Matija Kolarić

University College Algebra, Jun 2023.

INTRODUCTION TO DEVOPS

**Contents**

[1. Abstract 3](#_Toc136676156)

[2. Introduction 3](#_Toc136676157)

[3. Steps 3](#_Toc136676158)

[3.1. Source Code Retrieval 3](#_Toc136676159)

[3.2. Application Containerization 4](#_Toc136676160)

[3.3. Binary Creation 4](#_Toc136676161)

[3.4. Database Containerization 5](#_Toc136676162)

[3.5. Service Orchestration 6](#_Toc136676163)

[3.6. Deployment with Docker Swarm 7](#_Toc136676164)

[3.7. Image Publishing 7](#_Toc136676165)

[3.8. Automation with Makefile 8](#_Toc136676166)

[4. Conclusion 9](#_Toc136676167)

# Abstract

This project involves containerizing and deploying the TempConverter web application using Podman and Docker Swarm. The application, written in Python and utilizing a MySQL 8 database, converts temperature units and displays personalized information based on environment variables. The project is divided into several stages. First, the application's source code is retrieved from a GitHub repository, and a Docker image is built. Next, this image is updated with all necessary packages, exposed on port 5000, and configured to start the Flask application. The application is then built into a binary using PyInstaller. Next, a MySQL image is configured to connect to the application as a non-root user. Finally, a Podman Compose file is written for the test environment to orchestrate the application and database services, and environment variables are set to personalize the application's display. Finally, the application is deployed using Docker Swarm, a native clustering and orchestration solution for Docker. The Docker images are pushed to Docker Hub, and a Makefile is written to automate the build, package, and deployment processes. The Makefile also includes a clean rule to stop all running containers, remove Docker images, and stop Docker Swarm. This project demonstrates a complete DevOps workflow, from building an application from source code to deploying it in a production-like environment.

# Introduction

1. **Problem Definition**: The task is to containerize and deploy the TempConverter web application. This application, written in Python, requires a MySQL 8 database to function and uses environment variables for configuration and personalization. The challenge is to create a reliable, repeatable, and automated process for building and deploying this application.
2. **Existing Solutions**: Containerization and deployment of applications is a well-explored area, with numerous solutions available. Docker is a widely used platform that allows developers to package applications into containers — standardized software units containing everything the software needs to run, including libraries, system tools, code, and runtime. In addition, Podman Compose is a tool for defining and running multi-container Docker applications, and Docker Swarm is a native clustering and orchestration solution for Docker.
3. **Our Approach**: To solve the problem, we used Docker to containerize the TempConverter application and its MySQL database. We wrote Dockerfiles to define the environment for our application and database and a Podman Compose file to orchestrate these services. We also used PyInstaller to build the application into a binary and Docker Swarm to deploy the application.
4. **Results and Technical Solutions**: We have a fully automated process for building and deploying the TempConverter application. The application and its database are containerized, allowing them to be run on any system with Docker installed. The application is built into a binary, making it easy to distribute and run. The deployment process is managed by Docker Swarm, ensuring that the application is always running and available.
5. **Next Steps**: The next steps for this project include improving the automation process, such as integrating a continuous integration/continuous deployment (CI/CD) pipeline to automatically build, test, and deploy the application when changes are made to the source code. We could also explore scaling the application to handle more traffic using Docker Swarm's scaling capabilities. Finally, we could also investigate securing the application and its data, for example, by using Docker secrets to store sensitive information like database passwords securely.

# Steps

## Source Code Retrieval

The first store sensitive information like database passwords tempconverter application. The source code for this application is hosted on GitHub, a popular web-based hosting service for version control and source code management. To retrieve the source code, we used the git clone command followed by the repository's URL. This command creates a local copy of the repository on our machine, allowing us to access and modify the source code. For example, the command we used was:

**git clone** [**https://github.com/jstanesic/tempconverter**](https://github.com/jstanesic/tempconverter)

After running this command, a new temp converter directory was created in our current working directory. This directory contains all the files and folders from the GitHub repository, including the source code of the TempConverter application. It's important to note that the **git clone** command also creates a link between our local repository and the remote repository on GitHub. We can easily fetch updates from the remote repository or push our changes to it using Git's built-in commands. Once we had the source code on our machine, we could proceed with the project's next steps, such as building the Docker image for the application and creating the binary.

