Introduction to Docker

Made with Gamma

- Docker is an open platform for developing, shipping, and running applications.
- Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.
- With Docker, you can manage your infrastructure in the same ways you manage your applications.
- By taking advantage of Docker's methodologies for shipping, testing, and deploying code, you can significantly reduce the delay between writing code and running it in production.

The Docker platform

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security lets you run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you don't need to rely on what's installed on the host. You can share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.

Made with Gamma

Docker provides tooling and a platform to manage the lifecycle of your containers:

- Develop your application and its supporting components using containers.
- The container becomes the unit for distributing and testing your application.
- When you're ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

Understanding Virtualization and Containerization

Virtualization:

Virtualization is the process of creating a virtual version of something, such as a server, a storage device, a network, or even an entire operating system where the framework divides the resource into one or more execution environments.

Benefits of Virtualization:

Resource Efficiency: Better utilization of hardware resources.

- **Isolation:** Each VM is isolated from others, reducing security risks.
- **Scalability:** Easier to scale up and down based on demand.
- **Flexibility:** Supports multiple operating systems on the same physical hardware.

Made with Gamma

Containerization

Containerization involves encapsulating an application and its dependencies into a container that can run on any computing environment. Containers share the host system's OS kernel and isolate the application processes from other containerized applications.

- **Docker:** One of the most popular containerization platforms. It automates the deployment of applications inside lightweight containers.
- **Kubernetes:** An open-source platform for managing containerized applications across multiple hosts, providing basic mechanisms for deployment, maintenance, and scaling of applications.

Benefits of Containerization:

- **Lightweight:** Containers share the host OS kernel, making them more lightweight than VMs.
- **Portability:** Containers can run on any system that supports the container runtime environment, providing greater portability.
- **Consistency:** Ensures that applications run the same regardless of where they are deployed.
- **Fast Boot-Up:** Containers can start up quickly because they do not require booting up an entire OS.

Made with Gamma

Key Differences

1. **Architecture:**
   - **Virtualization:** Uses a hypervisor to create multiple VMs, each with its own OS.
   - **Containerization:** Runs containers that share the host OS kernel but isolate the application processes.
2. **Overhead:**
   - **Virtualization:** Higher overhead due to multiple OS instances.
   - **Containerization:** Lower overhead because containers share the host OS.
3. **Performance:**
   - **Virtualization:** Generally slower due to the additional layer of the hypervisor and multiple OS.
   - **Containerization:** Typically faster and more efficient due to shared OS and lightweight nature.
4. **Resource Utilization:**
   - **Virtualization:** Less efficient as it requires more resources to run multiple OS.
   - **Containerization:** More efficient as containers share the same OS and only encapsulate the application and its dependencies.

Made with Gamma

5. **Use Cases:**

- **Virtualization:** Suitable for running multiple different OS on the same hardware, isolated environments for development and testing.
- **Containerization:** Ideal for microservices architecture, continuous integration/continuous deployment (CI/CD) pipelines, and ensuring application portability across different environments.

Made with Gamma

Docker architecture

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.

Made with Gamma

The Docker daemon

The Docker daemon (`dockerd`) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker client

The Docker client (`docker`) is the primary way that many Docker users interact with Docker. When you use commands such as `docker run`, the client sends these commands to `dockerd`, which carries them out. The `docker` command uses the Docker API. The Docker client can communicate with more than one daemon.

Docker Desktop

Docker Desktop is an easy-to-install application for your Mac, Windows or Linux environment that enables you to build and share containerized applications and microservices. Docker Desktop includes the Docker daemon (`dockerd`), the Docker client (`docker`), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper.

Made with Gamma

Docker registries

A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker looks for images on Docker Hub by default. You can even run your own private registry.

When you use the `docker pull` or `docker run` commands, Docker pulls the required images from your configured registry. When you use the `docker push` command, Docker pushes your image to your configured registry.

Made with Gamma

Docker objects

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects

Images

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the `ubuntu` image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

Made with Gamma

Containers

Made with Gamma

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that aren't stored in persistent storage disappear.

Made with Gamma

What can I use Docker for?

Fast, consistent delivery of your applications

Docker streamlines the development lifecycle by allowing developers to work in standardized environments using local containers which provide your applications and services. Containers are great for continuous integration and continuous delivery (CI/CD) workflows.

Consider the following example scenario:

- Your developers write code locally and share their work with their colleagues using Docker containers.
- They use Docker to push their applications into a test environment and run automated and manual tests.
- When developers find bugs, they can fix them in the development environment and redeploy them to the test environment for testing and validation.
- When testing is complete, getting the fix to the customer is as simple as pushing the updated image to the production environment.

Lets do overview

Made with Gamma

Practical Oriented

Dockerfile

Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.

