.NET Microservices – Azure DevOps and AKS

Section 5: Docker Compose - Notes

Introduction to Docker Compose

Docker Compose is a tool that simplifies the management of multi-container Docker applications. It allows you to define and run applications with multiple services (such as web servers, databases, caching layers, etc.) using a single configuration file. Instead of manually running and linking several containers, Docker Compose orchestrates everything with one command.

Docker Compose uses a YAML file (docker-compose.yml) to configure the services, networks, and volumes that the application needs. With Compose, all the complexity of setting up and managing multiple Docker containers is abstracted into a single file, making it much easier to manage and scale applications.

Key Use Cases of Docker Compose

1. **Multi-Container Applications**: Many modern applications require multiple services to function, such as databases, APIs, message brokers, etc. Docker Compose allows you to define and link these services easily.
2. **Development Environments**: Compose is widely used in development to simulate a full stack of services locally. Developers can spin up containers for web apps, databases, and message queues with a single command.
3. **Testing and CI/CD**: In continuous integration pipelines, Docker Compose is often used to quickly spin up test environments, where the application and all its dependencies can be run in containers. This ensures consistent environments for testing.

Benefits of Docker Compose

1. **Simplified Multi-Container Management**
    - Without Docker Compose, managing multiple services requires manually running several Docker commands for each service, linking containers, and managing their lifecycles. With Compose, all services are defined in a YAML file, and one command (docker-compose up) can start and configure the entire application stack.
2. **Declarative Service Definitions**
    - The docker-compose.yml file provides a declarative way to define services, volumes, and networks. This makes it easy to version control your container setups and share configurations across teams.

3. **Consistent Environments**
    - Compose ensures that the services defined in your YAML file behave the same way across different environments, whether it's a local machine or a production environment. This reduces the chance of environment-specific bugs.
4. **Easy Networking**
    - Docker Compose automatically sets up networks for services so they can communicate with each other. Services are identified by their names, making it easy to create service dependencies. For example, your web app can simply reference the database service by name to connect.
5. **Scalability**
    - Docker Compose allows you to scale services effortlessly. You can specify how many instances of a service you want to run, and Compose will handle the networking and load balancing between them. This is useful for simulating production-like environments.
6. **Volume Management**
    - Compose provides built-in support for persistent storage with volumes, ensuring that data remains available even when containers are stopped and restarted. This is crucial for services like databases where data persistence is required.
7. **Isolation**
    - Compose keeps services isolated from each other unless explicitly defined to communicate. This ensures that different services don't interfere with each other unless necessary, improving the stability of your environment.

How Docker Compose Works

1. **Defining the Application**: All services, networks, and volumes are defined in a single docker-compose.yml file.
2. **Starting Services**: With the command docker-compose up, Docker Compose reads the YAML file and brings up all defined services. It ensures that any necessary networks and volumes are created, and it starts the containers.
3. **Stopping Services**: The command docker-compose down stops and removes all running containers and networks defined by the Compose file. Optionally, you can also remove volumes and images created by Compose.

Real-World Example

Let's say you're developing an e-commerce application that consists of the following services:

- **Web Server**: Runs your application code (e.g., Node.js or ASP.NET Core).
- **Database**: Stores the application's data (e.g., MySQL or PostgreSQL).

- **Cache**: Speeds up the application by caching data (e.g., Redis).

Instead of manually configuring each of these services and their networks, you can define them in a docker-compose.yml file. This file will specify how these containers should run, which ports to expose, how they communicate, and how their data is persisted. One command, docker-compose up, will start the entire application with all these services working together.

Advantages Over Docker Run

While Docker commands can individually create, start, and stop containers, Docker Compose provides a higher level of abstraction for orchestrating multiple containers together. Key differences:

1. **Single Command**: Docker Compose eliminates the need to issue multiple docker run commands. Everything is defined in a configuration file, and you only need one command to start the entire application.
2. **Easy Maintenance**: Making changes to your environment (e.g., adding a new service, modifying a configuration) is easier with Docker Compose. You modify the YAML file and simply restart the services. Docker Compose handles container creation, destruction, and updates without requiring complex commands.

