

Docker



docker®

By Eng: Abdelrahman Ayman Lotfy

DevOps track

Table of contents

Life Before Docker Compose	3
Manual Container Management	3
Networking Challenges	4
Configuration Sprawl	4
Scaling Issues	4
Environment Inconsistency	5
Life After Docker Compose	5
Centralized Configuration	5
Simplified Networking	6
Easy Environment Replication	6
Scaling Made Simple	7
Simplified Lifecycle Management	7
Integration with CI/CD	7
Core Components of a docker-compose.yml	8
My App	9
Notes on Scaling	10
Conclusion	11

Docker compose

Life Before Docker Compose

Before Docker Compose, developers relied solely on the Docker CLI and manual configuration to run and manage multiple containers. While this worked for single-container applications, it quickly became painful for multi-service systems.

Manual Container Management

- To run a simple multi-service application (e.g., a **web app + database**), each service had to be started with a long docker run command:

```
docker run -d --name db \  
-e POSTGRES_USER=user \  
-e POSTGRES_PASSWORD=pass \  
-v mydata:/var/lib/postgresql/data \  
postgres:latest
```

```
docker run -d --name web \  
-p 8080:80 \  
--link db:db \  
my-web-app:latest
```

Problems:

- Each container required remembering multiple flags (ports, env vars, volumes).
- Commands were hard to share between team members.
- If a container crashed, restarting it with the same options was tedious.

Networking Challenges

- By default, containers are isolated.
- Developers had to explicitly link containers or create custom networks.
- Example: making a web service talk to a database required using `--link` (deprecated now) or manually configuring networks.
- Scaling added more issues: if multiple containers of the same service were running, other services couldn't easily resolve which instance to connect to.

Configuration Sprawl

- Configurations were scattered across:
 - Bash scripts to start containers.
 - `.env` files for environment variables.
 - Documentation notes explaining “run this command first, then that one.”
- No **single source of truth**—every developer had a slightly different setup.
- Setting up a new environment often meant copy-pasting dozens of commands.

Scaling Issues

- Running multiple instances of a service (e.g., scaling a web server) was manual:

```
docker run -d -p 8081:80 my-web-app
```

```
docker run -d -p 8082:80 my-web-app
```

```
docker run -d -p 8083:80 my-web-app
```

- Load balancing had to be managed outside Docker (with an external reverse proxy).
- There was no easy way to scale down or replicate an entire multi-service environment.

Environment Inconsistency

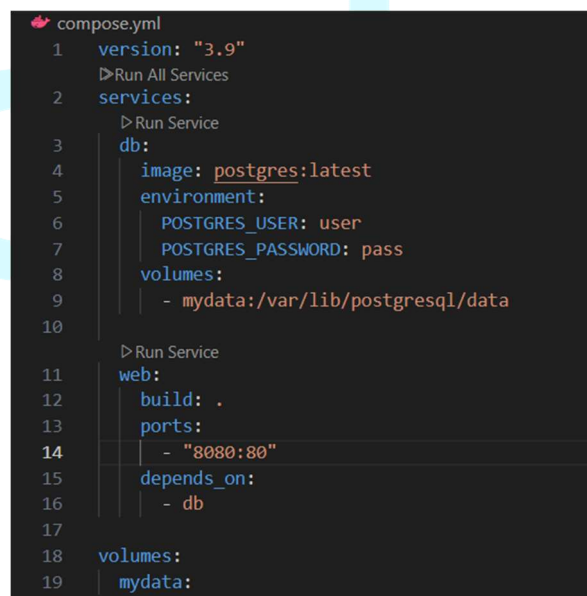
- Developers' local environments rarely matched staging or production.
- Minor differences in container run commands caused “it works on my machine” problems.
- Teams spent more time debugging setup issues than writing features.

Life After Docker Compose

Docker Compose changed the way developers build and manage multi-container applications. Instead of long, repetitive CLI commands and scattered configs, everything is described in a single YAML file (docker-compose.yml). With one command (docker-compose up), the entire stack can be started, stopped, scaled, or rebuilt.

Centralized Configuration

- All services are declared in a **single file**:



```
compose.yml
1  version: "3.9"
   ↳ Run All Services
2  services:
   ↳ Run Service
3    db:
4      image: postgres:latest
5      environment:
6        POSTGRES_USER: user
7        POSTGRES_PASSWORD: pass
8      volumes:
9        - mydata:/var/lib/postgresql/data
10
11   ↳ Run Service
12  web:
13    build: .
14    ports:
15      - "8080:80"
16    depends_on:
17      - db
18  volumes:
19    mydata:
```

Benefits:

- No need to memorize long docker run commands.
- Easy to **share** with teammates—just clone the repo and run docker-compose up.
- Acts as **documentation** and **source of truth** for the system.

