



# Virtualization and Containerization (Docker)

## Training Material

## **Table of Contents**

<b>Chapter 1: Introduction.....</b>	<b>4</b>
1. Overview of Virtualization and Containerization	
2. <b>Significance in Contemporary Development</b>	
<b>Chapter 2: Virtualization Concepts.....</b>	<b>4</b>
1. What is Virtualization?	
2. Types of Virtualization	
3. Essential Elements	
4. Advantages of Virtualization	
5. Practical Applications	
6. Benefits of Virtualization	
<b>Chapter 3: Containerization Concepts.....</b>	<b>8</b>
1. What is Containerization?	
2. How Containerization Works	
3. Benefits of Containerization	
<b>Chapter 4: Docker Overview.....</b>	<b>9</b>
1. What is Docker?	
2. Docker Architecture	
3. Docker vs. Traditional Virtualization	
<b>Chapter 5: Getting Started with Docker.....</b>	<b>12</b>
1. Installing Docker	
2. Basic Docker Commands	
<b>Chapter 6: Creating and Managing Containers.....</b>	<b>14</b>
1. Creating Containers	
2. Managing Containers	
3. Stopping and Removing Containers	
<b>Chapter 7: Working with Docker Images.....</b>	<b>18</b>
1. Understanding Docker Images	
2. Creating Custom Images	
3. Managing Images	

**Chapter 8: Deploying Applications with Docker.....22**

1. Introduction to Docker Compose
2. Multi-Container Applications
3. Best Practices for Deployment
4. **How to Deploy Application with docker desktop**

**Chapter 9: Conclusion.....25**

## Chapter 1. Introduction

### 1. Overview of Virtualization and Containerization

Virtualization and containerization represent two fundamental technologies that enhance resource efficiency, boost application scalability, and simplify the deployment process in contemporary software development.

### 2. Significance in Contemporary Development

These technologies empower developers to establish isolated environments for applications, guaranteeing uniformity across development, testing, and production phases.

## Chapter 2. Virtualization Concepts

### 1. What is Virtualization?

Virtualization enables several operating systems to operate on a single physical machine by forming virtual machines (VMs). Each VM possesses its unique OS and resources.



## 2. Types of Virtualization:

**Server Virtualization:** Segments a physical server into numerous virtual servers (VMs), each able to operate its own operating system and applications. This enhances hardware utilization and streamlines management.

**Desktop Virtualization:** Permits users to connect to desktop environments hosted on a centralized server. This may encompass virtual desktops or remote desktop solutions.

**Storage Virtualization:** Distills physical storage devices, consolidating them into a singular storage resource that can be overseen more effectively.

**Network Virtualization:** Merges hardware and software network resources into one, software-driven administrative unit, facilitating improved network management and adaptability.

## 3. Essential Elements:

- a) **Hypervisor:** A software layer that generates and supervises virtual machines. There are two varieties of Type 1 (Bare-metal Hypervisor) - Operates directly on the hardware (e.g., VMware ESXi, Microsoft Hyper-V) and Type 2 (Hosted Hypervisor) - Functions over an existing operating system (e.g., Oracle VirtualBox, VMware Workstation).
- b) **Virtual Machine (VM):** A self-contained instance that mimics a physical computer, executing its own operating system and applications.

**Guest Operating System:** The operating system operating within a virtual machine.

**Host Machine:** The physical machine that furnishes resources for the virtual machines.

## 4. Advantages of Virtualization:

**Resource Efficiency:** Optimizes the utilization of physical hardware, permitting multiple VMs to operate on a solitary server.

**Scalability:** Flexibly adjust resources up or down as necessary without major hardware modifications.

**Isolation:** VMs are segregated from one another, boosting security and stability. If one VM fails, others are not impacted.

**Simplified Management:** Centralized oversight of VMs enables simpler updates, backups, and resource distribution.

**Cost Savings:** Lowers hardware expenses, energy usage, and spatial needs in data centers.

ters.

## 5. Practical Applications

**Development and Testing:** Enables developers to establish isolated settings for application testing without affecting production systems.