Docker Compose YAML File and Important Properties

The core of Docker Compose is the **docker-compose.yml** file. This file contains the configuration for all the services, networks, and volumes that are needed for the application. It is written in **YAML (YAML Ain't Markup Language)**, which is a human-readable data format, making it easy to define the setup for multi-container applications.

Let's dive into the structure of a Docker Compose file and the most important properties.

Basic Structure of docker-compose.yml

A typical docker-compose.yml file contains three main sections:

1. **Version**: Specifies the version of Docker Compose file format.
2. **Services**: Defines the individual containers (e.g., web, database, cache) that make up the application.
3. **Networks and Volumes**: Optionally defines custom networks and volumes for inter-service communication and data persistence.

Here's an example of a basic Docker Compose file:

```
version: '3.8'  # Specifies the version of the Docker Compose file format
```

```
services:       # Defines all the services (containers)
  web:          # Defines a service named 'web'
    image: nginx:latest   # Pulls the NGINX image from Docker Hub
    ports:
      - "8080:80"         # Maps port 80 in the container to port 8080 on the host machine

  db:           # Defines a service named 'db' (database)
    image: mysql:5.7      # Pulls the MySQL image from Docker Hub
    environment:          # Passes environment variables to the container
      MYSQL_ROOT_PASSWORD: example

networks:       # Optionally define custom networks
  default:      # 'default' network is created automatically
    driver: bridge  # Default network driver is bridge (used for inter-container communication)
```

Key Properties in docker-compose.yml

1. **version**

```
version: '3.8'
```

- Specifies the version of Docker Compose syntax being used.
- As of now, 3.8 is commonly used, and it provides a set of features for modern Compose files.
- Ensure compatibility between the Docker engine and Docker Compose version by checking which versions support the specific features you need.

2. **services**
3. services:
4. web:
5. image: nginx:latest
6. ports:

- "8080:80"

- **Services** are the individual containers that form your multi-container application. Each service has its own configuration like which Docker image to use, how to expose ports, and environment variables.
- **image**: Specifies the Docker image to use for this service. If the image doesn't exist locally, it is pulled from a Docker registry like Docker Hub.
- **ports**: Maps ports between the host and the container. In the example, port 8080 on the host is mapped to port 80 in the NGINX container (commonly used for web traffic).

7. **build**
8. services:
9. web:
10. build:
11. context: ./app

dockerfile: Dockerfile

- Instead of pulling an image from a registry, you can **build** your own image directly within the Compose file.
- **context**: Specifies the build context (i.e., the directory containing your Dockerfile and source code).
- **dockerfile**: Points to a specific Dockerfile, in case it's not located in the root directory of the build context.

12. **environment**
13. services:
14. db:
15. image: mysql:5.7
16. environment:

MYSQL_ROOT_PASSWORD: example

- The **environment** property allows you to pass environment variables into your container. These variables are often required for configuring services like databases (e.g., setting up passwords or connection details).
- Environment variables defined here are equivalent to using the -e flag with docker run.

17. **volumes**

18. services:
19. web:
20. volumes:

- ./code:/var/www/html

- **Volumes** allow you to mount a directory from the host machine into the container. This is useful for persisting data or for sharing source code between the host and the container.
- In this example, the local directory ./code is mounted to /var/www/html inside the container.
- Volumes are often used with databases to persist data, so it remains intact even if the container is restarted.

21. **depends_on**
22. services:
23. web:
24. depends_on:

- db

- The **depends_on** property defines dependencies between services. In this example, the web service depends on the db (database) service. This ensures that Docker Compose starts the db service before the web service.
- However, **depends_on** does not wait for the database service to be ready—it only ensures that it is started first. For readiness checks, you may need to implement health checks.

