

Docker Assignment Practical

Part-1: Docker Basics

Task 1.1: Running Your First Containers

```
>> docker run -d -t --name my-alpine alpine # Pull and run basic containers
>> docker run -d -t --name my-busybox busybox
>> docker ps # List all containers
>> docker ps -a
>> docker image ls # Check downloaded images
```

```
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker run -d -t --name my-alpine alpine
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
Digest: sha256:4bcff63911fcb4448bd4fdacec207030997caf25e9bea4045fa6c8c44de311d1
Status: Downloaded newer image for alpine:latest
3a702d2e16d7fee90d856b7530b07026ac75a1b7b51ce67bcc78e5963992d23c
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker run -d -t --name my-busybox busybox
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
499bcf3c8ead: Pull complete
Digest: sha256:d82f458899c9696cb26a7c02d5568f81c8c8223f8661bb2a7988b269c8b9051e
Status: Downloaded newer image for busybox:latest
9049231673f6174f1c9c82f670d888dd6b1a4c70064cd8084a05d4e293f26a9c
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS          NAMES
9049231673f6   busybox    "sh"                    8 seconds ago Up 7 seconds   my-busybox
3a702d2e16d7   alpine     "/bin/sh"               29 seconds ago Up 29 seconds  my-alpine
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker ps -a
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS          NAMES
9049231673f6   busybox    "sh"                    14 seconds ago Up 14 seconds  my-busybox
3a702d2e16d7   alpine     "/bin/sh"               35 seconds ago Up 35 seconds  my-alpine
580aceab6cbc   gcr.io/k8s-minikube/kicbase:v0.0.47 "/usr/local/bin/entr..." 4 weeks ago    Exited (137) 4 weeks ago minikube
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker image ls
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
dipakp22/docker-todo-app    latest    b3f13a5ebd35   4 weeks ago    1.73GB
docker-getting-started-app-backend    latest    5b4ce80d65c6   4 weeks ago    1.82GB
docker-getting-started-app-client      latest    8b00168f45f6   4 weeks ago    1.82GB
docker/welcome-to-docker              latest    c4d56c24da4f   2 months ago    22.9MB
traefik                                v3.4       06ddf61ee653   2 months ago    272MB
alpine                                  latest     4bcff63911fc   2 months ago    13.3MB
gcr.io/k8s-minikube/kicbase            v0.0.47    631837ba851f   4 months ago    1.76GB
gcr.io/k8s-minikube/kicbase            <none>     6ed579c9292b   4 months ago    1.76GB
mysql                                   9.3        b9d8b7ec6e6a   5 months ago    1.19GB
phpmyadmin                             latest     efe7c91bca65   8 months ago    820MB
busybox                                 latest     d82f458899c9   12 months ago    6.21MB
```

Q&A

1. What's the difference between `docker ps` and `docker ps -a`?

Docker ps -> It lists containers that are currently active (in the "Up" state).

Ps -a -> It shows all containers, including running, stopped, and failed to start container

2. Why are Alpine and BusyBox images so small?

Most Linux distributions (like Ubuntu or Debian) use the glibc (GNU C library). Alpine and BusyBox use musl libc, a much smaller, simpler implementation of the standard C library. BusyBox combines many basic Unix utilities (like ls, cp, cat, grep, etc.) into a single binary with multiple symlinks.

Typical Linux uses individual binaries:

/bin/ls

/bin/cp

/bin/mv

BusyBox replaces them with:

/bin/busybox

Containers usually run a single process (like a web server, a database, etc.), not an entire operating system. So: You don't need init systems, login shells, or hardware management tools. You just need a small filesystem and a C library. That's exactly what Alpine and BusyBox provide.

BusyBox: best for ultra-minimal images, embedded systems, or when you just need basic shell tools.

Alpine: best for lightweight containers where you still want a package manager and a POSIX-compliant environment.

