Containers Technologies Foundation - Final Project

Jacques Adam

# Introduction

The objective of this project is to demonstrate correct application of the principles learnt in the course on containerisation and deployment of applications and dependent components such as databases.

This exercise is designed to provide a guide through the process of taking a functional web application and making it scalable.. The specific example used is modelled on a Django-based gallery website allowing a user to upload images that get stored to a storage point and is connected to a database.

The tasks include building a the Django web application and using two separate Docker containers, host the web app in one and and hosting the database running PostgreSQL in the other. A bridge network would then connect the two containers allowing them to operate as a unit.

Once the application is containerised, then the next task is to scale it using Kubernetes.

We are hopeful of gaining hands-on experience with how Docker simplifies the packaging of an application and how Kubernetes takes over to provide the scalable infrastructure needed for a successful web service.

# Implementation

The process required that a copy of the gallerysite Django application be downloaded from github (<https://github.com/JaydenKing32/gallerysite>).

The application was pre-built to use mysql and azure storage and needed to be refactored to connect and operate on PostgreSQL and a local folder to store uploaded images.

## Set-up

My setup included the following:

Fedora Workstation 42 as my main operating system and desktop for this project (as an opportunity to learn more about Linux).

The final setup (including this file and the powerpoint file) has been uploaded to github at: [**https://github.com/gadam/gallerysite.git**](https://github.com/gadam/gallerysite.git)

The video is located at: **https://drive.google.com/file/d/1QIi\_kNy61fwdCqc7ECMDz2iV2wcUEAY2/view?usp=sharing**

### Initial preparation

1. Installed PostgreSQL and the Django framework
2. Downloaded a copy of the project as a zip file
3. Refactored the following files to change the database from MySQL to PostgreSQL and Azure storage to local storage.
4. Created a Python virtualenv and installed dependencies:

python -m venv .venv

source .venv/bin/activate

pip install -r requirements.txt

1. Tested the application in my local environment

### Container development

1. Created a Dockerfile to build an image of the Django application.
   1. I chose the 3.12-slim python image as the base for its relatively small size. An app/ subfolder is also created in the image as the application’s home directory on the image and the contents of the entire local folder are copied to it.
   2. Finally, the command the Django app is primed to run by running the database migration via

pip install -r requirements.txt

1. A docker compose file was also created to build two containers, the first container (named web) hosting the Django application and the second container (named db) hosting the PostgreSQL database instance. The two containers are connected by a bridge network also named db. Local storage mounts were created:
   1. PostgreSQL data is persisted from the container to the local project folder in folder db/
   2. Uploaded images are persisted to another local project folder named media/images

The Django container definition includes a dependency to the PostgreSQL container and must wait for PostgreSQL to be started before the app container can also be started.

1. Tested the container development using the docker compose up command and the [http://localhost:8000](http://localhost:8000/) URL

### Kubernetes development

Implementing the application was several orders of magnitude more difficult than creating the two docker containers. Several manifest files needed to be created to support the deployment of the gallery application as follows:

- secret-db.yaml: Secret storing Postgres credentials

- postgres-pvc.yaml: PVC for Postgres data

- postgres-deployment.yaml: Postgres Deployment

- postgres-service.yaml: ClusterIP Service for Postgres

- media-pvc.yaml: PVC for Django media/uploads

- django-deployment.yaml: Django Deployment

- django-service.yaml: ClusterIP Service for Django

- k6-gallery-test.js: k6 load test script

#### secret-db.yaml

I chose to research how kubernetes handles critical information that should handled separately such as passwords and certificate tokens. File secret-db.yaml holds the user names and passwords for the PostgreSQL database used in the application. Normally this would not be included in the package uploaded to Github but provided separately.

#### Postgres-pvc.yaml

This manifest defines the Persistent Volume Claim connecting the PostgreSQL database to a persistent location to locate its data, and is a dependency for the PostgreSQL deployment manifest. An allocation of 1Gi was used as a default amount of starting block storage for the pod.