25. **networks**
26. networks:
27. app-network:

driver: bridge

- Docker Compose can define custom **networks** to control how containers communicate with each other.
- By default, services within the same Compose file are placed in a shared network, but you can also define your own.
- **driver**: Specifies the type of network driver (e.g., bridge, overlay).

28. **volumes (external)**
29. volumes:
30. data-volume:

external: true

- You can define **external volumes**, which are shared across multiple Docker Compose projects or containers. These volumes are not deleted when the services are taken down.

Example of a Comprehensive docker-compose.yml

```
version: '3.8'

services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8080:80"
    depends_on:
      - db
    environment:
      - APP_ENV=development

  db:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: example
    volumes:
      - db-data:/var/lib/mysql

networks:
```

```
  app-network:
    driver: bridge

volumes:
  db-data:
    external: false
```

- **web** service is built from a Dockerfile, exposing port 8080 and depending on the db service.
- **db** service uses the MySQL image, with a password passed through environment variables, and data is persisted using a volume (db-data).
- **networks**: Custom network app-network is defined to allow services to communicate.
- **volumes**: A local volume db-data is created to store MySQL data.

Benefits of Using docker-compose.yml

1. **Simplified Configuration**: All services, volumes, and networks are defined in one file, making the infrastructure easier to understand and modify.
2. **Portability**: The Compose file can be shared across teams or environments, ensuring consistency across different setups.
3. **Service Isolation**: Compose isolates services by default, allowing only defined services to communicate with each other.
4. **Easy Scaling**: With docker-compose, scaling services is as easy as adding scale parameters or adjusting the YAML file.
5. **Automation**: Docker Compose abstracts complex setups, reducing the number of manual Docker commands needed to start multi-container environments.

Docker Compose Commands in Terminal (Windows)

Docker Compose provides a powerful command-line interface (CLI) to manage multi-container applications. These commands help you define, run, stop, and manage your services as described in the docker-compose.yml file.

Let’s go through the most commonly used Docker Compose commands, particularly in the Windows terminal.

1. docker-compose up

The docker-compose up command is the most fundamental one. It creates and starts all services defined in your docker-compose.yml file.

docker-compose up

- **How it works**:
    - It reads the docker-compose.yml file and starts all the containers in the order specified by depends_on if defined.
    - If any services are not built (e.g., if they use a build directive), Compose will build the necessary images.
    - If the services are already running, the command will leave them as-is, unless changes have been made.
- **Flags**:
    - -d: Runs the containers in detached mode (i.e., in the background).

docker-compose up -d

Running in detached mode lets you continue using the terminal while the services run in the background.

- --build: Forces a rebuild of the service images before starting the containers.

docker-compose up --build

This is helpful when you’ve made changes to the Dockerfile or source code and need to rebuild the service images.

- --force-recreate: Recreates all containers even if their configuration or images haven’t changed.

docker-compose up --force-recreate

Use this when you need to refresh your container environment without changing the underlying configuration.

2. docker-compose down

The docker-compose down command stops and removes all containers, networks, and volumes that were created by docker-compose up.

docker-compose down

- **How it works**:
    - This command stops all the services and then removes the containers.
    - By default, it doesn’t remove any volumes unless explicitly specified.
- **Flags**:
    - --volumes: Removes all volumes that were created by Docker Compose.

docker-compose down --volumes

This is useful when you want to clear out all persistent data (e.g., databases) and start fresh.

    - --rmi all: Removes all the images created by the docker-compose setup.

docker-compose down --rmi all

This is useful if you want to free up disk space by removing unused images.

3. docker-compose build

The docker-compose build command builds or rebuilds the images for services defined in the Compose file. This is particularly useful if you're developing locally and need to continuously build and test changes.

docker-compose build

- **How it works**:
    - It reads the build context in the docker-compose.yml file and builds Docker images for the specified services.
- **Flags**:
    - --no-cache: Forces Docker Compose to build the images without using the cache.

docker-compose build --no-cache

This can be used when changes to the Dockerfile aren’t being picked up, or when you want to ensure everything is built from scratch.

