DAT515 Final Report

Group 16

Jakub Mroz
Maarij Sheraz Satti
Muhammad Fahad Nawaz Rana
j.mroz@stud.uis.no
ms.satti@stud.uis.no
mf.rana@stud.uis.no
University of Stavanger, Norway

ABSTRACT

The objective of this project was to learn how to deploy a full-stack web application on Amazon Cloud Services, by dockerizing it into images and then orchestrating those images with Kubernetes. The web application chosen for this project is divided into frontend, backend and a database. The report serves as a comprehensive documentation of the entire process of development and learning of the mentioned technologies.

1 INTRODUCTION

The goal was to develop and deploy a web app on Amazon AWS and have every part of the application working and communicating as it should, in addition to benefiting from Kubernetes orchestration.

The development and deployments of the web application in this project was done in 4 phases:

The first step begins with the selection of an appropriate web application and technologies used, taking into consideration factors such as complexity and compatibility with containerization technologies.

The next phase phase focuses on Docker containerization, where the selected web application is encapsulated within Docker containers. This process enhances portability and ensures consistent operation across diverse environments. The report delves into the nuances of Docker containerization, covering the creation of a Dockerfile and the implementation of best practices to optimize the container image.

Moving forward, the project incorporates Kubernetes orchestration, which enables efficient management and scaling of the containerized web application. Kubernetes configurations and deployment procedures are discussed to show on the orchestration aspect of the project.

The culmination of the project is the deployment of the containerized web application on AWS EC2. This phase involves configuring the AWS environment, launching an EC2 instance, transferring Docker and Kubernetes files, executing the application, and configuring port access for user accessibility. The report provides a detailed account of each step, emphasizing the importance of a well-structured AWS setup for the deployment.

Throughout the project, valuable insights are gained, including a deeper understanding of AWS services such as EC2 and security group configurations. The experience further reinforces knowledge of Docker and Kubernetes, enabling their practical application. In each of the named phases we will also discuss problems which we have encountered during the process.

The code for our project is available on GitHub, together with deployment files and instructions. We did not use any agile methodology, as our group is only 3 people we felt that we did not need it. We have discussed on who does what during lab meetings and on project group chat.

Lecturer: Prachi Vinod Wadatkar.

Final Report (DAT515), IDE, UiS
2023.

2 APPLICATION SELECTION

Firstly in the project we had to choose the scope of the application, its complexity and the technologies which will be used.

- Multi-container: we wanted our application to consist of more than one docker container, so that there is networking in between the containers.
- Relative simplicity: we did not want the app to contain features like user authentication, as they could be overly complex and take most of our focus away from the deployment aspect

With those criteria we have chosen to develop a simple web application which allows a person to sign in for a newsletter and receive a confirmation email in return: In addition to that a person can use a form to leave a comment on the website, which is then stored in the database and is visible to everyone who visits the site.

This application is divided into three main parts:

- Frontend: This is the part in which the user interacts with the website. Technologies we chose for it are React framework, with programming languages javascript and typescript used. The nice thing about the React is that it allows for easy connecting of html templates in form of jsx code. React also allows for client side routing, with changing the url without making a new call to the backend. Both javascript and typescript languages can be used in the same project, with typescript offering more features like strong typing and interfaces. This can make life easier with concepts like conditional rendering, when we want to render a specific html element of the website depending on the situation

- Backend: This is the part which contains logic of the application and it communicates with the database. For the backend we have used the Flask micro framework for Python. It is a simple framework which offers basic features like http requests and sending json files.
- Database: This is the part which stores information of the web app. For the database technology we have chosen Postgresql. It is a popular database, and we chose it because it was used in many other projects of similar scope. There were no hard reasons why we chose it over other technologies, as most of them can fill our basic database needs. Postgre was easy to setup and to communicate with in the backend.

