SKILLS | DevOps and Cloud Computing

Docker Compose

Objective

- Understand the purpose of Docker Compose and when to use it.
- Define and configure multi-container applications using docker-compose.yml.
- Start, stop, and manage multi-container environments witl docker-compose up and docker-compose down.
- Use environment variables and scaling options in Docker Compose for flexibility and efficiency.

PW SKILLS

Explaining what Docker Compose is and why it is useful for managing multi-container applications

Let’s see

Docker Compose is a tool designed to simplify the management and deployment of multi-container applications. It uses a YAML configuration file (docker-compose.yml) to define all the services, networks, volumes, and dependencies required for an application. This allows developers to orchestrate containers as a single cohesive unit, rather than managing them individually.

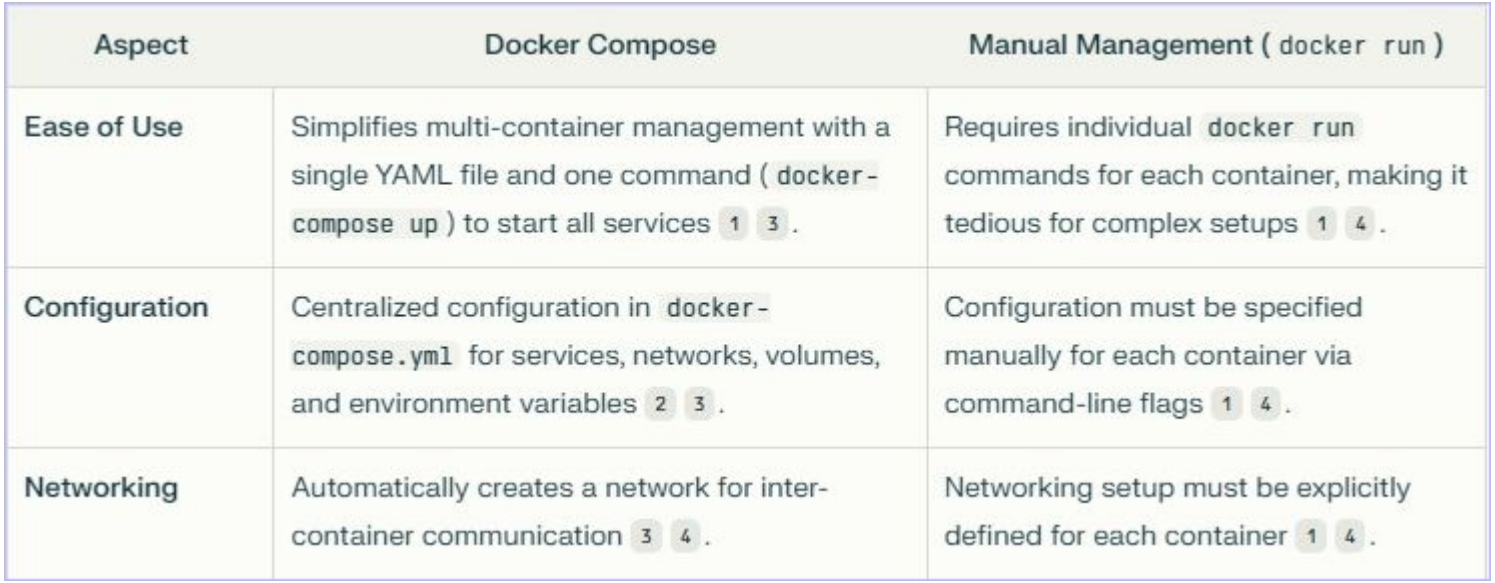
Why Docker Compose is Useful:

- Simplified Management
- Inter-container Communication
- Reproducibility
- Scalability
- Persistent Data

