Cre8tive Devs Software

created by

Akshay Kamble

Index

Introduction to Docker :

Docker is comparable to a software magic box. Imagine having a customised box that can house any software as well as the tools and settings necessary for it to function. Regardless of the computer, whether your laptop, a friend's PC, or even a very powerful server, this box functions in the same way on all of them. You may easily share your software with others by putting it in this enchanted container with Docker. It helps ensure that your programme functions properly everywhere it goes while saving time.

Keep in mind that Docker's main goal is to make software extremely portable and usable across several systems. Your software can use it as a virtual container to ensure that everything it requires is always packed and ready to go.

Install Docker On Windows :

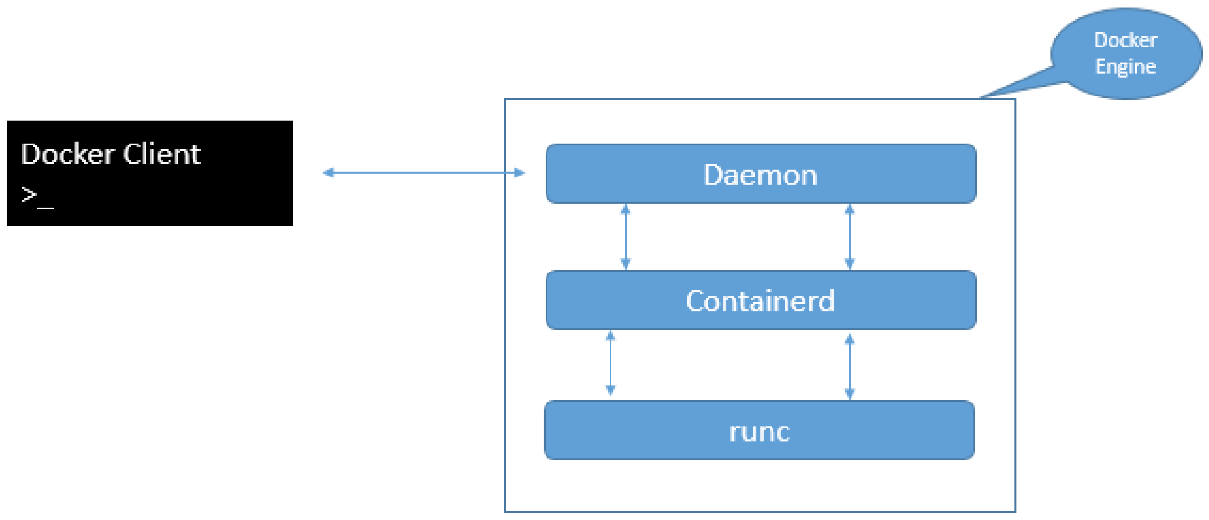
- Make sure your Windows version is 64-bit and supports virtualization technology by checking the system requirements.
- Get Docker Desktop here:

  Go to the Docker website (https://www.docker.com/products/docker-desktop) and select "Get Docker Desktop for Windows." Get the installer.
- Run the installation by finding and starting the downloaded installer file. obey the directions displayed on the screen.
- Activate Hyper-V: When requested, activate this virtualization technology that Docker needs, Hyper-V. Your computer might need to restart.
- Install Docker Desktop by selecting options like "Enable Windows Subsystem for Linux" during installation, if necessary. Docker Desktop will start up automatically after installation.
- Create a new account or sign in using your Docker Hub credentials (optional). Although optional, this step is advised.
- Docker Desktop Interface: Your system tray will display the Docker Desktop icon. To access Docker options, simply right-click.
- To confirm Docker is correctly installed, open a command prompt or PowerShell window and type the following command:

  ```
  docker –version
  ```
- Test Docker Run**:** Run a simple test container to ensure Docker is functioning:

  ```
  docker run hello-world
  ```

Docker Architecture:



- **Docker Client:**

  The Docker Client functions as an interface hub. It is used to interact with Docker and issue commands. When you need to do something with containers, it is your "go-to" resource. The Docker Client is provided commands like docker run, docker build, and docker stop.