Figure 1: Web app newsletter

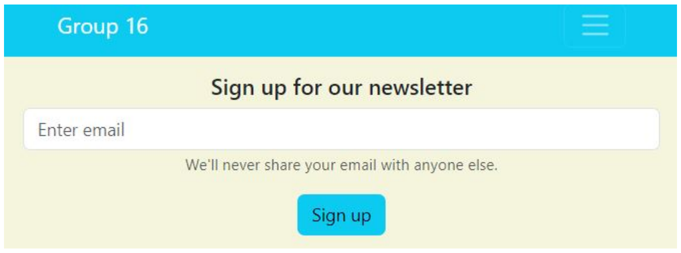
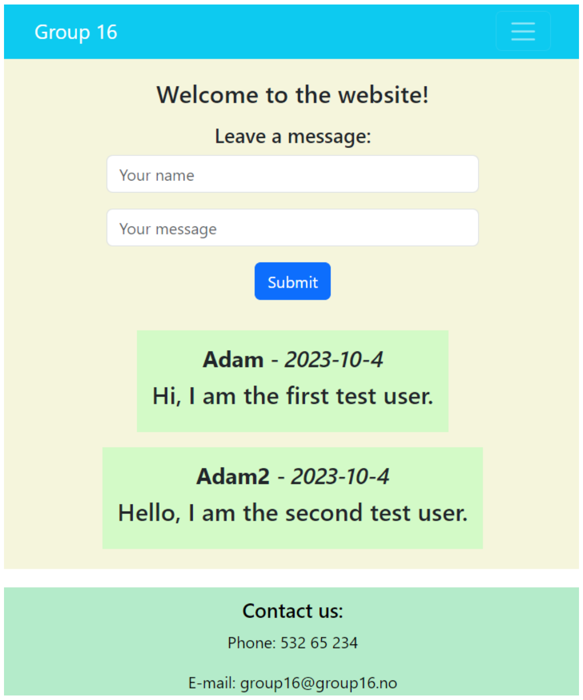

Group 16
Sign up for our newsletter
Enter email
We'll never share your email with anyone else.
Sign up

Figure 2: Web app comments



Adam - *2023-10-4*
Hi, I am the first test user.

Adam2 - *2023-10-4*
Hello, I am the second test user.

Contact us:
Phone: 532 65 234
E-mail: group16@group16.no

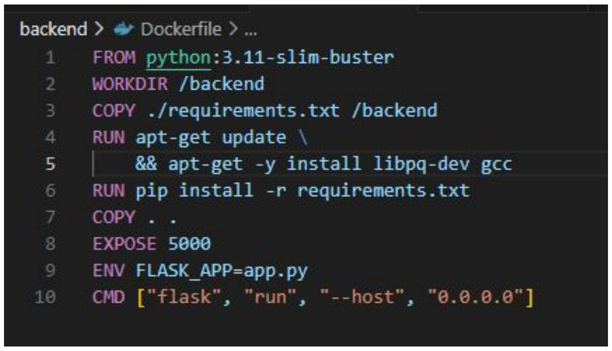
The source code for this selected application can be found on our GitHub repository https://github.com/JakubMroz4/DAT515-H23-cloud-computing.

3 DOCKER CONTAINERIZATION

The Docker containerization phase drew upon the insights and experience gained in the labs of this course. Our web application was packaged into Docker images, one for each part of it. We ourselves had to build the frontend and backend images, while Postgresql image was available on the Docker Hub and we only had to set it up with environmental variables for connection.

- Learning Docker Fundamentals: We have started this phase with getting the basic knowledge on docker on course lectures and labs. We have learned that containers are a different approach than launching the software on virtual machines. Container approach allows for abstracting the environment of the application, without simulating the OS like VMs do. This allows for better control over resources and infrastructure. Docker is a platform for developing, shipping, and running applications in containers. Containers are lightweight, portable, and isolated environments that package an application and its dependencies together. Docker images serve as templates for containers, and Docker registries store and distribute these images.
- Dockerfile Creation: To containerize the selected web application, we created a Dockerfile. This file defines the environment and dependencies required for the application to run within a Docker container. In the file we specified the necessary instructions, including installing dependencies, configuring environment variables (this step is often done later when running the container, but some variables can be defined early), and setting up the application. While working on the Dockerfile, we encountered challenges related to optimizing the image size and installing dependencies. To overcome these challenges, we followed best practices, such as using multi-stage builds, and ensured that unnecessary files and packages were not included in the final image.

Figure 3: Dockerfile of the backend



```
FROM python:3.11-slim-buster
WORKDIR /backend
COPY ./requirements.txt /backend
RUN apt-get update \
    && apt-get -y install libpq-dev gcc
RUN pip install -r requirements.txt
COPY . .
EXPOSE 5000
ENV FLASK_APP=app.py
CMD ["flask", "run", "--host", "0.0.0.0"]
```

- Image Building: Building the docker images involved running said Dockerfiles. During this process we got feedback on whether something was wrong with Dockerfiles, or it was good and could be built. Usually there was no problem with the building itself, if the Dockerfile was setup correctly, but in one scenario where we tried building the image on the VM we were provided with on school infrastructure which was running Ubuntu 22.04, the build process could not be finished as docker could not connect to PyPi and download Flask dependencies. We solved this by building the image on our machines and uploading it to Docker Hub.

