

# **Dockerizing a Microservice Application over an AWS EC2 Instance or Docker Desktop and Deploying Containers using Docker Compose**

## INTRODUCTION

This research project focuses on “Dockerizing a Microservice Application over an AWS EC2 Instance or Docker Desktop and Deploying Containers using Docker Compose”. It implements a “flight-service” maven application provided as a project resource to demonstrate the student's understanding of Docker and the use of containers to deploy microservices. The project documents the steps and processes involved in containerizing the flight-service application and describes the tools and resources used for the implementation of the project. Key terminologies are defined, and demonstrations of steps and processes are provided as screenshots within this document. All demonstrations during this exercise were performed using Docker Desktop v4.24.2.

The project discusses Docker and describes the relationship between Docker Images, Docker Containers, and Docker files. A brief discussion on the phrase “don’t use docker-compose in production” summarizes the student’s opinion on the phrase with a closer look at the challenges of implementing a dockerize microservice application and deploying containers using docker-compose.

## DEFINITIONS

**Docker** - A platform that packages your application and all its dependencies together in the form of containers. It is an open platform for developing, shipping, and running applications and enables you to separate your applications from your infrastructure so you can deliver software quickly.

**Containers** - A container is a runnable instance of an image. is a standalone, executable package that includes an application and all its dependencies, such as libraries and configurations. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

**Docker image** – A docker image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization.

**Docker client** - The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to , which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

**Docker daemon** - The Docker daemon listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

**Docker Desktop** - An easy-to-install application for Mac, Windows, or Linux environments that enables you to build and share containerized applications and microservices. It includes the Docker daemon the Docker client (docker), and Docker Compose.

**Dockerfile** - A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.

**Docker registries** - A Docker registry stores Docker images. Docker Hub is a public registry available for use by anyone, and Docker is configured to look for images on Docker Hub by default. Docker allows you to run and manage your private registry.

**Docker compose** – A subcommand used to build and manage multiple services in Docker containers. It uses a YAML file to configure the services, networks, and volumes for a group of containers, making it easier to manage complex applications.

Docker provides several benefits, including portability, consistency, and scalability. Developers can package their applications and dependencies in containers, which can then be easily deployed and run on any system that supports Docker. This streamlines the development and deployment process and ensures that applications run consistently, regardless of the underlying infrastructure. Docker has become a popular technology in the world of software development and deployment due to its ability to simplify and streamline the software development lifecycle, improve resource utilization, and increase application reliability. It has a rich ecosystem of tools and a large community of users and contributors, making it a versatile and well-supported platform.

## IMPLEMENTATION

the following software applications will be required for carrying out this exercise:

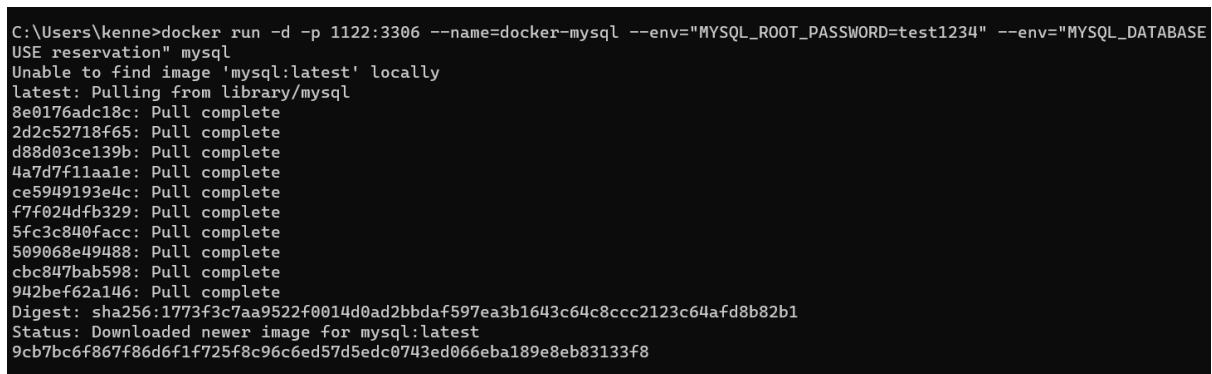
- **Eclipse IDE** – with Eclipse IDE, you will be able to compile the application file which has been provided for the assignment. NB: Remember to install from Eclipse marketplace, Spring tools in other run your application locally as a Springboot application.
- **Postman** – this tool allows you to test your running application using its URL by issuing various requests to the running application such as GET, POST, PUT, DELETE, etc.
- **MySQL Workbench** – this tool allows you to connect to instances of MySQL Servers and execute queries for creating and managing MySQL databases.
- **Docker Desktop** – Docker Desktop is a client application that can be installed on different operating systems environments which enables you to build and share containerized applications and microservices to the Dockerhub.