#### Postgres-deployment.yaml

This manifest sets up the pod for PostgreSQL using PostgreSQL:17.6-alpine3.21 as its base image. This image was chosen for its relatively small size.

The file includes a liveness probe which is a periodic health check that the kubelet runs to decide whether the container is still functioning. Also included is a readiness probe field that allows kubernetes to determine whether that container is ready to receive traffic.

#### Media-pvc.yaml

This manifest will connect the gallery-django deployment to a folder that will store the images uploaded by the user.

#### Django-deployment.yaml

The file includes an initContainer that checks that the postgreSQL pod has already been started before continuing with the rest of the gallery app deployment.

The app deployment makes use of the kubernetes secrets database to retrieve the user name and password to connect to the PostgreSQL database.

The deployment retrieves the application image from my docker hub repository jaga/gallerysite-web:latest where the docker image built during the docker development stage was stored.

In Django, the ALLOWED\_HOSTS environment variable controls which Host headers the application will accept. Setting it to "\*" tells Django to allow requests for any Host header — this effectively disables the Host-header check that would otherwise raise DisallowedHost when DEBUG is False.

The same change was required in the settings.py config file in order to resolve the DisallowedHost errors I was encountering during testing.

#### k6-gallery-test.js

This file uses Grafana’s load testing capabilities to test scaling up and down the number of virtual users and requests and is used to test the ability of the gallery app to 200 users.

### Main artefacts

The main artefacts developed for this assessment are:

- Django app source: gallery/, gallerysite/ (Django project)

- Docker Compose: compose.yaml (development)

- Kubernetes manifests: k8s/ (Deployments, Services, PVCs, Secret, k6 script)

# Discussion

### Difficulties during developmemt

Developing the Kubernetes manifest files was by far the most difficult task in this project. I restarted the project from scratch 3 times with the first 2 failures really due to:

* Misunderstanding of guidance provided by the course facilitator – I spent a fairly long time trying to resolve a bug where whenever I uploaded an image, the UI would re-display the image field in red and the image was not saved nor added to the gallery\_posts table in the database. By building the application on my local setup, I realised that whilst refactoring from Azure storage to local storage, I misunderstood the guidance to delete the references to Azure and delete the gallery/migrations/0002\_alter\_post\_image.py.
* Insufficient grasp of the added level of complexity that Kubernetes brings to the container deployment model. Whilst I did all the class exercises, the focus was more on understanding the functions that the kubectl command unlocked and not much focus on the intricacies of crafting a correct set of manifest files.

For full disclosure, in my third attempt, I relied heavily on co-pilot to guide me on where my code was falling short and to understand the root cause of the many errors that I was encountering. It was very precise in dissecting what was wrong and how to correct the issue with suggestions on best practices such as including init sections to wait for dependent pods to be spun up and health checks, how to code secrets, etc.

### Potential for future improvements

Future improvements would have to include replacing the runserver module and replacing it with an industrial strength server like WSGI for better concurrency and resilience.

The database, whilst robust, requires a back up strategy that delivers high availability.

The approach to implementing secrets still relies on at least one plain text file to store key information that is insecure. Consider using a vault in production.

Turn off DEBUG and change ALLOWED\_HOSTS from “\*” to list of IP addresses.

# Conclusion

This project ensured that the core principles of containerisation and application deployment using Docker and Kubernetes can be applied. By taking a pre-existing Django application and transforming it into a scalable, containerised web service, I needed to demonstrate skills in system configuration and container orchestration.

While Docker simplified the initial packaging and local deployment, the shift to Kubernetes introduced significant complexity, especially in managing dependencies, secrets, persistent storage, and deployment sequencing. The learning curve was steep, requiring multiple iterations and external assistance, but ultimately led to a functioning, scalable application.

The project highlighted the importance of:

* Proper understanding of container orchestration tools like Kubernetes,
* Secure handling of sensitive configuration,
* Best practices in deployment (e.g. using init containers, probes, and production-ready settings).