Task 1.2: Container Interaction

```
>> docker exec -t my-alpine ls / # Execute commands in running containers
>> docker exec -t my-busybox ps aux
>> docker exec -it my-alpine sh # Open interactive shell sessions
>> docker stop my-alpine # Container lifecycle management
>> docker start my-alpine
>> docker rm -f my-busybox
```

```
dipakprasad@Dipaks-MacBook-Pro docker-poc %
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker exec -t my-alpine ls /
bin dev etc home lib media mnt opt proc root run sbin srv sys tmp usr var
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker exec -t my-busybox ps aux
PID USER TIME COMMAND
1 root 0:00 sh
7 root 0:00 ps aux
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker exec -t my-busybox ls /
bin dev etc home lib lib64 proc root sys tmp usr var
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker exec -it my-alpine sh
/ # ls
bin dev etc home lib media mnt opt proc root run sbin srv sys tmp usr var
/ # ls -la
total 64
drwxr-xr-x 1 root root 4096 Oct 8 09:15 .
drwxr-xr-x 1 root root 4096 Oct 8 09:15 ..
-rwxr-xr-x 1 root root 0 Oct 8 09:15 .dockerenv
drwxr-xr-x 2 root root 4096 Jul 15 10:42 bin
drwxr-xr-x 5 root root 360 Oct 8 09:15 dev
drwxr-xr-x 1 root root 4096 Oct 8 09:15 etc
drwxr-xr-x 2 root root 4096 Jul 15 10:42 home
drwxr-xr-x 6 root root 4096 Jul 15 10:42 lib
drwxr-xr-x 11 root root 4096 Jul 15 10:42 var
/ # pwd
/
/ # exit
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker stop my-alpine
my-alpine
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker start my-alpine
my-alpine
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
9049231673f6 busybox "sh" 21 minutes ago Up 21 minutes my-busybox
3a702d2e16d7 alpine "/bin/sh" 21 minutes ago Up 6 seconds my-alpine
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker rm -f my-busybox
my-busybox
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3a702d2e16d7 alpine "/bin/sh" 22 minutes ago Up 25 seconds my-alpine
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
dipakp22/docker-todo-app latest b3f13a5ebd35 4 weeks ago 1.73GB
docker-getting-started-app-backend latest 5b4ce80d65c6 4 weeks ago 1.82GB
docker-getting-started-app-client latest 8b00168f45f6 4 weeks ago 1.82GB
docker/welcome-to-docker latest c4d56c24da4f 2 months ago 22.9MB
traefik v3.4 06ddf61ee653 2 months ago 272MB
alpine latest 4bcff63911fc 2 months ago 13.3MB
gcr.io/k8s-minikube/kicbase v0.0.47 631837ba851f 4 months ago 1.76GB
gcr.io/k8s-minikube/kicbase <none> 6ed579c9292b 4 months ago 1.76GB
mysql 9.3 b9d8b7ec6e6a 5 months ago 1.19GB
phpmyadmin latest efe7c91bca65 8 months ago 820MB
busybox latest d82f458899c9 12 months ago 6.21MB
```

Part 2: Docker Networking

Task 2.1: Default Bridge Network

```
>> docker run -d --name nginx-default nginx:latest # Run Nginx container
>> docker inspect nginx-default # Inspect the container
>> docker exec -it nginx-default curl localhost:80 # Test connectivity
```

```
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker run -d --name nginx-default nginx:latest
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
e363695fcb93: Pull complete
edd736256ac6: Pull complete
348644581cc5: Pull complete
3766556f3395: Pull complete
f3ff5b8e6cee: Pull complete
e3e8c796c790: Pull complete
54959f07be7f: Pull complete
Digest: sha256:f79cde317d4d172a392978344034eed6dff5728a8e6d7a42f507504c23ecf8b8
Status: Downloaded newer image for nginx:latest
f417469d8de9699d6738427367e7b8904374bbef7b4f44bf57440a3a7b0455f6
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker inspect nginx-default
[
```

```
{
  "Id": "f417469d8de9699d6738427367e7b8904374bbef7b4f44bf57440a3a7b0455f6",
  "Created": "2025-10-08T09:42:06.178257214Z",
  "Path": "/docker-entrypoint.sh",
  "Args": [
    "nginx",
    "-g",
    "daemon off;"
  ],
  "State": {
    "Status": "running",
    "Running": true,
    "Paused": false,
    "Restarting": false,
    "OOMKilled": false,
    "Dead": false,
    "Pid": 978,

```

```
-
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker exec -it nginx-default curl localhost:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Task 2.2: Port Forwarding

```
>> docker rm -f nginx-default
>> docker run -d -p 8080:80 --name nginx-exposed nginx:latest # Run Nginx with port mapping
>> curl localhost:8080 # Test access
```

Task 2.3: Custom Bridge Network