**Disaster Recovery:** Streamlines backup and recovery tasks by allowing VMs to be effortlessly duplicated and restored.

**Cloud Computing:** Constitutes the backbone of many cloud services, permitting effective resource allocation and management in public, private, and hybrid clouds.

## 6. Benefits of Virtualization

Resource Optimization: Effectively utilizes hardware assets.

Isolation: Offers a secure setting for testing and development.

Flexibility: Rapidly create and dismantle environments.

# Chapter 3. Containerization Concepts

## 1. What is Containerization?

Containerization is an innovative approach that enables applications to operate within distinct environments known as containers. These containers bundle the application code alongside its dependencies, libraries, and configuration files, guaranteeing consistent application performance across diverse computing landscapes.

**Prominent characteristics of containerization encompass:**

**Separation:** Each container functions autonomously, preventing applications from interfering with each other.

**Mobility:** Containers can be deployed on any system compatible with containerization, facilitating effortless application transitions between various environments (e.g., development, testing, production).

**Economy:** Containers utilize the host operating system's kernel, rendering them lightweight and faster to initiate than conventional virtual machines.

**Expansibility:** Containers can be swiftly adjusted up or down to accommodate shifting workloads, typically overseen by orchestration platforms like Kubernetes.

In essence, containerization streamlines application deployment, optimizes resource use, and boosts adaptability in both development and operational processes.

Containerization is an efficient variation of virtualization that bundles applications and their dependencies into containers, utilizing the host OS kernel while operating in isolated settings.

## 2. How Containerization Works

Containerization operates by utilizing a blend of operating system-level virtualization and resource management to form secluded environments for applications. Here's a more in-depth analysis of how it functions:

## 1. Images and Containers

**Images:** A container image is a nimble, self-contained, executable bundle that encompasses everything essential to execute an application, including the code, runtime, libraries, and environment variables. Images are created using a series of instructions outlined in a file known as a Dockerfile (for Docker containers, the most widely used containerization tool).

**Containers:** When an image is instantiated, it transforms into a container. Containers represent the active instances of these images, and they share the kernel of the host system while remaining distinct from one another.

## 2. Namespaces

Containerization employs Linux namespaces to offer isolation. Namespaces enable containers to possess their own perspectives on system resources such as process IDs (PID), network interfaces, user IDs, and file systems. This guarantees that processes within one container are oblivious to or unable to engage with processes in another.

## 3. Control Groups (cgroups)

cgroups regulate and constrain the resource utilization (CPU, memory, disk I/O, etc.) of containers. This ensures that no individual container can dominate the host system's resources, allowing for equitable distribution and enhanced performance.

## 4. Union File System

Containers frequently leverage a union file system, which permits multiple layers of file systems to be overlaid. This means that images can be developed gradually, sharing common layers between different images, which diminishes storage requirements and accelerates deployment.

## 5. Networking

Containers can interact with one another through designated networking configurations. They can be allocated unique IP addresses and can connect to external networks or other containers, according to the setup.

## 6. Orchestration

To manage multiple containers, orchestration tools like Kubernetes or Docker Swarm are employed. These tools automate the deployment, scaling, and management of containerized applications, simplifying the handling of intricate systems with numerous interdependent services.

## 7. Deployment and Lifecycle Management

Containers can be effortlessly deployed, initiated, halted, or eliminated as necessary. This adaptability enables continuous integration and continuous deployment (CI/CD) practices, where applications can be refreshed frequently without downtime.

Containerization operates by taking advantage of sophisticated operating system features to create lightweight, portable, and secluded environments for running applications, establishing it as a fundamental aspect of contemporary software development and deployment practices.

### 3. Benefits of Containerization

Containerization presents numerous significant advantages:

#### 1. Portability

Containers bundle applications alongside their dependencies, guaranteeing consistent performance across diverse settings (development, testing, production).

#### 2. Isolation

Each container functions independently, preventing application conflicts and boosting security. This separation facilitates debugging and testing processes.

