Arun KL

November 2, 2023 | 18m

Understand Docker Containers With TheSecMaster

Explore

Some of our readers have requested information about Docker and container technology. We have decided to publish an article on Docker Containers to help many users understand the technical aspects of Docker and containers. In this blog post, we will clarify the concept of containers and their different types, explain what Docker is, explore the relationship between Docker and containers,

discuss the appropriate use cases for Docker, delve into the problems Docker solves in development, address whether Docker replaces virtualization, explain the differences between Docker containers and virtual machines, and answer a few more questions that will aid in understanding Docker.

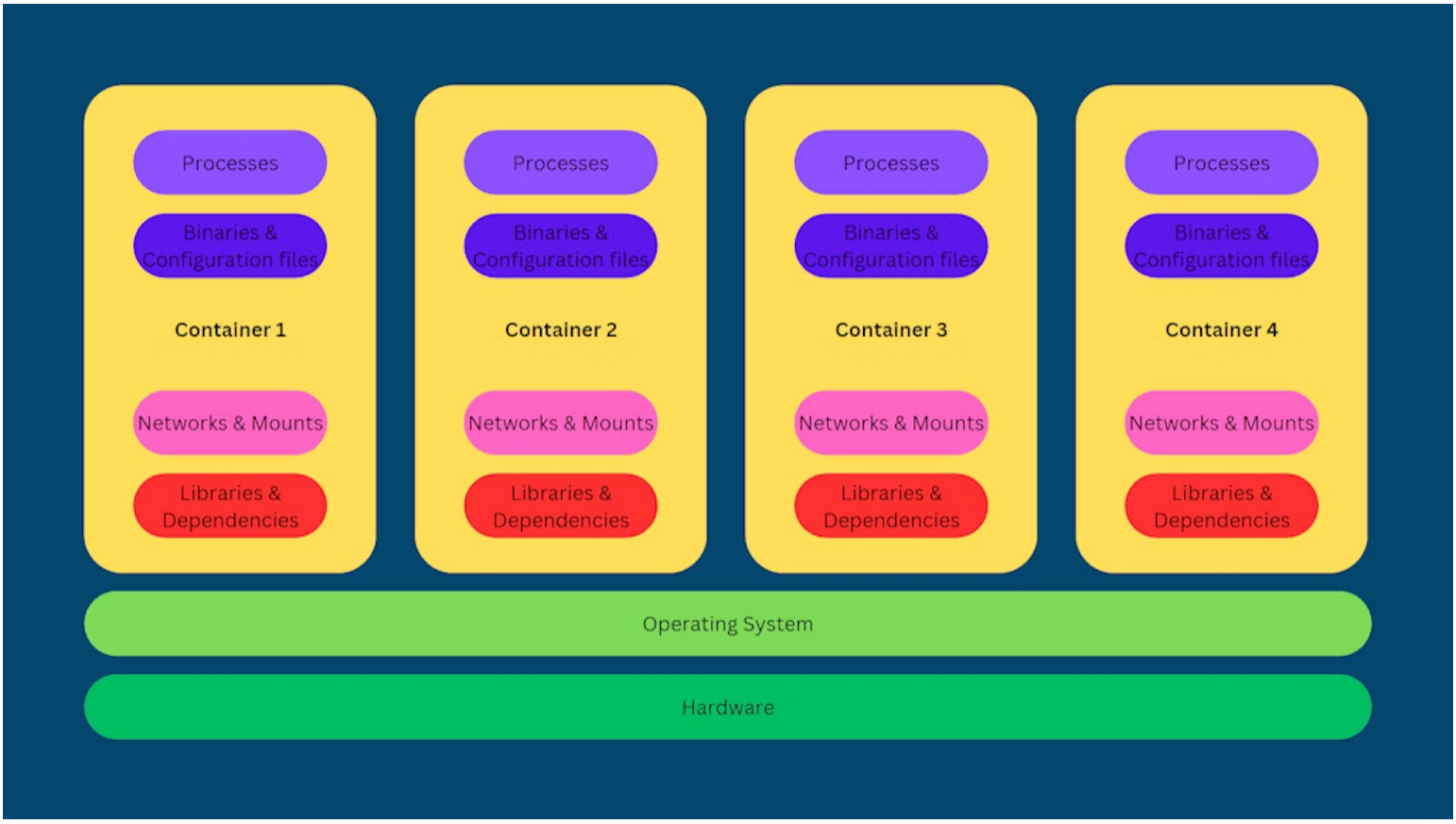
Before we delve directly into Docker, let's begin by exploring containers, as it will make it easier to comprehend Docker if you are familiar with containers.

