INTRO TO DEVOPS

Hervé Valentin

Algebra University, June 2024.

Contents

1. Abstract

The main goal of this project is to demonstrate how to deploy and containerize a modern web application using two different container orchestration systems. The jstanestic's TempConverter web application ([1]https://github.com/jstanesic/tempconverter), which connects to a MySQL 8 database, is containerized and deployed using both a simpler orchestration system and a more complex system. This process includes creating container images, configuring environment variables for database connectivity, and setting up load balancing and scaling. The project highlights the steps and considerations necessary for deploying a scalable web app in different orchestration environments.

2. Introduction

This project focuses on containerizing, and deploying the TempConverter web application, enhancing understanding of both containers and microservices, and the creation of scalable modern websites. The TempConverter app requires a MySQL 8 database and specific environment variables for configuration.

Initially, a container image for the app is created, updated, and configured to expose port 5000, ensuring it connects to the database as a non-root user. The image is pushed to a container registry with tags for both the latest and development versions. Deployment is tested locally using Podman, including the database container to ensure connectivity.

Two container orchestration systems are selected for deployment: one simpler (Docker Swarm) and one more complex (Kubernetes with Minikube). Templates and scripts are created for each system to manage deployment, load balancing, and scaling. The project includes deploying two app instances and one database instance, ensuring app replicas are on separate nodes, and documenting the scaling process.

A comparative reflection on the orchestration systems concludes the project, evaluating their effectiveness and suitability for different scenarios.

The use of many resources has been required in order to solve this project. The main resources used were as follows:

- Introduction to DevOps course - Algebra University, for an overall understanding of what containers and orchestration systems are in DevOps.
- [2]RedHat Academy DO180 course - Operating a Production Cluster, as an introduction to administering an OpenShift cluster with Kubernetes workloads.
- [3]docker.docs, for general help in using docker and docker swarm.
- [4]Minikube documentation, for an easier deployment of Kubernetes

3. Implementation

3.1. Environment

To solve this project, I used a virtual machine running CentOS 9 Stream. Here's the version of the tools I used:

- Podman: version 5.0.2
- Docker: version 26.1.4
- Minkube: version 1.33.1
- Kubectl: uses of Minikube kubectl – version 1.30.0

3.2. Setup

Firstly, I forked the tempconverter git repository found on [1]https://github.com/jstanesic/tempconverter in order to be able to push on my own github account the configuration files I will use throughout the project. Here is the link to my github repository: [5]https://github.com/Vazelek/tempconverter/tree/devops-project. I finally cloned the repository on my virtual machine (VM).

3.3. Deploy the app locally

The first step of this project is to deploy the application locally on the VM.

3.3.1. Create container image

The first objective is to create an image of the application that exposes port 5000 TCP, install all required requirements based on the requirements.txt file of this repository, update all packages and configure the correct command to start the flask application. All these steps are done with a single [(1)]Dockerfile, which will create the image for us. Let's go through this [(1)]Dockerfile steps by steps:

```
FROM python:3.9-slim
```

This first line sets the base image for the Docker container to python:3.9-slim, which is a minimal image with Python 3.9 installed. The "slim" variant is a smaller image, which is beneficial for reducing the size of the final container. The image resulting from this [(1)]Dockerfile will then be based on the python 3.9 image, which is required in order to run the flask application.

```
WORKDIR /app
```

This sets the working directory inside the container to /app. All the following commands will be executed relative to this directory.

```
COPY . .
```

This copy the contents of the current directory on the host machine (where the [(1)]Dockerfile is located) into the /app directory in the container.

```
RUN apt-get update && apt-get upgrade -y && \
    pip install --no-cache-dir -r requirements.txt
```

The first line of this command updates all packages in the image, and the second installs the Python packages listed in the requirements.txt file using pip. The --no-cache-dir option prevents pip from using its cache, reducing the image size.

```
EXPOSE 5000
```