- **Docker Engine:**

  The heart of Docker is the Docker Engine. Everything is controlled by it in the background. Your container fantasies will come true thanks to the cooperation of Containerd and the Docker Daemon.

- **Docker Daemon:**

  The worker bee is analogous to the Docker Daemon. It handles the labor-intensive lifting. The Docker Daemon executes your command when you say "docker run." Use the following command to launch the Docker Daemon:

  ```
  sudo systemctl start docker
  ```

- **Containerd:**

The Docker Daemon has a helper called Containerd. It takes care of the minute particulars involved in managing containers. It is comparable to the assistant checking to make sure everything is in order. You don't need to use direct commands to use Docker because Containerd handles it in the background.

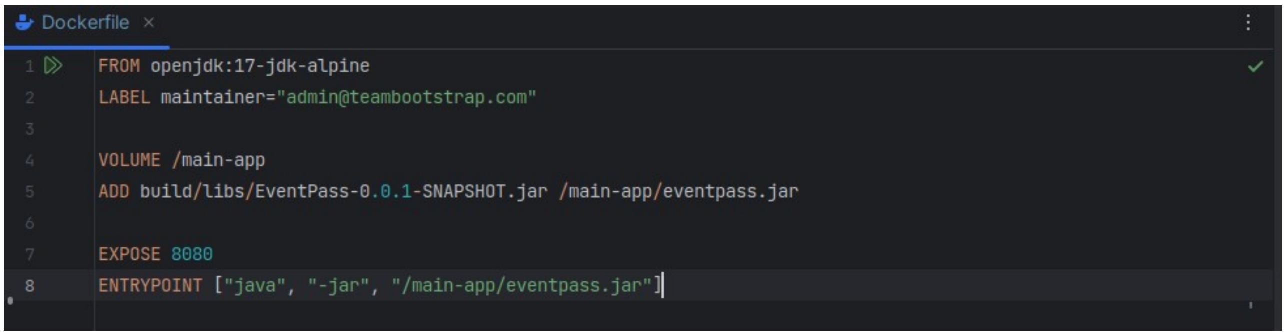
- **Runc:**

  Like a stage manager for a play, runc serves as the executor. Runc prepares and launches the real container when you say "docker run." Runc is handled by Docker, so you don't need to deal with it directly.

Converting Non-Docker Project to Docker Project :

We had a Spring Boot project named "EventPass" that we wanted to convert into a Docker application. To achieve this, we needed to create a Dockerfile in the main directory of our application, specifically within the "/EventPass" folder. This Dockerfile allowed us to transform our Spring Boot application into a Dockerized version.

DockerFile :