Note: Remember to create a **Dockerhub** account as you'll need one to be able to push an image to the Dockerhub. It is recommended to use the free version for learning purposes.

## STEP 1: Pull a MySQL image to run a container to host the flight-service database.

- run a mysql container on your local machine, you will need to pull a based image of MySQL from the Dockerhub. The command for pulling a MySQL from Dockerhub onto your local machine (within Docker Desktop) is shown in the screenshot below.

Figure 1.1 docker run docker-mysql image name reservation



```
C:\Users\kenne>docker run -d -p 1122:3306 --name=docker-mysql --env="MYSQL_ROOT_PASSWORD=test1234" --env="MYSQL_DATABASE
USE reservation" mysql
Unable to find image 'mysql:latest' locally
latest: Pulling from library/mysql
8e0176adc18c: Pull complete
2d2c52718f65: Pull complete
d88d03ce139b: Pull complete
4a7d7f11aa1e: Pull complete
ce5949193e4c: Pull complete
f7f024dfb329: Pull complete
5fc3c840facc: Pull complete
509068e49488: Pull complete
cbc847bab598: Pull complete
942bef62a146: Pull complete
Digest: sha256:1773f3c7aa9522f0014d0ad2bbdaf597ea3b1643c64c8ccc2123c64afd8b82b1
Status: Downloaded newer image for mysql:latest
9cb7bc6f867f86d6f1f725f8c96c6ed57d5edc0743ed066eba189e8eb83133f8
```

docker run – this command pulls an image from the Dockerhub and starts a container based on the image. For this exercise, the MySQL image is pulled from Dockerhub.

- - name – this command is used to provide a custom name for the mysql image when pulled from Dockerhub. For this exercise, we named the image: docker-mysql

-p – this command is used to specify the port on which the container of the image would listen for incoming requests. Two ports are specified and separated by a colon(,) the first port (left) specifies the port from which a request can be made to the container port which is the second port (right) on which the MySQL server listens for incoming requests.

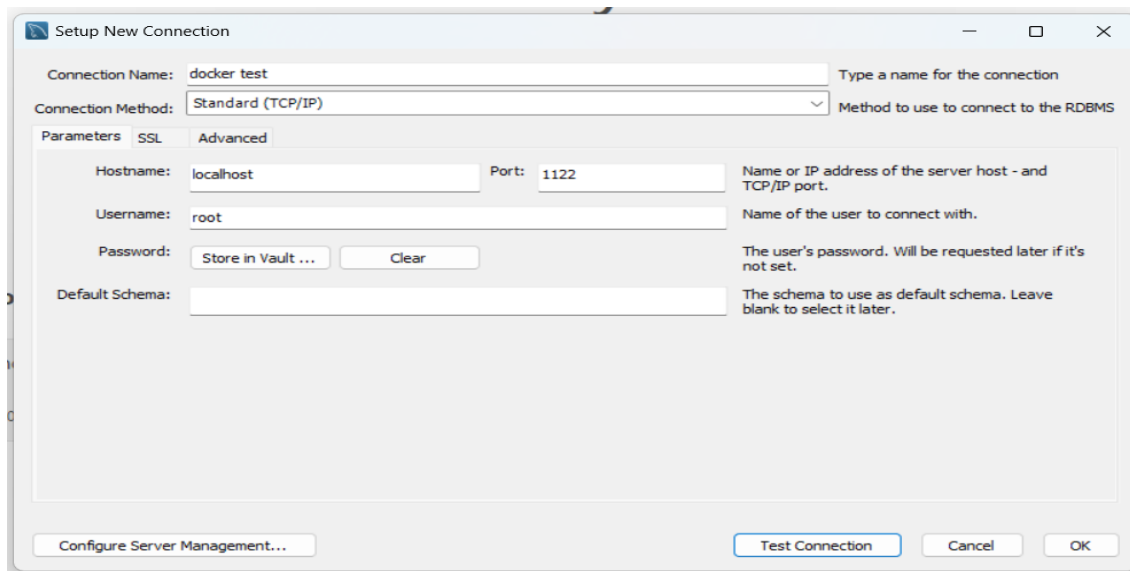
-e – this command specifies and sets the environment variables for the MySQL server. In this example, the environment variable specified is the database root password.

Lastly is specifying the name based image from which you are building this exercise was mysql name reservation database.