What is a Container?

A container is a lightweight and isolated environment that allows you to package and run an application and its dependencies. It encapsulates the software and all its dependencies into a single, self-contained unit that can run consistently across different computing environments.

Containers provide a consistent and reliable runtime environment, ensuring that an application will run the same way regardless of the host system. They achieve this by leveraging operating system-level virtualization, which allows multiple containers to run on the same host while remaining isolated from one another.

Each container includes the necessary libraries, binaries, and configuration files needed for the application to run successfully. This eliminates the common issue of "it works on my machine" by ensuring that the application's runtime environment is consistent across different development, testing, and production environments.



The history and development of containers can be traced back to the early 2000s, although the concept of process isolation and virtualization dates back even further. Let's explore the key milestones in the evolution of containers: The history and development of containers can be traced

back to the early 2000s, although the concept of process isolation and virtualization dates back even further. Let's explore the key milestones in the evolution of containers:

1. Chroot: In the 1970s, the Unix operating system introduced the chroot system call, which allowed a process to have its root directory set to a different location. This provided a level of isolation by limiting a process's access to files and directories outside its designated root directory.
2. Jails: In the early 2000s, the FreeBSD operating system introduced a feature called "jails." Jails provided a lightweight form of virtualization by creating isolated environments within a host system. Each jail had its own file system, network stack, and processes, allowing multiple applications to run independently on the same physical machine.
3. Linux Containers (LXC): In 2008, the Linux Containers project was launched, aiming to provide lightweight operating system-level virtualization on Linux. LXC introduced the concept of control groups (cgroups) and namespaces, which allowed for resource allocation and isolation of processes, networks, and file systems. LXC was a significant step towards the modern container technology we use today.
4. Docker: In 2013, Docker was released and had a transformative impact on the container ecosystem. Docker simplified the process of creating, managing, and distributing containers by introducing a user-friendly interface and a powerful toolset. It popularized the use of container images, which are pre-built, portable packages containing the application and its dependencies. Docker also introduced a distributed registry called Docker Hub, making it easy to share and discover container images.
5. Container Orchestration: As container adoption grew, the need for managing and orchestrating large-scale deployments emerged. Several container orchestration platforms, such as Kubernetes, Docker Swarm, and Apache Mesos, were developed to address this demand. These platforms provided capabilities for automated scaling, load balancing, service discovery, and fault tolerance, enabling the management of containerized applications at scale.
6. Container Standards: To ensure interoperability and portability between different container platforms, industry standards like the Open Container Initiative (OCI) were established. The OCI defined specifications for container runtime and image formats, fostering compatibility across various container runtimes and tools.

Since then, containers have gained widespread adoption and have become a fundamental building block in modern application development and deployment. They have revolutionized software delivery pipelines, enabling faster development cycles, improved resource utilization, and simplified deployment across diverse computing environments. The container ecosystem continues to evolve, with ongoing advancements in container runtimes, management tools, and security measures.

Types of Containers:

There are various container technologies available, each with its own unique features and use cases. And most of them were built on Linux Kernel. Here are some of the different types of containers:

1. LXC (Linux Containers): LXC is an operating system-level virtualization method that provides a lightweight and isolated environment for running multiple Linux distributions on a single host. LXC utilizes kernel features like cgroups and namespaces to achieve process isolation and resource

control.