This makes port 5000 available for connections from outside the container. It indicates that the application inside the container will listen for network requests on this port. This will allow us to connect the application using localhost:5000.

```
ENV DB_USER=db_user
ENV DB_PASS=dv_password
ENV DB_HOST=tempconverter-db
ENV DB_NAME=tempconverter_db
ENV STUDENT="Valentin Hervé"
ENV COLLEGE=Algebra
```

These lines set environment variables inside the container. They can be accessed by the application running inside the container. They are used to connect the mysql database and print some data in the web app.

```
CMD ["python", "app.py"]
```

Finally, this last line specifies the command to run when the container starts. In this case, it runs python app.py, which will start the application defined in the app.py file.

By running the following command in the directory where the [(1)]Dockerfile is located (tempconverter/), it will create the container image with the required specifications and the tag ‘latest’:

```
podman build -t docker.io/vherve/tempconverter:latest .
```

3.3.2. Push image to docker hub

Now that we created an image, we want to push it to docker hub. This is done in two simple commands:

```
podman login docker.io
```

The first one is used to login to my docker hub account

```
podman push docker.io/vherve/tempconverter:latest
```

This second command push the image docker.io/vherve/tempconverter:latest to my docker hub.

3.3.3. Update HTML title and push image with new tag

The goal is now to update something in the app we want to containerize (the title of the web app) and to push the updated image with a new tag. We first need to update the web application title running the following command and update the <title> tag:

```
vim templates/index.html
```

Then we need to build a new image with the Dockerfile with the tag 'dev':

```
podman build -t docker.io/vherve/tempconverter:dev .
```

And finally push this new image to docker hub:

```
podman push docker.io/vherve/tempconverter:dev
```

3.3.4. Deploy the application locally using Podman

Now that we have our image built for the tempconverter web application, we want to deploy it using Podman. We also want to deploy the database container and ensure the app container connects to it.

In order to do it, we first need to create a pod as follows:

```
podman pod create --name tempconverter-pod -p 5000:5000 -p 3306:3306
```

A pod is a group of one or more containers that share the same network namespace, allowing the tempconverter web app and the MySQL database to communicate seamlessly via localhost. This command creates a pod, mapping the port 5000 on the host to the port 5000 in the pod for the web application, and the same for the MySQL database with the port 3306.

Then the next steps are to run both the MySQL and the tempconverter container, and connect them with the pod we just created:

```
podman run -d --pod tempconverter-pod --name tempconverter-db \
  -e MYSQL_ROOT_PASSWORD=password \
  -e MYSQL_DATABASE=tempconverter_db \
  -e MYSQL_USER=db_user \
  -e MYSQL_PASSWORD=db_password \
  mysql:8
```

This command runs the ‘mysql:8’ container (from docker hub) as ‘tempconverter-db’ inside the ‘tempconverter-pod’ pod and set environment variables that will be used by MySQL to create a database ‘tempconverter_db’ and create a user and a password that will be required to connect to the database.

```
podman run -d --pod tempconverter-pod --name tempconverter-web \
  -e DB_USER=db_user \
  -e DB_PASS=db_password \
  -e DB_HOST=tempconverter-db \
  -e DB_NAME=tempconverter_db \
  -e STUDENT="Valentin Hervé" \
  -e COLLEGE=Algebra \
  docker.io/vherve/tempconverter:latest
```

This command is similar to the previous one and runs the ‘docker.io/vherve/tempconverter:latest’ container as ‘tempconverter-web’ inside the ‘tempconverter-pod’ pod and set environment variables that will be used by the app in order to connect to the database and display some data in the web application.