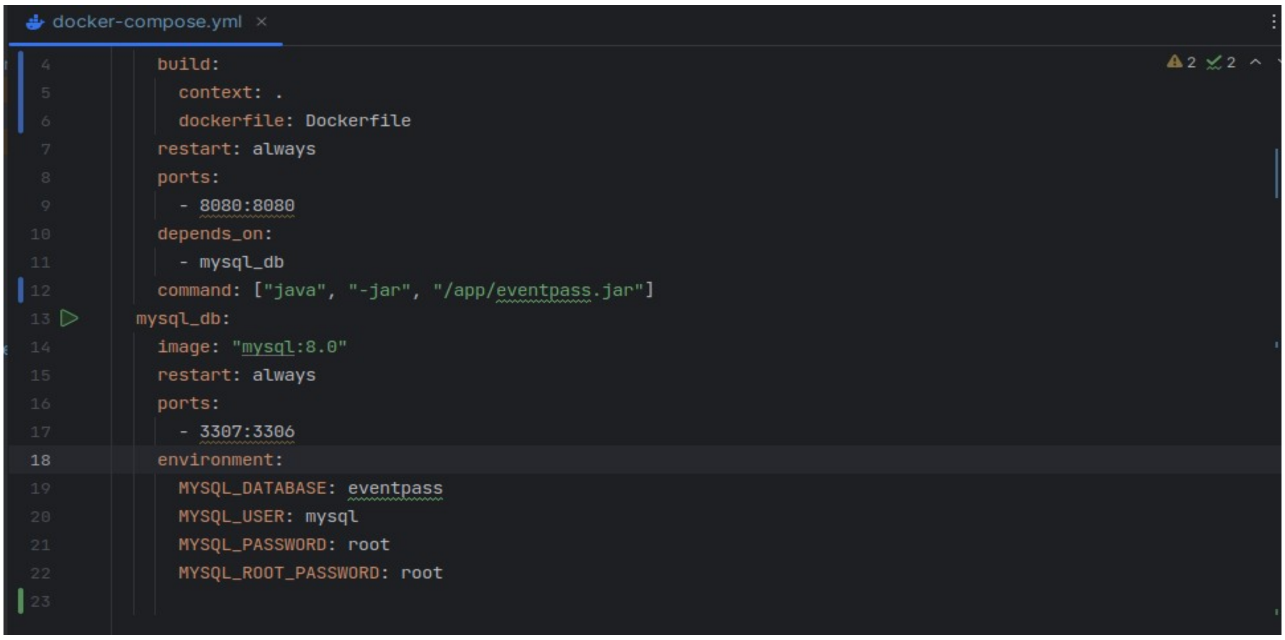


```
FROM openjdk:17-jdk-alpine
LABEL maintainer="admin@teambootstrap.com"

VOLUME /main-app
ADD build/libs/EventPass-0.0.1-SNAPSHOT.jar /main-app/eventpass.jar

EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/main-app/eventpass.jar"]
```

- FROM openjdk:17-jdk-alpine: Our decision to use a foundational image obtained from Docker Hub led to the choice of this directive. We successfully provided the required environment to ease the execution of Java applications by choosing an image that includes Java 17 and Alpine Linux. The requirement to guarantee compatibility and ideal runtime circumstances for our programme served as the primary impetus for this choice.

- maintainer LABEL: This line was added so that we could provide the email address of the person in charge of either generating or maintaining the Docker image. We can now incorporate important information regarding the image's creation and management, which will help with documentation and accountability.
- VOLUME /main-app: This command, which we presented, creates a "/main-app" data volume inside the container. Our goal was to make it possible for smooth data interchange between various containers as well as between the container and the host system. Volumes were used to guarantee that data could survive after the container did, promoting effective data management and sharing.
- ADD /main-app/eventpass.jar build/libs/EventPass-0.0.1-SNAPSHOT.jar: In order to complete this step, we added the Spring Boot JAR file that had been compiled (EventPass-0.0.1-SNAPSHOT.jar) to the container's "/main-app" directory from the local machine's build/libs directory. This step was conducted to get the container ready for our application's executable JAR file
- EXPOSE 8080: We indicated that the container would expose port 8080 in order to achieve this goal. Even though the port was not yet actively disclosed, it was noted that the programme inside the container will use it for communication.
- ENTRYPOINT ["java", "-jar", "/main-app/eventpass.jar"]: We configured the entry point command within the container for this reason. This command specified that the "java -jar" command will be used to launch the Java application when the container was started. It was possible to successfully launch our Spring Boot application because the container's path to the executable JAR file was given.

Docker-compose file :



docker-compose.yml

```
    build:
      context: .
      dockerfile: Dockerfile
    restart: always
    ports:
      - 8080:8080
    depends_on:
      - mysql_db
    command: ["java", "-jar", "/app/eventpass.jar"]
  mysql_db:
    image: "mysql:8.0"
    restart: always
    ports:
      - 3307:3306
    environment:
      MYSQL_DATABASE: eventpass
      MYSQL_USER: mysql
      MYSQL_PASSWORD: root
      MYSQL_ROOT_PASSWORD: root
```

We have our application "EventPass," which relies on a MySQL database. To ensure the smooth operation of both our application and the MySQL database, we require a Docker Compose file. This file serves as a crucial tool to facilitate the simultaneous execution and coordination of our "EventPass" application and the MySQL database. In essence, our need for a Docker Compose file arises from the necessity to effectively manage, deploy, and maintain our integrated application ecosystem. By leveraging Docker Compose, we ensure coherence, streamlined development, and effective coordination of both our "EventPass" application and the MySQL database, leading to a seamless and efficient operational environment.