Dockerfile Syntax

FROM: Specifies the base image for the Docker image.

FROM image_name:tag

Example

FROM ubuntu:20.04

Made with Gamma

WORKDIR: Sets the working directory for subsequent instructions.

```
WORKDIR /path/to/directory

# Example

WORKDIR /app
```

COPY: Copies files or directories from the build context to the container.

```
COPY host_source_path container_destination_path

# Example

COPY . .
```

Made with Gamma

RUN: Executes commands in the shell.

```
RUN command1  && command2

# Example

RUN apt-get update  && apt-get install -y curl
```

ENV: Sets environment variables in the image.

```
ENV KEY=VALUE

# Example

ENV NODE_VERSION=14
```

Made with Gamma

EXPOSE: Informs Docker that the container listens on specified network ports at runtime.

```
EXPOSE port

# Example

EXPOSE 8080
```

CMD: Provides default commands or parameters for an executing container.

```
CMD ["executable","param1","param2"]

# Example

CMD ["npm", "start"]
```

Made with Gamma

ENTRYPOINT: Configures a container that will run as an executable.

```
ENTRYPOINT ["executable","param1","param2"]

# Example

ENTRYPOINT ["node", "app.js"]
```

ARG: Defines variables that users can pass at build-time to the builder with the docker build command.

```
ARG VARIABLE_NAME =default_value

# Example

ARG VERSION=latest
```

Made with Gamma

VOLUME: Creates a mount point for external volumes or other containers.

```
VOLUME /path/to/volume

# Example

VOLUME /data
```

LABEL: Adds metadata to an image in the form of key-value pairs.

```
LABEL key="value"

# Example

LABEL version="1.0" maintainer="Adrian"
```

Made with Gamma

USER: Specifies the username or UID to use when running the image.

```
USER user_name

# Example

USER app
```

ADD: Copies files or directories and can extract tarballs in the process.

```
ADD source_path destination_path

# Example

ADD ./app.tar.gz /app
```

Made with Gamma

Dockerfile Example

vi Dockerfile

```
FORM ubutnu:latest                    ( #To pull the image whatever you want to use)

RUN  /project                         (# To create directory)

WORKDIR  /project                     (# To setting working directory)

RUN apt-get  update -y                (# Before install the packages we need to update the OS)

RUN apt-get install default-jdk -y

RUN apt-get install mysql-server -y   (# This is how install the packages)

RUN apt-get install git -y

COPY .  /project

ADD <here provide url> .              (# Search for tomcat apache2 download→official website→copy tar.gz)
```

Made with Gamma

Commands to work with Dockcefile

→ docker build -t <your _image_name> . (# "-t" means we are tagging the image)

→ docker images (# For images list)

→ docker inspect <image-name or container id> (#To get detailed information of objects)

How to push images to the hub.docker.com

→ docker tag <image_name> <username/image_name> (#Before push the image need to tag it)

Login to the docker (# Here need to provide hub.docker.com credentials)

→ docker push <image_name> <username/image_name> (#if you want to see pushed images login your dockr)

How to deploy docker images inside the container

docker run -it —name <image_name> /bin/bash (# '-it' means interactive mode)

docker run -it —name <image_name> 8081:80 <image_name>

Made with Gamma

Dockerfile Example

vi Dockerfile

FROM node:20-alpine

WORKDIR /app # For hands on experience

COPY package*.json ./

RUN npm install COPY . . EXPOSE ENV 8080

Expose port 8080 to the outside world

NODE_ENV=production

CMD node app.js

Made with Gamma

Dockerfile Example

vi Dockerfile

```
FROM ubuntu:latest

RUN apt-get update -y

RUN mkdir  /work_dir                    # For hands on experience

WORKDIR /work_dir

RUN apt-get install apache2 -y

COPY index.html   /var/www/html

RUN service apache2 start

EXPOSE 80
```

Let's do overview

Made with Gamma

Docker Image

Docker images are **read-only templates that contain instructions for creating a container**. A Docker image is a snapshot or blueprint of the libraries and dependencies required inside a container for an application to run.

Made with Gamma

1. **Build an image from a Dockerfile**

#Example

docker -t image_name path_to_dockerfile build myapp .