3.3.5. Test local deployment

Now that the containers are deployed, we can check with podman if they run:

```
podman ps
```

```
[student@localhost ~]$ podman ps
CONTAINER ID  IMAGE                                COMMAND         CREATED       STATUS        PORTS                                                 NAMES
bb976e536d3d  localhost/podman-pause:5.0.2-1715074917                24 hours ago  Up 8 seconds  0.0.0.0:3306->3306/tcp, 0.0.0.0:5000->5000/tcp  601f54f1801b
-infra
c518ad48708e  docker.io/library/mysql:8            mysqld          24 hours ago  Up 8 seconds  0.0.0.0:3306->3306/tcp, 0.0.0.0:5000->5000/tcp  tempconverte
r-db
7dcbee0d19e4  docker.io/vherve/tempconverter:latest  python app.py   24 hours ago  Up 3 seconds  0.0.0.0:3306->3306/tcp, 0.0.0.0:5000->5000/tcp  tempconverte
r-web
```

Figure 1. Result of ‘podman ps’ command demonstrating successful application start-up

We can see the status “Up” for the mysql and tempconverer containers, meaning that they are running well. We can then test the connection to the web application by connecting to http://localhost:5000:

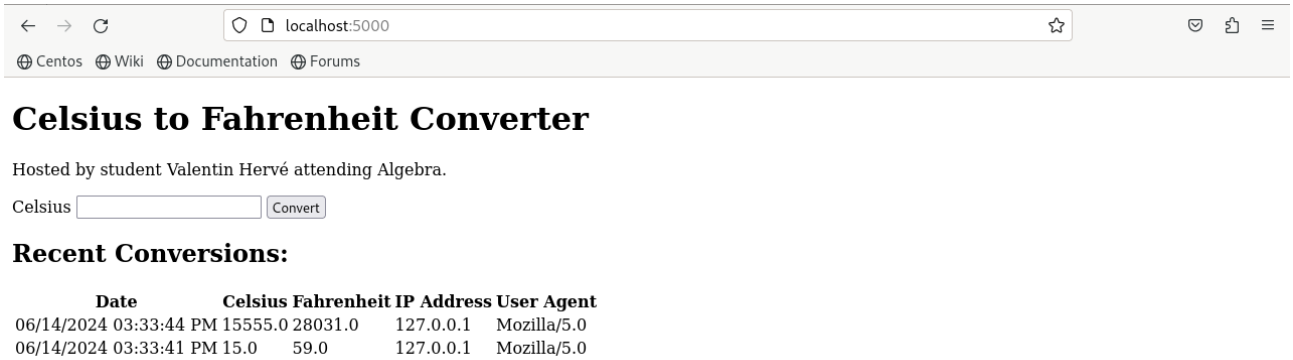

Figure 2. Demonstration of the correct operation of the locally deployed application

We can see here that we can successfully connect to the app at port 5000. We can also see that the application successfully connects to the MySQL database by sending a value in degrees Celsius and observing its display converted to degrees Fahrenheit in the table at the bottom.

3.4. Deploy the app using a simple orchestration system – Docker Swarm

[6]Docker Swarm is a native clustering and orchestration tool for Docker containers, allowing us to manage a group of Docker engines as a single virtual system for high availability and scalability. It can be used to deploy the tempconverter app and the MySQL database to ensure automatic load balancing, fault tolerance and easy scaling of the services.

3.4.1. Create template for Docker Swarm deployment

To facilitate deployment with docker swarm, we're going to create a [(2)].yml docker compose file that will enable us to deploy the application with swarm in just a few commands. The application thus created must be accessible from port 80, and there must be one instance of the database and two instances of the application.

