Tomislav Keščec

University College Algebra, June 2024.

Intro to DevOps

**Contents**

[1. Abstract ---------------------------------------------------------------------------------------------------------------------------3](#_Toc169015235)

[2. Introduction ----------------------------------------------------------------------------------------------------------------------3](#_Toc169015236)

[3. Explanation -----------------------------------------------------------------------------------------------------------------------4](#_Toc169015237)

[3.1. Project cloning -----------------------------------------------------------------------------------------------------------4](#_Toc169015238)

[3.2. Docker image ------------------------------------------------------------------------------------------------------------4](#_Toc169015239)

[3.3. Creating a Network -----------------------------------------------------------------------------------------------------6](#_Toc169015240)

[3.4. Starting containers ------------------------------------------------------------------------------------------------------6](#_Toc169015241)

[3.5. Simple container orchestration system -------------------------------------------------------------------------------7](#_Toc169015242)

[3.6. Kubernetes Deployment ------------------------------------------------------------------------------------------------9](#_Toc169015243)

[4. Conclusion ---------------------------------------------------------------------------------------------------------------------13](#_Toc169015244)

# Abstract

The project focuses on implementing and deploying the web application 'TempConverter' using Docker and Kubernetes. The application consists of a web interface that enables the conversion of temperature between Celsius and Fahrenheit measurement units. The process includes creating a Docker image, pushing the image to a registry, configuring and deploying using Docker Sworm and Kubernetes. The goal is to automate the deployment process and enable easy scaling of the application.

# Introduction

The problem this project addresses involves the absence of a temperature conversion application in a containerized form that facilitates straightforward deployment and scaling. Many users and businesses require temperature data conversion, but the existing solutions do not offer the flexibility and efficiency of containerization, hindering their broader applicability and ease of use.

Existing solutions typically involve deploying temperature conversion applications directly on servers without the use of container technology. This approach complicates the management and scaling of the application as it ties the deployment to specific hardware and requires manual intervention for updates, scaling, and management. As a result, the process becomes less efficient and more prone to errors, making it challenging to maintain service continuity and performance stability.

This project proposes containerizing the application to address these issues. A Docker image of the application has been created, which can be run on any system that supports Docker, regardless of the underlying hardware. Kubernetes is used for orchestration, allowing for automated management, scaling, and recovery of the application, ensuring high availability and resource efficiency.

The results and technical implementations demonstrate that the application is now available as a Docker image, which can be easily deployed and scaled in any Kubernetes environment. This solution significantly simplifies deployment processes, reduces the potential for human error, and enhances the scalability and reliability of the application. The application can now be deployed across multiple environments without needing extensive infrastructure adjustments.

The next steps include integrating additional functionalities into the application and optimizing it for greater flexibility and security. Future enhancements will focus on advanced user settings for conversion preferences and improved security features to protect data during the conversion process. These improvements aim to make the application more robust and user-friendly, meeting the growing needs of diverse user bases.

# Explanation

## Project cloning

The first step is to clone the project from GitHub repository.

git clone https://github.com/jstanesic/tempconverter

## Docker image

After the project is cloned, it is necessary to create a Docker file within the root directory, which can be used to build an image that will meet the project's requirements:

* Update all packages as part of the image build process
* Expose port 5000 TCP
* Install all required requirements
* Configure the correct command to start the flask application

**Dockerfile**

# Choose the base image

FROM python:3.9-slim

# Set the working directory

WORKDIR /app

# Copy the current directory contents into the container at /app

COPY . /app

# Update the package lists

RUN apt-get update && apt-get upgrade -y && rm -rf /var/lib/apt/lists/\*

# Install required Python libraries

RUN pip install --no-cache-dir -r requirements.txt

# Expose port 5000

EXPOSE 5000

# Set the environment variables

ENV FLASK\_APP=tempconverter.py

ENV STUDENT="Tomislav Keščec"

ENV COLLEGE="Algebra University College"

# Start flask application

CMD ["flask", "run", "--host=0.0.0.0"]

A Dockerfile has been created that defines how the application is packaged into a Docker container. This file includes instructions for installing necessary dependencies, copying the application's source code into the container, and setting up commands to run the application.