Steps for connecting to the MySQL custom image and setting up your flight-service database.

➤ Using MySQL Workbench, connect to the mysql server instance by providing the port number specified when you ran the docker run command above for pulling the mysql base image. In this exercise, we specify port 1122. (see -p 1122:3306)

**figure 1.2 – Connecting to mysql container using MySQL Workbench**



➤ Once connected to the mysql server, run the sql commands provided for the flight-service assignment. Once the commands are executed, it creates a database (reservation), and three tables (flight, passenger, and reservation) then insert values into the flight table.

**Figure 1.3 – Inside the mysql container using exec with bash command**

```
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| reservation |
| sys |
+-----+
5 rows in set (0.00 sec)

mysql> use reservation;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show * from flights;
```

**Figure 1.4 – viewing tables created inside of the mysql container**

```
mysql> use reservation;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show * from flights;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for
from flights' at line 1
mysql> select * from flight;
```

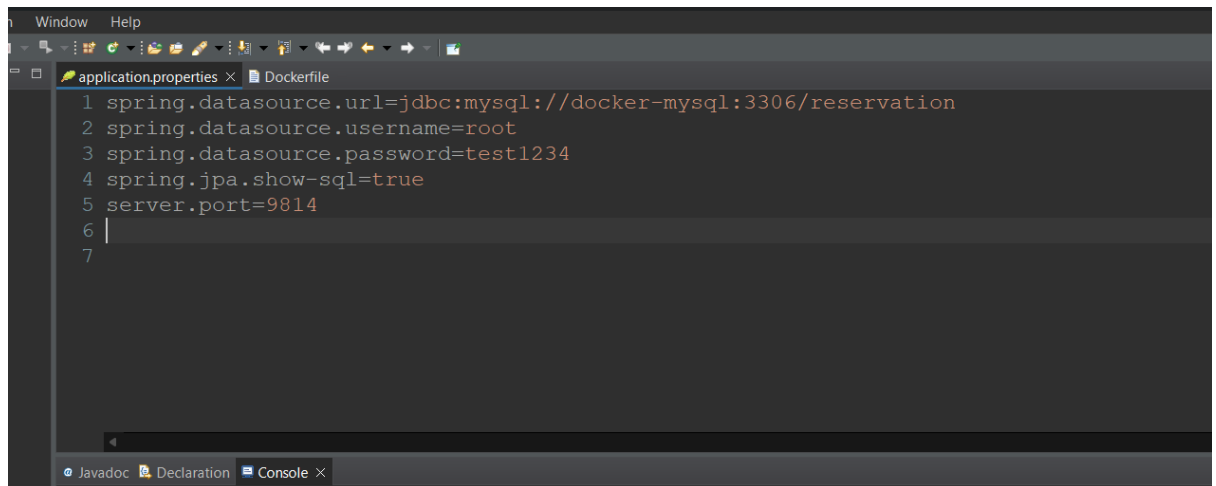
ID	FLIGHT_NUMBER	OPERATING_AIRLINES	DEPARTURE_CITY	ARRIVAL_CITY	DATE_OF_DEPARTURE	ESTIMATED_DEPARTURE_TIME
1	AA1	American Airlines	AUS	NYC	2022-02-05	2022-02-05 03:14:07
2	AA2	American Airlines	AUS	NYC	2022-02-05	2022-02-05 05:14:07
3	AA3	American Airlines	AUS	NYC	2022-02-05	2022-02-05 06:14:07
4	SW1	South West	AUS	NYC	2022-02-05	2022-02-05 07:14:07
5	UA1	United Airlines	NYC	DAL	2022-02-05	2022-02-05 10:14:07
6	UA1	United Airlines	NYC	DAL	2022-02-05	2022-02-05 10:14:07
7	SW1	South West	AUS	NYC	2022-02-06	2022-02-06 07:14:07
8	SW2	South West	AUS	NYC	2022-02-06	2022-02-06 08:14:07
9	SW3	South West	NYC	DAL	2022-02-06	2022-02-06 10:14:07
10	UA1	United Airlines	NYC	DAL	2022-02-06	2022-02-06 10:14:07

```
10 rows in set (0.00 sec)
```

## STEP 2. Create a Dockerfile that will build an image for the “flight-service” application

- Open Eclipse and import from existing maven project.  
“flightservices”
- Locate and edit the application properties file.  
Select the src/main/resources and click on the application.properties
- Update the URL and port as needed.  
Change the URL to the updated url shown below  
And create a server port of your choice i.e for this is port 9814
- Save the changes.