2. LXD (Linux Container Daemon): LXD is a system container manager that builds upon LXC. It offers a more user-friendly and streamlined experience for managing LXC containers. LXD provides a REST API and a command-line interface to manage containers, supports live migration, and offers features like clustering and snapshotting.
3. LXCFS (Linux Containers Filesystem): LXCFS is a userspace file system specifically designed for LXC containers. It provides a virtualized view of the host system's resources to the containers, allowing containers to have their own /proc, /sys, and /dev file systems. This enables better resource isolation and improves compatibility with containerized applications.
4. Docker: Docker is a popular containerization platform that simplifies the creation, deployment, and management of containers. It introduced the concept of container images, which are portable, self-contained units that include application code, dependencies, and configurations. Docker provides a rich ecosystem of tools and services, including Docker Engine, Docker Compose, and Docker Swarm, for building and orchestrating containerized applications.
5. Podman: Podman is a container runtime and management tool that aims to be a compatible drop-in replacement for Docker. It allows users to run and manage containers without requiring a separate daemon process. Podman uses the same container image format as Docker and provides a command-line interface similar to Docker CLI.
6. rkt (pronounced "rocket"): rkt is a container runtime developed by CoreOS (now part of Red Hat). It focuses on security, simplicity, and composability. rkt supports the industry-standard App Container (appc) specification and provides features like image signing, secure bootstrapping, and integration with system-level services.

These are just a few examples of container technologies available in the ecosystem. Each technology has its own strengths and use cases, and the choice depends on factors such as specific requirements, familiarity, and compatibility with existing infrastructure.

Containers can be further divided by their application. Let's see the different types of containers from the application dimension.

1. **Application Containers**: These containers are specifically focused on packaging and running individual applications. They include all the necessary dependencies and libraries required for the application to run successfully. Application containers provide isolation and portability, allowing developers to build and deploy applications across different environments consistently.
2. **System Containers:** System containers, also known as OS containers or lightweight VMs, encapsulate an entire operating system within a container. They emulate a complete OS environment and allow running multiple instances of the same or different operating systems on a single host. System containers provide stronger isolation compared to application containers and are suitable for scenarios where full OS separation is required, such as running legacy applications or different distributions on the same machine.
3. **Data Containers:** Data containers are specialized containers designed to persistently store and manage data. They separate data from application containers, allowing easy sharing and persistence of data across different instances. Data containers are commonly used in database management systems and distributed file systems.
4. **Cloud Containers:** Cloud containers, or cloud-native containers, are containers designed to

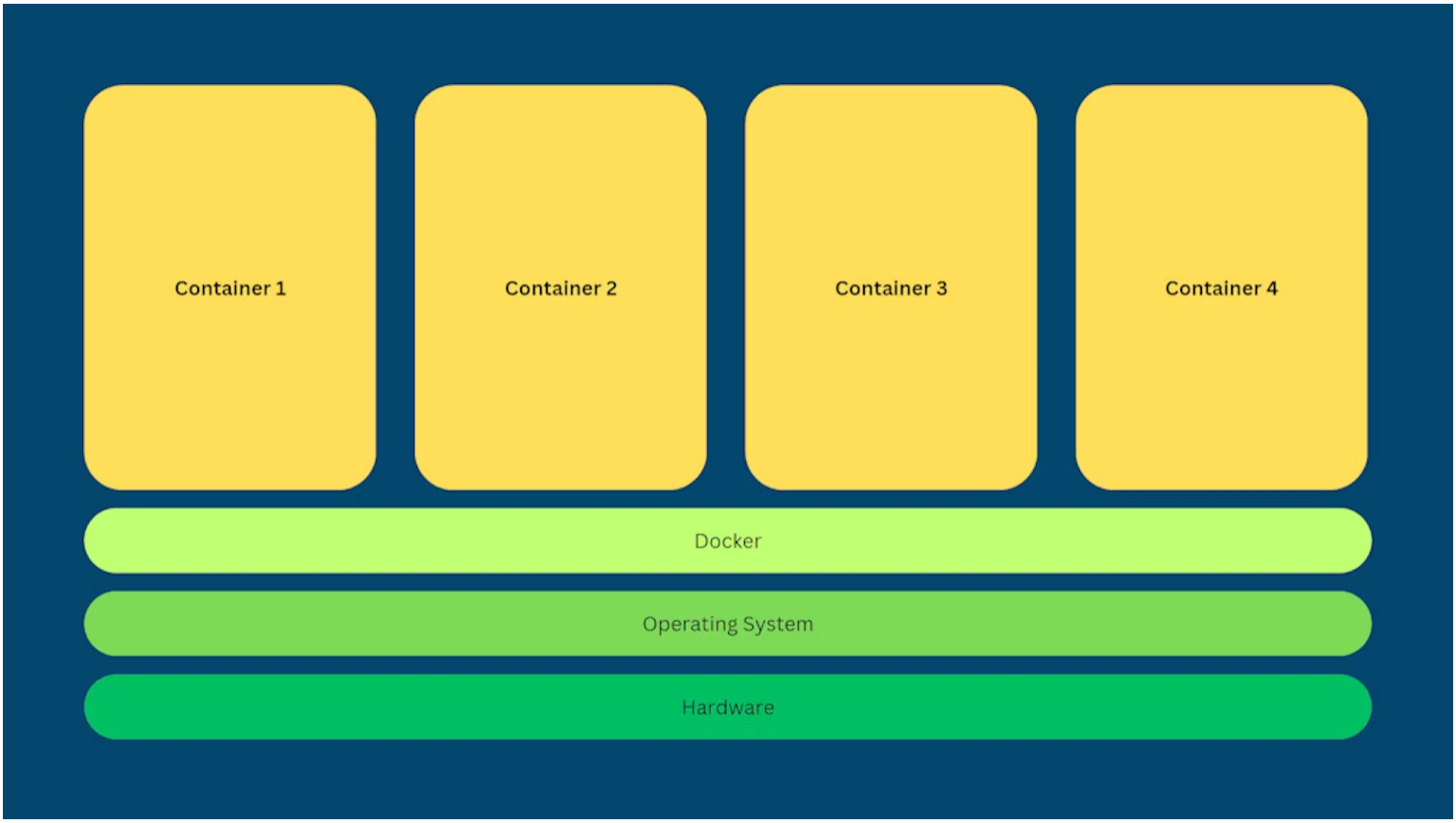
leverage cloud services and architectures fully. They are optimized for scalability, flexibility, and integration with cloud platforms. Cloud containers often make use of container orchestration systems like Kubernetes to manage container deployments across clusters of machines.