**Create docker image**

docker build -t tempconverter:latest .

A Docker image has been built using the previously defined Dockerfile.

**Push Docker image to registry**

docker login

docker tag tempconverter:latest codetome1503/tempconvert:latest

docker push codetome1503/tempconvert:latest

The Docker image was pushed to a Docker registry to make it available for deployment in Kubernetes.

A screenshot of a computer

Description automatically generated

Figure 1. DockerHub repository with latest tag

After changing the title tag in the **index.html** file, an image was built with the dev tag and pushed to the repository.

docker build -t tempconverter:dev .

docker tag tempconverter:dev codetome1503/tempconvert:dev

docker push codetome1503/tempconvert:dev

A screenshot of a computer

Description automatically generated

Figure 2. DockerHub repository with new dev tag

## Creating a Network

It is necessary to create a network through which the containers will communicate. The container hosting the application must be able to communicate with the container hosting the database.

docker network create converter\_network

## Starting containers

After creating the network, the MySQL database container is started, using the newly created network for communication.

docker run --name mysql-db --network converter\_network -e MYSQL\_ROOT\_PASSWORD=rootpass -e MYSQL\_USER=user -e MYSQL\_PASSWORD=pass -e MYSQL\_DATABASE=tempconvert -d mysql:8

After that, the application container is started, also using the newly created network for communication.

docker run --name tempconverter --network converter\_network -e DB\_HOST=mysql-db -e DB\_USER=user -e DB\_PASS=pass -e DB\_NAME=tempconvert -p 5000:5000 -d tempconverter:latest

As a result, a functional application can be seen.

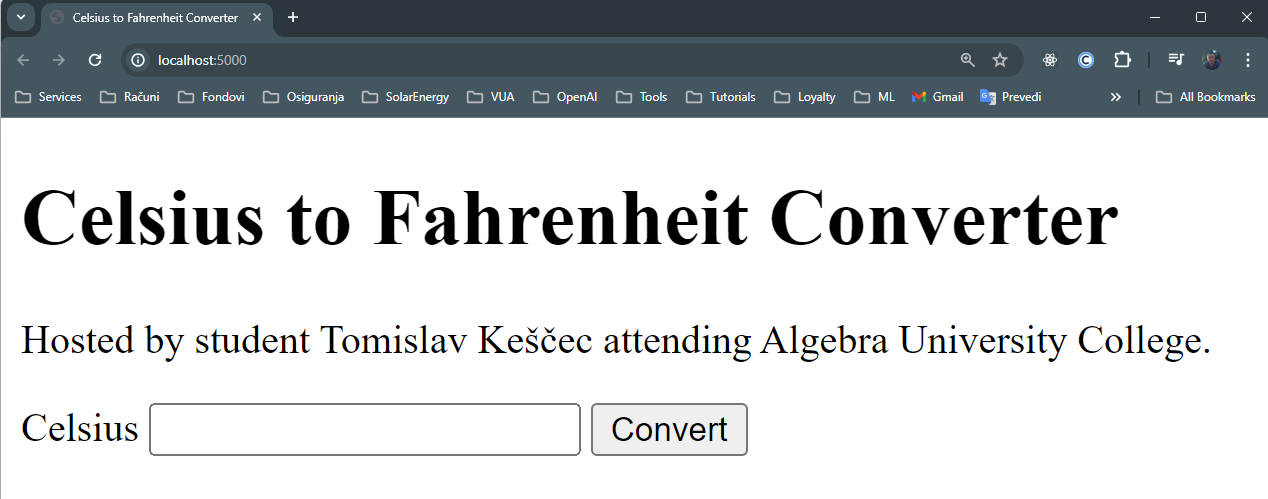


Figure 3. Result

## Simple container orchestration system

The choice of a container orchestration system often depends on the specific needs of the project, such as scalability, ease of setup, maintenance, and existing infrastructure compatibility. For the TempConverter project, we have a list of simpler orchestration systems to choose from:

* Docker Swarm
* AWS ECS
* Azure Container Apps
* Google Cloud Run
* HashiCorp Nomad