## Application Containerization

The next step in the project was to containerize the TempConverter application. Containerization involves packaging an application and its dependencies into a container, which can be run consistently on any platform supporting containerization. For this project, we used Docker, a popular platform for containerization. Docker allows us to define the environment for our application using a Dockerfile, a text file that contains all the commands needed to build a Docker image.

* We started with a base image of Python 3.7 using the **FROM** keyword. This image includes the Python runtime necessary to run our application.  
  **FROM python:3.7-slim AS build**
* We set the working directory in the container to **/app** using the **WORKDIR** keyword. This is the directory where our application's source code will be placed in the container.  
  **WORKDIR /app**
* We updated all packages in the image using the **RUN** keyword and the **apt-get update && apt-get upgrade -y** command. This ensures that all the software in the image is up-to-date.  
  **RUN apt-get update && apt-get upgrade -y**
* We added the binary of our application into the container at **/app** using the **ADD** keyword. This includes our application's binary that was previously built using PyInstaller.  
  **ADD bin/app /app**
* We exposed port 5000 using the **EXPOSE** keyword. This is the port that our Flask application listens on.  
  **EXPOSE 5000**
* Finally, we configured the correct command to start the application binary.  
  **CMD ["./app"]**

Once the Dockerfile was written, we used the docker build command to build a Docker image for our application. This command reads the Dockerfile and executes all its commands, resulting in a Docker image containing our application and its dependencies.

With the Docker image, we can now run our application in a Docker container, which can be deployed on any Docker system installed. This ensures that our application will run the same way, regardless of the underlying system.

## Binary Creation

A binary is a pre-compiled version of the application that can be run directly without needing a Python interpreter. This makes the application easier to distribute and run. We used PyInstaller, a Python package that converts Python applications into stand-alone executables under Windows, GNU/Linux, Mac OS X, and more for this task.

* We started with a base image of Python 3.7 using the FROM keyword. This image includes the Python runtime necessary to run our application.  
  **FROM python:3.7-slim AS build**
* We set the working directory in the container to /app using the WORKDIR keyword. This is the directory where our application's source code will be placed in the container.  
  **WORKDIR /app**
* We updated all packages in the image using the RUN keyword and the apt-get update -y command. This ensures that all the software in the image is up to date.  
  **RUN apt-get update -y**
* We installed binutils using the RUN keyword and the apt-get install -y binutils command. Binutils is a collection of binary tools necessary for the PyInstaller to work.  
  **RUN apt-get install -y binutils**
* Using the ADD keyword, we added the current directory contents into the container at /app. This includes our application's source code and other files in our current directory.  
  **ADD . /app**
* We installed PyInstaller and any needed packages specified in requirements.txt using the RUN keyword and the pip install command. PyInstaller is used to build our application into a binary, and requirements.txt lists the Python packages that our application depends on.  
  **RUN pip install --no-cache-dir -r requirements.txt pyinstaller**
* Finally, we used the ENTRYPOINT keyword to specify the command to build the application into a binary. The pyinstaller --onefile command created a single executable file. The --add-data option included the index.html file from the template’s directory in the binary. The app.py argument specifies the main script of our application, and the --distpath dist option specifies the output directory for the binary.  
  **ENTRYPOINT pyinstaller --onefile --add-data="templates/index.html:templates/" app.py --distpath dist**

With the binary created, we could run our application without needing a Python interpreter or any dependencies. This makes distributing our application and running it on different systems easier.

## Database Containerization

After the application was containerized and the binary was created, the next step was to containerize the MySQL 8 database that the application relies on. Containerizing the database allows it to run in a consistent environment and makes it easier to manage and deploy. We used Podman Compose, a tool for defining and running multi-container applications, for this task. Podman Compose allows us to define the environment for our database using a YAML file, which is easier and more flexible than writing a Dockerfile.

Here's a breakdown of the steps we took to containerize the database:

* We started by defining a new service in our Podman Compose file. This service, named db, represents our database.