**Figure 2.1 – viewing updated application.properties from the flightservice**

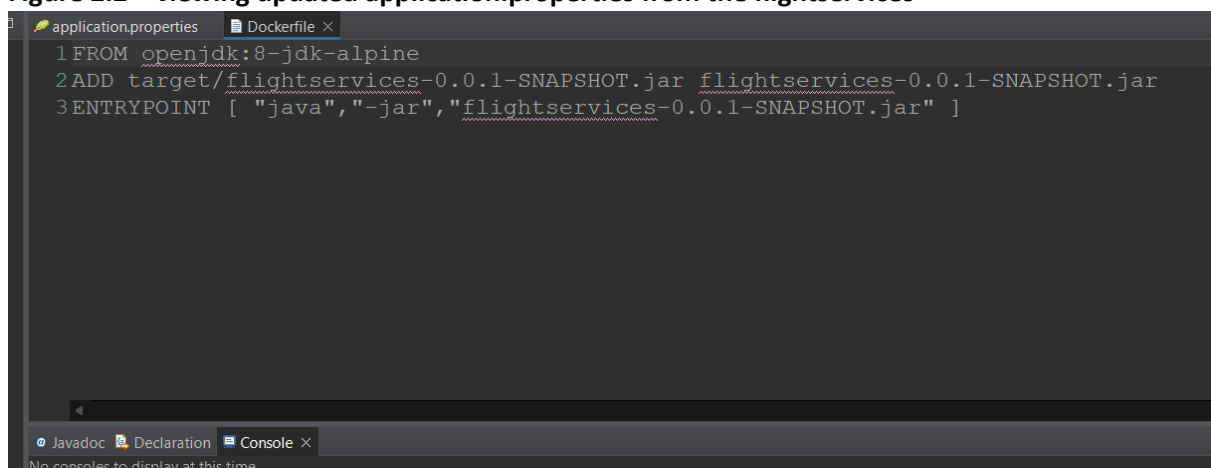
A screenshot of an IDE window with a dark theme. The 'application.properties' file is open, showing the following content:

```
1 spring.datasource.url=jdbc:mysql://docker-mysql:3306/reservation
2 spring.datasource.username=root
3 spring.datasource.password=test1234
4 spring.jpa.show-sql=true
5 server.port=9814
6
7
```

The IDE interface includes a top menu bar with 'Window' and 'Help', a toolbar with various icons, and a bottom status bar with tabs for 'Javadoc', 'Declaration', and 'Console'.

- On flightservices directory in your project.
- Create a new Dockerfile .
- Open the Dockerfile.
- Edit the Dockerfile to define the necessary commands for your Docker image, including specifying the base image, copying files, setting environment variables, and any other requirements for your application.
- Save the changes to the Dockerfile.

**Figure 2.2 – viewing updated application.properties from the flightservices**

A screenshot of an IDE window with a dark theme. The 'Dockerfile' is open, showing the following content:

```
1 FROM openjdk:8-jdk-alpine
2 ADD target/flightservices-0.0.1-SNAPSHOT.jar flightservices-0.0.1-SNAPSHOT.jar
3 ENTRYPOINT [ "java", "-jar", "flightservices-0.0.1-SNAPSHOT.jar" ]
```

The IDE interface includes a top menu bar with 'Window' and 'Help', a toolbar with various icons, and a bottom status bar with tabs for 'Javadoc', 'Declaration', and 'Console'. A message at the bottom states 'No consoles to display at this time.'

### Understanding the Dockerfile Commands

**FROM:** this command is used to specify the based image upon which your custom image will be based, ie the software dependency needed to run your application.

**ADD:** this command is used to add your project file by specifying the location and name (in this exercise, the flight-service jar file) to the working directory which docker uses to build an image.

**Entrypoint:** specifies the command that should be executed when a container is started from the Docker image.

Dockerfile is instructing Docker to build an image based on the "openjdk:8-jdk-alpine" base image, copy your application JAR file into the image, and configure the container to run the JAR file as the entry point when a container is created from this image.

### Rebuild the maeve project.

- Navigate to the root directory of the "flightservices" project.
- Run the Maeve build command: `maeve build flightservices`
- On the goal run this command "clean package -DskipTests" - basically means you are telling Maven to clean the target directory, compile and package your code, and skip the execution of unit tests during this process.
- After that it should display the build success as shown below

**Figure 2.3 – success maeve build from java**

```
Downloaded from : https://repo.maven.apache.org/maven2/org/apache/maven/s
Downloaded from : https://repo.maven.apache.org/maven2/com/google/guava/g
Replacing main artifact with repackaged archive
-----
BUILD SUCCESS
-----
Total time: 26.352 s
Finished at: 2023-11-01T22:32:37Z
-----
```

Open up a Terminal from CMD and navigate within your project folder and issue the docker command as seen in the screenshot below.

