Deploying a Microservice Application using Docker Swarm and Kubernetes

& Review on Docker Swarm and Kubernetes.

Table of Contents

Introduction

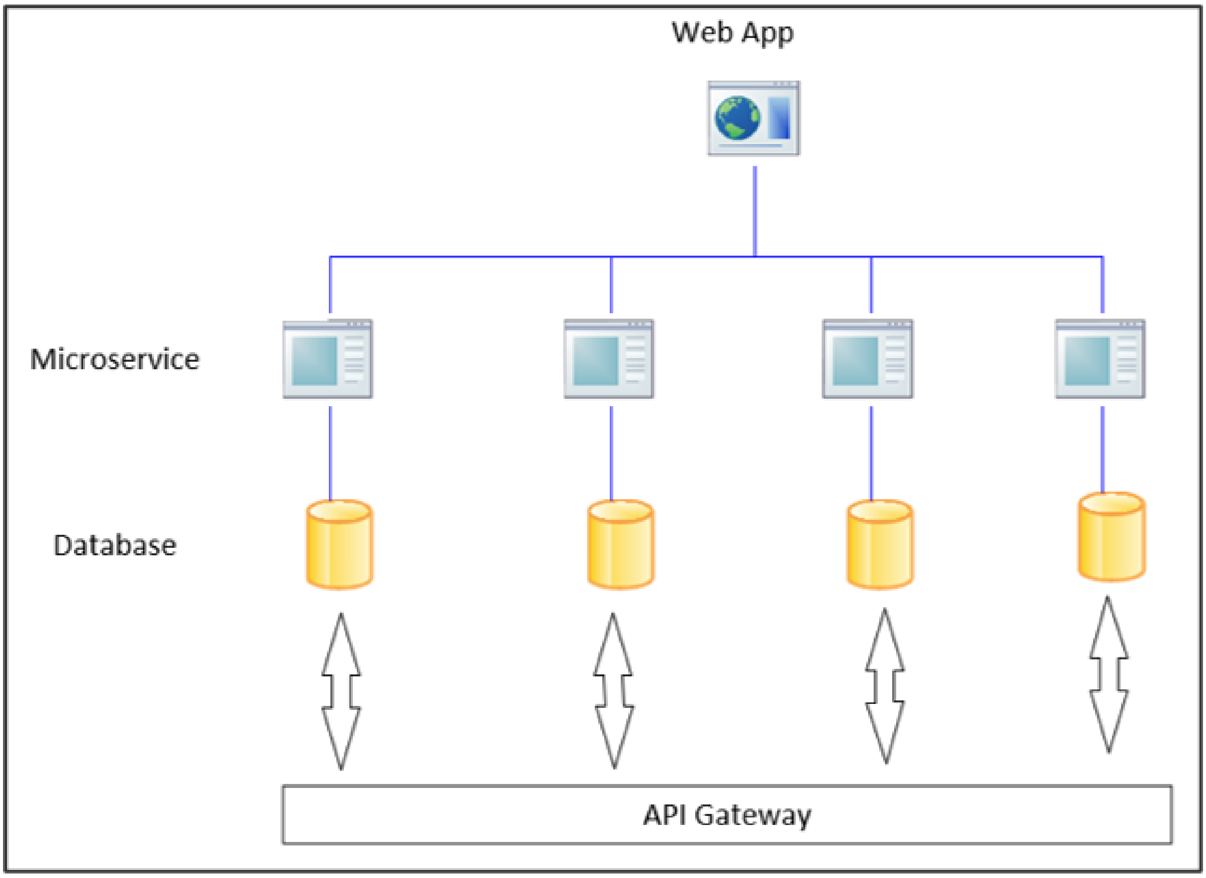
Concept of microservices and containerization

The introduction of Cloud Computing has marked a significant revolution in the burgeoning technological industry. It serves as a model that enables convenient, on-demand network access to a shared pool of configurable computing resources, allowing rapid provisioning and release with minimal service provider interaction or management effort. For technology professionals and most organizations, cloud computing has become not just an option but, if not imperative. This advancement has significantly complemented microservices and containerization, providing scalable and cost-effective infrastructure. It facilitates the deployment of microservices-based applications through container orchestration platforms like Kubernetes within cloud environments.

Microservices: Microservices is an architectural style where an application is structured as a collection of loosely coupled, independently deployable services. Each service focuses on a specific business capability and communicates with other services via APIs. This approach enables flexibility, scalability, and easier maintenance as different parts of the application can be developed, updated, and scaled independently.

Containerization: Containerization is a method of packaging, deploying, and running applications and their dependencies in isolated environments called containers. Containers encapsulate the application, libraries, and necessary configurations, allowing them to run consistently across different computing environments. This technology, popularized by tools like Docker, provides a lightweight, portable, and efficient way to deploy software, ensuring consistency from development to production environments.

Figure 1.1 Example of microservices architecture [1]



In other words, we can sum this up as:

- Microservices are about the design of software.
- Containers are about packaging software for deployment.

So, we can choose whether to use a container for hosting a microservice. But to get full value from both, it is significantly better to run microservices within containers.

Deploying an entire application to a single VM introduces a single point of failure risk, whether or not a microservice architecture has been used. But spreading the application through microservices across multiple containers results in fully exploiting the value of both by providing resilience as well as agility through scaling and improvements targeting specific services without negatively impacting the entire application.

Importance of container orchestration for microservices deployment

Container orchestration plays a pivotal role in the deployment and management of microservices-based applications. One of the biggest benefits of container orchestration is that it simplifies operations. Automating tasks not only helps to minimize the effort and complexity of managing containerized apps, it also translates into many other advantages.

Reliable application development: Container orchestration tools help make app development faster and repeatable. This increases deployment velocity and makes them ideal for supporting agile development approaches like DevOps.