- Local Testing: To test the application locally, we have set up a docker-compose file which starts container of frontend, backend and database from one file, sets up most of environmental variables and port exposure used for networking. This allowed us to test the application as a whole relatively quickly. The docker-compose file can be started on any machine with docker, so it can be used to test connectivity to different environments like AWS as well.
- Understanding Benefits: Containerization offers improved isolation, ensuring that the application and its dependencies are encapsulated and do not interfere with the host system or other containers. It also enhances application portability, allowing for seamless deployment across different environments, such as development, testing, and production. These benefits simplify the management of complex applications, promote consistency, and streamline the deployment process. By following these steps and understanding Docker fundamentals, we successfully containerized the web application, optimizing its deployment, and facilitating efficient local testing

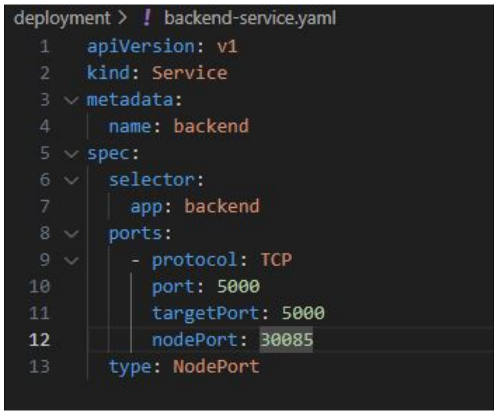
Problems which we have encountered with docker were also related to wrong order of operations done in Dockerfiles. This resulted in some files imported from dockerfile directory being overwritten, which would make the software inside malfunction with missing dependencies or configurations. We have fixed this problem by reading more on Dockerfile commands and changing the order of operations.

4 KUBERNETES ORCHESTRATION

Kubernetes orchestration is a powerful tool for container management.

- Kubernetes Architecture: Normally the Kubernetes cluster architecture consists of control plane and worker nodes. The control plane controls and schedules the cluster, while worker nodes run the containers. The smallest particle in kubernetes is a Pod, which runs the container or multiple containers. Pods are often abstracted and not runned directly. Kubernetes Deployment helps in keeping the containers in pods in desired state, by specifying things like environment variables and health checks. Kubernetes Services are responsible for setting up networking for pods.
- Configuration Creation: For our application we have separate deployments and services for every part of the application, meaning each frontend, backend and db gets its own deployment and service. In addition to that, the database has persisent volume and persistent volume claim files which allow for files to be stored when the kubernetes pod with db is stopped. Our application is rather small, so we run the pods on the control plane to simplify the deployment process.
- Local Deployment: To deploy our application in Kubernetes, we need to first set Kubernetes up on the machine, on top of the Docker installation. Kubelet, kubeadm, and kubectl have to be installed, as well as containerd - the container runtime used by us. We also use networking plugins to enable networking between containers. Most of this setup is taken from the course labs.
- Advanced Features: Discuss any experiments or implementations of advanced features within Kubernetes, such as automatic scaling and rolling updates. Health checks allow Kubernetes to know when to restart a pod. Health checking was added to the backend deployments file. We have experimented with kubernetes secrets, which are meant to store sensitive data like password securely, but we decided not to use it during the development as many environmental variables were often changing and it would make deployment a longer process. This would increase the iteration time between changes considerably.

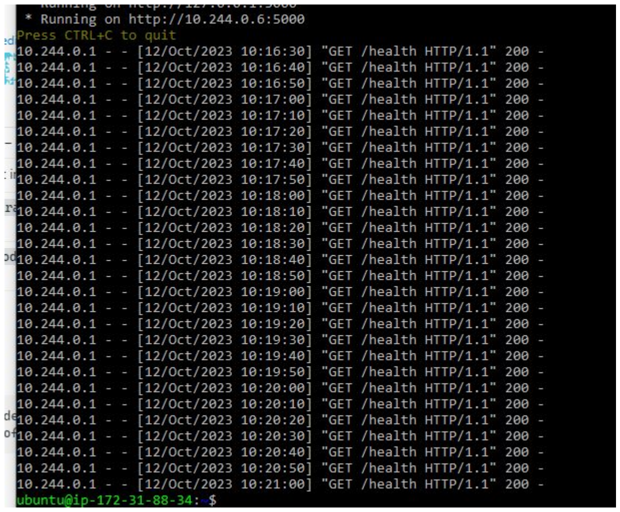
Figure 4: Backend service file



deployment > backend-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  selector:
    app: backend
  ports:
    - protocol: TCP
      port: 5000
      targetPort: 5000
      nodePort: 30085
  type: NodePort
```

Figure 5: Backend container receiving health checks