**services:**

**db:**

* We then specified the Docker image to use for this service. We used the arm64v8/mysql:oracle image from Docker Hub for this project. This image includes a pre-configured MySQL 8 server.

**image: arm64v8/mysql:oracle**

* We defined a volume for our service. This volume, db\_data, is mounted at /var/lib/mysql in the container. This is where MySQL stores its data. Using a volume ensures that our data persists even if the container is stopped or deleted.

**volumes:**

**- db\_data:/var/lib/mysql**

* We set several environment variables to configure the MySQL server. These variables include the root password, the name of the database, and the username and password for a non-root user. The MySQL image uses these variables to configure the server when it starts.

**environment:**

**MYSQL\_ROOT\_PASSWORD: $DB\_ROOT\_PASS**

**MYSQL\_DATABASE: $DB\_NAME**

**MYSQL\_USER: $DB\_USER**

**MYSQL\_PASSWORD: $DB\_PASS**

* Finally, we used the docker-compose-up command to start our database service. This command reads the Podman Compose file and starts all the services defined in it, including our database service.

**podman-compose up**

With Podman Compose, we can run our MySQL server in a Docker container, which can be deployed on any system with Docker installed. This ensures that our database will run the same way, regardless of the underlying system. Furthermore, by running the database in a container, we can easily manage it using Podman Compose commands. For example, we can start, stop, or restart the database by running a single Podman Compose command. We can also easily connect our application to the database by linking their containers.

## Service Orchestration

After containerizing the application and the database, the next step was orchestrating these services together. Service orchestration involves managing multiple services (in this case, the application and the database) so that they can interact with each other and function as a cohesive system. We used Podman Compose, a tool for defining and running multi-container applications, for this task. Podman Compose allows us to define our services and their configurations in a YAML file, which is easier and more flexible than managing each service individually.

Here's a breakdown of the Podman Compose file we used for service orchestration:

* We defined two services: db for our database and web for our application. Each service is defined with its configuration, such as the Docker image to use, the environment variables to set, and the networks to connect to.

**services:**

**db:**

**...**

**web:**

**...**

* For the db service, we used the arm64v8/mysql:oracle image from Docker Hub, and we set several environment variables to configure the MySQL server. Next, we built a Docker image for the web service using the Dockerfile located at .docker/Dockerfile.app. We also set several environment variables to configure the application, including the database host, name, user, and password. Finally, we defined a volume named db\_data for the db service. This volume is used to persist the data of our MySQL server.

**volumes:**

**db\_data:**

* We defined a network named tempconverter-cloud for our services. This network allows our services to communicate with each other.

**networks:**

**tempconverter-cloud:**

**driver: bridge**

* Finally, we used the docker-compose up command to start our services. This command reads the Podman Compose file and starts all the services defined in it.

**podman-compose up**

With Podman Compose, we can run our application and database as a cohesive system, with each service running in its container. This makes it easier to manage our system and ensures that our services can communicate.

## Deployment with Docker Swarm

After the application and the database were containerized and orchestrated, deploying these services using Docker Swarm was the final step. Docker Swarm is a native clustering and scheduling tool for Docker. It turns a pool of Docker hosts into a single, virtual Docker host, which allows you to scale your applications seamlessly.

* First, we initialized a Docker Swarm using the docker swarm init command. This command sets up the current Docker host as a Swarm manager, which allows us to manage the Swarm and its services.

**docker swarm init**

* Next, we deployed our services to the Swarm using the docker stack deploy command. This command reads a Podman Compose file and deploys the services defined in it as a stack. A stack is a group of interrelated services that share dependencies and can be orchestrated and scaled together.

**docker stack deploy -c docker-compose.yml tempconverter**

In the command above, -c docker-compose.yml specifies the Podman Compose file to use, and tempconverter is the stack's name.

* The Podman Compose file we used for deployment is like the one we used for service orchestration, but with a few differences. For example, we replaced the build directive with the image directive for the web service, as Docker Swarm does not support building images. We also removed the container\_name directive, as Docker Swarm automatically assigns names to its created containers. Finally, we used the **docker service ls** command to verify that our services were running. This command lists all the Swarm services, their status, and other information.

docker service ls

With Docker Swarm, we can run our application and database as a scalable and resilient system. In addition, Docker Swarm automatically manages the services in our stack, ensuring that they are always running and can handle varying loads. This makes it easier to manage our system and ensures that our services are always available to users.

