16-Dec-2024

Internship Day - 101 Report:

INTRODUCTION DOCKER

What is Docker?

Docker is an open-source platform designed to automate the deployment, scaling, and management of applications. It uses containerization technology to package applications and their dependencies into a standardized unit called a container. This ensures that the application runs consistently, regardless of the environment in which it's deployed, such as on a developer's local machine, a testing server, or a production cloud server.

At its core, Docker enables developers to package their applications along with all necessary libraries, frameworks, and system tools into lightweight containers. These containers can run on any system that supports Docker, making it possible to eliminate the classic “it works on my machine” problem in software development.

What is a Docker Container?

A Docker container is a runnable instance of an image. Containers encapsulate the application and its dependencies, running in an isolated environment. Docker containers are lightweight because they share the host operating system's kernel but run independently of each other. Each container has its own filesystem, system resources, and process space, ensuring isolation and security.

Why Use Docker?

Docker is primarily used for:

1. **Consistency Across Environments:**
    - One of the major challenges in software development is ensuring that an application behaves consistently across different environments. This includes development, staging, and production environments. With Docker, you can package the application along with its entire ecosystem (such as libraries, system tools, and settings) into a container, ensuring that it runs the same way everywhere.
2. **Microservices Architecture:**
    - In modern application development, the microservices architecture is gaining popularity. Microservices break large, monolithic applications into smaller, independent services that communicate via APIs. Docker containers are ideal for running microservices because each service can be packaged and run in its

own isolated environment. This isolation allows developers to update and scale individual services without affecting the entire application.

3. **Streamlining Development and Testing:**
    - Docker is great for setting up development and testing environments. Developers can quickly spin up containers that mirror the production environment, making it easier to test applications in conditions identical to those in production. Docker Compose, an extension of Docker, allows developers to define multi-container applications and manage them easily during development and testing.
4. **Simplified Continuous Integration and Deployment (CI/CD):**
    - Docker integrates well with CI/CD pipelines, allowing teams to automate the building, testing, and deployment of applications. By using Docker images in the CI/CD pipeline, developers can build and test code in a container that is identical to the production environment. This minimizes deployment errors and improves the reliability of releases.
5. **Portability and Scalability:**
    - Containers built with Docker can run anywhere, on any machine that supports Docker. Whether you're running the container on your local machine, a virtual machine, or a cloud server, it will behave the same way. Docker also integrates with orchestration tools like Kubernetes to help scale applications efficiently.

Advantages of Docker

1. **Consistent Development and Production Environments:** Docker eliminates the problem of different environments by packaging everything the application needs to run into a container. Whether it's a developer's laptop, a testing server, or a production server in the cloud, the container ensures that the application behaves the same way. This consistency helps developers and operations teams avoid "environment drift," where minor differences between development and production environments cause unexpected errors.
2. **Efficient Resource Utilization:** Docker containers are more lightweight than traditional virtual machines (VMs). Instead of creating a separate OS for each container, Docker containers share the host machine's kernel. This leads to reduced overhead, allowing multiple containers to run on the same host without consuming significant additional resources. As a result, Docker can run more containers on a single physical machine than virtual machines.
3. **Faster Start-up and Deployment Times:** Docker containers start almost instantly compared to virtual machines, which need to boot an entire OS. This rapid startup speed makes Docker ideal for dynamic environments where applications need to scale quickly

in response to changing workloads. This also means that Docker containers are easier to deploy, reducing the time it takes to push changes from development to production.

4. **Isolation and Security:** Each Docker container runs in isolation, with its own filesystem, memory, CPU allocation, and network stack. This isolation ensures that issues in one container do not affect others. For example, if a container crashes or encounters a bug, it will not bring down the entire system or other containers. Docker also uses kernel-level security features, such as control groups (cgroups) and namespaces, to enhance isolation between containers.
5. **Simplified Application Deployment:** Docker simplifies the deployment of complex, multi-component applications. With tools like Docker Compose, you can define an entire multi-container application, including the application server, database, caching server, and load balancer, in a single configuration file (docker-compose.yml). This configuration can be shared across development, testing, and production environments, ensuring consistent deployment with minimal effort.
6. **Facilitates Microservices Architecture:** Docker is widely adopted in microservices-based architectures. Each microservice can run in its own container, which can be built, tested, and deployed independently of other services. This allows for greater flexibility in how applications are built, scaled, and maintained.
7. **Support for Multi-cloud and Hybrid Cloud Deployments:** Docker provides flexibility in terms of where applications can run. Containers are platform-agnostic and can run on any infrastructure that supports Docker, whether it's on a developer's machine, in a data center, or in any cloud provider's environment (AWS, Azure, Google Cloud, etc.). This makes it easier to adopt hybrid or multi-cloud strategies, where applications are deployed across multiple cloud providers or between on-premise and cloud environments.