**Figure 2.4– building a docker image flight\_app**

```
C:\Users\kenne\OneDrive\Desktop\flightservices>docker build -f Dockerfile -t flight_app .
[+] Building 21.8s (8/8) FINISHED          docker:default
=> [internal] load build definition from Dockerfile    0.1s
=> => transferring dockerfile: 208B                    0.0s
=> [internal] load .dockerignore                      0.1s
=> => transferring context: 2B                          0.0s
```

**docker build:** This is the Docker command used to build a Docker image.

**-f Dockerfile:** This option specifies the path to the Dockerfile that Docker should use to build the image. In this case, it is set to "Dockerfile. You can provide a different file path if your Dockerfile is in a different directory.

**-t flight\_app:** This option specifies the name and optional tag for the Docker image you're building. In this case, you're giving the image the name "flight\_app." You can add a tag to the image (e.g., "flight\_app:1.0") if you want to version it.

### Testing your containerized application

Run the command **docker run -t --name=flight-app --link docker-mysql:mysql -p**

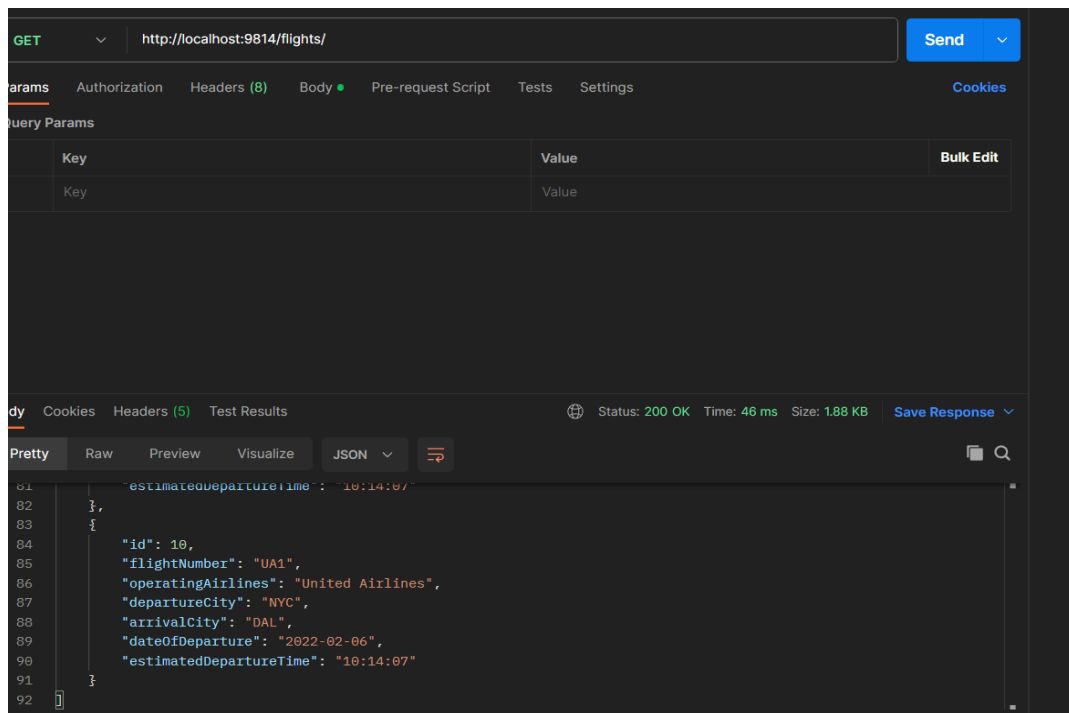
**9814:9814flight\_app**, a Docker container named "flight-app" will be created from the "flight\_app" image. This container will have a network link to the "docker-mysql" container, and port 9814 in the container will be mapped, allowing you to access the application inside the container via your web browser or other tools.





Test the flight-service containerized application from Postman. <http://localhost:9814/flights/> See the screenshot below.

**Figure 2.7** success test of the application



## Step 3: Uploading the flight-service Application to Docker Hub

Once you've containerized the flight-service application through the creation of a custom image, the next task is to upload this image to Docker Hub, making it accessible to other developers. To successfully push your image to Docker Hub, ensure you are logged in through your terminal using your Docker Hub credentials, and then follow the commands demonstrated in the provided screenshot.