5. **Serverless Containers:** Serverless containers combine the benefits of containers and serverless computing models. They allow developers to deploy applications as event-driven functions or microservices without the need to manage the underlying infrastructure. Serverless containers automatically scale based on demand and execute functions in isolated, ephemeral containers.
6. **IoT Containers:** Internet of Things (IoT) containers are tailored for deploying applications on resource-constrained devices in IoT networks. These containers are optimized for low memory and processing power requirements, enabling efficient execution of applications on IoT devices.

What is Docker?

Docker is an open-source platform that enables developers to automate the deployment and management of applications within containers. It provides a standardized and efficient way to package applications and their dependencies into portable, self-contained units called container images.

If you read out previous sections, you could have learned the basic idea of the Docker. To say in one sentence, Docker is a facade of Containers. Docker provides user interfaces to create, configure, and manage Containers. By the way, Docker uses LXC (Linux Containers). In essence, a docker is a high-level tool that provides several functionalities to operate Containers.



At its core, Docker utilizes containerization technology to create isolated environments where applications can run consistently across different computing environments. Containers created with Docker are lightweight, fast to start, and share the host system's operating system kernel, while still

maintaining process isolation from one another.

Docker introduced the concept of container images, which are built from a set of instructions called a Dockerfile. These images encapsulate everything needed to run an application, including the code, runtime, libraries, and system tools. They provide a consistent environment, ensuring that an application behaves the same way regardless of where it is deployed.

With Docker, developers can easily package their applications along with their dependencies into container images and distribute them to any environment where Docker is installed. Docker provides a robust ecosystem of tools and services, such as Docker Compose for managing multi-container applications and Docker Swarm for orchestrating container clusters.

The benefits of using Docker include improved application portability, scalability, and efficiency. It allows developers to build, test, and deploy applications in a reproducible manner, speeding up the development process and reducing the “it works on my machine” problem. Docker also simplifies the management of complex distributed systems and enables seamless deployment across various environments, including local development machines, on-premises servers, and cloud platforms.