To make use of the proper features and capabilities for our deployment, we have defined the version of the Docker Compose syntax as "3.7".

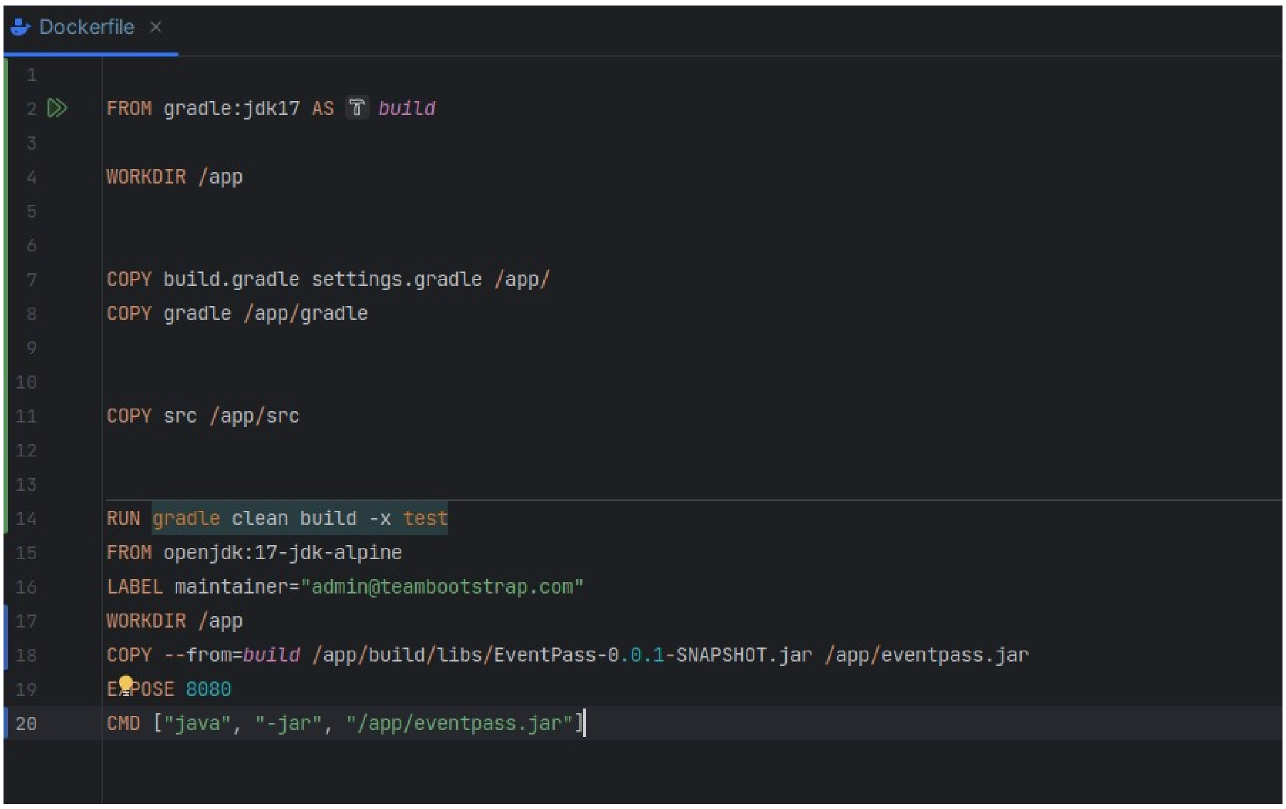
We define two different services under the "services" section. The "api_service" service is our Spring Boot programme, "EventPass." Using the current directory's context and the specified Dockerfile, we have told Docker to build this service. The service is configured to automatically resume if it abruptly stops in order to guarantee continuous availability. In order to facilitate external access, the service is mapped to communicate via port 8080 on the host system. It is dependent on the "mysql_db" service, indicating that it will not launch until the MySQL database is operational. Using the path to the application JAR file and the "java -jar" command, the provided command will run the Spring Boot application.

