DevOps Concepts & Docker Lab
A Summary of Docker Concepts and Commands

Amo Samuel

Docker: A Powerful Platform for Containerized Application Deployment

As I continue to explore tools for streamlining application deployment and management, Docker has stood out as a key player in modern software development. It simplifies packaging applications by leveraging containers, enabling consistent behavior across various environments—whether I'm working on my local machine, in a test environment, or deploying to production.

Why Docker?

One of the main reasons it is embraced is its ability to eliminate the "it works on my machine" problem. Containers isolate the application and all its dependencies, so I can rest assured that the behavior remains consistent regardless of where it runs. This has been invaluable for ensuring smooth transitions between development, testing, and production environments.

Key Docker Components

- **Docker Engine**: The heart of Docker. It's the core component responsible for creating, managing, and running containers. The way it abstracts away system-level details and isolates applications allows me to focus purely on building and running my software.
- **Docker Image**: This is essentially a blueprint for a container. When I build a Docker image, I package not only my code but also everything it needs to run—libraries, system tools, and settings. It ensures consistency and eliminates potential conflicts.
- **Docker Container**: This is where the magic happens—a running instance of my Docker image. Containers provide isolated environments that allow me to run applications securely and reliably. Whether I need a single container for a simple app or multiple interconnected containers for a complex service, Docker ensures each one runs smoothly.

- **Docker Compose**: Managing multiple containers for an application can get complicated, which is where Docker Compose comes in handy. It allows developers to define services, networks, and volumes in a simple YAML file, making it easier to manage multi-container applications.

Advanced Docker Techniques

While the basics of Docker are powerful, diving into some advanced techniques has taken my workflow to the next level. These include optimizing Dockerfile for faster builds, using multi-stage builds to keep images small, and configuring custom networks to ensure containers can communicate efficiently.

Moreover, integrating Docker with CI/CD pipelines has been a game-changer. By ensuring containers are tested and built automatically, I can ensure the reliability of deployments.

Key Docker Commands I have Utilized

Building and Running Containers

- **docker-compose up**: I have relied heavily on this command to build and start services defined in a docker-compose.yml file. Whether it's spinning up a development environment or testing
- **docker-compose down**: When I need to stop and clean up the running containers, this command ensures everything is shut down and removed.
- **docker build . -t myapp**: When building images, I specify a tag (myapp) so I can easily identify and manage it later. This command compiles my Dockerfile and creates an image from the contents of the current directory.

Container Management

- **docker ps**: Lists all the running containers, helping me monitor and track active services. I use it regularly to check container IDs and statuses.
- **docker stop <container-id>**: If I need to manually stop a container (perhaps to update its image or reconfigure it), this command comes in handy.