```
 * Running on http://10.244.0.6:5000
Press CTRL+C to quit
10.244.0.1 - - [12/Oct/2023 10:16:30] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:16:40] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:16:50] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:17:00] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:17:10] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:17:20] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:17:30] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:17:40] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:17:50] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:18:00] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:18:10] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:18:20] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:18:30] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:18:40] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:18:50] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:19:00] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:19:10] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:19:20] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:19:30] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:19:40] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:19:50] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:20:00] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:20:10] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:20:20] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:20:30] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:20:40] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:20:50] "GET /health HTTP/1.1" 200 -
10.244.0.1 - - [12/Oct/2023 10:21:00] "GET /health HTTP/1.1" 200 -
ubuntu@ip-172-31-88-34:~$
```

5 AWS EC2 DEPLOYMENT

The primary focus of this report is the deployment of the containerized web application on an AWS EC2 instance. Key steps in this process included configuring the AWS environment, launching an EC2 instance, transferring Docker and Kubernetes images, executing the application, and configuring port access and networking. This phase provided practical experience in deploying containerized applications in a cloud environment

- AWS Setup: : Our primary focus in this phase was on the deployment of the containerized web application on an AWS EC2 instance. We began by setting up an AWS EC2 instance and configuring security groups to manage network access.
- Image Transfer: : To prepare our Docker image for deployment, we needed to transfer it to the AWS EC2 instance. We documented the process of transferring the Docker images. To get the deployment, service and volume files, we could clone the repository on the instance. This step was essential for making our containerized application available within the AWS environment.
- Application Execution: With the docker images on the cloud, and deployments files ready, we could start running them with kubelet. Obviously with each layer added to the deployment, the launch time for us was longer, so normally we were not testing the application in AWS.
- Cloud Proficiency: Throughout this deployment process on AWS, we expanded our cloud computing skills by working with various AWS services. We emphasized the use of AWS EC2 for hosting our application, but we also gained experience in managing security groups and optimizing network access within the AWS ecosystem. This hands-on experience contributed to our proficiency in utilizing cloud resources effectively.

6 CONCLUSION AND DISCUSSION

Project Reflection: In reflecting on our project, we've gained valuable insights and experiences. We faced various challenges along the way, especially with container networking and software problems inside the containers. We learned to adapt to new technologies and navigate the complexities of cloud environments.

Skills Acquired: Through this project, we acquired proficiency in key technologies that are highly relevant in today's technology landscape. Docker and Kubernetes played a crucial role in containerizing and orchestrating our application, enabling efficient deployment and management. Additionally, our experience with AWS EC2 enhanced our cloud computing skills, emphasizing the importance of cloud services in the modern development workflow. These newfound skills are invaluable assets that will undoubtedly benefit us in future projects.

7 FUTURE WORK

In future iterations of the web application, several enhancements could be considered:

- Kubernetes secrets: Utilizing Kubernetes Secrets to securely store sensitive information like database password, enhancing application security. These improvements would further optimize the application's security.
- More Health checks: Right now we only use liveness probe, but we should also use the readiness probe which is used to tell Kubernetes when the pod can start accepting traffic.
- Scalability: Vertical scaling, also known as "scaling up" or "resizing," is a crucial aspect of managing applications in Kubernetes, particularly when dealing with resource-intensive workloads or varying performance requirements. It involves adjusting the capacity of individual pods or containers within a deployment, StatefulSet, or ReplicaSet. In your specific case, vertical scaling in Kubernetes becomes relevant because our application does not need many resources, so we could minimize their usage. Scaling up could also be beneficial if there would be increased load on the backend server.
- More features: we could add a feature for user registration and login, so that the comments on the web application are tied to users. This would user authentication, which realistically would force us to change flask to a more powerful web framework for the backend.

REFERENCES

- https://github.com/JakubMroz4/DAT515-H23-cloud-computing
- https://www.airplane.dev/blog/deploy-postgres-on-kubernetes
- https://kubernetes.io/docs/tasks/configure-pod-container/configure-livenes
- https://kubernetes.io/docs/concepts/configuration/secret/
- https://kubernetes.io/docs/tutorials/kubernetes-basics/
- https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html
- https://www.freecodecamp.org/news/how-to-dockerize-a-react-applicatior
- https://www.freecodecamp.org/news/how-to-dockerize-a-flask-app/
- https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html