The [(2)]docker compose file first consists of defining all the individual services that make up the application. In our case, there are two services: tempconverter-db (for the MySQL database) and tempconverter-web (for the web application):

```
services:
  tempconverter-db:
    image: mysql:8
    environment:
      MYSQL_ROOT_PASSWORD: passsword
      MYSQL_DATABASE: tempconverter_db
      MYSQL_USER: db_user
      MYSQL_PASSWORD: db_password
    volumes:
      - tempconverter-db-data:/var/lib/mysql
    networks:
      - tempconverter-network
    deploy:
      replicas: 1
      update_config:
        parallelism: 1
        delay: 10s
      restart_policy:
        condition: on-failure
```

These lines are the definition of the service ‘tempconverter-db’ that uses the image mysql:8. We set some environment variables as before in order to create the MySQL database and a user to connect to it. The volume section means that we mount a Docker volume named tempconverter-db-data to the MySQL data directory for persistent storage. Then we connect the service to the network ‘tempconverter-netxork’ that we will create later. Finally, we set some parameters as follows:

- ‘replicas: 1’: Only one replica of this service should run
- ‘parallelism: 1’: Update one task at a time
- ‘delay: 10s’: Wait 10 seconds between updates to different tasks
- ‘condition: on-failure’: Restart the container if it fails

The ‘tempconverter-web’ service is defined in the same way.

```
  tempconverter-web:
    image: docker.io/vherve/tempconverter:latest
    ports:
      - "80:5000"
    environment:
      DB_USER: db_user
      DB_PASS: db_password
      DB_HOST: tempconverter-db
      DB_NAME: tempconverter_db
      STUDENT: "Valentin Hervé"
      COLLEGE: Algebra
    depends_on:
      - tempconverter-db
    networks:
      - tempconverter-network
    deploy:
      replicas: 2
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure
```

The differences are:

- The image used (docker.io/vherve/tempconverter:latest),
- 'ports: 80:5000' maps port 80 on the host to port 5000 in the container, making the web application accessible via port 80 on the host machine (via http://localhost:80),
- The environment variables, that are defined as before in order to connect to the database and display some data in the web application,
- 'depends_on: tempconverter-db' ensures that the tempconverter-db service is started before the tempconverter-web service,
- The replicas and parallelism are set to 2 in order to have two replicas of this service that run at the same time.

We now need to create the network as follows:

```
networks:
  tempconverter-network:
    driver: overlay
```

The lines define a network named tempconverter-network using the overlay driver, which is suitable for multi-host networking and used by Docker Swarm.

Finally, in order to have a persistent storage of the MySQL database we need to add the following lines:

```
volumes:
  tempconverter-db-data:
```

These last lines define a docker volume named tempconverter-db-data allowing to have a persistent storage for the MySQL data.

3.4.2. Deploy the app with Docker Swarm

Deploying the app now only result in one command:

```
docker stack deploy -c docker-compose-swarm.yml tempconverter
```

Successful application deployment can be observed using the following command:

```
docker service ls
```

```
[student@localhost tempconverter]$ sudo docker service ls
ID             NAME                                MODE         REPLICAS   IMAGE                          PORTS
5ynk3wz9b69w   tempconverter_tempconverter-db      replicated   1/1        mysql:8
dx5z5lybq392   tempconverter_tempconverter-web     replicated   2/2        vherve/tempconverter:latest    *:80->5000/tcp
```

Figure 3. Result of 'docker service ls' command demonstrating successful application start-up

The column 'REPLICAS' tells us that the application has successfully started. We do have 1/1 replica of the database, and 2/2 replicas of the applications that are running. We can then test the connection to the web application by connecting to http://localhost:80.