PW SKILLS

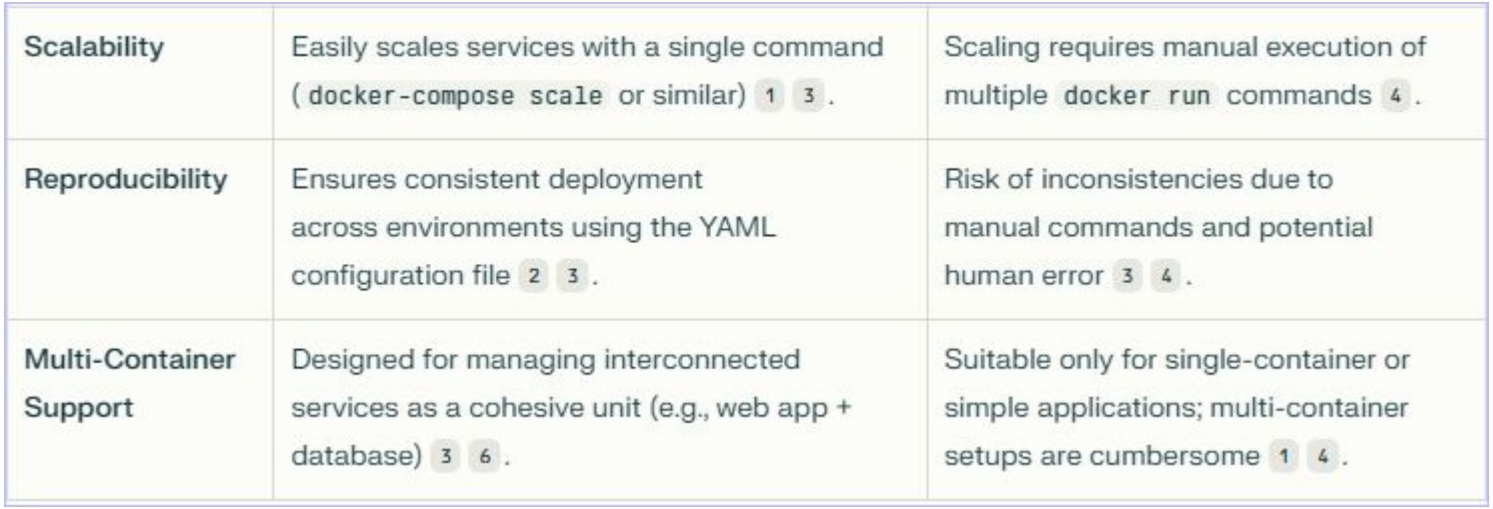
Comparing Docker Compose with manually managing multiple containers using docker run

Let’s compare



Aspect	Docker Compose	Manual Management (`docker run`)
Ease of Use	Simplifies multi-container management with a single YAML file and one command (`docker-compose up`) to start all services [1] [3].	Requires individual `docker run` commands for each container, making it tedious for complex setups [1] [4].
Configuration	Centralized configuration in `docker-compose.yml` for services, networks, volumes, and environment variables [2] [3].	Configuration must be specified manually for each container via command-line flags [1] [4].
Networking	Automatically creates a network for inter-container communication [3] [4].	Networking setup must be explicitly defined for each container [1] [4].

Let’s compare



Scalability	Easily scales services with a single command (`docker-compose scale` or similar) [1] [3].	Scaling requires manual execution of multiple `docker run` commands [4].
Reproducibility	Ensures consistent deployment across environments using the YAML configuration file [2] [3].	Risk of inconsistencies due to manual commands and potential human error [3] [4].
Multi-Container Support	Designed for managing interconnected services as a cohesive unit (e.g., web app + database) [3] [6].	Suitable only for single-container or simple applications; multi-container setups are cumbersome [1] [4].

PW SKILLS

Discuss common use cases, such as microservices, development environments, and production deployments.