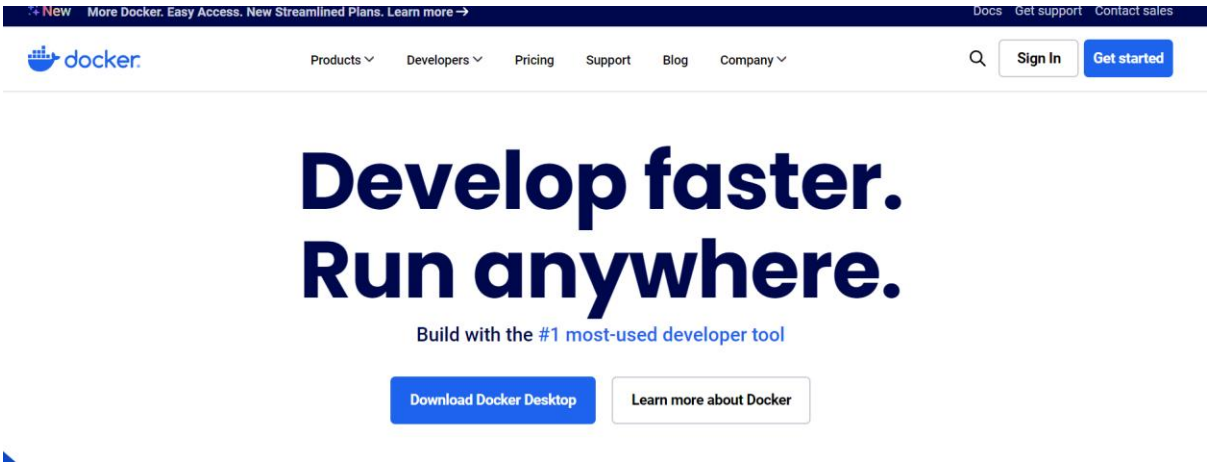
Key Docker Concepts

1. **Docker Images:**
    - A Docker image is a read-only template used to create containers. An image contains everything needed to run a container, including the operating system, application code, libraries, and dependencies. Images are built from Dockerfile instructions and can be stored in a Docker registry (such as Docker Hub) for sharing and reuse.
2. **Docker Containers:**
    - A Docker container is a runnable instance of a Docker image. Containers are lightweight and portable, providing isolation between applications and their environment. Multiple containers can be run on a single host, and they share the host OS’s kernel but have their own process and file namespace.

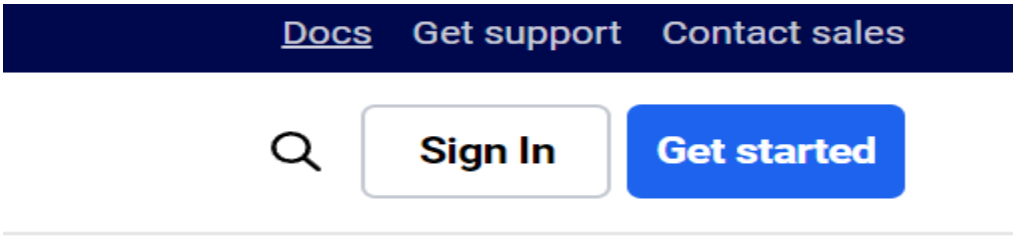
3. **Dockerfile:**
    - A Dockerfile is a text file that contains a series of instructions used to build a Docker image. Each instruction in a Dockerfile creates a new layer in the image. For example, you might use a FROM instruction to base your image on an existing image (like python:3.9-slim), and a COPY instruction to copy files from your local machine into the image.
4. **Docker Compose:**
    - Docker Compose is a tool that allows you to define and manage multi-container applications. Using a docker-compose.yml file, you can specify how to configure and run multiple services (like a web server, database, and caching layer) as a single application. This simplifies the orchestration of complex applications.