Relationship between Docker and Container

To better understand, let’s see how Docker and Container are interrelated. We hope this section will help you feel better about concepts. Docker and containers have a closely intertwined relationship. Docker is a platform that facilitates the creation, distribution, and management of containers. It builds upon containerization technology and provides a user-friendly interface and toolset to work with containers effectively.

Containers, on the other hand, are lightweight and isolated environments that encapsulate applications and their dependencies. They provide a consistent runtime environment for applications, ensuring that they run consistently across different computing environments. Containers achieve this by utilizing operating system-level virtualization and leveraging features like namespaces and cgroups.

Components of Docker

Docker leverages the underlying containerization technology to create and manage containers. It introduces several key components that simplify working with containers, including:

1. **Docker Engine:** This is the runtime that enables the creation and execution of containers. It provides the necessary tools and libraries to build and run containers on a host system. Docker Engine interfaces with the underlying operating system’s containerization capabilities, allowing the creation of isolated container environments.
2. **Docker Images:** Docker uses container images as a standardized format (Template) to package applications and their dependencies. Images are created from a set of instructions specified in a Dockerfile. They contain the application’s code, runtime, libraries, and configurations. Docker images serve as the building blocks for running containers.
3. **Docker Hub:** Docker Hub is a cloud-based registry where developers can publish, share, and

discover container images. It provides a vast repository of publicly available images, making it easy to access and utilize pre-built containers. Docker Hub allows users to collaborate and distribute container images across different environments.

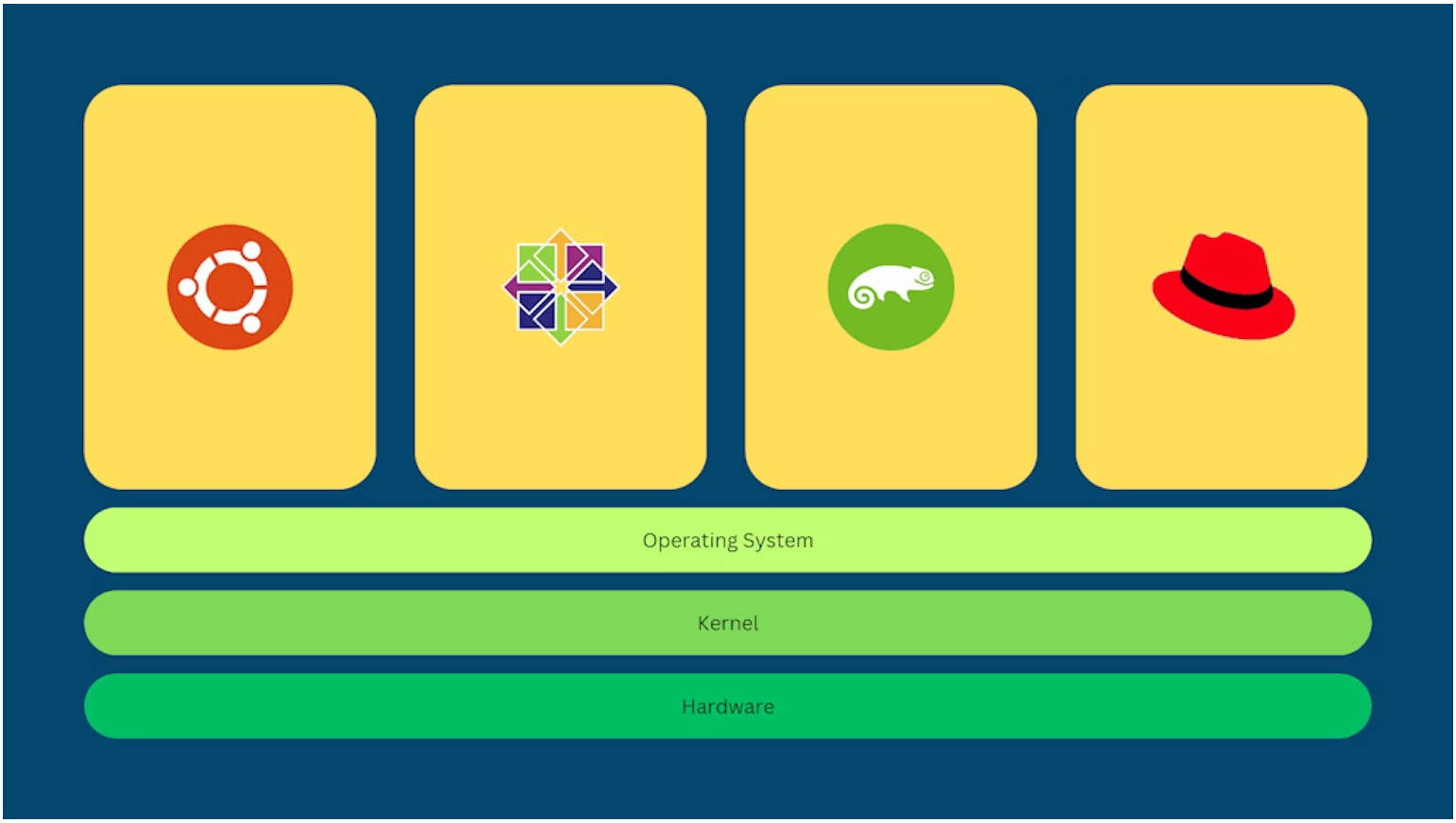
4. **Docker CLI:** Docker Command-Line Interface (CLI) is a command-line tool that allows users to interact with Docker and perform various container-related operations. It provides commands to build images, run containers, manage networks and volumes, and perform other essential tasks.