#### 3. Scalability

Containers can be effortlessly scaled up or down to accommodate fluctuating demands. Orchestration platforms like Kubernetes automate scaling, promoting efficient resource management.

#### 4. Resource Efficiency

Containers utilize the host OS kernel, rendering them lighter than conventional virtual machines. This results in superior utilization of system resources.

#### 5. Faster Deployment

Containers can be initiated nearly instantaneously, minimizing deployment time and supporting rapid development cycles and continuous integration/continuous deployment (CI/CD).

#### 6. Simplified Management

Container orchestration platforms enhance the deployment, management, and monitoring of applications, streamlining the management of intricate systems.

#### 7. Version Control

Containers are capable of versioning, allowing teams to revert to earlier versions effortlessly and ensuring a dependable deployment workflow.



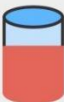




#### 8. Microservices Architecture

Containerization complements microservices effectively, allowing applications to be partitioned into smaller, manageable services that can be developed, deployed, and scaled independently.

**9. Efficiency:** Minimal overhead in comparison to VMs.

**10. Consistency:** Assures applications operate similarly across varied environments.



	Virtualization	Containerization
Startup time	 minutes	 seconds
Disk space		
Portability	Less Portable	
Efficiency		
Operating system/kernel	Dedicated	Shared

## Chapter 4. Docker Overview

### 1. What is Docker?

Docker is an open-source framework aimed at streamlining the deployment, scaling, and oversight of applications within nimble, transportable containers. It empowers developers to encapsulate applications along with all their dependencies into a uniform unit, ensuring that the software operates uniformly across various computational landscapes. Here's a more elaborate summary:

#### Key Features:

##### Containerization:

Docker employs container technology to establish isolated environments for applications. Each container comprises the application code, libraries, and any necessary binaries, permitting it to function independently from the host system.

Portability:

Docker containers can operate on any machine equipped with the Docker Engine, simplifying the transfer of applications between development, testing, and production settings without compatibility hurdles.

Efficiency:

Containers utilize the host operating system's kernel, rendering them more lightweight than conventional virtual machines (VMs). This leads to swifter startup durations and enhanced resource utilization.

##### Version Control:

Docker images can be versioned, enabling developers to effortlessly revert to earlier versions of an application when required. This is particularly advantageous in C

I/CD pipelines where updates happen frequently.

### Microservices Support:

Docker is excellently suited for microservices architecture, where applications are divided into smaller, independently deployable services. Each service can operate in its own container, fostering scalability and easier management.

### Docker Hub:

Docker Hub is a cloud-centric registry for storing and disseminating Docker images. Developers can retrieve pre-crafted images or upload their own to the registry, promoting teamwork and reuse.

### Use Cases:

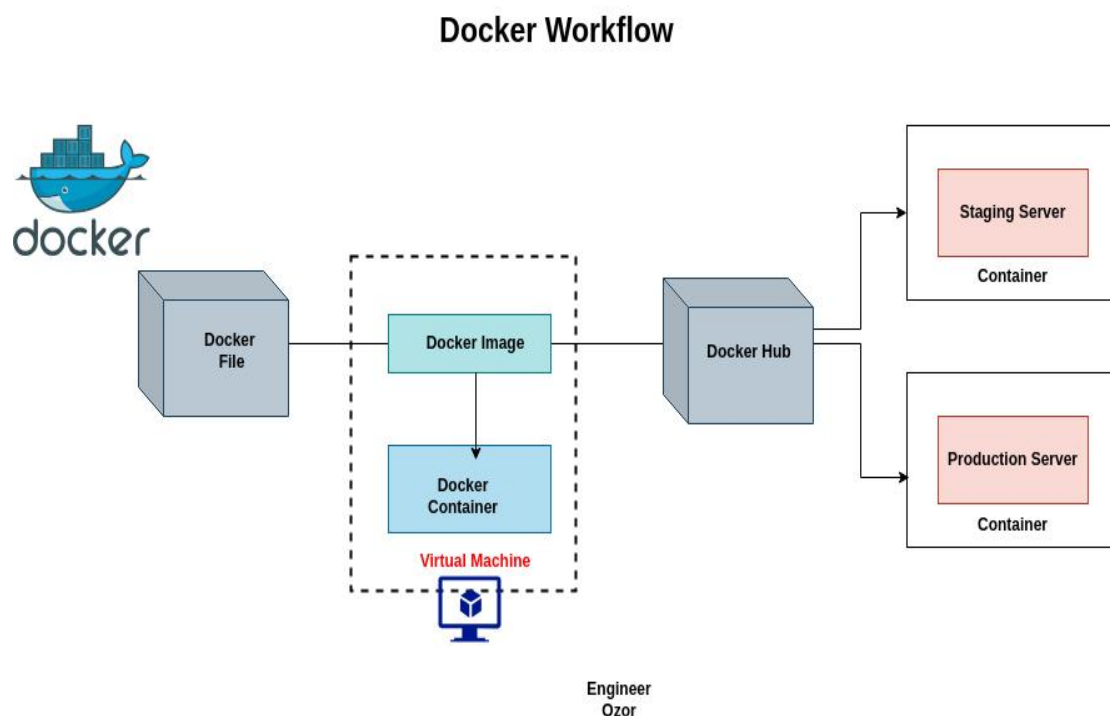
#### Development and Testing:

Developers can establish a consistent environment to create and evaluate applications, minimizing the "it works on my machine" dilemma.

**Continuous Integration/Continuous Deployment (CI/CD):** Docker facilitates swift deployment cycles, making it perfect for contemporary development workflows.

**Application Isolation:** Containers offer a method to execute multiple applications on the same host without conflict, boosting resource efficiency.

**Cloud Deployments:** Docker is frequently utilized in cloud environments, enabling applications to be easily deployed, scaled, and managed.



## 2. Docker Architecture

Docker's framework is crafted to enhance the development, rollout, and supervision of applications within containers. It is comprised of several essential elements that collaborate to establish an effective and scalable ecosystem. Below is an in-

depth examination of Docker's framework:

## 1. Docker Engine

The Docker Engine is the fundamental component of Docker, tasked with executing containers. It encompasses three primary segments:

**Docker Daemon (dockerd):** This is the background service that oversees Docker containers, images, networks, and volumes. It awaits API requests and can interact with additional Docker daemons.

**REST API:** This offers a mechanism for users and applications to engage with the Docker daemon. The API enables functionalities such as forming, launching, and administering containers.

**Docker CLI (Command Line Interface):** This command-line utility enables users to communicate with the Docker daemon via commands. Users can construct images, execute containers, and manage the entire Docker ecosystem using straightforward command-line instructions.

## 2. Docker Images

Docker images are immutable templates utilized to generate containers. An image encompasses everything required to execute an application, including:

- The application code
- Libraries and dependencies
- Environment variables
- Configuration files

Images are constructed in layers, allowing for effective storage and reuse. When modifications occur, only the altered layers need updating, saving disk space.

## 3. Docker Containers

Containers are active instances of Docker images. Each container remains isolated from others and operates as a distinct process on the host operating system. Key features of containers include:

**Isolation:** Containers function in their own environments, guaranteeing that applications do not compete with each other.

**Lightweight:** Containers utilize the host OS kernel, making them significantly smaller and swifter to initiate compared to traditional virtual machines.

**Ephemeral:** Containers can be effortlessly crafted, activated, halted, and eliminated, offering high flexibility.

## 4. Docker Registry

A Docker registry is a storage solution for Docker images. The most prevalent registry is Docker Hub, a public repository where users can discover and exchange images. There are also private registries for organizations requiring secure management of their images.

## 5. Docker Compose

Docker Compose is a tool for structuring and operating multi-

container applications. Through a simple YAML file, users can delineate the services, networks, and volumes essential for an application. This simplifies the management of intricate applications with numerous interconnected elements.

## 6. Networking

Docker presents various networking options to facilitate communication between containers, the host, and external systems. Key networking attributes include:

**Bridge Network:** The default network enabling containers to interact with each other.

**Host Network:** Containers share the host's networking architecture, which can enhance performance for certain applications.