```
>> docker network create my-network # Create custom network
>> docker network ls
>> docker run -d --network my-network --name web-server nginx:latest # Run containers in custom network
>> docker run -it --network my-network --name client alpine sh
>> ping web-server # Test name resolution inside the Alpine container
>> wget -qO- http://web-server
```

```
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker rm -f nginx-default
nginx-default
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker run -d -p 8080:80 --name nginx-exposed nginx:latest
c4324b8ca438d23bb3ca88b3c1d7f65719bc953b6bfe47df34fea698b2dac2
dipakprasad@Dipaks-MacBook-Pro docker-poc % curl localhost:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

```
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker network create my-network
5fb52434288196915776b3732edc32b199c1fb7ea443c5cdd313f702dcf1e2ce
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
456313f386ac        bridge             bridge              local
32c4802cc357        host               host                local
6042f0e6f136        minikube           bridge              local
5fb524342881        my-network         bridge              local
3418f3b47014        none              null                local
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker run -d --network my-network --name web-server nginx:latest
dc1d4e2cd6196dabb8d9c5f7860ef4723c8deaed53c3dfaac811c8ade47e7519
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker run -it --network my-network --name client alpine sh
/ # ping web-server
PING web-server (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.169 ms
64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.203 ms
64 bytes from 172.18.0.2: seq=2 ttl=64 time=0.186 ms
64 bytes from 172.18.0.2: seq=3 ttl=64 time=0.175 ms
^C
--- web-server ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.169/0.183/0.203 ms
/ # get -q0- http://web-server
sh: get: not found
/ # wget -q0- http://web-server
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
```

Q&A

1. Why can containers ping each other by name in custom networks but not in the default bridge?

Because Docker's built-in DNS-based service discovery only works in user-defined networks. Use a user-defined network whenever you want containers to communicate by name. If you use Docker Compose, it automatically creates a user-defined bridge network for your services.

Docker has two major types of "bridge" networks:

Type	Created by	Name	Example
Default bridge	Docker daemon (always exists)	bridge	docker network ls → bridge
User-defined bridge	You create it e.g. my_network	docker	network create my_network

When you create a user-defined bridge network, Docker automatically attaches an internal DNS server to that network. This DNS server lets containers resolve each other by container name or network alias. Default bridge network: Uses legacy behavior for compatibility. No automatic DNS-based name resolution. Only /etc/hosts file entries for linked containers (from the old --link option). User-defined bridge network: Has Docker-managed DNS server. Automatically adds DNS entries for containers in that network. Supports name resolution and service discovery. Containers are isolated from others not in that network.

-Default bridge

[web] 172.17.0.2

[db] 172.17.0.3

No DNS → Must use Ips

-Custom bridge (my_net)

Docker DNS provides:

web → 172.18.0.2

db → 172.18.0.3

So containers can:

ping web

ping db

2. What happens when you try to access the web server from your host machine in the custom network?

If a container is attached only to a custom bridge network, you cannot access it directly from the host via its container IP. You must publish a port (with -p or --publish) to expose the service on the host's network interface.

Each user-defined bridge network (like `my_net`) is a private, internal network managed by Docker. Containers inside it can talk to each other by name or IP. But the host machine (and the outside world) are not directly part of that network.

So if you start a web server like this:

```
>> docker network create my_net
>> docker run -d --name web --network my_net nginx
```

Docker assigns it an internal IP, e.g.:

172.19.0.2

If you try to access:

```
>> curl http://172.19.0.2
```

from your host machine, you'll get:

curl: (7) Failed to connect to 172.19.0.2 port 80: Connection refused

Because that IP is inside the Docker bridge namespace, not reachable from the host.

To make the web server accessible from your host, you must publish a port:

```
>> docker run -d --name web --network my_net -p 8080:80 nginx
```

Now Docker creates a NAT rule (iptables) that maps:

Host: 127.0.0.1:8080 → Container: 172.19.0.2:80

So from your host:

```
>> curl http://localhost:8080
```

In the default bridge network, containers are attached to a preconfigured network that also uses NAT.

You still need `-p` to reach containers from the host, but their IPs (e.g. 172.17.x.x) may sometimes be reachable directly from the host — depending on your host's network stack and Docker setup. In contrast, user-defined bridge networks are fully isolated namespaces — the host cannot reach them without explicit port mapping.