Let’s discuss

1. **Microservices Architecture**

- Docker Compose is ideal for orchestrating microservices, where an application is split into multiple services (e.g., APIs, databases, caching layers) running in separate containers.
- It simplifies the management of dependencies and communication between services by defining them in a single docker-compose.yml file.
- Example: A system with a frontend, backend, and database can be launched and managed as a unified stack.

Let’s discuss

2. **Development Environments**

- Developers can replicate production-like environments locally using Docker Compose without needing complex infrastructure.
- It allows for quick setup of isolated environments with all required services (e.g., databases, APIs) using a single command (docker-compose up).
- Example: Setting up a Django app with PostgreSQL or a Node.js app with Redis for local testing.

Let’s discuss

3. **Production Deployments**

- While Docker Compose is primarily designed for development, it can also be used for single-host production deployments.
- It ensures consistency across environments by using the same configuration file for development, staging, and production.
- Example: Deploying small-scale applications on a single server without requiring complex orchestration tools like Kubernetes.

PW SKILLS

Explaining the structure of a docker-compose.yml file and its key components

Docker compose structure

A docker-compose.yml file is a YAML configuration file used by Docker Compose to define and manage multi-container applications. Its structure includes three key components: Services, Networks, and Volumes.

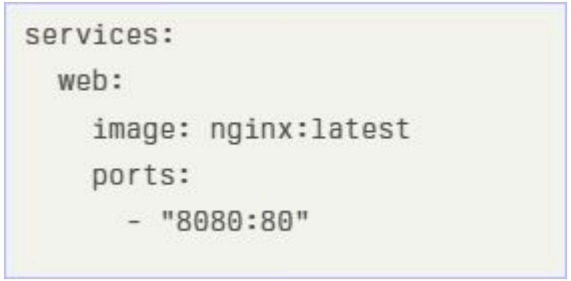
Key Components of a docker-compose.yml File:

1. **Services**

Defines the containers that make up your application. Each service specifies configurations such as:

- **Image:** The Docker image for the container.
- **Build:** Instructions to build the image.
- **Ports:** Port mappings between the host and container (e.g., "8080:80").
- **Environment Variables:** Configurations passed to the container.

Example:

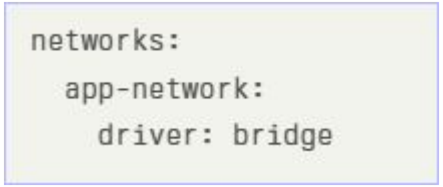


```
services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
```

2. Networks

Allows containers to communicate with each other securely. You can define custom networks for better isolation and control.

Example:

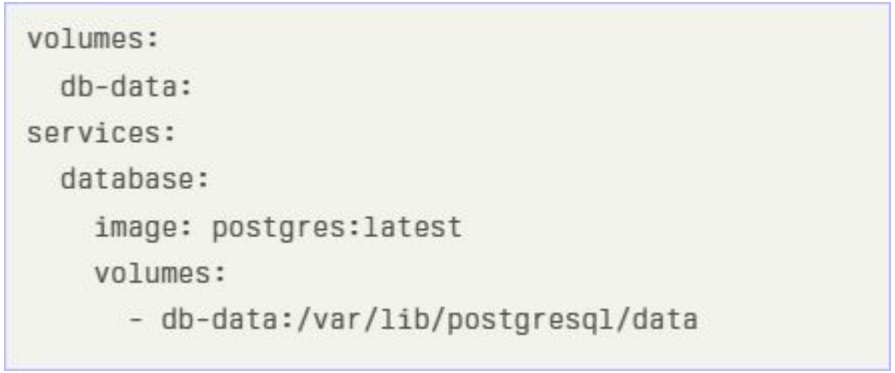


```
networks:
  app-network:
    driver: bridge
```

3. Volumes

Enables persistent storage by creating shared volumes that survive container restarts. Volumes are useful for storing data like databases or logs.