**Overlay Network:** Employed for networking across multiple hosts, allowing containers on different Docker hosts to communicate.

## 7. Volumes

Volumes are utilized for enduring data storage. Unlike container file systems, which are transient, volumes can preserve data independently of the container lifecycle. This is crucial for databases and applications that must retain data between container restarts.

## 3. Docker vs. Traditional Virtualization

Docker containers are more efficient than VMs, resulting in quicker deployment and enhanced resource utilization. Containers utilize the host OS kernel, while VMs operate distinct operating systems.

# Chapter 5. Getting Started with Docker

## 1. Installing Docker

**For Windows and macOS:** Download Docker Desktop from Docker's official site and adhere to the installation instructions.

**For Linux:** Refer to the official installation manual.

## 2. Basic Docker Commands

This list includes the most commonly used Docker commands to manage containers, images, networks, and volumes. Docker also has a wealth of options for each command, allowing for more granular control and functionality.

### Basic Commands

#### Container Management:

1. `docker run`: Create and start a container from an image.
2. `docker ps`: List running containers.
3. `docker ps -a`: List all containers (running and stopped).
4. `docker start <container_id>`: Start a stopped container.
5. `docker stop <container_id>`: Stop a running container.
6. `docker restart <container_id>`: Restart a container.
7. `docker rm <container_id>`: Remove a stopped container.

8. `docker exec -it <container_id> <command>`: Execute a command inside a running container.

## **Image Management**

1. `docker images`: List all Docker images on the local system.
2. `docker pull <image_name>`: Download an image from a registry (e.g., Docker Hub).
3. `docker build -t <image_name>:<tag> .`: Build an image from a Dockerfile in the current directory.
4. `docker rmi <image_id>`: Remove an image.
5. `docker tag <image_id> <new_image_name>`: Tag an image with a new name.

## **Volume Management**

1. `docker volume ls`: List all Docker volumes.
2. `docker volume create <volume_name>`: Create a new volume.
3. `docker volume rm <volume_name>`: Remove a volume.

## **Network Management**

1. `docker network ls`: List all Docker networks.
2. `docker network create <network_name>`: Create a new network.
3. `docker network rm <network_name>`: Remove a network.

## **Advanced Commands**

### **Container Logs and Information**

1. `docker logs <container_id>`: View logs from a container.
2. `docker inspect <container_id|image_id>`: Get detailed information about a container or image.
3. `docker stats`: Display a live stream of container resource usage statistics.

### **Container and Image Cleanup**

1. `docker system prune`: Remove unused data (stopped containers, unused networks, dangling images).
2. `docker image prune`: Remove unused images.
3. `docker container prune`: Remove stopped containers.

### **Docker Compose Commands**

1. `docker-compose up`: Start services defined in a `docker-compose.yml` file.
2. `docker-compose down`: Stop and remove containers defined in a `docker-compose.yml` file.
3. `docker-compose ps`: List containers managed by Docker Compose.

## Miscellaneous Commands

1. `docker version`: Display the Docker version information.
2. `docker info`: Display system-wide information about Docker.
3. `docker help`: Show help for Docker commands.

### Run a simple container:

```
docker run hello-world
```

## Chapter 6. Creating and Managing Containers

### 1. Creating Containers

To create a container, use the `docker run` command. For example, to run an Nginx container:

```
docker run -d -p 80:80 nginx
```

- `-d`: Run in detached mode (in the background).
- `-p`: Map port 80 on the host to port 80 in the container.

### How to create a container:

Creating a Docker container involves several steps, from installing Docker to running your first container. Here's a step-by-step guide:

#### Step 1: Install Docker

##### Download Docker:

1. Visit the Docker website and download the Docker Desktop application suitable for your operating system (Windows, macOS, or Linux).

##### Install Docker:

1. Follow the installation instructions specific to your OS. For Windows and macOS, this typically involves running the installer. For Linux, you might use package managers like `apt` or `yum`.