Scalability and load balancing: Container orchestration allows you to scale container deployments up or down based on changing workload requirements. You also get the scalability of cloud if you choose a managed offering and scale your underlying infrastructure on demand. Load balancing capabilities ensure even distribution of traffic across multiple containers, optimizing performance and resource utilization.

Lower cost: Containers require fewer resources than virtual machines, reducing infrastructure and overhead costs. In addition, container orchestration platforms require less human capital and time, yielding additional cost savings.

Enhanced security: Container orchestration allows you to manage security policies across platforms and helps reduce human errors that can lead to vulnerabilities. Containers also isolate application processes, decreasing attack surfaces and improving overall security.

High availability: It’s easier to detect and fix infrastructure failures using container orchestration tools. If a container fails, a container orchestration tool can restart or replace it automatically, helping to maintain availability and increase application uptime.

Better productivity: Container orchestration boosts developer productivity, helping to reduce repetitive tasks and remove the burden of installing, managing, and maintaining containers.

Docker Swarm Deployment

Docker Swarm is a container orchestration tool provided by Docker, designed to manage a cluster of Docker hosts and facilitate the deployment and scaling of containerized applications across multiple nodes.

Docker Node Roles

Manager/Leader: When a cluster is established, the Raft consensus algorithm is used to assign one of them as the "leader node." The leader node makes all of the swarm management and task orchestration decisions for the swarm. If the leader node becomes unavailable due to an outage or failure, a new leader node can be selected using the Raft consensus algorithm.

Worker Node: In a docker swarm with numerous hosts, each worker node functions by receiving and executing the tasks that are allocated to it by manager nodes. By default, all manager modes are also worker nodes and are capable of executing tasks when they have the resources available to do so.

Services: are the definition of tasks or jobs that need to be executed within the Swarm cluster. They define how containers should be deployed, including details like the number of replicas, networking, ports, and other configurations. They maintain a desired state, ensuring that a specified number of replica containers are always running across the worker nodes, and they automatically handle scaling, load balancing, and fault tolerance.

Docker Swarm Implementation

Play with Docker (PWD) is a browser-based Docker environment that allows users to run Docker commands in a virtual terminal and deploy services within a Docker environment. The steps for the implementation are outlined below using appropriate screenshots where applicable.

a) Deploying flight service in docker swarm

- Visit the Play with Docker website: Play with Docker
- Sign in with your Docker ID or continue as an anonymous user.
- Click on "Start" to create a new instance/session.

This will give you access to a terminal interface where you can run Docker commands see the figure below:

Figure2.1 shows managers and workers.

Click on the “Editor” button, **paste** the yaml file and save

Figure2.2 docker-compose.yaml

/root ✕

Save   Reload

```yaml
version: '3'
services:
  flight-app:
    image: kennedymwai/flightservice
    ports:
      - 9091:9091
    networks:
      - flightservice
    depends_on:
      - docker-mysql
    deploy:
      replicas: 1
  docker-mysql:
    image: mysql:5
    environment:
      MYSQL_ROOT_PASSWORD: test1234
      MYSQL_DATABASE: reservation
      MYSQL_ROOT_HOST: '%'
    deploy:
      replicas: 1
    ports:
      - "6666:3306"
    networks:
      - flightservice
networks:
  flightservice:

```

Enter this command on manager node **mv root docker-compose.yaml** This command renames root to docker-compose.yaml. After executing this command in the terminal, the file will be renamed and can be accessed and used as docker-compose.yaml. see on the figure below.

Figure2.3 Renaming the file

```
# The PWD team.                                                #
################################################################
[manager3] (local) root@192.168.0.4 ~
$ ls
root
[manager3] (local) root@192.168.0.4 ~
$ mv root docker-compose.yaml
[manager3] (local) root@192.168.0.4 ~
$ ls
docker-compose.yaml
[manager3] (local) root@192.168.0.4 ~
$
```

After renaming the file, write the command **docker stack deploy -c docker-compose.yaml flight-app** to deploy the flight and the mysql service

Figure2.4 deployment.

```
[manager3] (local) root@192.168.0.4 ~
$ docker stack deploy -c docker-compose.yaml flight-app
Creating network flight-app_flightservice
Creating service flight-app_docker-mysql
Creating service flight-app_flight-app
[manager3] (local) root@192.168.0.4 ~
$ docker service ls
```

Run the command **docker service ls** the confirm the services are running

Figure2.5 dockers services

```
$ docker service ls
ID             NAME                      MODE         REPLICAS   IMAGE                               PORTS
vnawa4jvjpsk   flight-app_docker-mysql   replicated   1/1        mysql:5                             *:6666->3306/tcp
rgsywgvdhwlr   flight-app_flight-app     replicated   1/1        kennedymwai/flightservice:latest    *:9091->9091/tcp
[manager3] (local) root@192.168.0.4 ~
```

b) Upscaling the services

To create a Docker Swarm service that can be scaled up or down, you can use a docker-compose.yml file to define your service and then deploy it using Docker Swarm commands.

Recreate a docker-compose.yml file with a service definition:

Figure2.6 new docker-compose.yml

Save Reload

```
version: '3'
services:
  flight-app:
    image: kennedymwai/flightservice
    ports:
      - 9091:9091
    networks:
      - flightservice
    depends_on:
      - docker-mysql
    deploy:
      replicas: 2
  docker-mysql:
    image: mysql:5
    environment:
      MYSQL_ROOT_PASSWORD: test1234
      MYSQL_DATABASE: reservation
      MYSQL_ROOT_HOST: '%'
    deploy:
      replicas: 2
    ports:
      - "6666:3306"
    networks:
      - flightservice
networks:
  flightservice:
```

Replace flight -service app by deploying the service again with the same command **docker stack deploy -c docker-compose.yaml flight-app.**

Alternatively

docker service scale"*container-id*"=3 This command will scale the webapp service to have 3 replicas. Adjust the number according to your specifications. Keep in mind that Docker Swarm will manage the distribution of replicas across available nodes in the Swarm cluster.