Example:

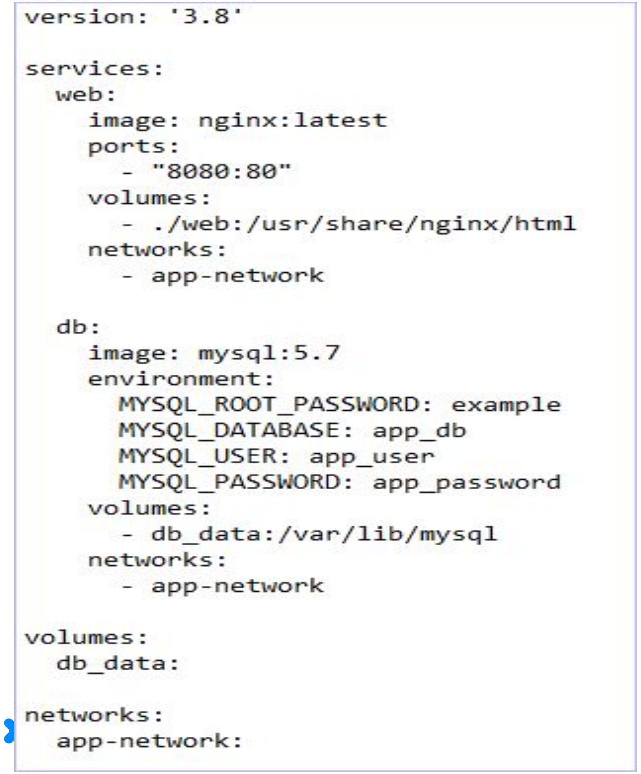


```
volumes:
  db-data:
services:
  database:
    image: postgres:latest
    volumes:
      - db-data:/var/lib/postgresql/data
```

PW SKILLS

Demonstrating running multiple containers with docker-compose up

Let’s do it



```
version: '3.8'

services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
    volumes:
      - ./web:/usr/share/nginx/html
    networks:
      - app-network

  db:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: example
      MYSQL_DATABASE: app_db
      MYSQL_USER: app_user
      MYSQL_PASSWORD: app_password
    volumes:
      - db_data:/var/lib/mysql
    networks:
      - app-network

volumes:
  db_data:

networks:
  app-network:
```

Pop Quiz

Q. Which section in the docker-compose.yml file is used to define the containers?

A

Services

B

Networks

Pop Quiz

Q. Which section in the docker-compose.yml file is used to define the containers?

A

Services

B

Networks

Take A 5-Minute Break!

- Stretch and relax
- Hydrate
- Clear your mind
- Be back in 5 minutes

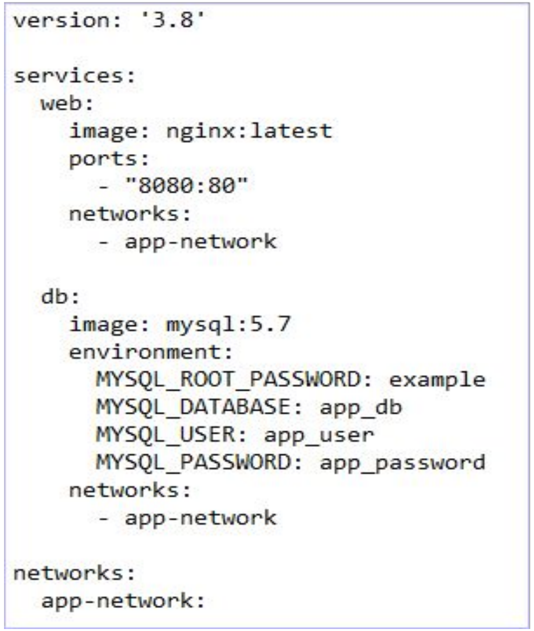
PW SKILLS

Demonstrating running multiple containers with docker-compose up

Let’s do it

1. **Create a docker-compose.yml File**

Define the services (containers) you want to run. For example, a web application with a database:

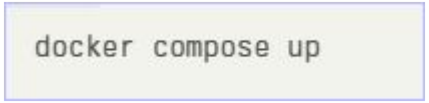


```
version: '3.8'

services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
    networks:
      - app-network

  db:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: example
      MYSQL_DATABASE: app_db
      MYSQL_USER: app_user
      MYSQL_PASSWORD: app_password
    networks:
      - app-network

networks:
  app-network:
```

Let’s do it

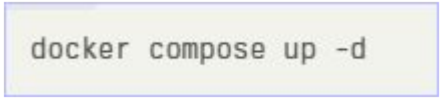
2. Run the Containers

Use the following command to start all containers defined in the docker-compose.yml file:



```
docker compose up
```

Add the -d flag to run them in detached mode (background):

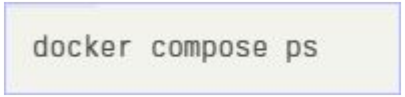


```
docker compose up -d
```

Let’s do it

3. Verify Running Containers

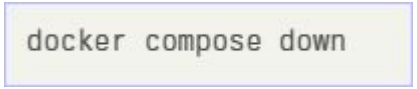
Check the status of running containers with:



```
docker compose ps
```

4. Stop and Remove Containers

To stop and remove all containers, use:



```
docker compose down
```

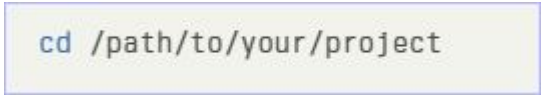
PW SKILLS

Explaining how to stop and remove containers using docker-compose down

Docker-compose down

1. **Navigate to the Project Directory**

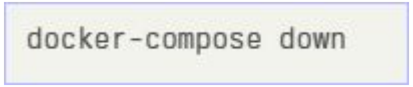
Go to the directory containing your docker-compose.yml file:



```
cd /path/to/your/project
```

2. **Run the Command**

Execute the docker-compose down command:



```
docker-compose down
```

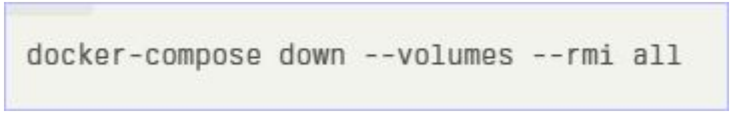
This stops all running containers and removes them along with their associated networks.

Docker-compose down

3. **Optional Flags**

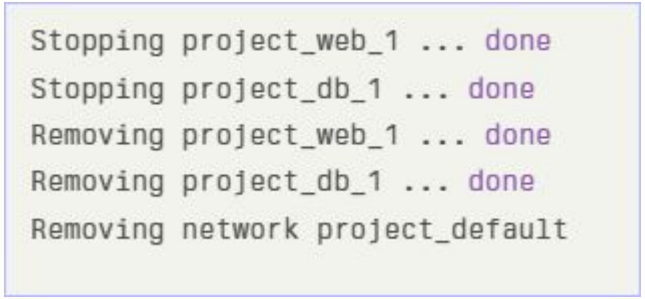
- --volumes: Removes volumes created by the application.
- --rmi all: Removes all images used by the services.

Example:



```
docker-compose down --volumes --rmi all
```

Example Output:



```
Stopping project_web_1 ... done
Stopping project_db_1 ... done
Removing project_web_1 ... done
Removing project_db_1 ... done
Removing network project_default
```

PW SKILLS

Discussing the benefits of running services in detached mode and restarting policies

Let’s discuss

Benefits of Running Services in Detached Mode:

1. Background Execution
2. Clean Terminal Output
3. Persistent Services
4. Multi-Service Management

Benefits of Restart Policies:

1. Automatic Recovery
2. Configurable Behavior
3. Enhanced Stability

PW SKILLS

The role of environment variables in configuring containerized applications dynamically.