Default bridge (legacy)

Host (172.17.0.1)

↳ may access containers (172.17.x.x)

Custom bridge (`my_net`)

Host —X—► [`my_net`]

└— web (172.19.0.2)

└— db (172.19.0.3)

Part 3: Docker Volumes

Task 3.1: Bind Mounts

```
>> mkdir shared-logs # Create a directory on your host
```

Run containers with bind mount

```
>> docker run -d -v $(pwd)/shared-logs:/app/logs --name logger1 alpine tail -f /dev/null
```

```
>> docker run -d -v $(pwd)/shared-logs:/app/logs --name logger2 busybox tail -f /dev/null
```

Create files from containers

```
>> docker exec logger1 sh -c "echo 'Log from container 1' > /app/logs/container1.log"
```

```
>> docker exec logger2 sh -c "echo 'Log from container 2' > /app/logs/container2.log"
```

Check from host: # Verify file sharing

```
>> ls shared-logs/
```

```
>> cat shared-logs/*.log
```

Check from containers:

```
>> docker exec logger1 ls /app/logs/
```

```
>> docker exec logger2 ls /app/logs/
```

```
dipakprasad@Dipaks-MacBook-Pro docker-poc % mkdir shared-logs
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker run -d -v $(pwd)/shared-logs:/app/logs --name logger1 alpine tail -f /dev/null
cd1fa2e279dca64b490798938eeaaaa4bd90aee1851c4d82aa2380e02f00ace
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker run -d -v $(pwd)/shared-logs:/app/logs --name logger2 busybox tail -f /dev/null
af1e462ce812b4632cc6e57574ef7e832097542c5ce9eff0e50b32ddc663d737
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker exec logger1 sh -c "echo 'Log from container 1' > /app/logs/container1.log"
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker exec logger2 sh -c "echo 'Log from container 2' > /app/logs/container2.log"
dipakprasad@Dipaks-MacBook-Pro docker-poc % ls shared-logs/
container1.log  container2.log
dipakprasad@Dipaks-MacBook-Pro docker-poc % cat shared-logs/*.log
Log from container 1
Log from container 2
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker exec logger1 ls /app/logs/
container1.log
container2.log
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker exec logger2 ls /app/logs/
container1.log
container2.log
```

Task 3.2: Docker Volumes

```
>> docker volume create app-data # Create and use Docker volume
>> docker volume ls
>> docker volume inspect app-data
# Mount volume in containers
>> docker run -d --mount source=app-data,target=/data --name data1 alpine tail -f /dev/null
>> docker run -d --mount source=app-data,target=/data --name data2 nginx:latest
>> docker exec data1 sh -c "echo 'Persistent data' > /data/test.txt" # Test data persistence
>> docker exec data2 cat /data/test.txt
```

```
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker volume create app-data
app-data
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker volume ls
DRIVER      VOLUME NAME
local       app-data
local       docker-getting-started-app_todo-mysql-data
local       minikube
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker volume inspect app-data
[
  {
    "CreatedAt": "2025-10-08T10:22:38Z",
    "Driver": "local",
    "Labels": null,
    "Mountpoint": "/var/lib/docker/volumes/app-data/_data",
    "Name": "app-data",
    "Options": null,
    "Scope": "local"
  }
]
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker run -d --mount source=app-data,target=/data --name data1 alpine tail -f /dev/null
2b4d456dace83a7c4d05bd673209108fdf4935841efe19c1ff1d9f148df8c7a9
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker run -d --mount source=app-data,target=/data --name data2 nginx:latest
650f9c8a8bd08adfe6f1e5acc357014bbd40d44b29b5c4aba5377e82d681ef99
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker exec data1 sh -c "echo 'Persistent data' > /data/test.txt"
dipakprasad@Dipaks-MacBook-Pro docker-poc % docker exec data2 cat /data/test.txt
Persistent data
```

Q&A

1. Explain the difference between bind mounts and Docker volumes. When would you use each?

Both bind mounts and volumes are ways to persist or share data outside a container's writable layer. This means when the container stops, your data doesn't vanish. However, they work very differently in terms of control, isolation, and portability.

A bind mount directly maps a specific directory or file on your host into the container's filesystem.

```
>> docker run -v /home/user/app:/usr/src/app node
```

```
>> docker run --mount type=bind,source=/home/user/app,target=/usr/src/app node
```

/home/user/app on your host is mounted as /usr/src/app inside the container. Any changes in one are instantly visible in the other.