2. **List all local images**

#Example

docker images

3. **Pull an image from Docker Hub**

#Example

docker pull image_name:tag

Made with Gamma

4. Remove a local image

#Example

```
docker rmi [image_name/image_id]
```

5. Inspect details of an image

#Example

```
docker image  inspect image_name/image_id
```

6. Prune unused images

```
docker image prune
```

7. Delete/remove all images at a time

```
docker rmi -f $(docker images -q)
```

Let's do overview

Made with Gamma

Container Lifecycle

In the context of Docker, “states” generally refer to the different states that a container can be in during its lifecycle. Docker containers can be in various states at different times, depending on their current status and the actions performed on them

1. Run a named container from an image

#Example

docker run --name container_name image_name:tag

2. List all running containers

docker ps

3. List all containers (including stopped ones)

docker ps -a

Made with Gamma

4. Stop a running container

docker stop container_name_or_id

5. Start a stopped container

docker start container_name_or_id

6. Remove a stopped container

docker rm container_name_or_id

7. Remove a running container (forcefully)

docker rm -f container_name_or_id

8. Inspect details of a container

docker inspect container_name_or_id

Made with Gamma

9. View container logs

```
docker logs container_name_or_id
```

10. Pause a running container

```
docker pause container_name_or_id
```

11. Unpause a paused container

```
docker unpause container_name_or_id
```

12. Stop all containers at a time

```
docker stop $( docker ps -a -q )
```

13. Delete all containers at a time

```
docker rm $( docker ps -a -q )
```

Let's do overview

Made with Gamma

Docker Volumes

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While **bind mounts** are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:

- Volumes are easier to back up or migrate than bind mounts.
- You can manage volumes using Docker CLI commands or the Docker API.
- Volumes work on both Linux and Windows containers.
- Volumes can be more safely shared among multiple containers.
- Volume drivers let you store volumes on remote hosts or cloud providers, encrypt the contents of volumes, or add other functionality.
- New volumes can have their content pre-populated by a container.
- Volumes on Docker Desktop have much higher performance than bind mounts from Mac and Windows hosts.

Made with Gamma

Docker Volume Architecture

Made with Gamma

Types of Docker Volumes

1. Anonymous Volumes
2. Named Volumes
3. Host Volumes (Bind Mounts)

1. Anonymous Volumes

Anonymous volumes are similar to named volumes, but they do not have a specific name assigned to them. Docker assigns them a unique identifier. These are typically used for temporary storage where the volume's content does not need to be retained once the container is deleted.

#Example

```
docker run -it  --name  <container_name>  -v  /<directory_name> <image_name>  /bin/bash
```

Real-time Use Case:

- **Temporary Data Storage:** Useful for temporary data that is needed during the container's lifecycle but does not need to persist after the container is removed.

Made with Gamma

2. Named Volumes

Named volumes are managed by Docker and are stored in a special location on the host filesystem. These volumes are independent of the host directory structure and provide a more flexible and portable way to persist data.

#Example

```
docker run -it  --name  <container_name>  -v  /<directory_name>:<volume _name>  <image_name>  /bin/bash
```

Real-time Use Case:

- **Database Storage:** Using named volumes to persist database data ensures that the data remains intact even if the container is re-created or updated.

Made with Gamma

3. Host Volumes (Bind Mounts)

Host volumes, also known as bind mounts, directly map a directory or file from the host machine's filesystem into the container. This type of volume is useful when you need to share configuration files or other persistent data between the host and the container.

#Example

docker run -it --name <container_name> -v /<directory_name>:<volume _name> <image_name> /bin/bash

docker run -it —name D2 -v /opt/veera:mohan ubuntu /bin/bash

Real-time Use Case:

- **Development Environment:** Developers can edit files on their local machine and immediately see changes reflected in the running container without rebuilding the image.

Made with Gamma

Share Volumes Container to Container

—> To share volumes to the container we have to use command **" --privileged=true --volume -from "**

--privileged=true means you are going to giving authorization to access the volume which is available in the container.

--volume -from means where the volume it is available and how we are trying to share the volume

#Example

docker run -it --name <container_name> --privileged=true --volume -from <container_name> <image_name> /bin/bash

Let's do overview

Made with Gamma

Docker Networks

Docker includes a networking system for managing communications between containers, your Docker host, and the outside world. Several different network types are supported, facilitating a variety of common use cases.

Docker Network Types

Docker networks configure communications between neighboring containers and external services. Containers must be connected to a Docker network to receive any network connectivity. The communication routes available to the container depend on the network connections it has.

1. Bridge Network

2. Host Network

3. None Network

Made with Gamma

1. Bridge Network

The bridge network is the default network type when you create a container. It allows containers on the same host to communicate with each other while isolating them from containers on other hosts.

Example: Imagine you have a web application consisting of a front-end container (running a web server) and a back-end container (running a database). Using a bridge network, these containers can communicate with each other.

```
#Create a bridge network

docker network create my_bridge_network

#Run the database container and attach it to the bridge network

docker run -d --name db --network my_bridge_network mongo

#Run the web server container and attach it to the bridge network

docker run -d --name web --network my_bridge_network nginx
```

Made with Gamma

2. Host Network

The host network removes network isolation between the container and the Docker host. The container shares the host's networking namespace and can access services running on the host.

Example: Suppose you want to run a containerized application that needs to bind to a specific port on the host.