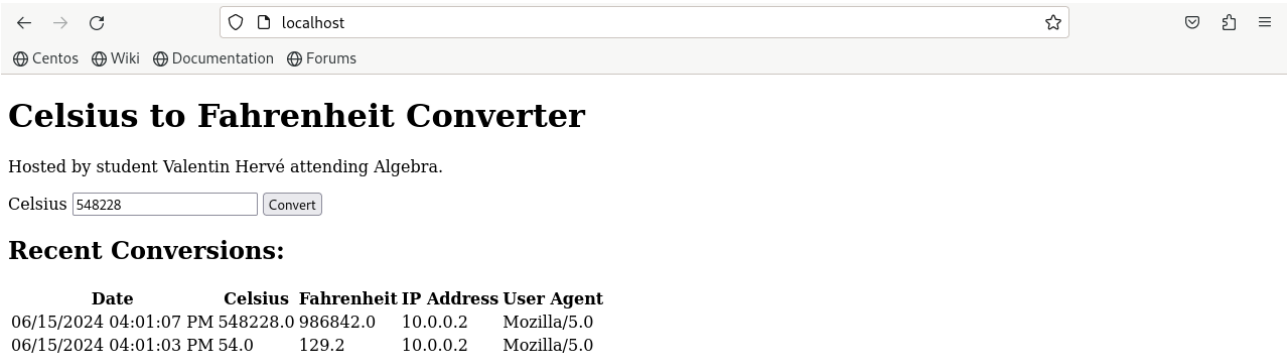

Figure 4. Demonstration of the correct operation of the docker swarm deployed application

We can once more see here that we can successfully connect to the app at port 80 (not displayed as it is the default port). We can also see that the application successfully connects to the MySQL database by sending a value in degrees Celsius and observing its display converted to degrees Fahrenheit in the table at the bottom.

Scaling the application with a different number of replicas can be done with the following command:

```
docker service scale tempconverter_tempconverter-web=3
```

We can see that we have now 3 replicas of the web app by running:

```
docker service ls
```

```
[student@localhost tempconverter]$ sudo docker service ls
ID             NAME                                MODE         REPLICAS   IMAGE                          PORTS
5ynk3wz9b69w   tempconverter_tempconverter-db      replicated   1/1        mysql:8
dx5z5lybq392   tempconverter_tempconverter-web     replicated   3/3        vherve/tempconverter:latest    *:80->5000/tcp
```

Figure 5. Result of 'docker service ls' command demonstrating successful application rescaling

3.5. Deploy the app using a complex orchestration system – Kubernetes (Minikube)

[7]Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications.

[4]Minikube quickly sets up a local Kubernetes cluster on macOS, Linux, and Windows. It is specifically designed to help application developers and newcomers to Kubernetes. Instead of having to install every needed components for kubernets and modifying the configuration files, minikube will do it for us in a few commands.

3.5.1. Create template for Kubernetes deployment

Kubernetes templates are quite similar to Docker Swarm compose template. To make things easier to understand, I've divided these files into three parts:

[3]mysql-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: tempconverter-db
spec:
  ports:
  - port: 3306
    name: mysql
  selector:
    app: mysql
```

This Service exposes the MySQL database on port 3306, allowing other pods to access it using the service name ‘tempconverter-db’. The service will be exposed on port 3306. The selector tag identifies the pods that this service targets, which are in our case all pods that match the name ‘app: mysql’.

[4]mysql-statefulset.yaml

This StatefulSet ensures that a MySQL database instance is deployed with persistent storage, maintaining the state even if the pod restarts. We associate the StatefulSet with the Service 'tempconverter-db' and set the number of replicas to 1. We set a selector as we did for the service to matche all pods with the label 'app: mysql'. We also define the container specifications: the container will be called 'mysql', it will use the 'mysql:8' image and listen to the port 3306. Some environment variables as set as before in order to create the database and an user. Finally, we mount a persistent storage to ensure data are retained even if the pod restarts, using a PersistentVolumeClaim template that requests 1Gi of storage.

[5]tempconverter-deployment.yaml

Finally, this Deployment ensures that two replicas of the tempconverter web application are deployed, with environment variables configured to connect to the MySQL database. In this file, we therefore set the number of replicas to 2, we call the app ‘tempconverter’, we use the image ‘docker.io/vherve/tempconverter:latest’ which we pull from docker hub if it is not present. We tell the container to listen on port 5000 and we set the same environment variables as before to connect to the MySQL database and print some data in the web application. By setting ‘DB_HOST’ to ‘tempconverter-db’, it connects the tempconverter web application to the MySQL database. The accompanying Service defined at the end of the file exposes the application on port 80, forwarding traffic to port 5000 in the pods, and uses a load balancer to distribute incoming traffic.

3.5.2. Deploy the app with Kubernetes

As said before, in order to deploy our application with Kubernetes, we will use Minikube to make it easier. The first step is to install Minikube:

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-
linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-
linux-amd64
```