Figure2.7 Scaling up the services

```
$ docker service ls
ID            NAME                    MODE        REPLICAS   IMAGE                               PORTS
vnawa4jvjpsk  flight-app_docker-mysql replicated  2/2        mysql:5                             *:6666->3306/tcp
rgsywgvdhwlr  flight-app_flight-app   replicated  1/1        kennedymwai/flightservice:latest    *:9091->9091/tcp
[manager3] (local) root@192.168.0.4 ~
$ docker service scale rgsywgvdhwlr=3
rgsywgvdhwlr scaled to 3
overall progress: 3 out of 3 tasks
1/3: running   [==================================================>]
2/3: running   [==================================================>]
3/3: running   [==================================================>]
verify: Service converged
[manager3] (local) root@192.168.0.4 ~
$ docker service ls
ID            NAME                    MODE        REPLICAS   IMAGE                               PORTS
vnawa4jvjpsk  flight-app_docker-mysql replicated  2/2        mysql:5                             *:6666->3306/tcp
rgsywgvdhwlr  flight-app_flight-app   replicated  3/3        kennedymwai/flightservice:latest    *:9091->9091/tcp
```

You could also Execute the same command by just writing the first 3 letters of the container id i.e. **docker services scale vna=2** as shown below

Figure 2.7 Scalling up services part 2

```
$ docker service ls
ID            NAME                    MODE        REPLICAS   IMAGE                               PORTS
vnawa4jvjpsk  flight-app_docker-mysql replicated  1/1        mysql:5                             *:6666->3306/tcp
rgsywgvdhwlr  flight-app_flight-app   replicated  1/1        kennedymwai/flightservice:latest    *:9091->9091/tcp
[manager3] (local) root@192.168.0.4 ~
$ docker service scale vna=2
vna scaled to 2
overall progress: 2 out of 2 tasks
1/2: running   [==================================================>]
2/2: running   [==================================================>]
verify: Service converged
[manager3] (local) root@192.168.0.4 ~
$ docker service ls
ID            NAME                    MODE        REPLICAS   IMAGE                               PORTS
vnawa4jvjpsk  flight-app_docker-mysql replicated  2/2        mysql:5                             *:6666->3306/tcp
rgsywgvdhwlr  flight-app_flight-app   replicated  1/1        kennedymwai/flightservice:latest    *:9091->9091/tcp
```

visualization gets the graphical representation or interface that provides a visual overview of the Docker Swarm cluster's nodes and services. It offers a real-time depiction of the Swarm's infrastructure, including the status of nodes, running containers, and services distributed across the cluster. This visualization tool aids in monitoring the Swarm's health, resource allocation, and container distribution, allowing users to easily observe the state and performance of the cluster. Thisa process is performed by running command **docker run -d -p 5000:8080 -v /var/run/docker.sock:/var/run/docker.sock dockersamples/visualizer**

docker run: command you've provided is used to start the Docker container dockersamples/visualizer, which is a containerized tool for visualizing Docker Swarm services. This command does the following:

-d: Runs the container in detached mode, which means it runs in the background.

-p 5000:8080: Maps port 8080 inside the container to port 5000 on the host machine, allowing access to the visualizer via port 5000 on the host.

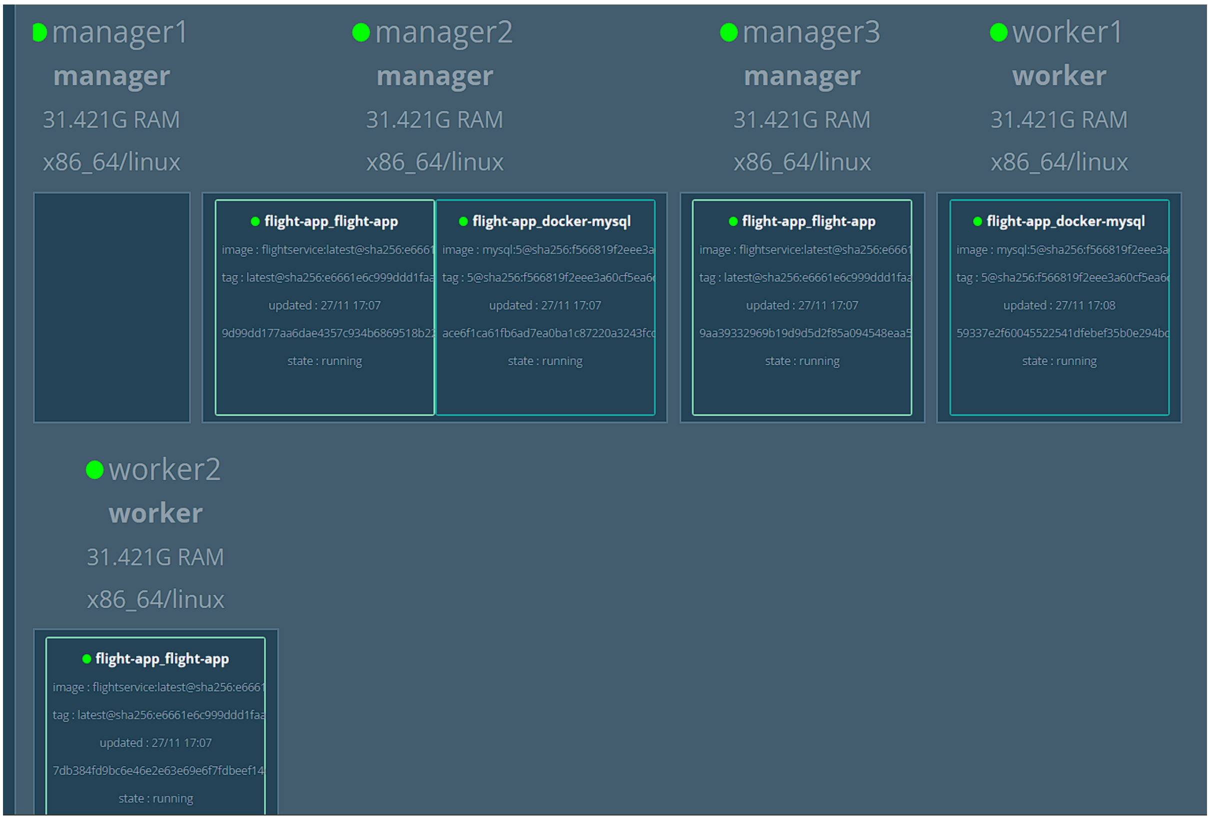
-v /var/run/docker.sock:/var/run/docker.sock: Mounts the Docker daemon socket inside the container. This allows the container to communicate with the Docker daemon on the host, enabling it to gather information about running containers and services.

