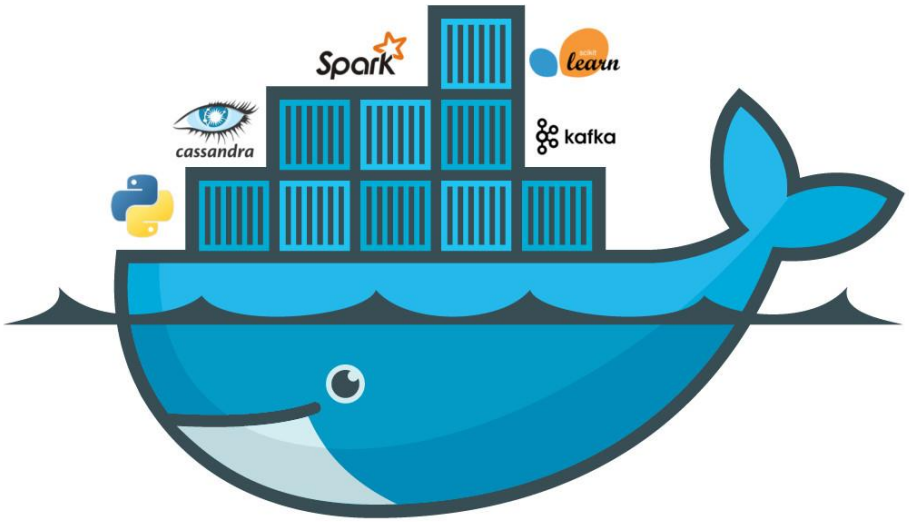


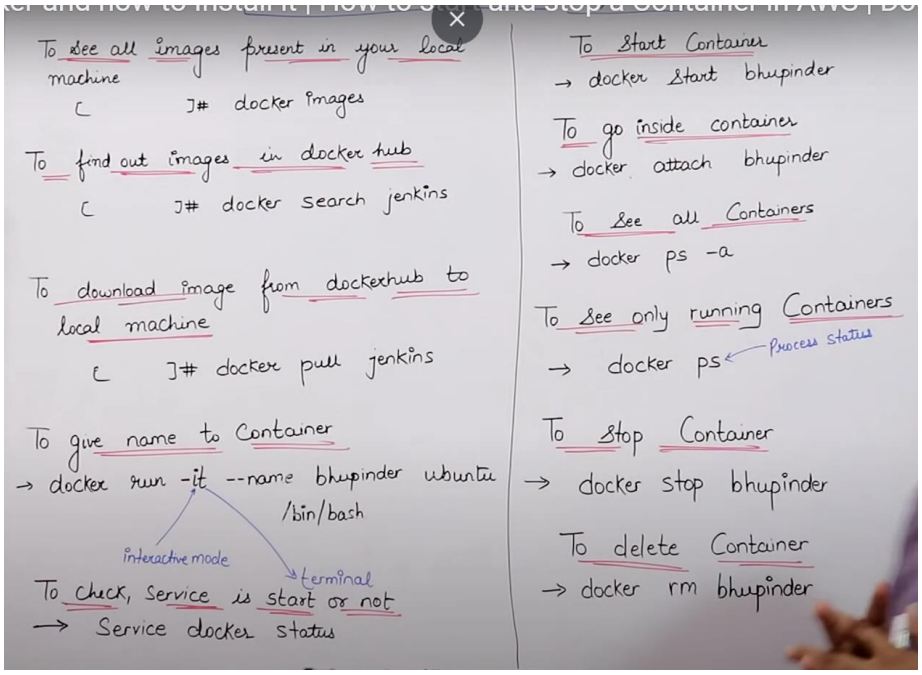
Docker Def: Docker is the open-source centralisd platform designed to create deploy and run the application