    - --pull: Always attempts to pull the latest version of the base images before building.

docker-compose build --pull

4. docker-compose stop

The docker-compose stop command stops all running containers without removing them.

docker-compose stop

- **How it works**:
    - Unlike docker-compose down, which stops and removes containers, docker-compose stop only stops the containers. They remain in a stopped state and can be restarted later using docker-compose start.

5. docker-compose start

The docker-compose start command starts existing containers that were stopped with docker-compose stop.

docker-compose start

- **How it works**:
    - This is used to bring back stopped containers without recreating or rebuilding them.

6. docker-compose logs

The docker-compose logs command displays the logs from all containers defined in the docker-compose.yml file. This is useful for debugging and monitoring services in real time.

docker-compose logs

- **Flags**:
    - -f: Follows the log output in real-time (similar to tail -f).

docker-compose logs -f

This flag is very useful when monitoring the live behavior of your services.

7. docker-compose ps

The docker-compose ps command shows the status of the containers created by Docker Compose.

docker-compose ps

- **How it works**:
    - It lists all the running containers, their states (e.g., running, exited), ports exposed, and the command used to start them.
    - This command helps in identifying if a service is running as expected or has crashed.

8. docker-compose exec

The docker-compose exec command allows you to run arbitrary commands inside a running service container. This is useful for debugging and inspecting containers.

docker-compose exec <service-name> <command>

- **Example**:

docker-compose exec web bash

  - This command opens an interactive Bash shell inside the web container. You can then inspect the file system, logs, or manually run commands.

9. docker-compose scale

The docker-compose scale command allows you to scale a specific service up or down. This means you can increase or decrease the number of container instances for a given service.

docker-compose up --scale <service-name>=<number-of-instances>

- **Example**:

docker-compose up --scale web=3

  - This command starts three instances of the web service and balances them across the defined network.

- **Note**: Scaling is useful in development to simulate production-like environments with multiple instances of the same service (e.g., for load balancing).

10. docker-compose config

The docker-compose config command validates and displays the configuration file. This is helpful to ensure that the docker-compose.yml file is correct before running any commands.

docker-compose config

- **How it works**:

  - It parses the docker-compose.yml file and checks for syntax errors, missing properties, or deprecated options.

11. docker-compose pull

The docker-compose pull command pulls the latest version of images defined in the Compose file. It is useful when you want to make sure that you're using the most up-to-date images for services that don't use local builds.

docker-compose pull

12. docker-compose restart

The docker-compose restart command restarts all or specific services. It's a useful command to refresh services without taking down the entire environment.

docker-compose restart <service-name>

Common Workflow Using Docker Compose Commands

1. **Start Your Services**:

docker-compose up -d

2. **Check Running Containers**:

docker-compose ps

3. **Monitor Logs**:

docker-compose logs -f

4. **Execute a Command in a Service**:

docker-compose exec <service> <command>

5. **Scale Services**:

docker-compose up --scale web=3

6. **Stop and Remove All Services**:

docker-compose down --volumes

Key Points to Remember (for Interview Preparation)

- **docker-compose up** starts and creates services, networks, and volumes defined in docker-compose.yml.
- **docker-compose down** stops and removes services, networks, and volumes, unless volumes are preserved explicitly.
- **docker-compose build** builds services from a Dockerfile if defined.
- **docker-compose exec** allows you to run commands inside a running container.

Scaling is possible using the --scale flag to simulate load balancing and multi-instance setups.

Log monitoring and **service inspection** are done using docker-compose logs and docker-compose ps.