**Factors to Consider:**

1. **Complexity and Learning Curve**: Systems like Docker Swarm and HashiCorp Nomad are generally easier to set up and manage compared to more complex systems like Kubernetes.
2. **Integration with Existing Tools**: Compatibility with existing CI/CD pipelines and development workflows. For instance, if already using AWS services extensively, AWS ECS might be a natural choice.
3. **Scalability and Manageability**: Although simpler, the system should still meet the scalability needs of the application. Docker Swarm, for example, integrates seamlessly with existing Docker configurations.
4. **Cost and Resource Efficiency**: Some systems may offer more cost-effective solutions especially in cloud environments like AWS ECS or Google Cloud Run.

In this project it is used a **Docker Swarm** due to its simplicity in setup and direct compatibility with Docker, which we're already using.

**Setting up Docker Swarm**

The first step is to initialize a Docker Sworm.

docker swarm init

This command will turn Docker engine into a swarm manager, and it will start listening for other machines to join the swarm as nodes.

A screenshot of a computer program

Description automatically generated

Figure 4. Swarm Initialized

**Creating the deployment configuration**

Docker Swarm uses a configuration similar to docker-compose but with swarm-specific options.

version: '3.7'

services:

  db:

    image: mysql:8

    environment:

      MYSQL\_ROOT\_PASSWORD: rootpass

      MYSQL\_DATABASE: tempconvert

      MYSQL\_USER: user

      MYSQL\_PASSWORD: pass

    deploy:

      replicas: 1

      placement:

        constraints: [node.role == manager]

    networks:

      - temp-network

  tempconverter:

    image: codetome1503/tempconverter:latest

    ports:

      - "5000:5000"

    depends\_on:

      - db

    environment:

      DB\_USER: user

      DB\_PASS: pass

      DB\_HOST: tempstack\_db

      DB\_NAME: tempconvert

      STUDENT: Tomislav Keščec

      COLLEGE: Algebra University College

    deploy:

      replicas: 2

      placement:

        constraints: [node.role == manager]

    networks:

      - temp-network

networks:

  temp-network:

    external: true

**Deploying the Stack**

docker stack deploy -c docker-compose.yml tempstack

This command will start all services defined in docker-compose.yml, managing them as a stack within the swarm.

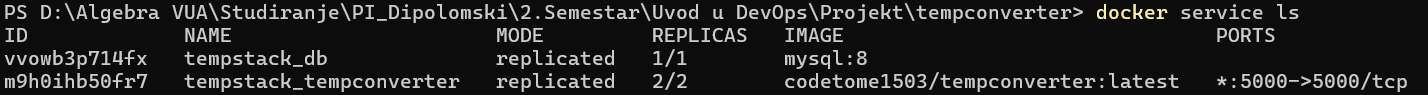
****

Figure 5. Active services

**Scaling the Application**

docker service scale tempstack\_tempconverter=3

This command changes the number of running instances of the TempConverter service to three.

A screen shot of a computer

Description automatically generated

Figure 6 Change service replicas.

This setup with Docker Swarm provides a robust yet simple solution for orchestrating the TempConverter application. It leverages Docker's native clustering capabilities. This choice suits the project's requirements for simplicity, scalability, and ease of integration with existing Docker-based workflows.

## Kubernetes Deployment

1. **Create the Kubernetes Deployment for TempConverter**

First, a deployment configuration for the TempConverter application is needed. This will define how Kubernetes should manage the application's pods.

apiVersion: apps/v1

kind: Deployment

metadata:

  name: tempconverter

  labels:

    app: tempconverter

spec:

  replicas: 2

  selector:

    matchLabels:

      app: tempconverter

  template:

    metadata:

      labels:

        app: tempconverter

    spec:

      containers:

      - name: tempconverter

        image: codetome1503/tempconverter:latest

        ports:

        - containerPort: 5000

        env:

        - name: DB\_USER

          value: "user"

        - name: DB\_PASS

          value: "pass"

        - name: DB\_HOST

          value: "db-service"

        - name: DB\_NAME

          value: "tempconvert"