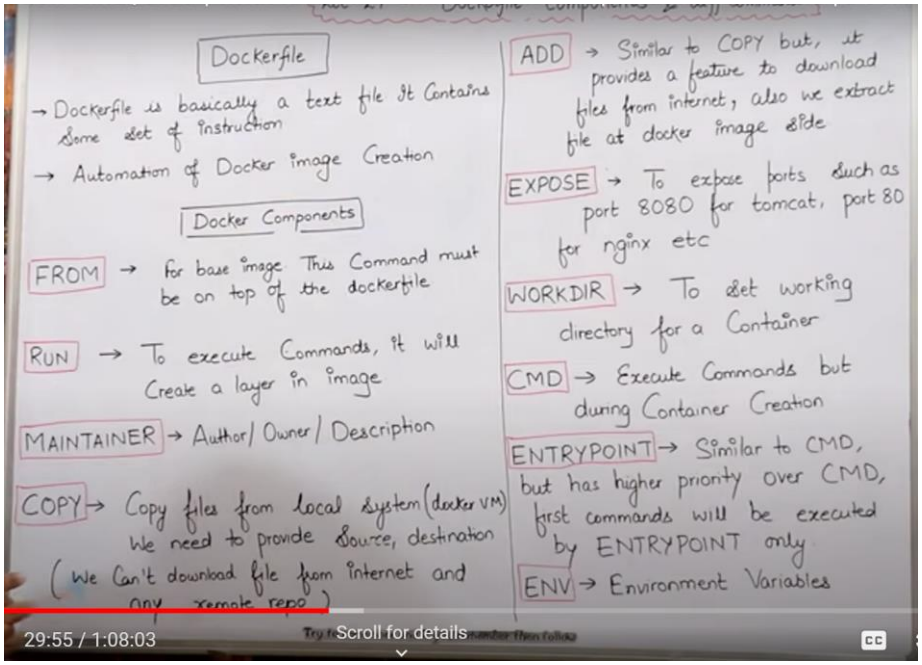
- Docker is written in "go" language
- Docker uses container on the host o.s to run applications it allows applications to use the same linux kernel as a system on the host computer rather than creating a whole virtual O.S
- We can install docker on any O.S but docker engine run natively on linux distribution.
- Docker is a tool that perform O.S level virtualization
- Docker was first release in march 2013 it is developed by Solomon hykes and sebastion rahl

Docker commands:

- **Docker images** to see the images
- **Docker search Jenkins** to find out images in docker hub
- **Docker pull Jenkins** download image from docker hub
- **Docker run -it –name Sohail ubuntu /bin/bash** to give name to container
- **Service docker start** to start the docker
- **Service docker stop** to stop docker
- **Service docker status** to check status of the docker
- docker rename (oldname)(new name)
- **docker run** `<image_name>`
- **docker -d** To start the Docker daemon.
- **docker info** To display Docker system-wide information.
- **docker version** To display the current installed Docker version.
- **docker login -u <username>** To login into Docker.
- **docker push <username>/<image_name>** To publish an image to Docker Hub.
- **docker commit <container_name>** To create a new image from a container's changes.
- **docker history <container_name>** docker history
- **docker rmi <container_name>** To remove one or more images.
- **docker tag** To tag an image into a repository.
- **docker run --name <container_name> <image_name>** To create and run a container from an image.

- **docker run -p <host_port>:<container_port> <image_name>** To run a container with port mapping.
- **docker rm <container_name>** To remove a stopped container.
- **docker ps**
- **docker ps -a** To list all docker containers.
- **docker volume create** To create a new Docker volume.





Dockerfile

1) → Create a file named Dockerfile (Capital)
2) → Add instructions in Dockerfile
3) → Build dockerfile to Create image
4) → Run image to Create Container

1. vi Dockerfile → i → :wq
2. FROM ubuntu
   RUN echo "Technical guftgu" >/tmp/testfile