Simplified Networking

- Compose automatically creates a **default network**.
- Containers can communicate using service names as hostnames (e.g., db instead of IP).
- No need for --link or manual docker network create.
- Example: the web service above can connect to the db service just by using db:5432.

Easy Environment Replication

- The same docker-compose.yml file can be used in:
 - Local development
 - Testing / staging
 - Continuous Integration (CI) environments
- Consistency reduces “it works on my machine” problems.
- New developers onboard in minutes by running:

`git clone <repo>`

`cd project`

`docker-compose up`

Scaling Made Simple

- Scaling services is built-in:

`docker-compose up --scale web=3`

- Compose automatically wires networking and service discovery for all replicas.
- Works seamlessly with load balancers and reverse proxies (like Nginx).

Simplified Lifecycle Management

- Common tasks become **one-liners**:

- Start everything: `docker-compose up -d`
- Stop everything: `docker-compose down`
- Rebuild services: `docker-compose up --build`
- View logs from all services: `docker-compose logs -f`

Integration with CI/CD

- Compose is widely used in pipelines for:
 - Spinning up temporary databases for testing.
 - Running multi-service integration tests.
 - Packaging reproducible dev/test environments.

Core Components of a docker-compose.yml

- Services
- Networks
- Volumes
- Environment variables

Let's make our first docker-compose file

Steps:

1. Think about services → In this app I need to run Image and attach to it data base so we here have 2 services (app & db)
2. Think about network you can use external one by make external: true but in our app I created a new one
3. You need to think about volumes to store your data
4. Also for each service you can put ports, volumes, network and healthcheck
5. Also you can put env variables in a separate file and call each variable in compose file like: \${name-of-var}
6. In our app db service should be made first so in app service put depends_on: db

