UNIT III – DEVOPS AND CI/CD CONCEPTS [9 hours]

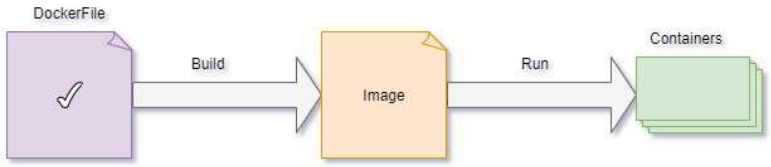
What is DevOps? Why is it used in industry?,CI/CD – Continuous Integration and Deployment, Introduction to GitHub Actions / Jenkins, Introduction to Docker and Dockerfile, Automating build and test for an app

---

DOCKER IMAGE

A **Docker Image** is a lightweight, standalone, and executable software package that includes everything needed to run an application: the code, a runtime, system tools, libraries, and settings.

The docker image includes the following to run a piece of software. A docker image is a platform-independent image that can be built in the Windows environment and it can be pushed to the docker hub and pulled by others with different OS environments like Linux.



If a Dockerfile is the **recipe**, then a Docker Image is the **perfectly packaged, ready-to-use meal** created from that recipe. This package is completely portable, meaning it will run the exact same way on your laptop, a teammate's computer, or a production server.

The Anatomy of a Docker Image

A Docker image isn't one large, monolithic file. It's cleverly constructed from a series of layers and stored in a registry.

Layers

An image is composed of a stack of **read-only layers**. Each instruction in a Dockerfile (like FROM, COPY, RUN) creates a new layer on top of the previous one.

- **Efficiency:** When you change an instruction in your Dockerfile, Docker only rebuilds that specific layer and the ones that follow it. This makes builds incredibly fast.
- **Sharing:** If multiple images share the same base layers (e.g., they all start with FROM ubuntu:22.04), those layers are only stored once on your system.

Base Image

This is the foundational layer of your image, specified by the FROM instruction in a Dockerfile. It can be a minimal operating system like alpine, a programming language runtime like python:3.9-slim, or an application like nginx.

Docker Registries

A **Docker Registry** is a storage and distribution system for Docker images. It's a library where you can pull official images, find images shared by the community, and push your own.

- **Docker Hub:** The default public registry and the largest library of container images.
- **Private Registries:** Companies often use private registries (like Amazon ECR, Google Artifact Registry, or Docker Hub private repos) to store their proprietary application images securely.

Commands Related to Docker Image Operations

The following are the some of the commands that are used for Docker Images:

Command	**Description**
docker image build	This command is used for building an image from the Dockerfile
docker image history	It is used for knowing the history of the docker image

docker image inspect	It is used for displaying the detailed information on one or more images
docker image prune	It used for removing unused images that are not associated with any containers
docker image save	This command helps in saving the docker images into a tar archived files
docker image tag	It helps in creating a tag to the target image that refers to the source image.

Docker Image Prune

Docker image prune is a command used in the docker host to remove the images that are not used or Docker image prune command is used to remove the unused docker images.

```
ubuntu@ip-172-31-17-89:~/dangling-image-example$ docker image prune
WARNING! This will remove all dangling images.
Are you sure you want to continue? [y/N] y
Total reclaimed space: 0B
```

All the unused images are also known as dangling images which are not associated with any containers

Docker Image Build

The Following is the command which is used to build the docker image.

docker build -t your_image_name:tag -f path/to/Dockerfile

- **Docker build:** Initiates the build process.
- **-t your_image_name:tag:** Gives the image you're creating a name and, if desired, a tag**.**

- **path/to/Dockerfile . : G**ives the location of the Dockerfile. Give the right path if it's not in the current directory. "(.) DOT" represents the current wordir.

Docker Image Tag

Docker tags are labels for container images, used to differentiate versions and variants of an image during development and deployment. Docker tags will help you identify the various versions of docker images and help distinguish between them. Docker image will help us to build continuous deployment very quickly

Docker Image Vs Docker Container

The following are the difference between Docker Image and Docker Container:

Docker image	**Docker container**
The Docker image is the Docker container's source code.	The Docker container is the running instance of the Docker image.
Dockerfile is a prerequisite to Docker Image.	Docker Image is a pre-requisite to Docker Container.
Docker images can be shared between users with the help of the Docker Registry.	Docker containers can't be shared between the users.
To make changes in the docker image we need to make changes in Dockerfile.	We can directly interact with the container and can make the changes required.

Structure Of Docker Image

The layers of software that make up a Docker image make it easier to configure the dependencies needed to execute the container.

- **Base Image:** The basic image will be the starting point for the majority of Dockerfiles, and it can be made from scratch.
- **Parent Image:** The parent image is the image that our image is based on. We can refer to the parent image in the Dockerfile using the **FROM** command, and each declaration after that affects the parent image.
- **Layers:** Docker images have numerous layers. To create a sequence of intermediary images, each layer is created on top of the one before it.

Create A Docker Image And Run It As Container

Follow the below steps to create a Docker Image and run a Container:

Step 1: Create a Dockerfile.

Step 2: Run the following command in the terminal and it will create a docker image of the application and download all the necessary dependencies needed for the application to run successfully.

docker build -t <name>:<tag>

This will start building the image.

DOCKERFILE

A Dockerfile is a simple text file that contains a sequence of instructions used to automate the process of building a Docker image. These instructions are command-line operations that define the base image, add files, install dependencies, and configure the container's runtime environment.

Common instructions include:

- FROM: Specifies the base image (e.g., FROM ubuntu:22.04 or FROM python:3.10-alpine).

- RUN: Executes commands in a new layer on top of the current image, for tasks like installing software or libraries (e.g., RUN apt-get update && apt-get install -y nginx).
- COPY: Copies files from the host machine into the container's filesystem.
- CMD: Provides defaults for an executing container, which can include the command to run the application (e.g., CMD ["nginx", "-g", "daemon off;"]).
- EXPOSE: Informs Docker that the container listens on the specified network ports at runtime.

For a detailed guide, refer to the official Docker Docs on writing a Dockerfile or resources like the H3ABioNet basic Docker guide (PDF).

Automating Build and Test for an App

Automation is a key benefit of using Docker, particularly in a DevOps workflow with Continuous Integration/Continuous Deployment (CI/CD) pipelines.

The Automated Workflow

1. Code Commit: Developers push code changes to a version control system like Git.
2. Trigger CI Pipeline: The commit automatically triggers a CI server (e.g., Jenkins, GitLab CI, GitHub Actions).
3. Build Image: The CI server uses the Dockerfile to build a new Docker image for the application. This ensures consistency and reproducibility across environments.
4. Automated Testing: After the image is built, automated unit, integration, and performance tests are run inside the containerized environment. This guarantees the application works as expected within its intended runtime environment.
5. Store Image: If tests pass, the validated Docker image is pushed to a container registry, such as Docker Hub or a private registry.
6. Deployment: The image is then available to be automatically or manually deployed to staging or production environments.

Key Benefits of Docker for Testing Automation

- Environment Consistency: Eliminates "it works on my machine" problems by ensuring the test environment is identical to the production environment.

- Isolation: Tests run in isolated containers, preventing interference between different test runs or applications.
- Speed and Efficiency: Containers spin up quickly, speeding up the testing cycle compared to traditional virtual machines.