To Create image out of dockerfile

```
docker build -t myimg .
```

(-t → tag)

```
docker ps -a
docker images
```

Creating Volume from Dockerfile

① Create a Dockerfile and Write :wq

```
FROM ubuntu
VOLUME ["/myvolume1"]
```

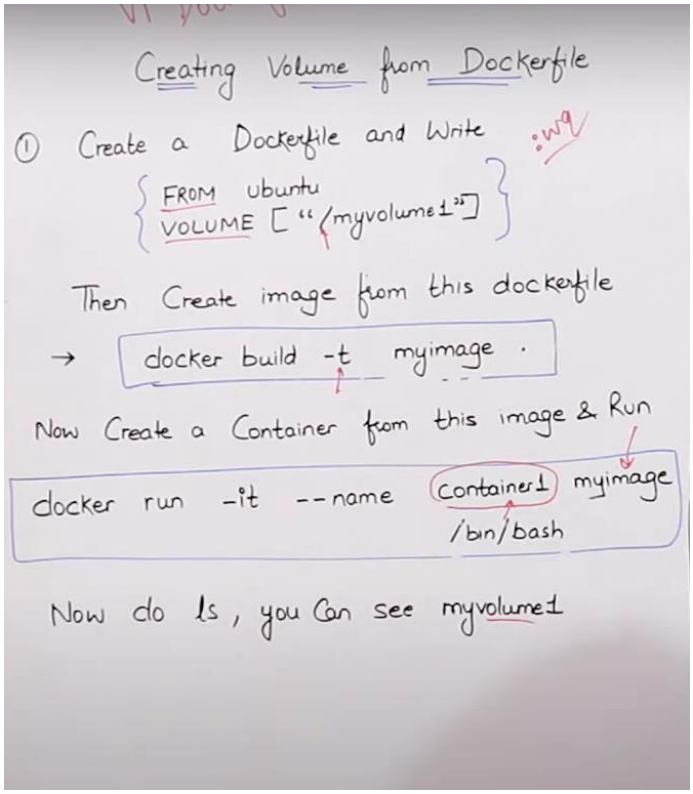
Then Create image from this dockerfile

→ `docker build -t myimage .`

Now Create a Container from this image & Run

`docker run -it --name container1 myimage /bin/bash`

Now do ls, you can see myvolume1



Docker Image: A Docker image is a file used to execute code in a Docker container .

Docker Container: A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application:

Deploying docker project using jenkins:

Install plugins

- CloudBees Docker Build and Publish plugin
- Docker API Plugin
- Docker Commons Plugin
- Docker Compose Build Step Plugin
- Docker Pipeline
- Docker plugin
- docker-build-step
- sonarscanner
- maven integration
- owsap dependency plugin
- jdk plugin

manage Jenkins configure the tools

Add Credentials for docker,git,sonarqube,

Create job

Select pipeline

Click ok

Pipeline select pipeline script

- pipeline {
- agent any
- tools{
- jdk 'jdk17'
- maven 'Maven'
- }
- environment{
- SCANNER_HOME=tool'Sonarqube'
- }

- stages {
- stage('GIT CHECKOUT') {
- steps {
    - git branch: 'main', url: 'https://github.com/GootySohail/Ekart.git'
- }
- }
- stage('COMPILE') {
- steps {
    - sh 'mvn compile'
- }
- }
- stage('SONAR ANALYSIS') {
- steps {
    - withSonarQubeEnv('Sonarqube') {
    - sh '"$SCANNER_HOME/bin/sonar-scanner \
        - -Dsonar.projectKey=Shopping-cart \
        - -Dsonar.sources=. \
        - -Dsonar.java.binaries=. \
        - -Dsonar.host.url=http://54.196.157.159:9000/ \
        - -Dsonar.login=squ_2e950989af73a77e9c419c4c80d179783091e080
- '

    - }
- }
- }
- stage('BUILD') {
- steps {
    - sh "mvn clean package -DskipTests=true"
- }
- }
- stage('DEPENDENCY CHECK') {
- steps {
    - dependencyCheck additionalArguments: '--scan target/', odcInstallation: 'DC'
    - dependencyCheckPublisher pattern: '**/dependency-check-report.xml'
- }

- }
- stage('BUILD DOCKER IMAGE') {
- steps {
    - script{

    - withDockerRegistry(credentialsId: 'b19f8c77-88fb-45a2-9a3c-bd2326b75cac', toolName: 'docker'){

        - sh "docker build -t shopping-cart -f docker/Dockerfile ."
        - sh "docker tag shopping-cart gootysohail/shopping-cart:latest"

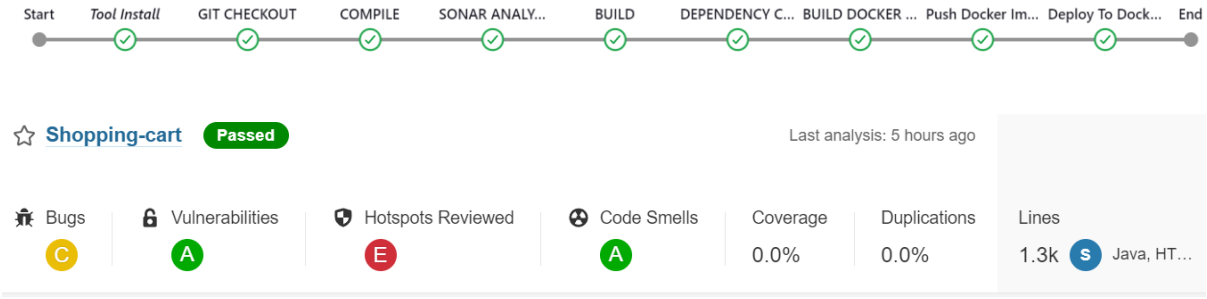
        - }
    - }
- }
- }
- stage('Push Docker Image') {
- steps {
    - script {
        - withDockerRegistry(credentialsId:'b19f8c77-88fb-45a2-9a3c-bd2326b75cac', toolName:'docker') {
        - sh "docker push gootysohail/shopping-cart:latest"
        - }
    - }
- }
- }