Then we need to start Minikube:

```
minikube start
```

If the user you are using is not in the sudoers, add it by adding this line:

```
<your username> ALL=(=ALL) NOPASSWD: /usr/bin/podman
```

in the file opened with the command:

```
sudo visudo
```

It will allow to use podman without sudo, and it is required by Minikube. Then try again to start Minikube.

Starting Minikube will install kubectl if it doesn't exist, so we don't have to modify some configuration files and install it manually, which makes installation much easier. In fact, Minikube install a custom kubectl that will be run if we create this alias:

```
alias kubectl="minikube kubectl --"
```

Then run this command to use Minikube's Docker deamon:

```
eval $(minikube docker-env)
```

Ensure the tempconverter image is built in docker:

```
docker build -t tempconverter:latest .
```

This command will use our firs Dockerfile to build the image. Then finally run these three commands:

```
kubectl apply -f mysql-statefulset.yaml
kubectl apply -f mysql-service.yaml
kubectl apply -f tempconverter-deployment.yaml
```

This will use our templates to deploy the MySQL database and the 2 replicas of the web application. To verify if everything works, we can run the following command:

```
minikube kubectl -- get pods # or kubectl get pods
```

```
[student@localhost tempconverter]$ minikube kubectl -- get pods
NAME                             READY   STATUS    RESTARTS       AGE
mysql-0                          1/1     Running   3 (139m ago)   27h
tempconverter-6467489b6c-pbfw8   1/1     Running   7 (136m ago)   27h
tempconverter-6467489b6c-qskkg   1/1     Running   7 (136m ago)   27h
```

Figure 6. Result of 'kubectl get pods' command demonstrating successful application start-up

We can see here the status 'Running' for our three pods, demonstrating the successful starting of our application. If we want to connect to our web application, we will first need to know on which IP we should connect. We can do it by running the command:

```
minikube service tempconverter --url
```

This command displays the address http://192.168.49.2:32082. If we connect to it, we got this result:

Figure 7. Demonstration of the correct operation of kubernetes deployed application

We can once more see here that we can successfully connect to the app at port 80 (not displayed as it is the default port). We can also see that the application successfully connects to the MySQL database by sending a value in degrees Celsius and observing its display converted to degrees Fahrenheit in the table at the bottom.

Scaling the application with a different number of replicas can be done with the following command:

```
kubectl scale deployment tempconverter --replicas=3
```

We can see that we have now 3 replicas of the web app by running:

```
minikube kubectl -- get pods # or kubectl get pods
```

```
NAME                             READY   STATUS    RESTARTS        AGE
mysql-0                          1/1     Running   3 (140m ago)    27h
tempconverter-6467489b6c-4td4c   1/1     Running   0               46s
tempconverter-6467489b6c-pbfw8   1/1     Running   7 (138m ago)    27h
tempconverter-6467489b6c-qskkg   1/1     Running   7 (138m ago)    27h
```

Figure 8. Result of ‘kubectl get pods’ command demonstrating successful application rescaling

4. Conclusion

During this project, we successfully containerized the tempconverter application and deployed it firstly locally, and then using both Docker Swarm and Kubernetes (with Minikube). Docker Swarm provided a simpler and quicker setup, making it ideal for small-scale deployments. On the other hand, Kubernetes offered a more complex but highly scalable and feature-rich environment, suitable for large-scale production deployments. Kubernetes without Minikube is really hard to install and use, but Minikube makes it really easier for beginners as I am.