##### Verify Installation:

1. Open a terminal or command prompt and run:

```
docker --version
```

2. You should see the installed Docker version.

#### Step 2: Pull a Docker Image

### **Open Terminal:**

1. Open a terminal or command prompt.

### **Pull an Image:**

1. To create a container, you need an image. For example, to pull the official Ubuntu image, run:

```
docker pull ubuntu
```

2. This downloads the latest Ubuntu image from Docker Hub.

### **Step 3: Create and Run a Container**

1. **Run the Container:** Use the `docker run` command to create and start a container. For example, to run an interactive shell in the Ubuntu container, use:

```
docker run -it ubuntu
```

The `-it` flag combines `-i` (interactive) and `-t` (pseudo-TTY) to allow interaction with the container.

2. **Verify the Container is Running:**

You'll be dropped into the Ubuntu shell. You can check if it's running by executing:

```
whoami
```

3. This should return `root`, as you are operating inside the container.

### **Step 4: Exit the Container**

1. **Exit the Shell:** To exit the container shell, type:

```
exit
```

2. This stops the container. If you want to keep the container running in the background, you can use `Ctrl + P` followed by `Ctrl + Q` to detach.

### **Step 5: List Containers**

1. **View Running Containers:** To see all running containers, run:

```
docker ps
```

2. **View All Containers (including stopped ones):**

To see all containers, use  
`docker ps -a`

## Step 6: Start/Stop a Container

1. **Start a Stopped Container:** If you exited the container and want to start it again, use:

```
docker start <container_id>
```

Replace <container\_id> with the actual ID or name of the container.

2. **Stop a Running Container:** To stop a running container, use:

```
docker stop <container_id>
```

## Step 7: Remove a Container

1. **Remove a Stopped Container:** To delete a stopped container, run:

```
docker rm <container_id>
```

## Step 8: Clean Up

1. **Remove Unused Images and Containers:** To clean up unused images and containers, you can run:

```
docker system prune
```

## Summary

1. Install Docker.
2. Pull an image using `docker pull`.
3. Create and run a container using `docker run -it`.
4. Exit the container using `exit` or detach with `Ctrl + P`, `Ctrl + Q`.
5. Manage containers with `docker ps`, `docker start`, `docker stop`, and `docker rm`.
6. Clean up with `docker system prune`.

## 2. Managing Containers

### List running containers:

```
docker ps
```

### View all containers:

```
docker ps -a
```

### Here is a step by step process:



Managing Docker containers involves several key operations, including starting, stopping, inspecting, and removing containers. Here's a step-by-step guide to effectively manage your Docker containers:

## Step 1: List Containers

### List Running Containers:

```
docker ps
```

This command displays all currently running containers.

### List All Containers:

```
docker ps -a
```

This shows both running and stopped containers, along with their statuses.

## Step 2: Start a Container

1. **Start a Stopped Container:** If you have a stopped container, you can start it using:

```
bash
Copy code
docker start <container_id>
```

Replace <container\_id> with the actual ID or name of the container.

## Step 3: Stop a Running Container

1. **Stop a Running Container:** To stop a container that is currently running, use:

```
docker stop <container_id>
```

## Step 4: Restart a Container

1. **Restart a Container:** To restart a running or stopped container, run:

```
docker restart <container_id>
```

## Step 5: Execute Commands in a Running Container

1. **Run Commands in a Running Container:** To execute commands inside a running container, use:

```
docker exec -it <container_id> <command>
```

For example, to open a shell in a running Ubuntu container:

```
docker exec -it <container_id> /bin/bash
```

## Step 6: Inspect a Container

1. **Get Detailed Information:**To view detailed information about a container (such as configuration, network settings, and resource limits), use:

```
docker inspect <container_id>
```

## Step 7: Remove a Container

1. **Remove a Stopped Container:**To delete a container that is no longer needed, first ensure it is stopped, then run:

```
docker rm <container_id>
```

2. **Force Remove a Running Container:**If you want to remove a running container (which will stop it), use the -f option:

```
docker rm -f <container_id>
```

## Step 8: View Container Logs

1. **Check Logs:**To view the logs of a container, which can help with debugging, run:

```
docker logs <container_id>
```

