Day-11 SRE Training

Topic: Docker

Docker is a containerization platform that allows developers to package applications and their dependencies into lightweight, portable containers. These containers ensure that applications run consistently across different environments.

What Does Docker Do?

- Encapsulates applications with all dependencies (libraries, configs, etc.).
- Ensures consistency across development, testing, and production.
- Reduces conflicts between different environments.
- Improves scalability and deployment speed.

What is a Docker Image?

- A **Docker Image** is a **read-only template** containing everything needed to run an application (OS, libraries, dependencies, and app code).
- It serves as a blueprint to create **containers**.
- Docker images are stored in a specific format known as the **Docker image format**, which consists of several layers and metadata.

What is a Docker Container?

- A **Docker Container** is a **running instance** of an image.
- It is an isolated environment that runs an application with all dependencies included.

Three Main Parts of Docker

1. **Docker CLI (Command-Line Interface)**
    - The **Docker CLI** allows users to interact with Docker using commands.
2. **Docker Daemon (dockerd)**
    - The Docker Daemon is the background service that manages containers, images, networks, and storage.

3. **Docker Registry**
    - A **Docker Registry** stores and distributes Docker images.

DockerFile:

```
Dockerfile > ...
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
ENV FLASK_APP=src/main.py
ENV FLASK_ENV=development
ENV PYTHONPATH=/app
EXPOSE 5000
CMD ["gunicorn","--bind","0.0.0.0:5000","src.main:app"]
```

1. **FROM python:3.9-slim**
    - This specifies the base image to use for the container. The `python:3.9-slim` is a lightweight image containing Python 3.9.
2. **WORKDIR /app**
    - Sets the working directory inside the container to `/app`. Any subsequent commands (like `COPY`, `RUN`) will execute relative to this directory.
3. **COPY requirements.txt .**
    - Copies the `requirements.txt` file from your local machine to the `/app` directory inside the container. This is used to install Python dependencies.
4. **RUN pip install --no-cache-dir -r requirements.txt**
    - Installs the Python dependencies listed in `requirements.txt` using `pip`. The `--no-cache-dir` flag ensures that pip doesn't cache the installation files, saving space.
5. **COPY . .**
    - Copies the entire content of your current directory (from your local machine) into the `/app` directory in the container. This would include your Python files, app code, etc.

6. **ENV FLASK_APP=src/main.py**
    - Sets an environment variable `FLASK_APP`, which Flask uses to know where to look for your app. Here, it's pointing to `src/main.py`.
7. **ENV FLASK_ENV=development**
    - Sets the `FLASK_ENV` environment variable to `development`, enabling debug mode and auto-reloading in Flask.
8. **ENV PYTHONPATH=/app**
    - Adds `/app` to the Python module search path. This allows Python to find and import files from this directory when running your application.
9. **EXPOSE 5000**
    - Exposes port 5000 for the container, allowing the Flask app to be accessible on this port when running inside the container.
10. **CMD ["gunicorn","--bind","0.0.0.0:5000","src.main:app"]**

- Specifies the default command to run when the container starts. It runs **Gunicorn** (a Python WSGI HTTP server) to serve the Flask app. It binds the server to `0.0.0.0:5000` and tells it to use the `src.main:app` entry point.

To build the image:

```
docker build -t python-docker-app .
```

This creates an image called python-docker-app.

```
docker images
```

`docker images` is a command that lists all the Docker images available locally on your machine.

After building the image, you start a container:

```
docker run -d -p 5000:5000 python-docker-app
```

`-d`: Runs in **detached mode** (background).

`-p 5000:5000`: Maps port **3000 (container)** to **3000 (host)**.

`python-docker-app`: The image name.

```
sudo docker tag python-docker-app:latest veena1700/python-docker-app:v1
```

- It creates a new reference (or alias) for the existing `python-docker-app:latest` image and tags it as `veena1700/python-docker-app:v1`.
- Tags allow you to version your Docker images, making it easy to identify different releases or versions of your application. For example, you can use tags like `v1`, `v2`, `latest`, or `stable` to mark specific versions of the image. This ensures you can deploy the correct version.
- Without tags, Docker assumes `latest` by default. If you build a new image without tagging, it will overwrite the `latest` tag. Tagging images ensures that previous versions aren't accidentally overwritten and can be kept for reference or rollback.

```
docker push veena1700/python-docker-app:v1
```

- This will upload the image to Docker Hub under your repository.
- Once the image is pushed, it becomes available on Docker Hub for others to pull and use.
- You can store the image on Docker Hub or other registries, so it's available for deployment on different systems or environments.

Docker_compose.yml

A `docker-compose.yml` file is a configuration file used by Docker Compose to define and manage multi-container applications. It specifies the services, networks, and volumes needed for an application, making it easy to deploy and manage all containers with a single command.

In simple terms, it's a way to automate running multiple Docker containers that work together.

With a single command (`docker-compose up`), you can start and stop all the containers defined in the `docker-compose.yml` file, making it easy to manage complex applications.

docker-compose.yml

```yaml
version: '3.8'
services:
  web:
    # build: .
    image: python-docker-app
    ports:
      - "5000:5000"
    volumes:
      - .:/app
    environment:
      - FLASK_APP=src/main.py
      - FLASK_ENV=development
    command: flask run --host=0.0.0.0
```

`version`: Specifies the Docker Compose file format version.

`services`: Defines the individual containers (services) in the application.

- Each service has properties such as:
  - `image`: The Docker image to use for the container.
  - `build`: Optionally, build the image from a Dockerfile.
  - `ports`: Define port mappings between the host and container.

```
docker-compose up --build
```

`docker-compose up --build` is a command used to build (if necessary) and start the services defined in the `docker-compose.yml` file.

- `--build`: Ensures that the Docker images for the services are built before starting them, even if the images already exist.
- `up`: Starts the services (containers) defined in the `docker-compose.yml` file, creating any necessary networks and volumes.