A volume is a managed data store created and maintained by Docker. Docker handles where it lives on disk (usually under /var/lib/docker/volumes/...).

```
>> docker volume create mydata
```

```
>> docker run -v mydata:/var/lib/mysql mysql
```

```
>> docker run --mount source=mydata,target=/var/lib/mysql mysql
```

Docker creates and manages mydata internally. Data persists even if the container is removed.

Part 4: Building Docker Images

Task 4.1: Create a Flask Application

```
>> mkdir flask-docker-app
>> cd flask-docker-app
>> vi app.py # Create application files
>> vi requirements.txt
>> vi Dockerfile
```

Docker-Fundamentals > Assignment-4 > flask-docker-app > Dockerfile > ...

```
1 FROM python:3.11-slim (last pushed 6 days ago)
2
3 WORKDIR /app
4
5 COPY requirements.txt .
6 RUN pip install --no-cache-dir -r requirements.txt
7
8 COPY app.py .
9
10 EXPOSE 5000
11
12 CMD ["python", "app.py"]
13
```

```
dipakprasad@Dipaks-MacBook-Pro Assignment-4 % mkdir flask-docker-app
dipakprasad@Dipaks-MacBook-Pro Assignment-4 % cd flask-docker-app
dipakprasad@Dipaks-MacBook-Pro flask-docker-app % touch app.py
dipakprasad@Dipaks-MacBook-Pro flask-docker-app % vi app.py
dipakprasad@Dipaks-MacBook-Pro flask-docker-app % touch requirements.txt
dipakprasad@Dipaks-MacBook-Pro flask-docker-app % vi requirements.txt
dipakprasad@Dipaks-MacBook-Pro flask-docker-app % touch Dockerfile
dipakprasad@Dipaks-MacBook-Pro flask-docker-app % vi Dockerfile
dipakprasad@Dipaks-MacBook-Pro flask-docker-app % docker build -t my-flask-app:v1.0 .
[+] Building 32.2s (11/11) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 205B
=> [internal] load metadata for docker.io/library/python:3.11-slim
=> [auth] library/python:pull token for registry-1.docker.io
=> [internal] load .dockerignore
```

docker:desktop-li

Task 4.2: Build and Test Image

```
>> docker build -t my-flask-app:v1.0 . # Build the image
>> docker run -d -p 5000:5000 --name flask-app my-flask-app:v1.0 # Run container
>> curl localhost:5000 # Test the application
>> curl localhost:5000/health
```

```
dipakprasad@Dipaks-MacBook-Pro flask-docker-app % docker run -d -p 8081:5000 --name flaskd-app my-flask-app:v1.0
5353260c7255e6d31124d6088e2f104b98e8d26a782c9370be364bcd8cc1852
dipakprasad@Dipaks-MacBook-Pro flask-docker-app % curl localhost:5000
dipakprasad@Dipaks-MacBook-Pro flask-docker-app % curl localhost:5000/health
```

Pretty-print ☒

```
{
  "container_id": "5353260c7255",
  "message": "Hello from Docker!"
}
```

Pretty-print ☐

```
{
  "status": "healthy"
}
```

Task 4.3: Multi-container Application

>> **Create docker-compose.yml**

>> **docker-compose up -d**

>> **docker-compose ps**

```
dipakprasad@Dipaks-MacBook-Pro flask-docker-app % docker-compose up -d
WARN[0000] /Users/dipakprasad/Downloads/docker-poc/flask-docker-app/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it
to avoid potential confusion
[+] Running 1/2
 ✓ Container flask-docker-app-redis-1 Running 0.0s
   Container flask-docker-app-web-1 Starting 0.0s
Error response from daemon: ports are not available: exposing port TCP 0.0.0.0:5000 -> 127.0.0.1:0: listen tcp 0.0.0.0:5000: bind: address already in use
dipakprasad@Dipaks-MacBook-Pro flask-docker-app % docker-compose ps
WARN[0000] /Users/dipakprasad/Downloads/docker-poc/flask-docker-app/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it
to avoid potential confusion
NAME                IMAGE                COMMAND                SERVICE    CREATED      STATUS      PORTS
flask-docker-app-redis-1 redis:alpine         "docker-entrypoint.s..." redis      2 minutes ago Up 2 minutes 6379/tcp
```

flask-docker-app — vi docker-compose.yml

```
Version: '3.8'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - app-logs:/app/logs
    networks:
      - app-network

  redis:
    image: redis:alpine
    networks:
      - app-network

volumes:
  app-logs:

networks:
  app-network:
```