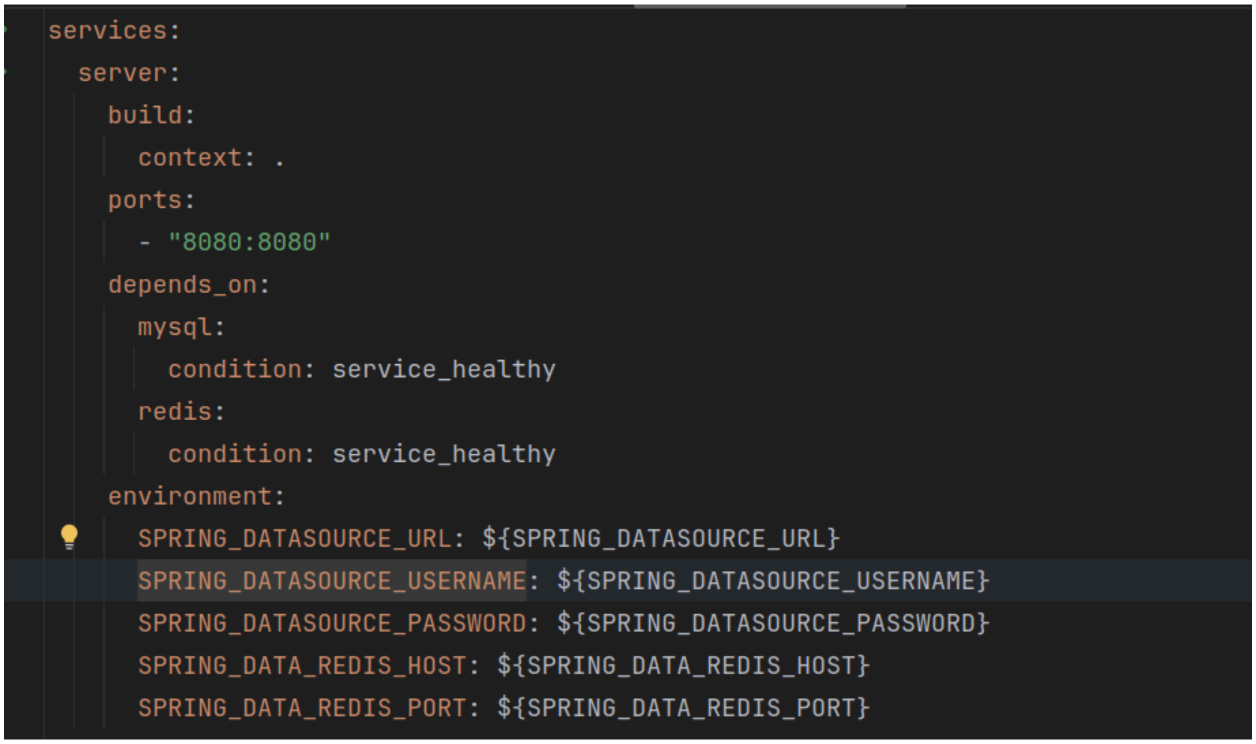
- **docker logs <container-id>**: Debugging issues in containers becomes straightforward with this command. I use it to inspect logs and ensure everything is running as expected.

Volumes and Networks

- **docker volume ls**: Lists all the Docker volumes on my system. Volumes are essential for data persistence, especially when containers are ephemeral.
- **docker network ls**: Lists Docker networks, which I often set up to enable communication between containers.
- **docker volume rm <volume-name>**: Occasionally, I'll need to clean up unused volumes to save space and reduce clutter, and this command does just that.