Figure 3.1 – docker push

```
C:\Users\kenne\OneDrive\Desktop\flightservices>docker tag flight_app kennedymwai/flightservice

C:\Users\kenne\OneDrive\Desktop\flightservices>docker push flight_app kennedymwai/flightservice
"docker push" requires exactly 1 argument.
See 'docker push --help'.

Usage:  docker push [OPTIONS] NAME[:TAG]

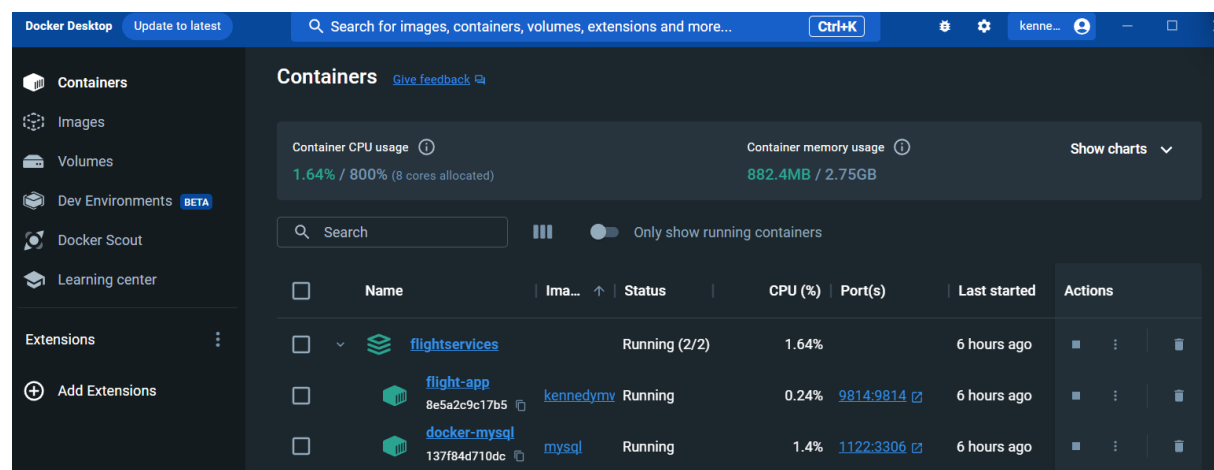
Upload an image to a registry

C:\Users\kenne\OneDrive\Desktop\flightservices>docker push kennedymwai/flightservice
Using default tag: latest
The push refers to repository [docker.io/kennedymwai/flightservice]
7d03ee46db67: Pushed
ceaf9e1ebef5: Mounted from kennedymwai/flightservice
9b9b7f3d56a0: Mounted from kennedymwai/flightservice
f1b5933fe4b5: Mounted from kennedymwai/flightservice
latest: digest: sha256:e6661e6c999ddd1faa2ee76afde697827f06ed0f8d97c3696d4d45c51f6eba35 size: 1159
```

**docker tag** – this command creates a tag of your image to be pushed to the docker hub using the local image. As shown above, flight\_app is our local image and kennedy/flightservice is our image to be pushed. You must always begin the naming of the image to be pushed to the docker hub with your docker hub username and a forward slash.