Figure 2.8 running the visualization

```
ID             NAME                       MODE         REPLICAS   IMAGE                                PORTS
vnawa4jvjpsk   flight-app_docker-mysql    replicated   2/2        mysql:5                              *:6666->3306/tcp
rgsywgvdhwlr   flight-app_flight-app      replicated   3/3        kennedymwai/flightservice:latest     *:9091->9091/tcp
[manager3] (local) root@192.168.0.4 ~
$ docker run -d -p 5000:8080 -v /var/run/docker.sock:/var/run/docker.sock dockersamples/visualizer
to see the visualizer
Unable to find image 'dockersamples/visualizer:latest' locally
latest: Pulling from dockersamples/visualizer
ddad3d7c1e96: Pull complete
3a8370f05d5d: Pull complete
71a8563b7fea: Pull complete
119c7e14957d: Pull complete
28bdf67d9c0d: Pull complete
12571b9c0c9e: Pull complete
e1bd03793962: Pull complete
3ab99c5ebb8e: Pull complete
94993ebc295c: Pull complete
021a328e5f7b: Pull complete
Digest: sha256:530c863672e7830d7560483df66beb4cbbcd375a9f3ec174ff5376616686a619
Status: Downloaded newer image for dockersamples/visualizer:latest
60ed764ac948a9e07393ddc318b24bdf9ee4f26932d82d65bae048cad4e187f0
```

The command, in summary, starts the **dockersamples/visualizer** container, providing it with access to the Docker daemon on the host machine and making the visualization accessible on port 5000.

Figure2.9 shows the visualization



C) Upgrade and rolling back

In Docker Swarm, upgrading a service involves updating the service to a new version, whether it's a newer image or configuration. Docker provides the **docker service update --image <new_image>:<tag> <service_name>**command to perform service upgrades in the Swarm cluster. In this case the flight service app will be **docker service update –image:5.7 vna**

Figure2.10 upgrading services

```
$ docker service ls
ID             NAME                       MODE         REPLICAS   IMAGE                                 PORTS
vnawa4jvjpsk   flight-app_docker-mysql    replicated   2/2        mysql:5                               *:6666->3306/tcp
rgsywgvdhwlr   flight-app_flight-app      replicated   3/3        kennedymwai/flightservice:latest      *:9091->9091/tcp
[manager3] (local) root@192.168.0.4 ~
$ docker service update --image mysql:5.7 vna
vna
overall progress: 2 out of 2 tasks
1/2: running   [==================================================>]
2/2: running   [==================================================>]
verify: Service converged
[manager3] (local) root@192.168.0.4 ~
$ docker service ls
ID             NAME                       MODE         REPLICAS   IMAGE                                 PORTS
vnawa4jvjpsk   flight-app_docker-mysql    replicated   1/2        mysql:5.7                             *:6666->3306/tcp
rgsywgvdhwlr   flight-app_flight-app      replicated   3/3        kennedymwai/flightservice:latest      *:9091->9091/tcp
```

Additionally, you can set a rollback condition in case of failure during the update rolling back a service to a previous version can be achieved using the **docker service rollback <service_name>** command followed by the service name. This command allows you to revert to the previous version.

Figure2.10 rolling back the service

```
$ docker service rollback vna
vna
rollback: manually requested rollback
overall progress: rolling back update: 2 out of 2 tasks
1/2: running   [==================================================>]
2/2: running   [==================================================>]
verify: Service converged
[manager3] (local) root@192.168.0.4 ~
$ docker service ls
ID             NAME                       MODE         REPLICAS   IMAGE                                 PORTS
vnawa4jvjpsk   flight-app_docker-mysql    replicated   2/2        mysql:5                               *:6666->3306/tcp
rgsywgvdhwlr   flight-app_flight-app      replicated   3/3        kennedymwai/flightservice:latest      *:9091->9091/tcp
[manager3] (local) root@192.168.0.4 ~
```

Docker Swarm features:

Cluster management integrated with Docker Engine: Use the Docker Engine CLI to create a swarm of Docker Engines where you can deploy application services. You don't need additional orchestration software to create or manage a swarm.

Scalability and Load Balancing: Docker Swarm allows easy scaling of services by adding or removing container replicas across the cluster. It automatically distributes incoming requests among the containers, providing load balancing for improved performance.

High Availability: Swarm ensures high availability by replicating containers across multiple nodes. If a node fails, containers are automatically rescheduled and restarted on other healthy nodes to maintain service availability.

Mult-host Networking: Swarm provides built-in overlay networking, allowing containers across different nodes to communicate securely. This enables seamless communication between containers regardless of the node they are running on.

Service Discovery: Swarm includes a built-in DNS service that enables service discovery within the cluster. Containers can easily discover and communicate with other containers using service names rather than specific IP addresses.

Rolling Updates: At rollout time you can apply service updates to nodes incrementally. The swarm manager lets you control the delay between service deployment to different sets of nodes. If anything goes wrong, you can roll back to a previous version of the service.

Kubernetes Deployment
An open-source container orchestration platform that consist of a set of worker machines, called nodes, designed for automating deployment, scaling, and management of containerized applications. It allows users to manage clusters of containers efficiently and helps in automating various tasks related to deploying, scaling, and maintaining application workloads.