## 3. Stop and Remove Containers

1. **Remove All Stopped Containers:**To remove all stopped containers in one go, run:

```
docker container prune
```

2. **Stop a running container:**

```
docker stop CONTAINER_ID
```

## Remove a container:

```
docker rm CONTAINER_ID
```

## Chapter 7. Working with Docker Images

### 1. Understanding Docker Images

A Docker image is a compact, self-sufficient, and executable bundle that encompasses everything necessary to operate a piece of software. This comprises the application code, libraries, dependencies, environment variables, and configuration files. Docker images are the foundational elements of containers, functioning as the template from which containers are forged.

## Key Traits of Docker Images

### 1. Layered Architecture:

Docker images are constructed in layers. Each layer signifies a modification or addition, like installing a library or incorporating files. This stratification allows for efficient storage and reuse, as shared layers among various images are possible.

### 2. Read-Only:

Once an image is generated, it becomes immutable (read-only). Alterations made while a container is utilizing that image do not influence the original image. Instead, a new layer is formed atop the existing layers in the container's writable layer.

### 3. Versioning:

Images can be versioned through tags. For instance, `myapp:1.0` and `myapp:2.0` can denote different iterations of the same application. This facilitates the management and deployment of specific versions.

### 4. Portable:

Docker images can be effortlessly shared and deployed across various environments, be it a developer's local setup, a testing stage, or production servers. This guarantees uniformity and eradicates the 'it works on my machine' dilemma.

### 5. Base Images:

Images can be constructed atop other images, referred to as base images. For example, an image can be founded on an official Ubuntu image, incorporating custom software and configurations on top of it.

## 2. Creating a custom Docker image on Docker Desktop for Windows:

It involves several steps. Below is a step-by-step guide to help you build your own Docker image.

### Step 1: Install Docker Desktop

#### 1. Download and Install Docker Desktop:

1. If you haven't already, download Docker Desktop for Windows from the Docker website.
2. Follow the installation instructions and start Docker Desktop.

### Step 2: Create a Project Directory

1. Create a Directory: Open a command prompt or PowerShell window.
2. Create a new directory for your project:

```
my-docker-image> cd my-docker-image
```

### Step 3: Create a Dockerfile

1. **Create a Dockerfile:** Inside your project directory, create a file named Dockerfile (no file extension). You can use any text editor (like Notepad or VSCode) for this.

Here's a simple example of a Dockerfile for a Node.js application:

```
# Use the official Node.js image as a base
FROM node:14

# Set the working directory in the container
WORKDIR /app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application files
COPY . .

# Expose the application port
EXPOSE 3000

# Define the command to run the application
CMD ["npm", "start"]
```

Modify this Dockerfile as needed based on your application requirements.

#### Step 4: Build the Docker Image

1. **Open Command Prompt:** Ensure you are still in the directory containing the Dockerfile.
2. **Build the Image:** Run the following command to build your custom image:

```
docker build -t my-custom-image:latest .
```

The -t flag tags the image with a name (my-custom-image) and a version (latest).

#### Step 5: Verify the Image

1. **List Docker Images:** To verify that your image has been created, run:

```
docker images
```

2. You should see my-custom-image listed among the available images.

#### Step 6: Run a Container from Your Image

1. **Run the Container:** Use the following command to create and start a container from your custom image:

```
docker run -p 3000:3000 my-custom-image
```

The -p flag maps port 3000 of your host machine to port 3000 of the container.

### Step 7: Access Your Application

1. **Open a Web Browser:** If your application is a web application, open a browser and go to `http://localhost:3000` to see it running.

### Step 8: Clean Up (Optional)

1. **Stop and Remove Containers:** To stop a running container, first find its ID using `docker ps`, then use:

```
docker stop <container_id>
```

2. **To remove the container after stopping it, run:**

```
docker rm <container_id>
```

3. **Remove the Image:** If you want to delete your custom image, use:

```
docker rmi my-custom-image
```

### Summary

1. Install Docker Desktop.
2. Create a project directory.
3. Create a Dockerfile with your application instructions.
4. Build your custom image using `docker build`.
5. Verify the image creation with `docker images`.
6. Run a container from your image with `docker run`.
7. Access your application via a web browser if applicable.
8. Clean up by stopping/removing containers and images as needed.