INSTALLATION OF DOCKER

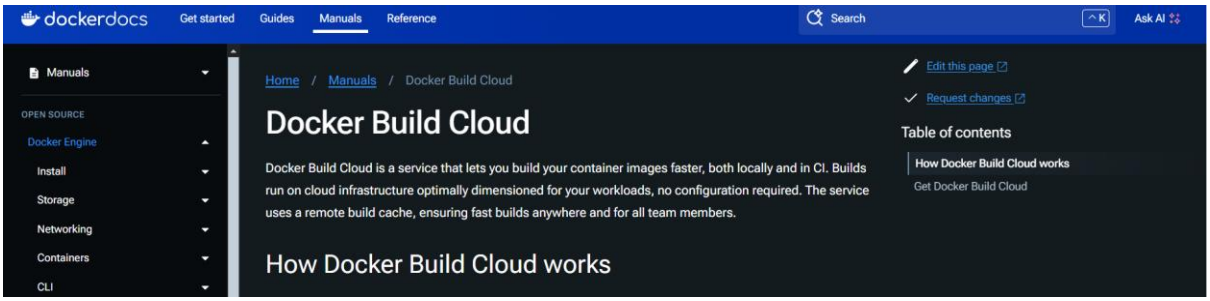
- Search docker.com on chrome & you get interface like that



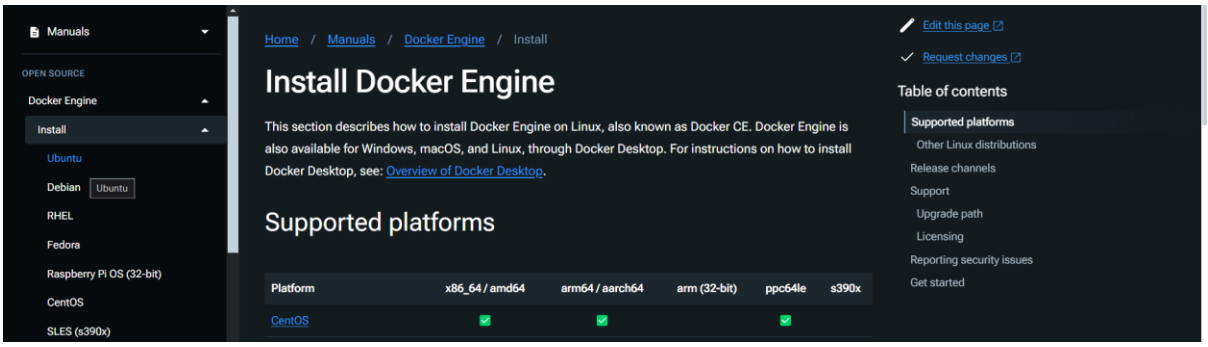
- Click on docs mention on the upper right of the interface



- Go to Docker Engine & click on install



- Click on Ubuntu

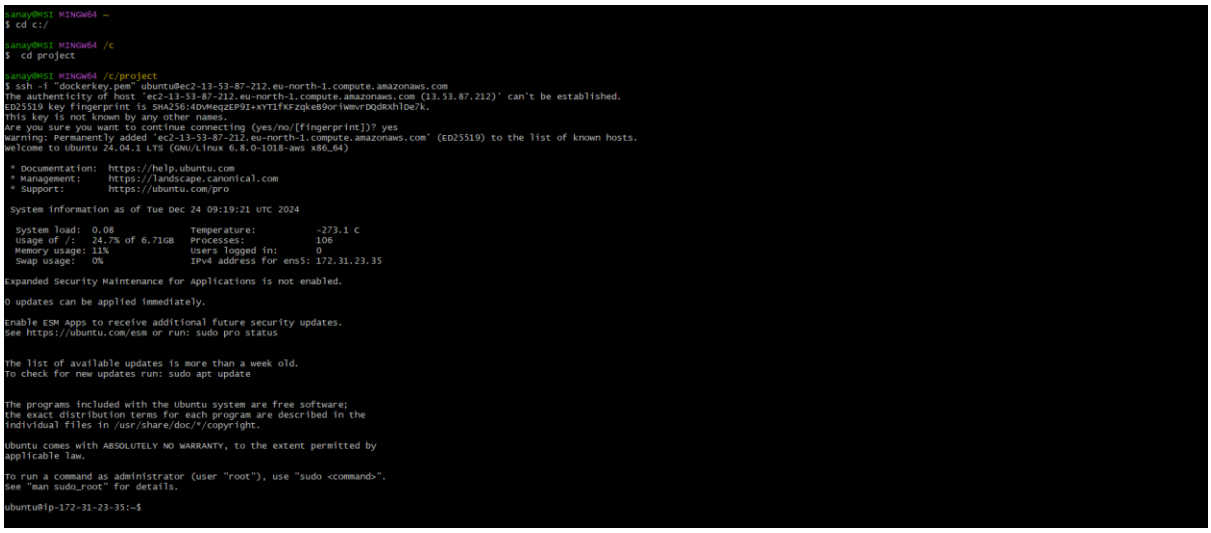


- Copy that setup a docker commands

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/dock
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] http
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