Kubernetes is considered the most popular container orchestration platform. Together with other tools in the container ecosystem, Kubernetes enables a company to deliver a highly productive platform-as-a-service (PaaS) that addresses many of the infrastructures- and operations related tasks and issues around cloud-native application development, so that development teams can focus exclusively on coding and innovation. Core components include

Kubelet: An agent that runs on each node in the cluster. It makes sure that containers are running in a pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

Pods: A pod is the smallest deployable unit in Kubernetes and represents one or more tightly coupled containers sharing resources such as storage volumes, networking, and an IP address. Containers within a pod share the same lifecycle and are scheduled together on the same node. Pods enable easy management of related containers that need to work together.

Deployment: is a higher-level abstraction that manages the creation and scaling of ReplicaSets. It also manages the lifecycle of pods.

Services: an abstraction that defines a logical set of Pods and a policy by which to access them by providing a stable endpoint for accessing the containers, regardless of their actual location or changes in the cluster's topology. It also provides a consistent way to expose a group of pods and make them accessible to other services or external users.

Control Plane components: The control plane consists of multiple components that manage and control the cluster's state and perform cluster management tasks. Key components include:

API Server: Exposes the Kubernetes API, which users, other components, and external systems interact with to manage the cluster.

Scheduler: Assigns pods to nodes based on resource availability and workload requirements.

Controller Manager: Monitors the state of the cluster and ensures that the desired state matches the actual state.

etcd: A distributed key-value store that stores cluster data, configurations, and states

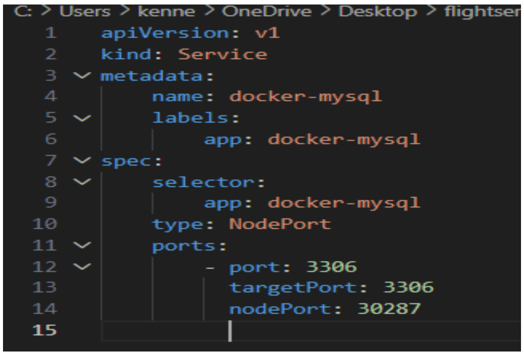
Kubernetes Implementation

For the implementation of deploying a microservice using Kubernetes, the project uses Docker Desktop.

STEP I. Composing YAML Files to deploy services.

In other to deploy our flight service microservice application, we created five (5) .yml files for both the flight application and the backend database. Attached are screenshots of the .yml file. The Service file configurations are used to expose the application to other services running within the cluster. The Deployment files are the instruction set used for deploying and scaling the microservice application within the cluster. The configmap file contains the details needed to set up the database and tables

Figure3.1 mysql_service.yml



C: > Users > kenne > OneDrive > Desktop > flightser

```yaml
apiVersion: v1
kind: Service
metadata:
    name: docker-mysql
    labels:
        app: docker-mysql
spec:
    selector:
        app: docker-mysql
    type: NodePort
    ports:
        - port: 3306
          targetPort: 3306
          nodePort: 30287
```

Figure 3.2 mysql_deplyment.yml

C: > Users > kenne > OneDrive > Desktop > flightservices > flightservices

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: docker-mysql
  labels:
    app: docker-mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: docker-mysql
  template:
    metadata:
      labels:
        app: docker-mysql
    spec:
      containers:
      - name: docker-mysql
        image: mysql
        ports:
        - containerPort: 3306
        env:
        - name: MYSQL_DATABASE
          valueFrom:
            configMapKeyRef:
              name: db-config
              key: database
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: password
        volumeMounts:
        - name: db-init-volume
          mountPath: /docker-entrypoint-initdb.d
        - name: db-data-volume
          mountPath: /var/lib/mysql/
      volumes:
      - name: db-init-volume
        configMap:
          name: db-config
      - name: db-data-volume
```

Figure 3.3 mysql_config.yml

C: > Users > kenne > OneDrive > Desktop > flightservices > flightservices > flight > ! db-config.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  database: "reservation"
  initdb.sql:
    USE reservation;

    CREATE TABLE flight
    (
      ID INT  NOT NULL AUTO_INCREMENT,
      FLIGHT_NUMBER VARCHAR(20)  NOT NULL,
      OPERATING_AIRLINES VARCHAR(20)  NOT NULL,
      DEPARTURE_CITY VARCHAR(20)  NOT NULL,
      ARRIVAL_CITY VARCHAR(20)  NOT NULL,
      DATE_OF_DEPARTURE DATE  NOT NULL,
      ESTIMATED_DEPARTURE_TIME TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
      PRIMARY KEY (ID)
    );

```

Figure3.4 flight_deployment.yml

C: > Users > kenne > OneDrive > Desktop > flightservices > flightse

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flight-app
  labels:
    app: flight-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: flight-app
  template:
    metadata:
      name: flight-app
      labels:
        app: flight-app
    spec:
      containers:
        - name: flight-app
          image: kennedymwai/flightservice

```

Figure3.5 flight_service.yml

C: > Users > kenne > OneDrive > Desktop > flightservices >

```yaml
apiVersion: v1
kind: Service
metadata:
    name: flight-app
    labels:
        app: flight-app
spec:
    type: NodePort
    ports:
        - port: 9814
          targetPort: 9814
          nodePort: 30288
    selector:
        app: flight-app

```

Step II: Deployment

➢ To deploy the microservice, run the command “kubectl apply -f follow by the name of the .yml files that have been written.

Figure3.6 Creating the services

```
Microsoft Windows [Version 10.0.22621.2715]
(c) Microsoft Corporation. All rights reserved.