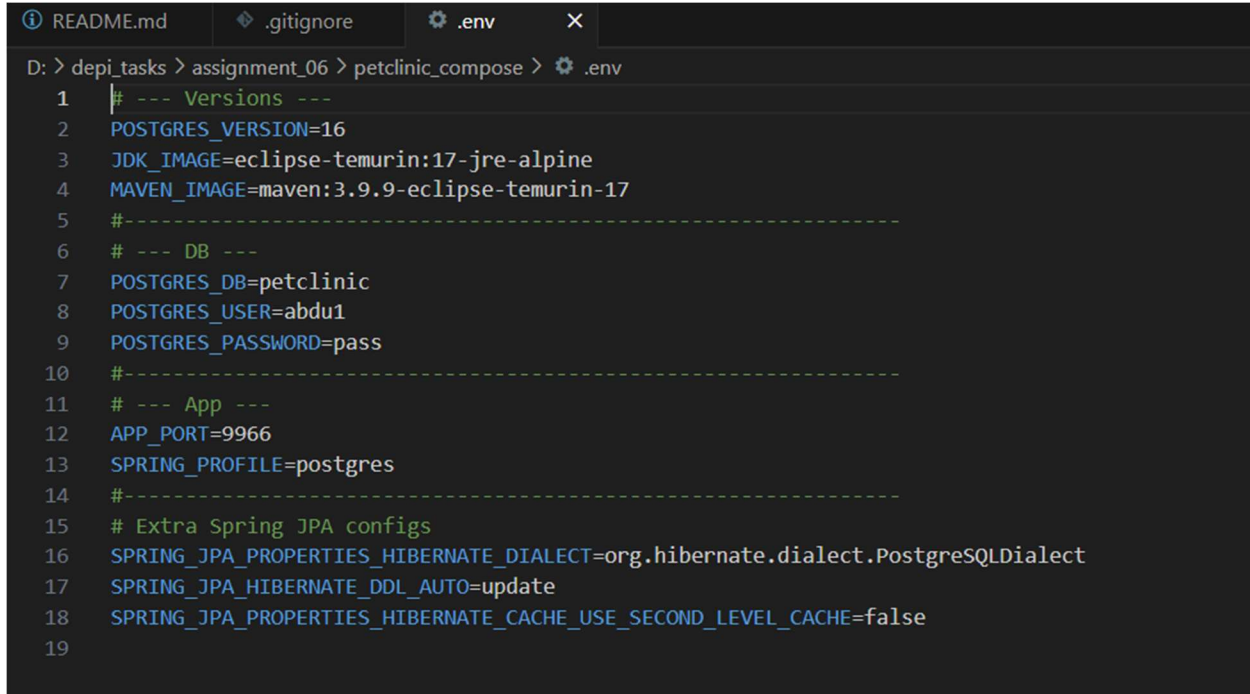
My Docker-compose file for spring pet clinic

```

1  version: '3.8'
2  services:
3    db:
4      image: postgres:${POSTGRES_VERSION}
5      container_name: meine_petclinic_db
6      environment:
7        POSTGRES_DB: ${POSTGRES_DB}
8        POSTGRES_USER: ${POSTGRES_USER}
9        POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
10     ports:
11       - "5434:5432"
12     volumes:
13       - vol1:/var/lib/postgresql/data
14     healthcheck:
15       test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d ${POSTGRES_DB}"]
16       interval: 1m30s
17       timeout: 30s
18       retries: 5
19       start_period: 30s
20     networks:
21       - abduu1
22
23  app:
24    build:
25      context: .
26      dockerfile: Dockerfile
27    args:
28      MAVEN_IMAGE: ${MAVEN_IMAGE}
29      JDK_IMAGE: ${JDK_IMAGE}
30    image: meine_petclinic_app
31    container_name: meine_petclinic_app_build
32    environment:
33      SPRING_DATASOURCE_URL: jdbc:postgresql://db:5432/${POSTGRES_DB}
34      SPRING_DATASOURCE_USERNAME: ${POSTGRES_USER}
35      SPRING_DATASOURCE_PASSWORD: ${POSTGRES_PASSWORD}
36      SPRING_JPA_PROPERTIES_HIBERNATE_DIALECT: ${SPRING_JPA_PROPERTIES_HIBERNATE_DIALECT}
37      SPRING_JPA_HIBERNATE_DDL_AUTO: ${SPRING_JPA_HIBERNATE_DDL_AUTO}
38      SPRING_JPA_PROPERTIES_HIBERNATE_CACHE_USE_SECOND_LEVEL_CACHE: ${SPRING_JPA_PROPERTIES_HIBERNATE_CACHE_USE_SECOND_LEVEL}
39
40    ports:
41      - "9990:9966"
42    depends_on:
43      - db
44    networks:
45      - abduu1
46  volumes:
47    vol1:
48
49  networks:
50    abduu1:

```

My .env file



```
D: > depi_tasks > assignment_06 > petclinic_compose > .env
1  # --- Versions ---
2  POSTGRES_VERSION=16
3  JDK_IMAGE=eclipse-temurin:17-jre-alpine
4  MAVEN_IMAGE=maven:3.9.9-eclipse-temurin-17
5  #-----
6  # --- DB ---
7  POSTGRES_DB=petclinic
8  POSTGRES_USER=abdu1
9  POSTGRES_PASSWORD=pass
10 #-----
11 # --- App ---
12 APP_PORT=9966
13 SPRING_PROFILE=postgres
14 #-----
15 # Extra Spring JPA configs
16 SPRING_JPA_PROPERTIES_HIBERNATE_DIALECT=org.hibernate.dialect.PostgreSQLDialect
17 SPRING_JPA_HIBERNATE_DDL_AUTO=update
18 SPRING_JPA_PROPERTIES_HIBERNATE_CACHE_USE_SECOND_LEVEL_CACHE=false
19
```

If you want to make scale:

Notes:

1. Take care about ports edit the previous ports to

ports:

- "9966"

Or

ports:

- "0:9966"

It will assign random ports to your application

2. Take care about your container name you can remove it and the name of the container will be :

<project_name>_<service_name>_<index>

Such as : depi_app_1 , depi_app_2 and so on.

3. Take care about your volume as many services will store in one volume so you can use **S3 bucket**.
4. Take care about Logging & Monitoring as when scaling logs are split across containers so you can use **ELK stack**, **Loki**, or cloud logging for better observability.
5. Take care about Environment Variables & Config as All replicas of a service will share the same environment variables so you can use a shared session store (**Redis**, **Memcached**).

Conclusion

Docker Compose makes it easy to define, run, and manage multi-container applications using a single YAML file. It simplifies networking, scaling, and environment consistency, making development faster and more reliable. While perfect for local development and small setups, larger production systems often require more advanced orchestration tools like **Kubernetes**.

docker®