both on separate endpoints)

CONCLUSION

During this project, I gained invaluable knowledge and hands-on experience with Kubernetes, which will be essential for my future career in IT. Through this documentation, I aim to demonstrate my ability to apply these newly acquired skills in upcoming projects and professional opportunities.

Finally, I would like to sincerely thank Jeroen Verbruggen for introducing me to Kubernetes and guiding me throughout the learning process. Your insightful teaching and support have been invaluable in helping me understand the complexities of Kubernetes and master its key concepts. Thanks to your help, I have gained the skills necessary to apply Kubernetes effectively in upcoming and even future real-world projects.