C:\Users\kenne>cd C:\Users\kenne\OneDrive\Desktop\flightservices\flightservices

C:\Users\kenne\OneDrive\Desktop\flightservices\flightservices>kubectl create -f docker-mysql-configmap.yml,docker-mysql-deployment.yml,docker-mysql-service.yml,flight-service-deployment.yml,flight-service-svc.yml
configmap/mysql-initdb-config created
deployment.apps/docker-mysql created
service/docker-mysql created
deployment.apps/flight-app created
service/flight-app created
```

➢ To view the services and deployments running within your Kubernetes cluster, run the following commands. “kubectl get pods” “kubectl get services” or “kubectl get deployments”.

Figure3.7 Shows deployments.

```
C:\Users\kenne\OneDrive\Desktop\flightservices\flightservices>kubectl create -f docker-mysql-configmap.yml,docker-mysql-deployment.yml,docker-mysql-service.yml,flight-service-deployment.yml,flight-service-svc.yml
configmap/mysql-initdb-config created
deployment.apps/docker-mysql created
service/docker-mysql created
deployment.apps/flight-app created
service/flight-app created

C:\Users\kenne\OneDrive\Desktop\flightservices\flightservices>kubectl get pods
NAME                           READY   STATUS    RESTARTS      AGE
docker-mysql-6bcf9c4cc9-45lls  1/1     Running   0             3m10s
flight-app-86d569755c-bzghz    1/1     Running   1 (80s ago)   3m10s

C:\Users\kenne\OneDrive\Desktop\flightservices\flightservices>kubectl get deplpyments
error: the server doesn't have a resource type "deplpyments"

C:\Users\kenne\OneDrive\Desktop\flightservices\flightservices>kubectl get deployments
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
docker-mysql   1/1     1            1           4m5s
flight-app     1/1     1            1           4m4s

C:\Users\kenne\OneDrive\Desktop\flightservices\flightservices>kubectl get services
NAME           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
docker-mysql   NodePort    10.97.103.177   <none>        3306:30287/TCP   4m26s
flight-app     NodePort    10.97.233.17    <none>        9814:30288/TCP   4m25s
kubernetes     ClusterIP   10.96.0.1       <none>        443/TCP          4d2h
```

Once the MySQL service is running in your Minikube cluster, you can retrieve the URL to access it by executing the following command:

Figure3.8 Test the Mysql connection.

```
C:\Users\kenne\OneDrive\Desktop\flightservices\flightservices\flights>minikube service mysql --url
http://127.0.0.1:1070
!   Because you are using a Docker driver on windows, the terminal needs to be open to run it.
```

The command above will output the URL that can use to access the MySQL service running within Minikube via Docker Desktop. “**Port 1070**”

Figure3.9 Mysql Connection configuration

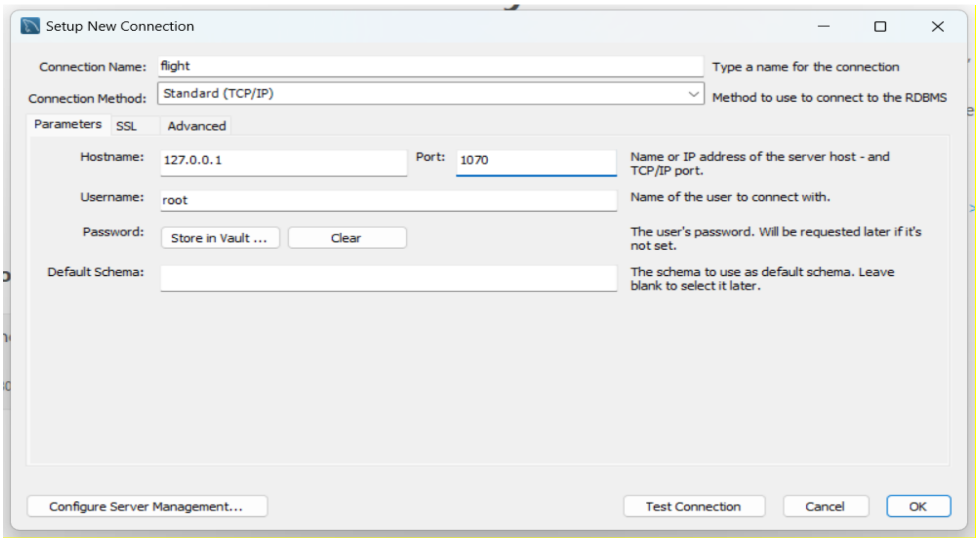
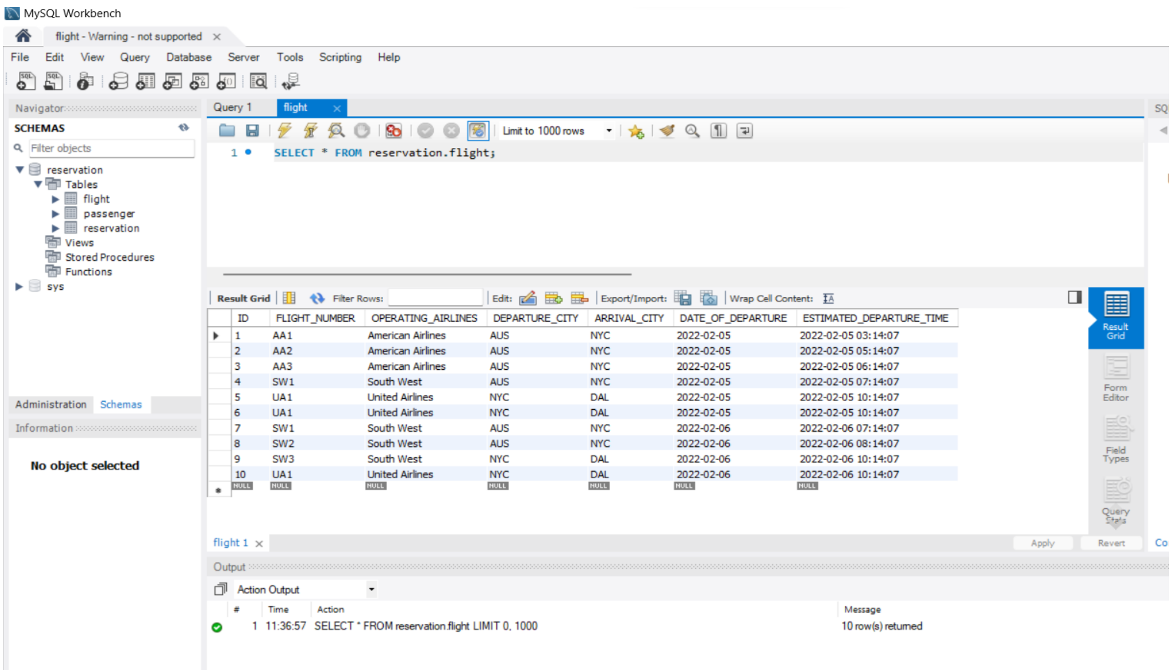


Figure3.10 Mysql tables



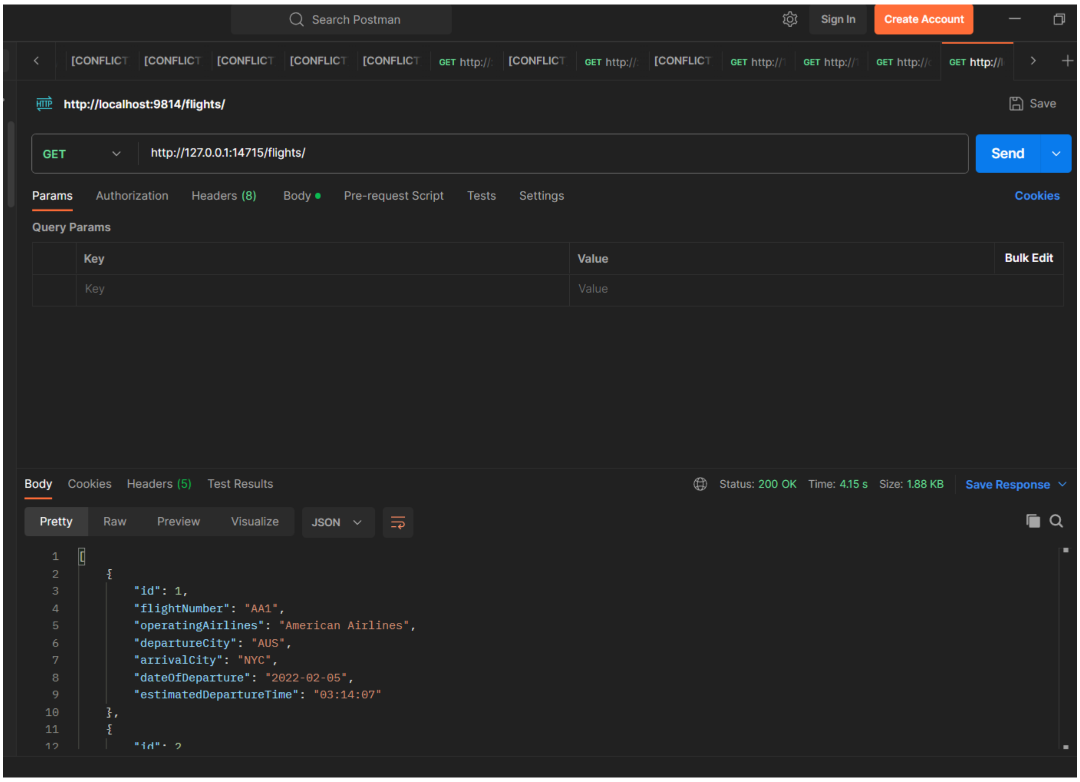
This command below retrieves the URL that you can use to access the flight-service running in your Minikube cluster. Executing this command in your terminal will output the URL you can use to access your service through postman

Figure3.11 flight service url