The official MySQL 8.0 Docker image is used by the "mysql_db" service. Like the "api_service," it is configured to restart automatically after quitting. External connection with the MySQL database is made possible via a mapping between port 3307 on the host machine and port 3306 inside the container. In order to set up the MySQL database settings, such as the database name, user credentials, and passwords, we have also defined environment variables.

Incorporating these configurations within our Docker Compose file enables the seamless deployment and coordination of our "EventPass" Spring Boot application and the MySQL database. This comprehensive setup ensures effective communication, synchronization, and stability between the components, resulting in a robust and functional application environment.

Enhancing Docker Projects for Improved Efficiency：

Updated dockerFile :


Dockerfile

```
FROM gradle:jdk17 AS build

WORKDIR /app


COPY build.gradle settings.gradle /app/
COPY gradle /app/gradle


COPY src /app/src


RUN gradle clean build -x test
FROM openjdk:17-jdk-alpine
LABEL maintainer="admin@teambootstrap.com"
WORKDIR /app
COPY --from=build /app/build/libs/EventPass-0.0.1-SNAPSHOT.jar /app/eventpass.jar
EXPOSE 8080
CMD ["java", "-jar", "/app/eventpass.jar"]
```

In our Dockerfile, we've implemented a series of optimizations to significantly enhance the efficiency of building our Docker image, leading to improved deployment and application performance. We've adopted a multi-stage build approach, leveraging specialized stages for distinct tasks. In the initial stage, we benefit from the "gradle:jdk17" base image, tailored for Gradle-based projects, ensuring a streamlined build process:

```
FROM gradle:jdk17 AS build
WORKDIR /app
COPY build.gradle settings.gradle /app/
COPY gradle /app/gradle
COPY src /app/src
RUN gradle clean build -x test
```

Optimizing file copying, we selectively copy only essential files required for building, reducing data transfer overhead:

```
FROM openjdk:17-jdk-alpine
LABEL maintainer="admin@teambootstrap.com"
WORKDIR /app

COPY --from=build /app/build/libs/EventPass-0.0.1-SNAPSHOT.jar /app/eventpass.jar
EXPOSE 8080
CMD ["java", "-jar", "/app/eventpass.jar"]
```

Selecting the lightweight Alpine version of OpenJDK 17 image further contributes to efficiency:

Moreover, caching Gradle dependencies during the build stage ensures faster builds and optimal resource utilization:

By consolidating multiple commands into a single RUN command within the build stage, we've streamlined the image creation process:

Finally, simplifying the entry point command directly initiates our Spring Boot application for a more straightforward startup process and seamless execution flow. In summary, these enhancements collectively contribute to a more efficient build process, smaller image size, and ultimately, faster deployment and improved performance for our "EventPass" application within its containerized environment.

Prameters in which we have made more efficent docker image :

1. **Multi-Stage Build:** We embraced a multi-stage build approach, leveraging distinct stages to streamline the image creation process. This optimization segregates build and runtime components, resulting in a final image with only necessary runtime dependencies.
2. **Focused File Copying:** We refined the copying of files by selectively including essential files needed for building the application. This approach reduces unnecessary data transfer and minimizes the image's overall size.
3. **Lightweight Base Image:** Our choice of the Alpine version of the OpenJDK 17 image as the base contributes to a smaller image footprint. This lightweight base image accelerates deployment and conserves storage resources.
4. **Caching Dependencies:** By caching Gradle dependencies during the build stage, we ensure that unchanged source code won't trigger redundant downloads. This significantly speeds up build times and resource utilization.
5. **Consolidated Commands:** We consolidated multiple commands into a single RUN command within the build stage, which simplifies the image's layer structure and enhances build efficiency.
6. **Simplified Entry Point:** Our streamlined entry point command initiates the Spring Boot application directly, simplifying the startup process and ensuring a smoother execution flow.