## Image Publishing

After the application and the database were containerized, the binary was created, services were orchestrated, and the system was deployed. The final step was to publish the Docker image of the application. Publishing the image allows it to be shared with others and deployed on any system that supports Docker. For this task, we used Docker Hub. This cloud-based registry service allows you to link to code repositories, build your images and test them, store manually pushed images, and links to Docker Cloud so you can deploy images to your hosts.

* First, we tagged our Docker image using the docker tag command. This command assigns a tag to the image, which can be used to version and track the image.

**podman tag tempconverter-app: latest tempconverter-app:<hash\_of\_file>**

In the command above, tempconverter-app: latest is the name and tag of our image, and <hash\_of\_file> is the new tag we want to assign to the image.

* Next, we logged into Docker Hub using the docker login command. This command prompts for a username and password, which are used to authenticate with Docker Hub.

**podman login**

* Once logged in, we pushed our image to Docker Hub using the docker push command. This command uploads our image to the registry, making it available for others to pull and use.

**podman push darwin0id/tempconverter-app:<hash\_of\_file>**

In the command above, **darwin0id/tempconverter-app:<hash\_of\_file>** is the name and tag of our image, and darwin0id is our Docker Hub username.

Finally, we verified that our image was successfully pushed by checking our Docker Hub account. We could see our image listed in our repositories, along with its tags and other information. With the image published, anyone can pull it from Docker Hub and run our application in a Docker container. This makes it easy to share and deploy our application on any system.

## Automation with Makefile

After completing all the steps manually, the final task was to automate the entire process using a Makefile. A Makefile contains a set of directives used with the **make build** automation tool. It simplifies the build and deployment process by automating complex workflows, making it more efficient and less error-prone.

* First, we created a Makefile in the root directory of our project. This file contains a set of targets, each representing a step in our build and deployment process.

**pipeline: build package deploy clean**

In the line above, the pipeline is a particular target representing the default goal. When make is run without arguments, it builds all targets, which builds the build, package, deploys, and clean targets in order.

* For each target, we defined a set of commands to be executed when the target is built. These commands correspond to the steps we took manually in the previous tasks.

For example, the build target might look like this:

**build:**

**podman build -t tempconverter-app .**

The build is the target name in the lines above, and the **podman build -t tempconverter-app .** Is the command to be executed when the target is built?

* We repeated this process for each step in our build and deployment process, creating a target for each step and defining the appropriate commands for each target. Finally, we used the make command to build our targets and automate our process. By running make with no arguments, we built all the targets, which built all the other targets in order.

**make**

The Makefile can automate our entire build and deployment process with a single command. This makes the process more efficient, less error-prone, and easier to manage.

# Conclusion

In conclusion, this project involved containerizing and deploying a web application, TempConverter, using Docker, Podman Compose, and Docker Swarm. The application's source code was retrieved from a GitHub repository, and a Dockerfile was written to build a Docker image for the application. The application was then built into a binary using PyInstaller, which made it easier to distribute and run. The MySQL 8 database the application relies on was also containerized using Docker. Finally, a Podman Compose file was written to define the application and database services, allowing them to be orchestrated and run together as a cohesive system. The services were then deployed using Docker Swarm, a native clustering and scheduling tool for Docker. Docker Swarm allowed the services to be scaled and managed as a unit, ensuring they could handle varying loads and recover from failures.

The Docker image of the application was published to Docker Hub, making it available for others to pull and use. This made it easy to share and deploy the application on any system. Finally, the entire process was automated using a Makefile. The Makefile simplified the build and deployment process by automating complex workflows, making the process more efficient and less error-prone.

Overall, this project demonstrated the power and flexibility of containerization and service orchestration. By using Docker, Podman Compose, and Docker Swarm, we were able to build a scalable and resilient system that can be easily managed and deployed. As a result, this project is a valuable reference for future containerization and service orchestration projects.

GitHub: <https://github.com/Darwin0id/introduction-to-devops>