Docker Compose Example Breakdown

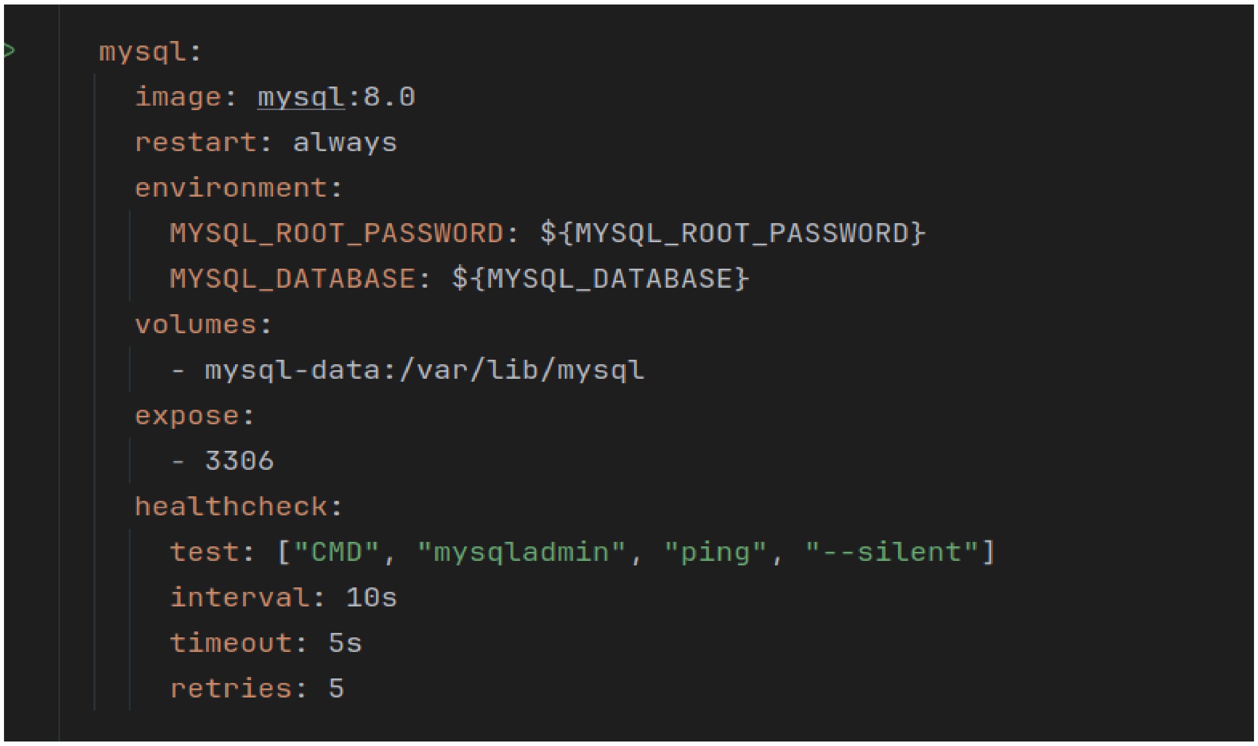
The given YAML file configures a multi-service application consisting of a Spring Boot server, MySQL database, and Redis.



```yaml
services:
  server:
    build:
      context: .
    ports:
      - "8080:8080"
    depends_on:
      mysql:
        condition: service_healthy
      redis:
        condition: service_healthy
    environment:
      SPRING_DATASOURCE_URL: ${SPRING_DATASOURCE_URL}
      SPRING_DATASOURCE_USERNAME: ${SPRING_DATASOURCE_USERNAME}
      SPRING_DATASOURCE_PASSWORD: ${SPRING_DATASOURCE_PASSWORD}
      SPRING_DATA_REDIS_HOST: ${SPRING_DATA_REDIS_HOST}
      SPRING_DATA_REDIS_PORT: ${SPRING_DATA_REDIS_PORT}
```

- **build**: Specifies that the server container should be built from a Dockerfile in the current context (working directory).
- **ports**: Maps port 8080 on the host to port 8080 in the container.
- **depends_on**: Ensures that the mysql and redis services are healthy before starting the server