- stage('Deploy To Docker Container') {
- steps {
    - script {
        - withDockerRegistry(credentialsId:'b19f8c77-88fb-45a2-9a3c-bd2326b75cac', toolName:'docker') {
        - sh "docker run -d --name shopping -p 8070:8070 gootysohail/shopping-cart:latest"
        - }
    - }
- }
- }

- }
- }

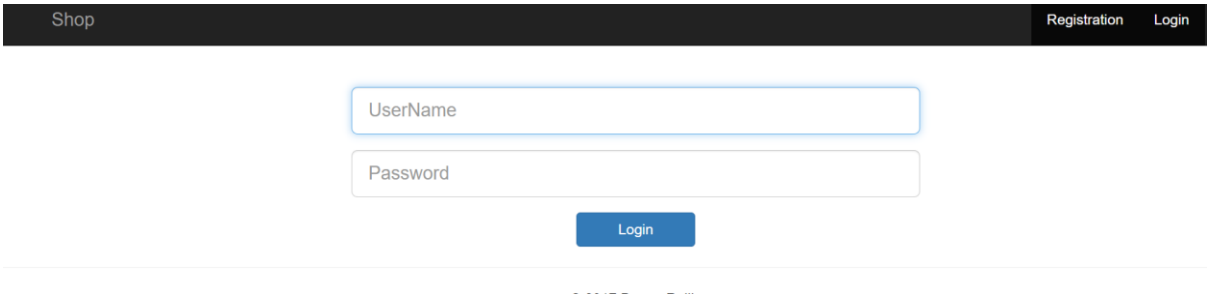
Click apply and save

Click build now

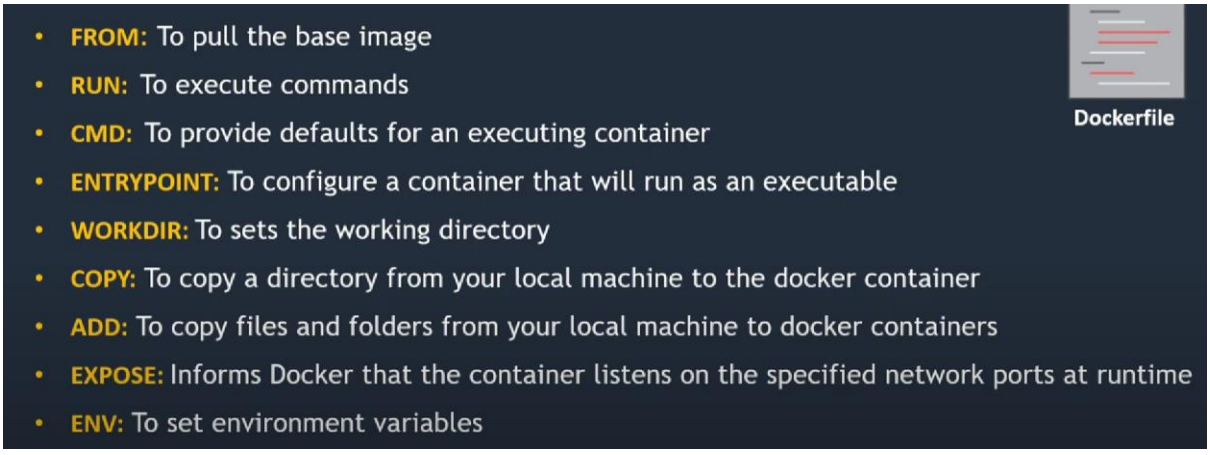


After build success

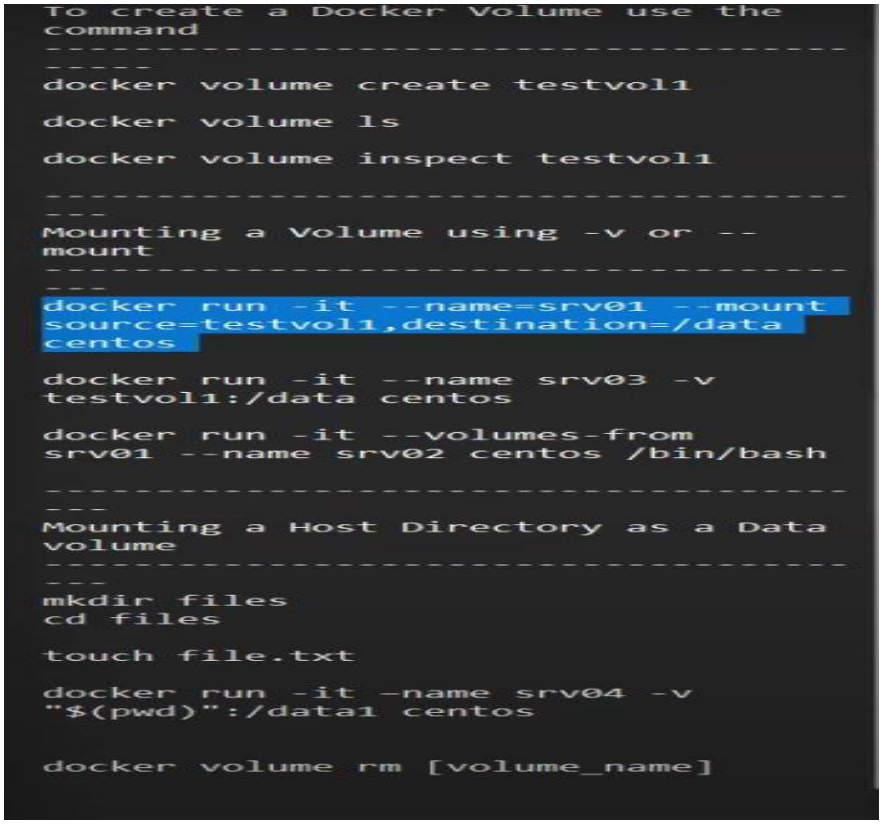
Copy the ip and browse in browser



Docker File




```
FROM centos:latest
RUN yum install java -y
RUN mkdir /opt/tomcat
WORKDIR /opt/tomcat
ADD https://dlcdn.apache.org/tomcat/tomcat-9/v9.0.54/bin/apache-tomcat-9.0.54.tar.gz
RUN tar -xvzf apache-tomcat-9.0.54.tar.gz
RUN mv apache-tomcat-9.0.54/* /opt/tomcat
COPY ./webapp.war /opt/tomcat/webapps
EXPOSE 8080
CMD ["/opt/tomcat/bin/catalina.sh", "run"]
~
```



Mulitstage Dockerfile

In simple words, a multi-stage build in Docker is a technique used to create smaller and more efficient Docker images. Instead of building a single Docker image with all the dependencies and tools needed for an application, a multi-stage build involves creating multiple stages in the Dockerfile, each serving a specific purpose.

The first stage, known as the "build" stage, is used to compile, build, and gather all the necessary files for the application. This stage might include installing dependencies, compiling code, and collecting artifacts.

The subsequent stages, often referred to as the "runtime" stages, take only the essential files from the build stage needed to run the application. These stages exclude unnecessary build tools and dependencies, resulting in a smaller and more streamlined final image.

The key advantage of multi-stage builds is that they help reduce the overall size of the Docker image, making it more efficient for distribution and deployment. This can lead to faster image downloads, quicker deployment times, and lower resource consumption.

DockerfileCopy code

 # Use an official Python runtime as a parent image

FROM python:3.8-slim

 # Set the working directory in the container

WORKDIR /app

 # Copy the current directory contents into the container at /app

COPY . /app

 # Install any needed packages specified in requirements.txt

RUN pip install --no-cache-dir -r requirements.txt

 # Run script.py when the container launches

CMD ["python", "script.py"]

In this example:

1. We start with the official Python image (**python:3.8-slim**), which is a lightweight version of Python.
2. We set the working directory inside the container to **/app**.
3. We copy the contents of the current directory (where your Dockerfile is located) into the container's **/app** directory.
4. If you have a **requirements.txt** file specifying Python dependencies, it is copied into the container, and the dependencies are installed using **pip**.
5. The final line specifies the command to run when the container starts. In this case, it runs a Python script named **script.py**. You should replace this with the actual name of your Python script.