For future projects, the choice between Docker Swarm and Kubernetes should be based on the project's complexity, scale, and specific requirements. Docker Swarm is recommended for smaller projects where ease of use and rapid deployment are essential. In contrast, Kubernetes is better suited for larger, more complex applications that require advanced orchestration capabilities and robust scalability.

5. Appendices

5.1. Dockerfile

```
FROM python:3.9-slim

WORKDIR /app

COPY . .

RUN apt-get update && apt-get upgrade -y && \
    pip install --no-cache-dir -r requirements.txt

EXPOSE 5000

ENV DB_USER=db_user
ENV DB_PASS=dv_password
ENV DB_HOST=tempconverter-db
ENV DB_NAME=tempconverter_db
ENV STUDENT="Valentin Hervé"
ENV COLLEGE=Algebra

CMD ["python", "app.py"]
```

5.2. docker-compose-swarm.yml

```
version: '3.8'

services:
  tempconverter-db:
    image: mysql:8
    environment:
      MYSQL_ROOT_PASSWORD: passsword
      MYSQL_DATABASE: tempconverter_db
      MYSQL_USER: db_user
      MYSQL_PASSWORD: db_password
    volumes:
      - tempconverter-db-data:/var/lib/mysql
    networks:
      - tempconverter-network
    deploy:
      replicas: 1
      update_config:
        parallelism: 1
        delay: 10s
      restart_policy:
        condition: on-failure

  tempconverter-web:
    image: docker.io/vherve/tempconverter:latest
    ports:
      - "80:5000"
    environment:
      DB_USER: db_user
      DB_PASS: db_password
      DB_HOST: tempconverter-db
      DB_NAME: tempconverter_db
      STUDENT: "Valentin Hervé"
      COLLEGE: Algebra
    depends_on:
      - tempconverter-db
    networks:
      - tempconverter-network
    deploy:
      replicas: 2
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure

networks:
  tempconverter-network:
    driver: overlay

volumes:
  tempconverter-db-data:
```

5.3. mysql-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: tempconverter-db
spec:
  ports:
  - port: 3306
    name: mysql
  selector:
    app: mysql
```

5.4. mysql-statefulset.yaml

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  serviceName: "tempconverter-db"
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:8
        ports:
        - containerPort: 3306
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: passsword
        - name: MYSQL_DATABASE
          value: tempconverter_db
        - name: MYSQL_USER
          value: db_user
        - name: MYSQL_PASSWORD
          value: db_password
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
  volumeClaimTemplates:
  - metadata:
      name: mysql-persistent-storage
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

5.5. tempconverter-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tempconverter
spec:
  replicas: 2
  selector:
    matchLabels:
      app: tempconverter
  template:
    metadata:
      labels:
        app: tempconverter
    spec:
      containers:
      - name: web
        image: docker.io/vherve/tempconverter:latest
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 5000
        env:
        - name: DB_USER
          value: db_user
        - name: DB_PASS
          value: db_password
        - name: DB_HOST
          value: tempconverter-db
        - name: DB_NAME
          value: tempconverter_db
        - name: STUDENT
          value: "Valentin Hervé"
        - name: COLLEGE
          value: "Algebra"
---
apiVersion: v1
kind: Service
metadata:
  name: tempconverter
spec:
  selector:
    app: tempconverter
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
  type: LoadBalancer
```

6. References

[1] jstanesic Github repository 'tempconverter', https://github.com/jstanesic/tempconverter
[2] Red Hat Academy, https://rha.ole.redhat.com/rha/app
[3] docker.docs, https://docs.docker.com/
[4] minikube, https://minikube.sigs.k8s.io/docs/
[5] Valentin Hervé Github repository 'tempconverter', https://github.com/Vazelek/tempconverter/tree/devops-project
[6] docker.docs, Swarm mode overview, https://docs.docker.com/engine/swarm/
[7] kubernetes, https://kubernetes.io/