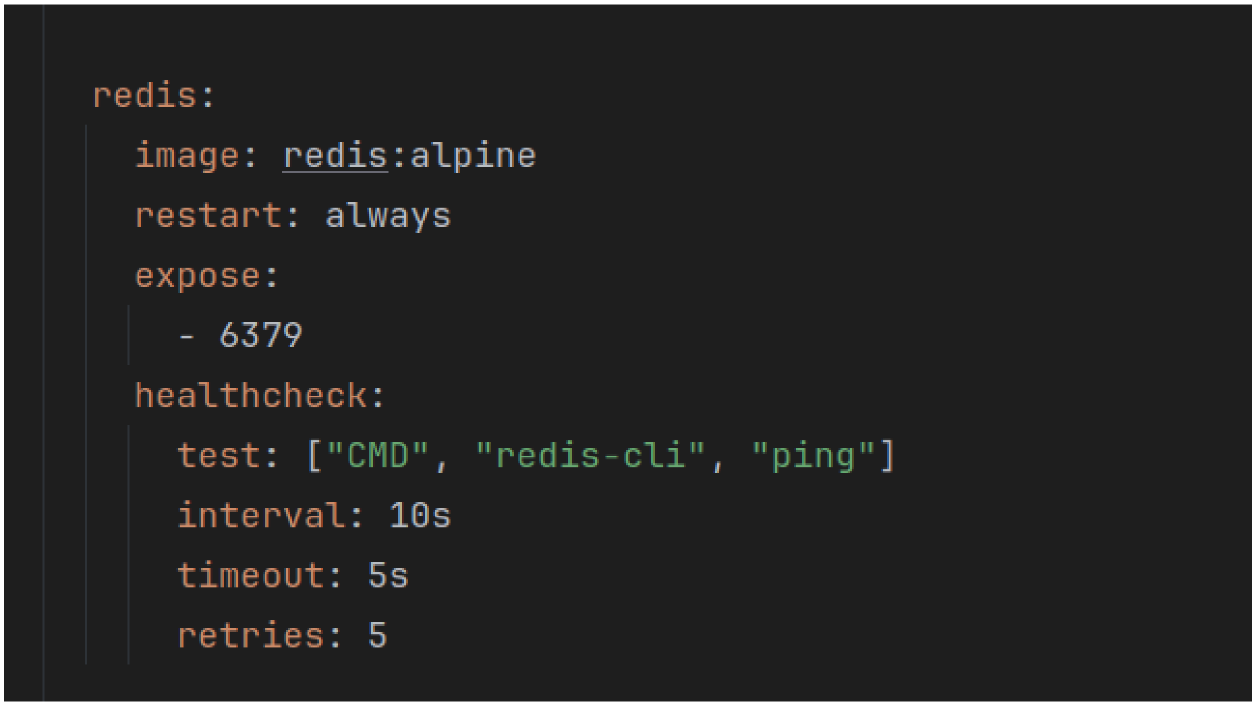
MySQL Service



```yaml
  mysql:
    image: mysql:8.0
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
      MYSQL_DATABASE: ${MYSQL_DATABASE}
    volumes:
      - mysql-data:/var/lib/mysql
    expose:
      - 3306
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "--silent"]
      interval: 10s
      timeout: 5s
      retries: 5
```

- **image**: Pulls the MySQL image version 8.0 from Docker Hub.
- **restart**: Restarts the container automatically if it crashes.
- **environment**: Environment variables are used to configure MySQL (root password and database name).
- **volumes**: Mounts a persistent storage volume (mysql-data) to /var/lib/mysql inside the container, ensuring that database data is preserved.
- **healthcheck**: Checks if MySQL is healthy by running the mysqladmin ping command

Redis Service



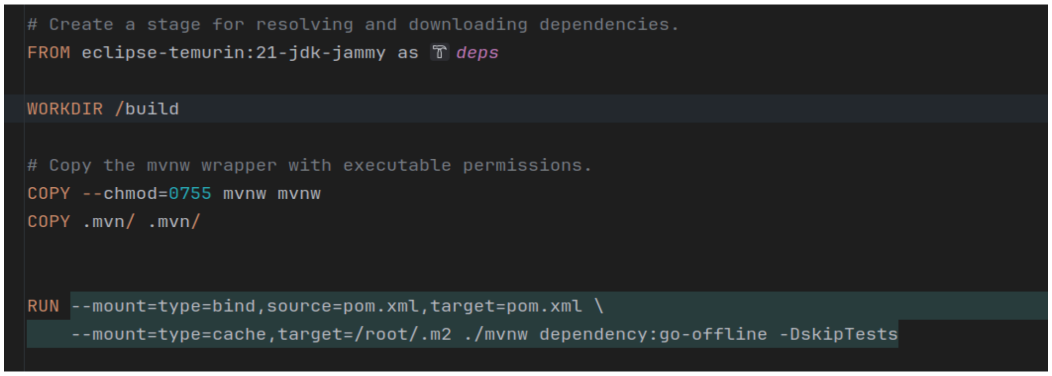
```
redis:
  image: redis:alpine
  restart: always
  expose:
    - 6379
  healthcheck:
    test: ["CMD", "redis-cli", "ping"]
    interval: 10s
    timeout: 5s
    retries: 5
```

- **image**: Uses the lightweight Alpine version of Redis.
- **expose**: Opens port 6379 inside the container, the default port for Redis.
- **healthcheck**: Uses the redis-cli ping command to check Redis' health.

