Docker Documentation

What is docker?

Basically docker is an open platform that provides a goto environment to run your code because of which you don't need to install anything in your actual machine.Its also used for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.

Why docker?

Fast, consistent delivery of your applications.

Developers create and test their work using Docker which provides a test environment. They fix any issues and test again until everything works perfectly. Once testing is done, they quickly update the customer's system by sending the fixed version through Docker.Docker makes the overall development process very easy and scalable.

Docker architecture:

Docker mainly use server-client architecture.

1. Docker Engine

The Docker Engine is the core of the Docker platform and consists of:

- **Docker Daemon**: It listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes.
- **REST API**: It Allows applications to interact with the Docker Daemon programmatically.
- **CLI (Command Line Interface)**: Enables users to execute Docker commands to manage images and containers.Its also called **Docker client**.

2. Images

Docker images are read-only templates that define the blueprint for creating containers.Basically it includes everything like files,binaries,os package,libraries. It’s read-only, meaning you can’t change it directly, but you can create new containers from it. It composed of many layers.

2.1 Image layers:

Image layers

Each layer in an image contains a set of filesystem changes - additions, deletions, or modifications. Let's look at a theoretical image:

1. The first layer adds basic commands and a package manager, such as apt.
2. The second layer installs a Python runtime and pip for dependency management.
3. The third layer copies in an application's specific requirements.txt file.
4. The fourth layer installs that application's specific dependencies.
5. The fifth layer copies in the actual source code of the application.

3. Containers

Containers are runnable instances of Docker images. They are lightweight, portable, and isolated from each other and the host system. Containers use the host's operating system kernel but have their own filesystem, networking, and process space.

4. Registries

Docker registries are storage locations for Docker images. Public registries like Docker Hub allow users to share and access images. So, its a centralized location to store and manage the container's image.

5. Docker Network

Docker provides a range of networking options for containers to communicate with each other and with external systems. Key networking modes include:

- **Bridge Network**: Default network for standalone containers.
- **Host Network**: Containers share the host's network namespace.
- **Overlay Network**: Used for communication between containers in a Docker Swarm.

6. Volumes

Volumes are used to persist data generated by containers. They allow containers to store and share data outside of their ephemeral lifecycle.

To dockerize a project:

First of you need to write a **Dockerfile:**
To build the image, you'll need to use a Dockerfile. A Dockerfile is simply a text-based file with

no file extension that contains a script of instructions. Docker uses this script to build a container image.

Here's an example of a simple docker file:

```
# Use the official Python image
FROM python:3.10-slim

# Set the working directory
WORKDIR /app

# Copy the project files
COPY . .

# Copy the requirements.txt file
COPY requirements.txt .

# Install dependencies from requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Expose the application port
EXPOSE 8000

# Run the application
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

In the above dockerfile

- **`FROM <image>` - this specifies the base image that the build will extend.**
- **`WORKDIR <path>` - this instruction specifies the "working directory" or the path in the image where files will be copied and commands will be executed.**
- **`COPY <host-path> <image-path>` - this instruction tells the builder to copy files**

  The first `.` refers to your local machine's current directory.

  The second `.` refers to the `/app` directory inside the

- **from the host and put them into the container image.**
- **`RUN <command>` - this instruction tells the builder to run the specified command.**
- **`ENV <name> <value>` - this instruction sets an environment variable that a running container will use.**
- **`EXPOSE <port-number>` - this instruction sets configuration on the image that indicates a port the image would like to expose.**

- **`USER <user-or-uid>` - this instruction sets the default user for all subsequent instructions.**
- **`CMD ["<command>", "<arg1>"]` - this instruction sets the default command a container using this image will run.**

Writing a **docker.yml** file:

The **docker-compose.yml** file is a configuration file used by **Docker Compose**, a tool that helps you define and manage multi-container applications. It allows you to set up and run multiple Docker containers with a single command, making it much easier to work with complex applications that involve more than one service (like a web server, database, and caching system).

Heres a docker-compose.yml:

```
services:
  web:
    build: .
    ports:
      - "8000:8000"
    volumes:
      - .:/app
```

Services: The **services** section defines all the containers (or services) that are part of your application.
Like **web** is the name of the service. It's an arbitrary name that you choose, and it represents a container that will run your web application.

Build: This tells Docker Compose to build the Docker image for the **`web`** service from a **Dockerfile** in the current directory (`.` refers to the current directory).

Volumes:
The `/app` directory is inside the Docker container. Since the **docker-compose.yml** file specifies that the **current directory** (`.`) on your host machine is mounted to `/app` inside the container, the contents of your **current directory** on your local machine will be reflected inside the container at the `/app` path.

To build,tag and publish an image:

docker image ls

This command will show the available images, its specific tag name and the size it needed. So, the tag mainly controls the version of a specific specific image according to the project requirements.

Volumes provide the ability to connect specific filesystem paths of the container back to the host machine. If you mount a directory in the container, changes in that directory are also seen on the host machine. If you mount that same directory across container restarts, you'd see the same files.