Part 5: Image Registry

Task 5.1: Push to Docker Hub

>> **docker login**

>> **docker tag my-flask-app:v1.0 dipakp22/flask-demo:v1.0**

>> **docker push dipakp22/flask-demo:v1.0**

>> **docker rmi my-flask-app:v1.0 dipakp22/flask-demo:v1.0**

>> **docker run -d -p 5001:5000 dipakp22/flask-demo:v1.0**

Login to Docker Hub

Tag image

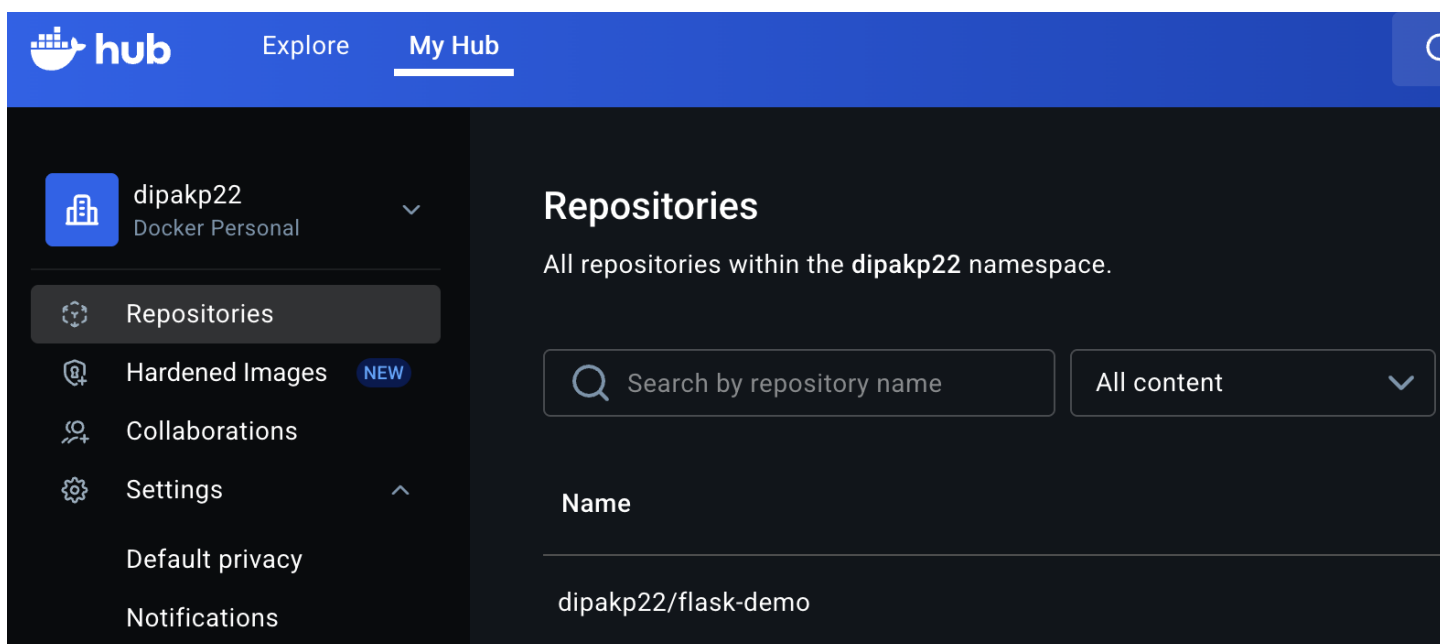
Push image to Docker Hub

Test pulling image

```
dipakprasad@Dipaks-MacBook-Pro flask-docker-app % docker login
Authenticating with existing credentials... [Username: dipakp22]

Info → To login with a different account, run 'docker logout' followed by 'docker login'

Login Succeeded
dipakprasad@Dipaks-MacBook-Pro flask-docker-app % docker tag my-flask-app:v1.0 dipakp22/flask-demo:v1.0
dipakprasad@Dipaks-MacBook-Pro flask-docker-app % docker push dipakp22/flask-demo:v1.0
The push refers to repository [docker.io/dipakp22/flask-demo]
6515746754d0: Pushed
73a2808109f1: Pushed
2be32b0beecd: Pushed
3e42af531f4c: Pushed
8dbbd075dad3: Pushed
ddb893bfd7ff: Pushed
17ab56e27011: Pushed
188f91a4ccf1: Pushed
e363695fcb93: Mounted from library/nginx
v1.0: digest: sha256:90844e898fe44cd33ef48ebb838c673a49da7ae53c9b96a8e4525719118bb414 size: 856
```