        - name: STUDENT

          value: "Tomislav Keščec"

        - name: COLLEGE

          value: "Algebra University College"

1. **Create the Kubernetes Service for TempConverter**

To make the TempConverter application accessible, a Service needs to be created that exposes it.

apiVersion: v1

kind: Service

metadata:

  name: tempconverter-service

spec:

  type: LoadBalancer

  ports:

  - port: 80

    targetPort: 5000

    protocol: TCP

  selector:

    app: tempconverter

1. **Create the Kubernetes Deployment for MySQL Database**

A Deployment for the MySQL database that TempConverter uses is also needed.

apiVersion: apps/v1

kind: Deployment

metadata:

  name: mysql

  labels:

    app: mysql

spec:

  replicas: 1

  selector:

    matchLabels:

      app: mysql

  template:

    metadata:

      labels:

        app: mysql

    spec:

      containers:

      - name: mysql

        image: mysql:8

        env:

        - name: MYSQL\_ROOT\_PASSWORD

          value: "rootpass"

        - name: MYSQL\_DATABASE

          value: "tempconvert"

        - name: MYSQL\_USER

          value: "user"

        - name: MYSQL\_PASSWORD

          value: "pass"

        ports:

        - containerPort: 3306

1. **Create the Kubernetes Service for MySQL**

The MySQL service will allow the TempConverter application to communicate with the database.

apiVersion: v1

kind: Service

metadata:

  name: db-service

spec:

  ports:

  - port: 3306

    targetPort: 3306

    protocol: TCP

  selector:

    app: mysql

These configurations need to be deployed to the Kubernetes cluster using *kubectl*.

kubectl apply -f mysql-deployment.yaml

kubectl apply -f mysql-service.yaml

kubectl apply -f tempconverter-deployment.yaml

kubectl apply -f tempconverter-service.yaml

This series of commands sets up both the TempConverter application and the MySQL database within the Kubernetes cluster, including internal networking and external access via a LoadBalancer for the TempConverter application.

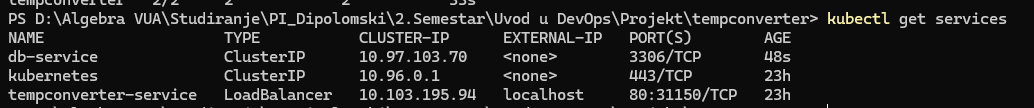


Figure 7. Active kubernetes services

Since Kubernetes is being used locally on Docker, port forwarding is necessary to test if the service is working and if the application is accessible.

kubectl port-forward service/tempconverter-service 8888:80

A screenshot of a computer

Description automatically generated

Figure 8. Result after port forwarding

# Conclusion

In the project, Docker was used to create images for the TempConverter application and a MySQL database. Specific Dockerfiles were crafted to define the operational environment, dependencies, and startup commands necessary for each component. These Dockerfiles guided the construction of Docker images, which are standardized, executable packages that include everything needed to run the application—code, runtime, libraries, and settings.

Once the Docker images were constructed, they were deployed as containers. Containers are runtime instances of Docker images, providing an isolated environment for the application to run. This isolation helps to eliminate inconsistencies in operation due to differences in development and production environments.

For orchestration, Docker Swarm was employed. Docker Swarm is a container orchestration tool that manages clusters of Docker engines, turning a group of Docker hosts into a single virtual host. It was utilized to handle the deployment, scaling, and management of the containers. The Docker Swarm setup involved initializing a swarm, defining services, and managing the desired state of the containers within the swarm environment.

In Kubernetes, similar functionalities were achieved without using Helm for the setup. Kubernetes deployments were defined in YAML files, specifying how many replicas of each container should be run, the Docker images to use, and the configuration necessary to connect the application to the MySQL database. Services were also defined to expose the application and database on specified ports, facilitating network access to these services within the Kubernetes cluster.

This straightforward technical approach underscored the effective use of Docker and Kubernetes in creating a scalable, manageable, and isolated environment for running web applications, focusing on the deployment mechanisms and orchestration capabilities of Docker Swarm and Kubernetes.