- Go to git bash and run the command Ubuntu



- Paste the docker installation command & installation start

- After that check docker install or not / check the version of it

```
root@ip-172-31-43-214:~# docker --version
Docker version 27.4.1, build b9d17ea
root@ip-172-31-43-214:~#
```

- Check Docker service status:

```
systemctl status docker
```

This checks the status of the Docker service to verify if it's running.

17-Dec-2024

Internship Day - 102 Report:

- List running containers:

```
docker ps
```

This lists all the Docker containers currently running on your machine.

- Run "hello-world" Docker container:

```
docker run hello-world
```

This runs a simple container that prints "Hello from Docker!" to check if Docker is working.

- List available Docker images:

```
docker images
```

This command lists all Docker images that are currently available on your system.

- Remove Docker images:

```
docker rmi <image-name>
```

This command deletes a Docker image from your system.

- Start a Docker container:

```
docker start <container-ID>
```

This command starts an existing Docker container by referencing its container ID.

- Stop a Docker container:

```
docker stop <container-ID>
```

This command stops a running container.

- Remove a stopped Docker container:

```
docker rm <container-ID>
```

This command removes a container that is no longer running.

MySQL Container Setup:

- Pull MySQL Docker image:

```
docker pull mysql
```

This downloads the latest MySQL image from Docker Hub.

- Run MySQL container:

```
docker run -d -p 3306:3306 --name mysql-container mysql
```

This runs a MySQL container in the background (-d), exposing port 3306 for MySQL.

- Connect to a running MySQL container:

```
docker exec -it <container-ID> mysql -u root -p
```

This allows you to execute commands inside the running MySQL container. It opens the MySQL prompt where you can log in as root.

- MySQL connection example (external):

```
mysql -h <host-IP> -P 3306 -u root -p
```

This connects to the MySQL server running in Docker from an external client using the host IP and port 3306.

Build a Docker Image:

- **Steps to create a custom Docker image:**

- Create a directory for your Docker image:

```
mkdir <folder-name>
cd <folder-name>
```

You create a directory for the project where your Docker image files (such as Dockerfile) will reside.

- Create a Dockerfile:

```
touch Dockerfile
vim Dockerfile
```

Create a file named Dockerfile (the blueprint of the Docker image), and open it in an editor like vim to add instructions.

- Writing the Dockerfile:

```
FROM ubuntu:latest
CMD ["echo", "Welcome to Docker"]
```

  - FROM ubuntu:latest: This tells Docker to base the image on the latest version of Ubuntu.
  - CMD ["echo", "Welcome to Docker"]: This sets the command that will run when the container starts. It will print "Welcome to Docker" to the console.

- Build the Docker image:

```
docker build -t <image-name>:v1 .
```

This command builds the Docker image using the instructions in the Dockerfile located in the current directory (.). The image is tagged with a name (<image-name>) and version (v1).

- Run the Docker container:

```
docker run <image-name>:v1
```

This runs the container from the Docker image you built. The output will display "Welcome to Docker," as defined in the CMD instruction of the Dockerfile.

18-Dec-2024

Internship Day - 103 Report:

SOME EXAMPLES:

Project 2

- **Create a directory for Project 2:**

```
mkdir Project2
cd Project2
```

Create a new directory named Project2 and navigate into it.

- **Create a Dockerfile:**

```
touch Dockerfile
vim Dockerfile
```

Create an empty Dockerfile using touch and then open it in an editor (vim) to add instructions.

- **Write the Dockerfile:**

```
FROM ubuntu:latest
ENTRYPOINT ["echo"]
```

FROM ubuntu:latest: This defines the base image for the container, which is the latest version of Ubuntu.

ENTRYPOINT ["echo"]: This sets the default executable to be echo. The ENTRYPOINT instruction ensures that the container runs echo when started.

- **Build the Docker image:**

```
docker build -t chandan:v2 .
```

This command builds a Docker image from the Dockerfile in the current directory (.), and tags it as chandan:v2.