Dockerfile Breakdown

The Dockerfile outlines how the application is built and optimized into a Docker image.

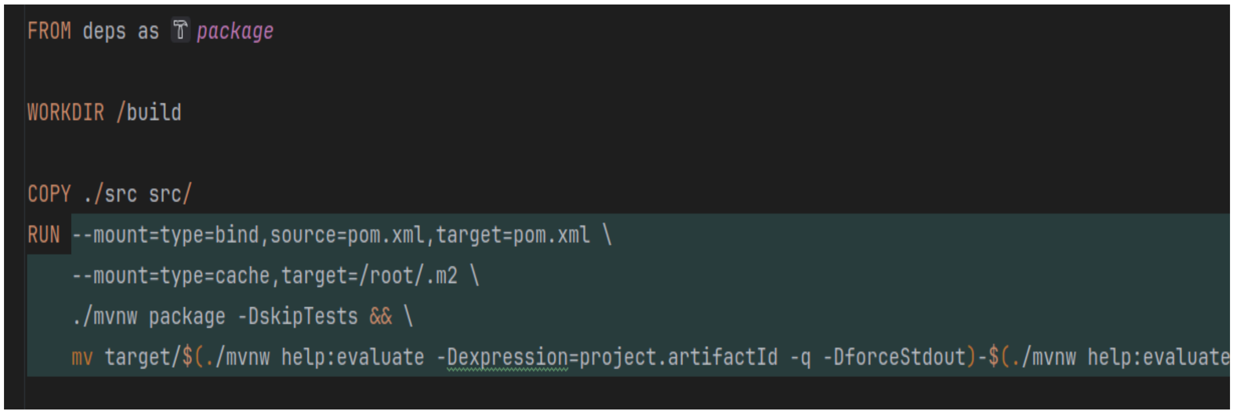
Resolving Dependencies Stage



```
# Create a stage for resolving and downloading dependencies.
FROM eclipse-temurin:21-jdk-jammy as deps

WORKDIR /build

# Copy the mvnw wrapper with executable permissions.
COPY --chmod=0755 mvnw mvnw
COPY .mvn/ .mvn/


RUN --mount=type=bind,source=pom.xml,target=pom.xml \
    --mount=type=cache,target=/root/.m2 ./mvnw dependency:go-offline -DskipTests
```

- **FROM eclipse-temurin:21-jdk-jammy**: Specifies the base image using Java Development Kit (JDK) 21 from Eclipse Temurin.
- **WORKDIR**: Sets the working directory inside the container to /build.
- **COPY**: Copies the Maven wrapper (mvnw) and Maven configurations to the container.
- **RUN**: Downloads all Maven dependencies offline to leverage Docker caching.

Building the Application



```
FROM deps as package

WORKDIR /build

COPY ./src src/
RUN --mount=type=bind,source=pom.xml,target=pom.xml \
    --mount=type=cache,target=/root/.m2 \
    ./mvnw package -DskipTests && \
    mv target/$(./mvnw help:evaluate -Dexpression=project.artifactId -q -DforceStdout)-$(./mvnw help:evaluate
```

- **COPY ./src src/**: Copies the source code to the container.
- **RUN**: Uses Maven to package the application into a JAR file.

6. Conclusion

Docker allows you to build, deploy, and manage applications in isolated environments with ease. By defining services in docker-compose.yml and creating efficient Dockerfiles, you can streamline development, ensure consistency, and optimize performance through caching and layered builds.