```
C:\Users\kenne\OneDrive\Desktop\flightservices\flightservices\flights>kubectl get services
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
app-service   NodePort    10.99.167.159   <none>        8080:30100/TCP   7m39s
kubernetes    ClusterIP   10.96.0.1       <none>        443/TCP          5d2h
mysql         NodePort    10.100.55.248   <none>        3306:30200/TCP   49m

C:\Users\kenne\OneDrive\Desktop\flightservices\flightservices\flights>minikube service app-service --url
http://127.0.0.1:14715
!   Because you are using a Docker driver on windows, the terminal needs to be open to run it.
```

Figure3.12 postman Test



Step III: Scaling the deployments.

To scale the services running within the cluster, run the command "kubectl scale - - replicas follow by the number of replicas you wish to scale your application to, then the service you wish to scale, and the name of the service

Figure3.13 scale replicas

```
flight-app     1/1     1           1           5m28s

:\Users\kenne\OneDrive\Desktop\flightservices\flightservices>kubectl scale deployment docker-mysql --replicas=5
deployment.apps/docker-mysql scaled

:\Users\kenne\OneDrive\Desktop\flightservices\flightservices>kubectl scale deployment flight-app --replicas=2
deployment.apps/flight-app scaled
```

Figure3.14 show scale replicas

```
C:\Users\kenne\OneDrive\Desktop\flightservices\flightservices>kubectl get deployments
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
docker-mysql   5/5     5            5           24m
flight-app     2/2     2            2           24m

C:\Users\kenne\OneDrive\Desktop\flightservices\flightservices>
```

Step IV: Upgrading and rolling back.

In the yml file, changed the image tag of the flight-service container from *kennedymwai/flight-service* to *kennedymwai/flight-service:v2.*

Figure3.15 Check Deployment History

```
C:\Users\kenne\OneDrive\Desktop\flightservices\flightservices\flights>kubectl rollout history deployment
deployment.apps/app-deployment
REVISION  CHANGE-CAUSE
2         <none>
3         <none>
4         <none>
5         <none>

deployment.apps/db-deployment
REVISION  CHANGE-CAUSE
1         <none>
```

After applying the updated app YAML to your Kubernetes cluster, it will deploy the application using the new image tag (v2). Then, you can proceed with the rollback steps to revert the deployment back to the previous state by rolling back to the original image (v1).

Figure3.16 shows the history versions.