- **Run the Docker container:**

```
docker run chandan:v2 Pankaj Sharma
```

This runs the Docker container based on the chandan:v2 image, passing Pankaj Sharma as an argument to the echo command.

- **List running containers:**

```
docker ps
```

This lists all currently running Docker containers.

- **List all containers (including stopped ones):**

```
docker ps -a
```

This lists all Docker containers, including those that are stopped.

19-Dec-2024

Internship Day - 104 Report:

Project 3

- **Create a directory for Project 3:**

```
mkdir Project3
cd Project3
```

Create a new directory named Project3 and navigate into it.

- **Create a Dockerfile:**

```
touch Dockerfile
vim Dockerfile
```

Again, create an empty Dockerfile and open it in a text editor to add content.

- **Write the Dockerfile:**

```
FROM ubuntu:latest
ENTRYPOINT ["echo"]
CMD ["welcome to Shimla"]
```

FROM ubuntu:latest: Use the latest Ubuntu as the base image.

ENTRYPOINT ["echo"]: Set echo as the command that will always be run when the container starts.

CMD ["welcome to Shimla"]: This provides default arguments to the echo command, so the container will print "welcome to Shimla" when run.

- **Build the Docker image:**

```
docker build -t sandeep:v3 .
```

This command builds a Docker image from the Dockerfile in the current directory (.), tagging it as sandeep:v3.

- **Run the Docker container:**

```
docker run sandeep:v3
```

This runs the Docker container based on the sandeep:v3 image. It will print "welcome to Shimla" since that is specified in the CMD instruction.

- **Run the container with arguments:**

```
docker run sandeep:v3 welcome to Haridwar
```

This runs the same Docker container but overrides the default message from CMD with "welcome to Haridwar". When using arguments, Docker replaces the CMD part but still runs the ENTRYPOINT command.

20-Dec-2024

Internship Day - 105 Report:

DOCKER COMPOSE

What is Docker Compose?

Docker Compose is a tool that simplifies the management of multi-container Docker applications. It allows developers to define and manage multi-container environments using a single configuration file, typically called docker-compose.yml. With Docker Compose, you can define the services that make up your application, specify their configurations, and manage how they interact with each other.

In essence, Docker Compose enables you to:

- **Define** multiple services in one place.
- **Configure** each service's environment, networking, and dependencies.
- **Orchestrate** starting, stopping, and scaling of services.
- **Run** multi-container Docker applications with a single command.

Key Components of Docker Compose

Docker Compose revolves around a few key components:

- **docker-compose.yml file**: This file defines the services, networks, and volumes required for a multi-container application. It specifies details such as the Docker images to use, the command to run, exposed ports, environment variables, volumes, and networks.
- **Services**: Each service represents a single container within the application. For example, a web service, a database service, and a cache service are all defined as separate services within the Compose file.
- **Volumes**: Volumes are used to persist data across container restarts. This is important for services like databases, where you want to ensure data is retained even if the container stops or is rebuilt.
- **Networks**: Networks allow services to communicate with each other. Docker Compose automatically creates a default network for services, but you can also define custom networks for more complex setups.

Why Use Docker Compose?

- **Simplified Multi-Container Application Management**

Docker Compose makes it easier to define and manage multiple containers. Instead of manually starting each container with complex docker run commands, you can simply define everything in a docker-compose.yml file and start the entire system with a single docker-compose up command.

- **Declarative Infrastructure**

With Docker Compose, the configuration for your application is stored in code (in the YAML file). This ensures consistency and allows teams to manage their infrastructure declaratively. It provides a clear record of how services should be configured and run.

- **Service Isolation and Networking**

Docker Compose enables seamless service isolation. Each service runs in its own container, but Docker Compose automatically connects them via internal networking. Services can communicate with each other by referring to their service name (e.g., a web service can connect to a database by referring to the "db" service). This eliminates the need to manage IP addresses or manual port forwarding.

- **Scalability**

Docker Compose allows you to scale services easily. For example, if you want to scale the number of web containers in a system, you can simply use docker-compose up --scale web=3. This command will launch three instances of the "web" service, distributing the workload.

- **Environment-Specific Configurations**

Docker Compose supports different environments, such as development, testing, and production. Using features like .env files and service profiles, you can customize the configuration of your services depending on the environment. For example, in a development environment, you may want to expose debugging ports or run additional services that are not needed in production.