Sample Code

```
version: "3.8"
services:
 mysql-container:
  image: mysql:8.3.0
  environment:
   - MYSQL_ROOT_PASSWORD=admin
  ports:
   - "3306:3306"
  volumes:
   - ./mysql-init:/docker-entrypoint-initdb.d
  networks:
   - ecommerce-network
  hostname: mysql-host-productsmicroservice

 products-microservice:
  image: harshamicroservices/ecommerce-products-microservice:v1.0
  environment:
```

```
  MYSQL_HOST: mysql-host-productsmicroservice
  MYSQL_PASSWORD: admin
 ports:
  - "8080:8080"
 networks:
  - ecommerce-network


postgres-container:
 image: postgres:16.1
 environment:
  - POSTGRES_USER=postgres
  - POSTGRES_PASSWORD=admin
  - POSTGRES_DB=eCommerceUsers
 ports:
  - "5432:5432"
 volumes:
  - ./postgres-init:/docker-entrypoint-initdb.d
 networks:
  - ecommerce-network

users-microservice:
 image: harshamicroservices/ecommerce-users-microservice:v1.0
 environment:
  - POSTGRES_HOST=postgres-container
  - POSTGRES_PASSWORD=admin
 ports:
  - "9090:9090"
 networks:
  - ecommerce-network
```

```
networks:
 ecommerce-network:
  driver: bridge
```

Sample Code Explanation

1. version: "3.8"

- **Explanation**:
    - Specifies the version of Docker Compose syntax used in this file. Version 3.8 is compatible with newer Docker Compose features.
    - Each version brings new functionalities and deprecates older ones, so using the correct version ensures compatibility with the Docker Engine and features like networks and volumes.

2. services:

- **Explanation**:
    - This is the main section where you define your application services. Each service runs in its own container, and you can configure how they interact with each other.

MySQL Service Definition (mysql-container):

3. mysql-container:

- **Explanation**:
    - This defines a container named mysql-container that will run the MySQL database.
    - Services are given unique names, and this will be used as a reference elsewhere (like in networking or linking).

4. image: mysql:8.3.0

- **Explanation**:
    - This line pulls the MySQL Docker image version 8.3.0 from Docker Hub. This container will run a MySQL database instance.

5. environment:

- **Explanation**:
    - Environment variables are used to pass configuration parameters to the container.

6. - MYSQL_ROOT_PASSWORD=admin

- **Explanation**:
    - This sets the environment variable MYSQL_ROOT_PASSWORD to admin, which configures the root password for the MySQL server.

7. ports:

- **Explanation**:
    - This maps ports between the container and the host machine so that the service is accessible outside of the Docker network.

8. - "3306:3306"

- **Explanation**:
    - The first 3306 is the port on the host, and the second 3306 is the port inside the container. This mapping exposes MySQL on port 3306 (the default MySQL port) to the host machine.
    - You can connect to the MySQL instance on localhost:3306 from the host.

9. volumes:

- **Explanation**:
    - This section mounts volumes to persist data or execute scripts. Volumes allow files from the host system to be made available inside the container.

10. - ./mysql-init:/docker-entrypoint-initdb.d

- **Explanation**:
    - This maps the ./mysql-init directory on the host to /docker-entrypoint-initdb.d inside the container.
    - The MySQL container will run any SQL scripts placed in this directory during initialization, which is a common way to pre-load the database with schema or seed data.

11. networks:

- **Explanation**:
    - Defines the network that the service will be a part of. Networking allows containers to communicate with each other.

12. - ecommerce-network

- **Explanation**:
    - Specifies that this container is part of the ecommerce-network, enabling communication with other containers on this network.

13. hostname: mysql-host-productsmicroservice

- **Explanation**:
    - Sets the hostname of the MySQL container to mysql-host-productsmicroservice, which other services (like the Products Microservice) can use to connect to this container.

Products Microservice (products-microservice):

14. products-microservice:

- **Explanation**:
    - Defines a service named products-microservice, which is a microservice for handling product-related functionality in the eCommerce system.

15. image: harshamicroservices/ecommerce-products-microservice:v1.0

- **Explanation**:
    - This service will run a pre-built Docker image harshamicroservices/ecommerce-products-microservice:v1.0 (presumably hosted on Docker Hub or a private repository). This image is for the Products Microservice.