```
C:\Users\kenne\OneDrive\Desktop\flightservices\flightservices\flights>kubectl rollout history deployment app-deployment --revision=4
deployment.apps/app-deployment with revision #4
Pod Template:
  Labels:       app=web-server
        pod-template-hash=67f9d89698
  Containers:
   flight-service:
    Image:      kennedymwai/flightservice
    Port:       8080/TCP
    Host Port:  0/TCP
    Environment:        <none>
    Mounts:     <none>
  Volumes:      <none>


C:\Users\kenne\OneDrive\Desktop\flightservices\flightservices\flights>kubectl rollout history deployment app-deployment --revision=5
deployment.apps/app-deployment with revision #5
Pod Template:
  Labels:       app=web-server
        pod-template-hash=5d95775bd6
  Containers:
   flight-service:
    Image:      kennedymwai/flightservice:v2
    Port:       8080/TCP
    Host Port:  0/TCP
    Environment:        <none>
    Mounts:     <none>
  Volumes:      <none>
```

The following command will rollback the app-deployment to the state it was in before revision 2, which is the previous version (v1 in this case). Adjust the revision number based on your actual deployment history of your choice.

Figure3.17 shows the rollback app

```
C:\Users\kenne\OneDrive\Desktop\flightservices\flightservices\flights>kubectl rollout undo deployment app-deployment --to-revision=4
deployment.apps/app-deployment rolled back

C:\Users\kenne\OneDrive\Desktop\flightservices\flightservices\flights>
```

Kubernetes Features

- **Container deployment**: Kubernetes deploys a specified number of containers to a specified host and keeps them running in the desired state.
- **Rolling Updates and Rollbacks**:Deployments in Kubernetes enable seamless updates of application versions by gradually rolling out new changes across pods, minimizing downtime. It also supports easy rollbacks to previous stable versions if issues arise
- **Service discovery**: Services in Kubernetes enable stable networking and service discovery by providing a consistent endpoint to access a group of pods, and it distributes traffic across these pods to achieve load balancing.
- **Storage provisioning**: Developers can set Kubernetes to mount persistent local or cloud storage for your containers as needed.
- **Load balancing and scalability**: When traffic to container spikes, Kubernetes can employ load balancing and scaling to distribute it across the network to ensure stability and performance. (It also saves developers the work of setting up a load balancer.)
- **Self-healing for high availability**: When a container fails, Kubernetes can restart or replace it automatically. It can also take down containers that don't meet your health-check requirements.
- **Support and portability across multiple cloud providers**: Kubernetes enjoys broad support across all leading cloud providers. This is especially important for organizations deploying applications to a hybrid cloud or hybrid multi-cloud environment.
- **A growing ecosystem of open-source tools:** Kubernetes also has an ever-expanding stable of usability and networking tools to enhance its capabilities via the Kubernetes API. These include Knative, which enables containers to run as serverless workloads; and Istio, an open-source service mesh.

Evaluation and Conclusion

In summary, it's crucial to underscore the significance of Kubernetes and docker swam in the realm of DevOps and modern application development and deployment. Orchestrations plays a pivotal role in accelerating software delivery and minimizing the time gap between code creation and its deployment in production. It adeptly addresses the issue of disparate development environments and dependencies by encapsulating applications and their prerequisites within containers.

"**CrashLoopBackOff**" errors are common challenges encountered in Kubernetes. The "CrashLoopBackOff" error occurs when a pod repeatedly crashes after starting due to application errors, misconfigurations, or insufficient resources. Troubleshooting involves inspecting pod logs, checking resource configurations, and verifying application settings to identify and resolve the underlying issue. After reviewing the application's configuration, environment variables, and initialization processes. I rectified any misconfigurations causing the application to fail during startup and worked up perfecty.

Docker Swarm is known for its simplicity and user-friendly approach compared to Kubernetes. The task was completed since it was relatively easier to set up, manage, and understand, making it more accessible for users new to container orchestration.

Simplicity: Docker Swarm provides a straightforward and easy-to-understand approach for deploying and managing containers. It integrates seamlessly with Docker, leveraging its familiar commands and concepts.

Fast Deployment: Docker Swarm offers rapid deployment of containerized applications, making it suitable for smaller-scale projects or use cases that require quick setup.

Built-in Functionality: Docker Swarm includes several functionalities like load balancing, rolling updates, and service discovery, making it a complete solution for simpler container orchestration needs.

Learning curve:

Kubernetes typically has a steeper learning curve due to its extensive features and complexity, requiring more time and effort for users to grasp its concepts effectively.

Docker Swarm's learning curve is generally gentler, making it more approachable for users new to container orchestration.

Use cases:

Kubernetes is well-suited for enterprise-grade applications or scenarios demanding high scalability, robustness, and a vast ecosystem of tools and integrations.

Docker Swarm is often favoured for smaller or less complex projects, development environments, or scenarios where a more straightforward orchestration solution suffices.

Kubernetes and Docker Swarm cater to different needs and preferences. Kubernetes offers extensive features and scalability but comes with a higher learning curve, making it suitable for complex and large-scale deployments. On the other hand, Docker Swarm prioritizes simplicity and ease of use, making it a viable option for simpler setups, users who prefer straightforward container orchestration solutions. The choice between the two often depends on the specific requirements, complexity, and scalability needs of the project or organization.

References

1. Containers vs Microservices: What’s The Difference? – BMC Software | Blogs.

https://www.bmc.com/blogs/containers-vs-microservices/

2. What Is container orchestration | Google Cloud.

https://cloud.google.com/discover/what-is-container-orchestration

3. Swarm mode overview | Docker Docs

https://docs.docker.com/engine/swarm/

4. Kubernetes Components | Kubernetes

https://kubernetes.io/docs/concepts/overview/components/