Following these steps will allow you to create and manage a custom Docker image on Docker Desktop for Windows

## 3.Managing Images

### Build an image:

```
docker build -t myapp .
```

### List images:

```
docker images
```

### Remove an image:

```
docker rmi IMAGE_ID
```

## Chapter 8. Deploying Applications with Docker

### 1. Introduction to Docker Compose

Docker Compose is a tool for defining and managing multi-container applications using a YAML file.

### 2. Multi-Container Applications

A sample docker-compose.yml might look like this:

```
yaml
Copy code
version: '3'services:
  web:
    image: nginx
    ports:
      - "8080:80"
  app:
    build: .
    volumes:
      - ./app:/app
```

### 3. Best Practices for Deployment

- **Optimize Dockerfiles:** Reduce image size by minimizing layers.
- **Use .dockerignore:** Exclude unnecessary files from your images.
- **Regularly Update Images:** Keep dependencies and base images up to date for security.

### 4. How to Deploy Application with docker desktop in windows as OS

Deploying an application using Docker Desktop on Windows involves several steps, from setting up your Docker environment to running your application in a container. Here's a step-by-step guide to help you through the process:

#### Step 1: Install Docker Desktop

##### Download and Install:

1. If you haven't already, download Docker Desktop for Windows from the Docker website.
2. Follow the installation instructions and start Docker Desktop.

#### Step 2: Prepare Your Application

1. **Create a Project Directory:** Open a command prompt or PowerShell and create a new directory for your application:

```
mkdir my-appcd my-app
```

2. **Add Your Application Files:** Place your application code and any necessary files in this directory. For example, if it's a Node.js application, you might include app.js, package.json, etc.

### Step 3: Create a Dockerfile

1. **Create a Dockerfile:** Inside your project directory, create a file named Dockerfile (no file extension).

Here's a simple example for a Node.js application:

```
# Use the official Node.js image as a base
FROM node:14

# Set the working directory in the container
WORKDIR /usr/src/app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application files
COPY . .

# Expose the application port
EXPOSE 3000

# Define the command to run the application
CMD ["npm", "start"]
```

Adjust the Dockerfile based on your application requirements.

### Step 4: Build the Docker Image

1. **Open Command Prompt:** Ensure you are still in your project directory.
2. **Build the Image:** Run the following command to build your Docker image:

```
docker build -t my-app-image .
```

This command builds the image and tags it as my-app-image.

### Step 5: Run the Container

1. **Run the Container:** Use the following command to create and start a container from your image:

```
docker run -p 3000:3000 --name my-app-container my-app-image
```

The `-p` flag maps port 3000 of your host to port 3000 of the container. You can adjust the port numbers as needed.

## Step 6: Access Your Application

1. **Open a Web Browser:** If your application is a web app running on port 3000, open a browser and navigate to:

```
http://localhost:3000
```

You should see your application running.

## Step 7: Manage the Container

1. **List Running Containers:**  

```
docker ps
```
2. **Stop the Container:**  

```
docker stop my-app-container
```
3. **Start the Container** (if it's stopped):  

```
docker start my-app-container
```
4. **Remove the Container:**  

```
rm my-app-container
```

## Step 8: Clean Up (Optional)

1. **Remove the Image** (if needed):  

```
docker rmi my-app-image
```

## Summary

1. Install Docker Desktop on Windows.
2. Prepare your application and create a project directory.
3. Create a Dockerfile tailored to your application.
4. Build the Docker image with `docker build`.
5. Run a container from the image using `docker run`.
6. Access your application via a web browser.
7. Manage the container using Docker commands.



## Chapter 9. Conclusion

Understanding virtualization and containerization is crucial for modern software development. Docker provides a powerful platform for leveraging containerization to build, deploy, and manage applications efficiently