Let’s see

Environment variables play a crucial role in dynamically configuring containerized applications by allowing developers to externalize configuration details without altering the application code. Here’s their role and benefits:

Role of Environment Variables

1. Dynamic Configuration
2. Separation of Code and Configuration
3. Portability

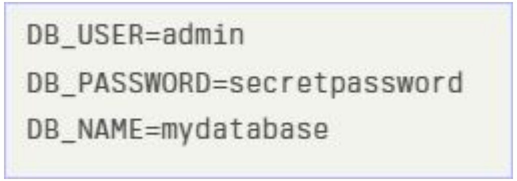
PW SKILLS

Demonstrating passing environment variables via the .env file and docker-compose.yml

Let’s do it

1. **Create a .env File**

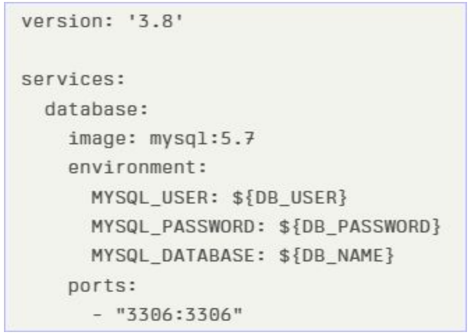
The .env file contains key-value pairs of environment variables. For example:



```
DB_USER=admin
DB_PASSWORD=secretpassword
DB_NAME=mydatabase
```

2. **Reference the .env File in docker-compose.yml**

In the docker-compose.yml, use variable interpolation to dynamically insert values from the .env file:

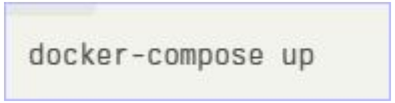


```
version: '3.8'

services:
  database:
    image: mysql:5.7
    environment:
      MYSQL_USER: ${DB_USER}
      MYSQL_PASSWORD: ${DB_PASSWORD}
      MYSQL_DATABASE: ${DB_NAME}
    ports:
      - "3306:3306"
```

Let’s do it

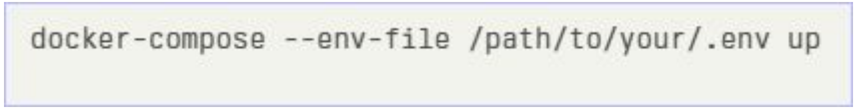
3. Run Docker Compose

Docker Compose automatically loads the .env file if it is in the same directory as the docker-compose.yml. Run the following command:



```
docker-compose up
```

If the .env file is located elsewhere or has a different name, specify it explicitly using the --env-file flag:



```
docker-compose --env-file /path/to/your/.env up
```

PW SKILLS

Discuss scaling services using docker-compose up --scale, explaining when and why scaling is necessary.

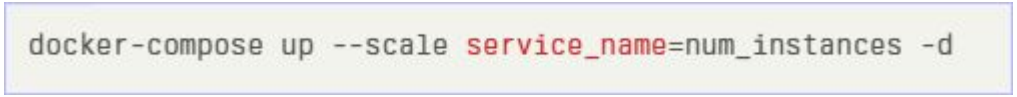
Let’s discuss

Scaling Services with docker-compose up --scale

The --scale flag in Docker Compose allows you to dynamically adjust the number of running instances (containers) for a specific service. This is useful for handling varying workloads or traffic demands efficiently.

Command Example

To scale a service:



```
docker-compose up --scale service_name=num_instances -d
```

Let’s discuss

When and Why Scaling is Necessary

1. Handling Increased Traffic
2. Load Balancing
3. Resource Optimization
4. Fault Tolerance

Best Practices for Scaling:

1. Stateless Services
2. Port Management
3. Monitoring

PW SKILLS

Time for case study!

Important

- Complete the post-class assessment
- Complete assignments (if any)
- Practice the concepts and techniques taught in this sessio
- Review your lecture notes
- Note down questions and queries regarding this session ar consult the teaching assistants

Thanks
!