**docker push** – pushes your image to the docker hub using the name specified in step one.

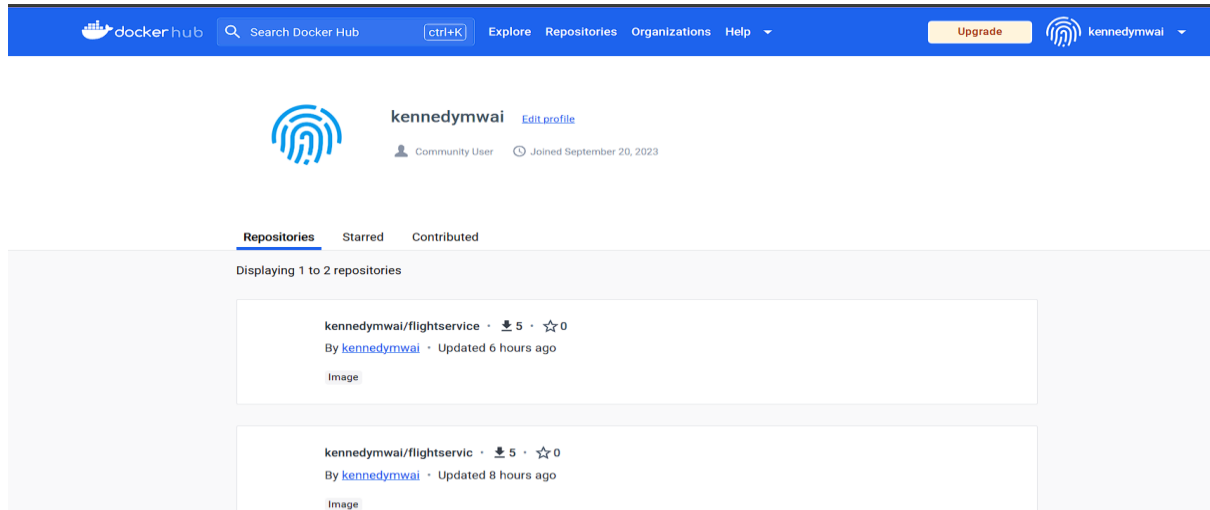
Figure 3.2 – docker hub



The screenshot shows the Docker Desktop application interface. The left sidebar contains navigation options: Containers, Images, Volumes, Dev Environments (marked BETA), Docker Scout, and Learning center. The main area is titled 'Containers' and displays system metrics: Container CPU usage at 1.64% / 800% (8 cores allocated) and Container memory usage at 882.4MB / 2.75GB. Below the metrics is a search bar and a toggle for 'Only show running containers'. A table lists the running containers:

	Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
<input type="checkbox"/>	flightservices		Running (2/2)	1.64%		6 hours ago	[Stop] [Restart] [Delete]
<input type="checkbox"/>	flight-app	kennedymwai/flight-app:latest	Running	0.24%	9814:9814	6 hours ago	[Stop] [Restart] [Delete]
<input type="checkbox"/>	docker-mysql	mysql	Running	1.4%	1122:3306	6 hours ago	[Stop] [Restart] [Delete]

**Figure 3.3 – docker hub**

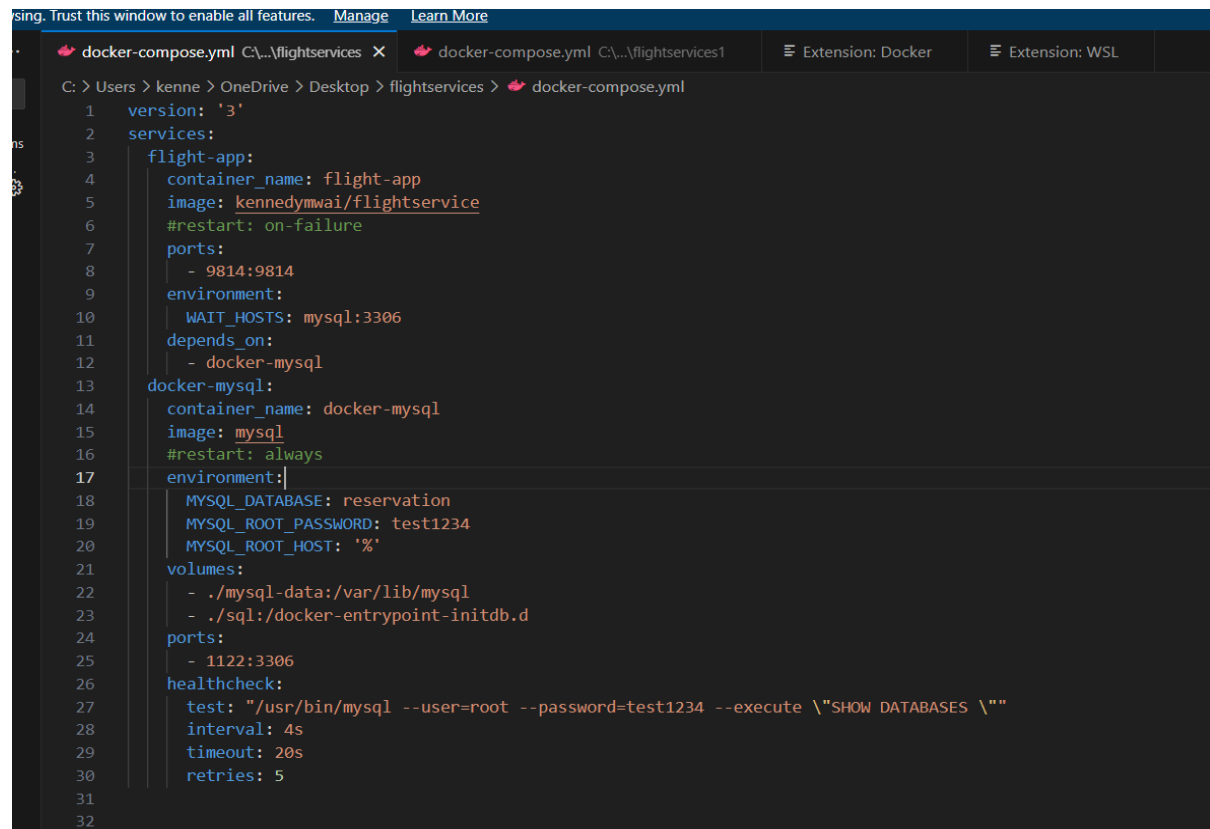


pushing Docker images to a container registry, such as Docker Hub, streamlines the sharing, collaboration, and deployment of containerized applications, making it a fundamental aspect of container-based development and deployment workflows.