Make sure to adjust the Dockerfile and other files (like **script.py** and **requirements.txt**) according to your project structure and dependencies. You can then build and run the Docker image using Docker commands.

```
# Stage 1: Build stage
FROM python:3.8-slim AS build-stage

# Set the working directory in the build stage
WORKDIR /app

# Copy only the requirements file to the build stage
COPY requirements.txt .

# Install dependencies in the build stage
RUN pip install --no-cache-dir -r requirements.txt

# Stage 2: Runtime stage
FROM python:3.8-slim AS runtime-stage
```

In this multi-stage Dockerfile:

1. The first stage (**build-stage**) is responsible for installing dependencies. It copies only the **requirements.txt** file and installs the dependencies. This helps reduce the size of the final image by excluding unnecessary files.
2. The second stage (**runtime-stage**) starts with a fresh Python image. It copies only the essential files needed to run the application from the **build-stage**. This includes the installed Python dependencies and the application code.
3. The final image is smaller because it only contains the necessary files, excluding the build tools and dependencies.

You can build and run this Docker image using standard Docker commands. Make sure to adapt the Dockerfile based on your project structure and requirements.

Set the working directory in the runtime stage

WORKDIR /app

Copy only the necessary files from the build stage

COPY --from=build-stage /usr/local/lib/python3.8/site-packages /usr/local/lib/python3.8/site-packages

COPY --from=build-stage /app .

Run script.py when the container launches

CMD ["python", "script.py"]

Project Multistage

- FROM ubuntu:latest
- # Set the working directory
- WORKDIR /app
- # Update the package repository and install necessary dependencies
- RUN apt-get update && apt-get install -y python3 python3-pip python3-venv
- # Copy the requirements file before creating the virtual environment
- COPY requirements.txt .
- # Debug: Check if the requirements file was copied correctly
- RUN ls -la /app && cat /app/requirements.txt
- # Create a virtual environment
- RUN python3 -m venv /app/venv
- # Install the dependencies in the virtual environment
- RUN /app/venv/bin/pip install --no-cache-dir -r requirements.txt
- # Copy the application files
- COPY app.py .
- COPY templates/ ./templates
- COPY static ./static
- # Expose the application port
- EXPOSE 5000
- # Set environment variables
- ENV FLASK_APP=app.py
- ENV FLASK_RUN_HOST=0.0.0.0
- # Run the application within the virtual environment
- CMD ["/app/venv/bin/flask", "run"]

https://github.com/GootySohail/python-calculator.git

- # Stage 1: Builder Stage
- FROM ubuntu:latest AS builder
- # Set the working directory
- WORKDIR /app
- # Update and install necessary dependencies
- RUN apt-get update && apt-get install -y python3 python3-pip python3-venv
- # Copy the requirements file
- COPY requirements.txt .

- # Debugging: List files and check the contents of requirements.txt
- RUN ls -la /app && cat /app/requirements.txt
- # Create a virtual environment
- RUN python3 -m venv /app/venv
- # Install dependencies in the virtual environment
- RUN /app/venv/bin/pip install --no-cache-dir -r requirements.txt
- # Copy the application files
- COPY app.py .
- COPY templates/ ./templates
- COPY static ./static
- # Stage 2: Final Stage
- FROM python:3.9-slim
- # Set the working directory
- WORKDIR /app
- # Copy the application files and virtual environment from the builder stage
- COPY --from=builder /app /app
- # Expose the application port
- EXPOSE 5000
- # Set environment variables
- ENV FLASK_APP=app.py
- ENV FLASK_RUN_HOST=0.0.0.0
- # Run the application using the virtual environment
- CMD ["/app/venv/bin/flask", "run"]

Distroless Image

- Distroless images are minimalistic images that only include the necessary components to run your application. Here's a multi-stage Dockerfile example that uses a Distroless image for the runtime stage:
- # Stage 1: Build stage
- FROM python:3.8-slim AS build-stage
- # Set the working directory in the build stage
- WORKDIR /app
- # Copy only the requirements file to the build stage
- COPY requirements.txt .
- # Install dependencies in the build stage
- RUN pip install --no-cache-dir -r requirements.txt
-
- # Stage 2: Runtime stage with Distroless image
- FROM gcr.io/distroless/python3:debug AS runtime-stage
- # Set the working directory in the runtime stage
- WORKDIR /app
- # Copy only the necessary files from the build stage

- COPY --from=build-stage /usr/local/lib/python3.8/site-packages /usr/local/lib/python3.8/site-packages
- COPY --from=build-stage /app .
- # Specify the entry point for the Distroless image
- ENTRYPOINT ["python", "script.py"]