#Run a container using the host network

docker run -d --network host nginx

3. None Network

The none network disables all networking for a container. This can be useful for containers that do not require network access.

Example: Suppose you want to run a container for a task that does not need any network connectivity, such as a local data processing job.

#Run a container with no network

docker run -d --network none my_data_processing_image

Made with Gamma

Docker Compose

Docker Compose is a tool for defining and running multi-container applications. It is the key to unlocking a streamlined and efficient development and deployment experience.

Compose simplifies the control of your entire application stack, making it easy to manage services, networks, and volumes in a single, comprehensible YAML configuration file. Then, with a single command, you create and start all the services from your configuration file.

Compose works in all environments; production, staging, development, testing, as well as CI workflows. It also has commands for managing the whole lifecycle of your application:

- Start, stop, and rebuild services
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service

Made with Gamma

How Compose works

Docker Compose relies on a YAML configuration file, usually named `compose.yaml`.

The `compose.yaml` file follows the rules provided by the **Compose Specification** in how to define multi-container applications. This is the Docker Compose implementation of the formal **Compose Specification**.

The default path for a Compose file is `compose.yaml` (preferred) or `compose.yml` that is placed in the working directory. Compose also supports `docker-compose.yaml` and `docker-compose.yml` for backwards compatibility of earlier versions. If both files exist, Compose prefers the canonical `compose.yaml`

Made with Gamma

Why use Compose?

Key Benefits of Docker Compose

Using Docker Compose offers several benefits that streamline the development, deployment, and management of containerized applications:

- Simplified control: Docker Compose allows you to define and manage multi-container applications in a single YAML file. This simplifies the complex task of orchestrating and coordinating various services, making it easier to manage and replicate your application environment.
- Efficient collaboration: Docker Compose configuration files are easy to share, facilitating collaboration among developers, operations teams, and other stakeholders. This collaborative approach leads to smoother workflows, faster issue resolution, and increased overall efficiency.
- Rapid application development: Compose caches the configuration used to create a container. When you restart a service that has not changed, Compose re-uses the existing containers. Re-using containers means that you can make changes to your environment very quickly.

Made with Gamma

- Portability across environments: Compose supports variables in the Compose file. You can use these variables to customize your composition for different environments, or different users.
- Extensive community and support: Docker Compose benefits from a vibrant and active community, which means abundant resources, tutorials, and support. This community-driven ecosystem contributes to the continuous improvement of Docker Compose and helps users troubleshoot issues effectively.

Common Use Case Of Docker Compose

Compose can be used in many different ways. Some common use cases are outlined below.

Development Environment

When you're developing software, the ability to run an application in an isolated environment and interact with it is crucial. The Compose command line tool can be used to create the environment and interact with it.

The **Compose file** provides a way to document and configure all of the application's service dependencies (databases, queues, caches, web service APIs, etc). Using the Compose command line tool you can create and start one or more containers for each dependency with a single command (`docker compose up`).

Made with Gamma

Automated Testing Environment

An important part of any Continuous Deployment or Continuous Integration process is the automated test suite. Automated end-to-end testing requires an environment in which to run tests. Compose provides a convenient way to create and destroy isolated testing environments for your test suite. By defining the full environment in a **Compose file**, you can create and destroy these environments in just a few commands.

```
docker compose up -d

docker compose down
```

Single Host Deployments

Compose has traditionally been focused on development and testing workflows, but with each release we're making progress on more production-oriented features.

For details on using production-oriented features, see **Compose in production**.

Made with Gamma

Docker Compose File Reference

Docker Compose File Syntax

version:

Specifies the version of the Docker Compose file format

#Example

```
version: '3.8'
```

services:

Defines the services/containers that make up the application.

#Example

```
services:
  web:
    image: nginx:latest
```

Made with Gamma

networks:

Configures custom networks for the application.

#Example:

```
networks:
  my_network:
      driver: bridge
```

volumes:

Defines named volumes that the services can use.

#Example:

```
volumes :
  my_volume :
```

Made with Gamma

environment:

Sets environment variables for a service.

#Example:

environment :

- NODE_ENV=production

ports:

Maps host ports to container ports.

#Example:

ports :

- "8080:80"

Made with Gamma

depends_on:

Specifies dependencies between services, ensuring one service starts before another.

#Example:

depends_on:

- db

build:

Configures the build context and Dockerfile for a service.

#Example:

build:

context: .

dockerfile : Dockerfile.dev

Made with Gamma

volumes_from:

Mounts volumes from another service or container.

#Example:

volumes_from :

- service_name

command:

Overrides the default command specified in the Docker image.

#Example:

command : ["npm", "start"]

Made with Gamma

Lets Work With Real Time Projects

1. Transaction History
2. Docker Case Study

THE END

Made with Gamma