Q&A

1. Explain the key differences between Docker containers and virtual machines.

Feature	Docker Containers	Virtual Machines
Isolation	Process-level isolation (via Linux namespaces & cgroups)	Full OS-level isolation (hypervisor-based)
OS	Share the host kernel	Each VM has its own guest OS
Startup time	Seconds (lightweight)	Minutes (boot full OS)
Resource usage	Very low (no separate OS overhead)	Higher (each VM runs its own OS)
Portability	Highly portable (same image runs anywhere Docker runs)	Limited (depends on hypervisor, OS image)
Use case	Microservices, CI/CD, lightweight apps	Full system emulation, strong isolation, multiple OS types

In short: Containers virtualize the OS; VMs virtualize the hardware.

2. Why do containers in custom bridge networks have DNS resolution while default bridge network containers don't?

Network type	DNS behavior	Why
--------------	--------------	-----

-----	-----	-----

Default `bridge`	No container name resolution	Legacy mode; no built-in Docker DNS; containers must use IPs unless linked
Custom user-defined bridge	Automatic DNS resolution by container name	Docker's embedded DNS server is attached to user-defined networks; containers register automatically
In short: Only user-defined networks get Docker-managed DNS. The default bridge is legacy and lacks this feature for backward compatibility.		

3. When would you choose bind mounts over Docker volumes and vice versa?

Use Case	Choose	Reason
-----	-----	-----
Local development (edit code live, test)	Bind mount	Syncs host and container files instantly
Production app data (databases, uploads)	Docker volume	Managed by Docker, portable, safe, easy backup
Share data between containers	Volume	Centralized, consistent data store
Access specific host paths/configs	Bind mount	Direct host file access needed (e.g., logs, configs)
Rule of thumb: Dev → bind mounts Prod → volumes		

4. What strategies could you use to reduce Docker image size?

1. Use minimal base images - Prefer alpine, scratch, or distroless images. Example:
FROM python:3.12-alpine
 2. Multi-stage builds - Build in one stage, copy only the binary/runtime into a smaller final stage.
FROM golang:1.21 AS builder
WORKDIR /app
COPY . .
RUN go build -o app

FROM alpine
COPY --from=builder /app/app /app/app
CMD ["/app/app"]
 3. Minimize layers - Combine related RUN instructions:
>> RUN apt-get update && apt-get install -y curl && rm -rf /var/lib/apt/lists/*
 4. Clean up temporary files - Remove caches, package lists, etc., in the same layer.
 5. Use .dockerignore - Exclude unnecessary files (logs, node_modules, tests, etc.).
 6. Avoid unnecessary tools - Don't install editors or shells in production images.
- Result: smaller, faster-to-pull images, better security footprint.

5. What are three security best practices when building Docker images?

Best Practice	Description
-----	-----
Avoid running as root	Use `USER` directive to drop privileges: `USER appuser`
Use minimal base images	Smaller surface area = fewer vulnerabilities
Keep images updated	Rebuild often to include patched dependencies
Don't hardcode secrets	Use environment variables or Docker secrets instead
Verify image sources	Always use trusted base images (`FROM ubuntu:22.04`, not random tags)
Bonus: Enable image scanning (e.g., docker scan, Trivy) in CI/CD.	

6. What additional considerations would you need for running containers in production?

Running containers at scale introduces operational and security challenges beyond “just Docker run.”

- Orchestration - Use Kubernetes, Docker Swarm, or ECS for:
High availability, Auto-scaling, Service discovery, Rolling updates
- Networking - Use user-defined networks for isolation. Apply network policies (limit inter-container communication). Configure secure ingress/egress (TLS termination, reverse proxies).
- Storage - Use Docker volumes or external storage drivers (EFS, NFS, CSI). Ensure data persistence for stateful workloads.
- Security - Run containers as non-root. Scan images regularly. Use read-only root filesystems where possible. Limit capabilities (`--cap-drop all`).
- Monitoring & Logging - Centralized logs (ELK, Loki, CloudWatch, etc.) Metrics (Prometheus, Grafana) Health checks (HEALTHCHECK in Dockerfile)
- Resource limits - Set CPU and memory limits:
>> `docker run --memory=512m --cpus=1.0 ...`
- Secrets management - Use secret stores (Vault, Docker secrets, AWS Secrets Manager) — never bake credentials into images.