## STEP 4: Using Docker compose to manage the flight-service Microservice

Docker compose allows you to manage multiple containerized applications by compiling the multiple containers into a .yaml file. For this exercise, the docker-compose file is shown in the screenshot below

Figure 4.1 docker compose file



```
1 version: '3'
2 services:
3   flight-app:
4     container_name: flight-app
5     image: kennedymwai/flightservice
6     #restart: on-failure
7     ports:
8       - 9814:9814
9     environment:
10      WAIT_HOSTS: mysql:3306
11     depends_on:
12       - docker-mysql
13   docker-mysql:
14     container_name: docker-mysql
15     image: mysql
16     #restart: always
17     environment:
18       MYSQL_DATABASE: reservation
19       MYSQL_ROOT_PASSWORD: test1234
20       MYSQL_ROOT_HOST: '%'
21     volumes:
22       - ./mysql-data:/var/lib/mysql
23       - ./sql:/docker-entrypoint-initdb.d
24     ports:
25       - 1122:3306
26     healthcheck:
27       test: "/usr/bin/mysql --user=root --password=test1234 --execute \"SHOW DATABASES \""
28       interval: 4s
29       timeout: 20s
30       retries: 5
31
32
```

### Understanding the Docker-compose file

The docker-compose file consists of configuration settings for two services, our flight-service application and the mysql database on which the application depends. As seen in the figure above, the flight-app service refers to the flight-service application whereas the docker-mysql refers to the mysql db.

These services, when run using the docker-compose command, will be run as individual containers, therefore, we must specify the tags/names for each container to be created and a base image from which they will be instantiated. For the flight-app, it is based on a custom image that was built using the flight-service jar file provided for this exercise, whereas the docker-mysql uses a mysql image pulled from the docker hub as its based image. Both applications rely on different environment variables and dependencies to run, therefore we specified each as per the service and exposed the relevant listening ports. Because the flight-service application relies on a backend database to execute, notice within the docker-compose file that the flight-app service depends on the docker-

```

[+] Running 3/3
  ✓ Container flight-app          Removed
  ✓ Container docker-mysql       Removed
  ✓ Network flightservices_default Removed

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

```

## Conclusion

In summary, it's crucial to underscore the significance of Docker in the realm of DevOps and modern application development and deployment. Docker 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.

The Docker daemon boasts a range of built-in commands for image and container management, enabling smooth production delivery. Among these, Docker Compose stands out as a prominent tool. Docker Compose, a subcommand, simplifies the creation and administration of multi-container applications within Docker. It empowers users to articulate and configure application services through a .yaml file, facilitating the simultaneous launch of multiple containers running diverse services as a cohesive unit.

Nonetheless, it's worth noting that there are certain limitations associated with employing Docker Compose for constructing and operating multiple containerized services, prompting arguments against its use in production environments.

Drawing from my experience with this research project, I've identified several reasons why I concur that Docker Compose might not be the ideal choice for production-grade development setups. One notable drawback is that, with Docker Compose, tasks such as installing new applications, adding features, or performing application updates necessitate the services to be taken offline, resulting in downtime for the running application and services. Additionally, Docker Compose lacks the full suite of features for container orchestration and clustered services.

Hence, some Docker enthusiasts contend that Docker Swarm could potentially address Docker Compose limitations, while others advocate Kubernetes as a superior and preferred option for managing and orchestrating multiple container services.

It's important to note that this research project exclusively delves into Docker and its implementation, without delving deeply into an exhaustive analysis of the two highly efficient tools for clustered container deployment and orchestration.

## References

1. Getting Started with Docker  
Available at: <https://docs.docker.com/get-started/overview/> [cited 03 November 2023]
2. Docker hub  
Available at: <https://hub.docker.com/> [cited 03 November 2023]
3. Docker compose  
Available at: [https://docs.docker.com/get-started/08\\_using\\_compose/](https://docs.docker.com/get-started/08_using_compose/) [cited 03 November 2023]