16. environment:

- **Explanation**:
    - Passes environment variables required by the service to function properly.

17. MYSQL_HOST: mysql-host-productsmicroservice

- **Explanation**:
    - This sets the MYSQL_HOST environment variable to mysql-host-productsmicroservice, which was defined as the hostname for the MySQL container. This tells the microservice where to connect to the MySQL database.

18. MYSQL_PASSWORD: admin

- **Explanation**:
    - The MYSQL_PASSWORD environment variable is set to admin, which is the password to connect to the MySQL database. This should match the password used in the MySQL container.

19. ports:

- **Explanation**:
    - Maps the internal port of the service to the host machine.

20. - "8080:8080"

- **Explanation**:
    - Maps port 8080 inside the container to port 8080 on the host machine. This exposes the Products Microservice to localhost:8080.

21. networks:

- **Explanation**:
    - The products-microservice is also connected to the ecommerce-network, enabling communication with the MySQL container.

PostgreSQL Service Definition (postgres-container):

22. postgres-container:

- **Explanation**:
    - Defines a service named postgres-container that will run a PostgreSQL database instance for managing user data.

23. image: postgres:16.1

- **Explanation**:
    - This line pulls the PostgreSQL Docker image version 16.1 from Docker Hub.

24. environment:

- **Explanation**:
    - Sets the necessary environment variables for PostgreSQL.

25. - POSTGRES_USER=postgres

- **Explanation**:
    - This sets the POSTGRES_USER environment variable, specifying the username to be used when connecting to PostgreSQL. The username is postgres.

26. - POSTGRES_PASSWORD=admin

- **Explanation**:
    - This sets the POSTGRES_PASSWORD environment variable, which is the password to authenticate the POSTGRES_USER.

27. - POSTGRES_DB=eCommerceUsers

- **Explanation**:
    - Specifies the name of the database that will be created by default, which in this case is eCommerceUsers.

28. ports:

- **Explanation**:
    - Maps the internal PostgreSQL port to the host machine.

29. - "5432:5432"

- **Explanation**:
    - Maps port 5432 inside the container to port 5432 on the host machine. PostgreSQL is now accessible via localhost:5432.

30. volumes:

- **Explanation**:
    - Mounts the ./postgres-init directory on the host to the /docker-entrypoint-initdb.d directory in the container. This allows initialization scripts for PostgreSQL.

31. - ./postgres-init:/docker-entrypoint-initdb.d

- **Explanation**:
    - Any SQL scripts inside the ./postgres-init folder on the host will be executed when the container initializes.

32. networks:

- **Explanation**:
    - Connects the PostgreSQL container to the ecommerce-network so that it can communicate with other services.

Users Microservice (users-microservice):

33. users-microservice:

- **Explanation**:
    - Defines a service named users-microservice for handling user-related functionality in the eCommerce system.

34. image: harshamicroservices/ecommerce-users-microservice:v1.0

- **Explanation**:
    - This service will run a pre-built Docker image for the Users Microservice, pulled from the Docker registry.

35. environment:

- **Explanation**:
    - Sets environment variables for connecting to the PostgreSQL database.

36. - POSTGRES_HOST=postgres-container

- **Explanation**:

- This sets the POSTGRES_HOST environment variable to postgres-container, allowing the microservice to connect to the PostgreSQL container.

37. - POSTGRES_PASSWORD=admin

- **Explanation**:
  - The POSTGRES_PASSWORD environment variable is set to admin, which should match the password defined in the postgres-container.

38. ports:

- **Explanation**:
  - Exposes the Users Microservice on port 9090.

39. - "9090:9090"

- **Explanation**:
  - Maps port 9090 in the container to port 9090 on the host machine, allowing access to the Users Microservice at localhost:9090.

40. networks:

- **Explanation**:
  - Connects the users-microservice to the ecommerce-network, allowing it to communicate with the PostgreSQL container and other services.