Docker simplifies the process of working with containers by providing an abstraction layer and a comprehensive toolset. It streamlines container creation, distribution, and management, making it accessible to a wider audience. While Docker is a prominent container platform, it is important to note that there are other containerization technologies and platforms available, each with its own features and benefits.

How Does Docker Work?

As you know, Containers, much like virtual machines, offer fully isolated environments where processes, services, network interfaces, and mounts can operate independently. However, the key distinction is that containers share the same operating system (OS) kernel. This sharing of the kernel ensures efficient resource utilization while maintaining isolation. Let's explore this in more detail.

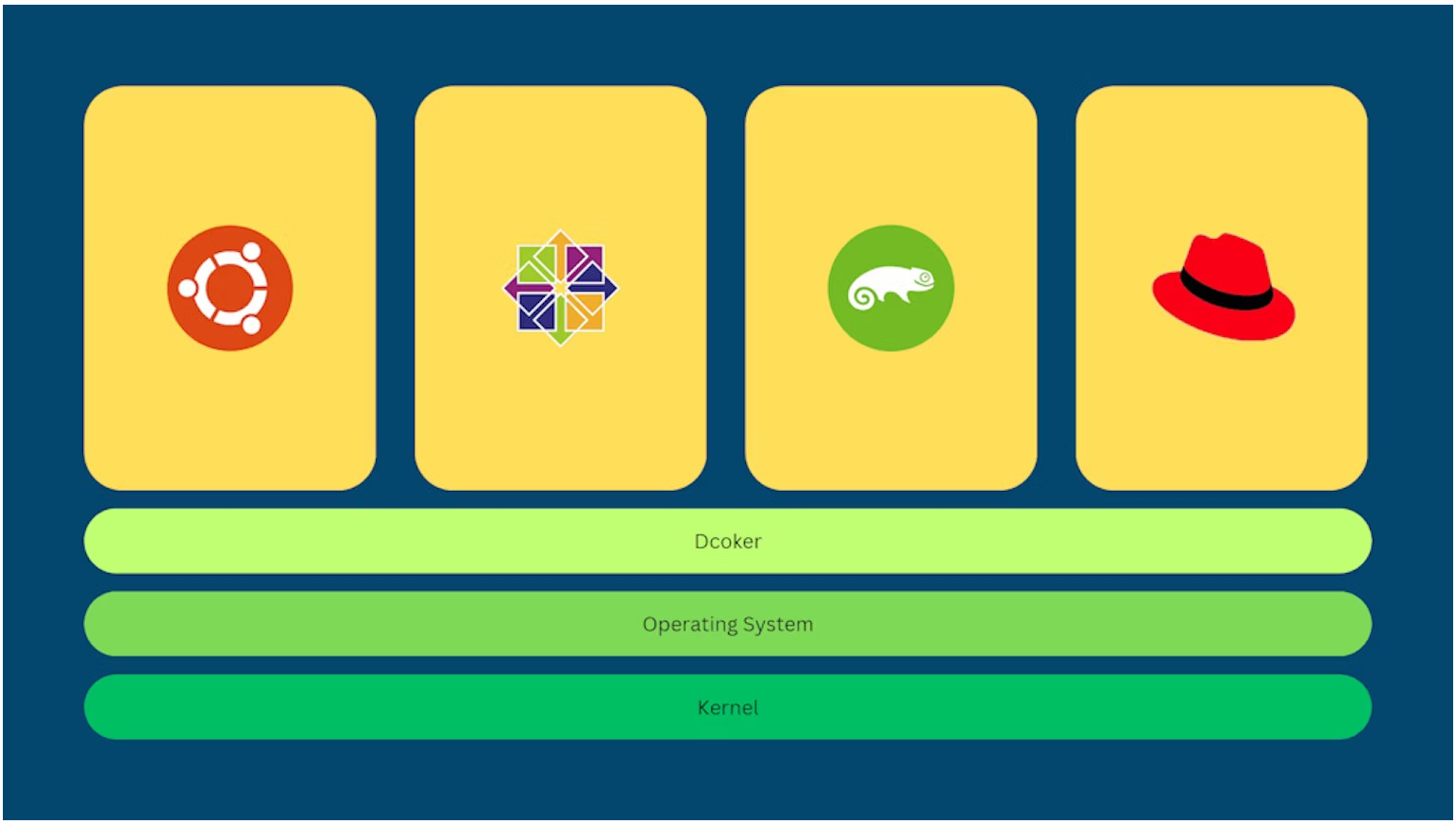
Docker Containers on Linux:



To grasp the significance of containerization, it's essential to revisit fundamental concepts of operating systems. Operating systems like Ubuntu, Fedora, Suse, or Centos consist of two primary components: the OS kernel and a set of software.

The OS kernel acts as an intermediary between the hardware and software layers. In the case of

Linux-based systems, the kernel remains the same across various distributions. It is the software layer above the kernel that distinguishes different operating systems, encompassing aspects such as user interfaces, drivers, compilers, file managers, and developer tools.

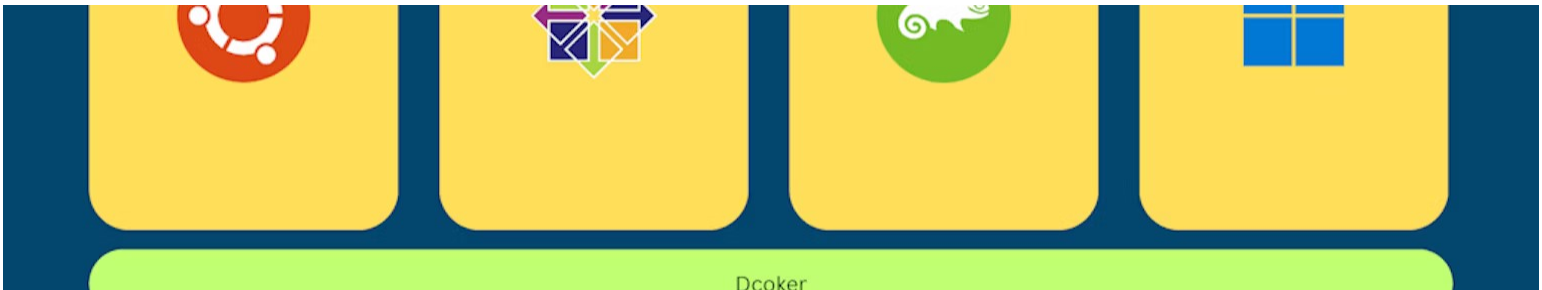


Suppose we have a system with Ubuntu as the underlying OS and Docker installed. Docker enables the execution of containers based on different OS distributions as long as they share the same Linux kernel.

For example, if the host OS is Ubuntu, Docker can run containers based on distributions like Debian, Fedora, Suse, or Centos. Each Docker container contains only the additional software that distinguishes the respective operating systems. The underlying kernel of the Docker host seamlessly interacts with all the supported OS distributions.

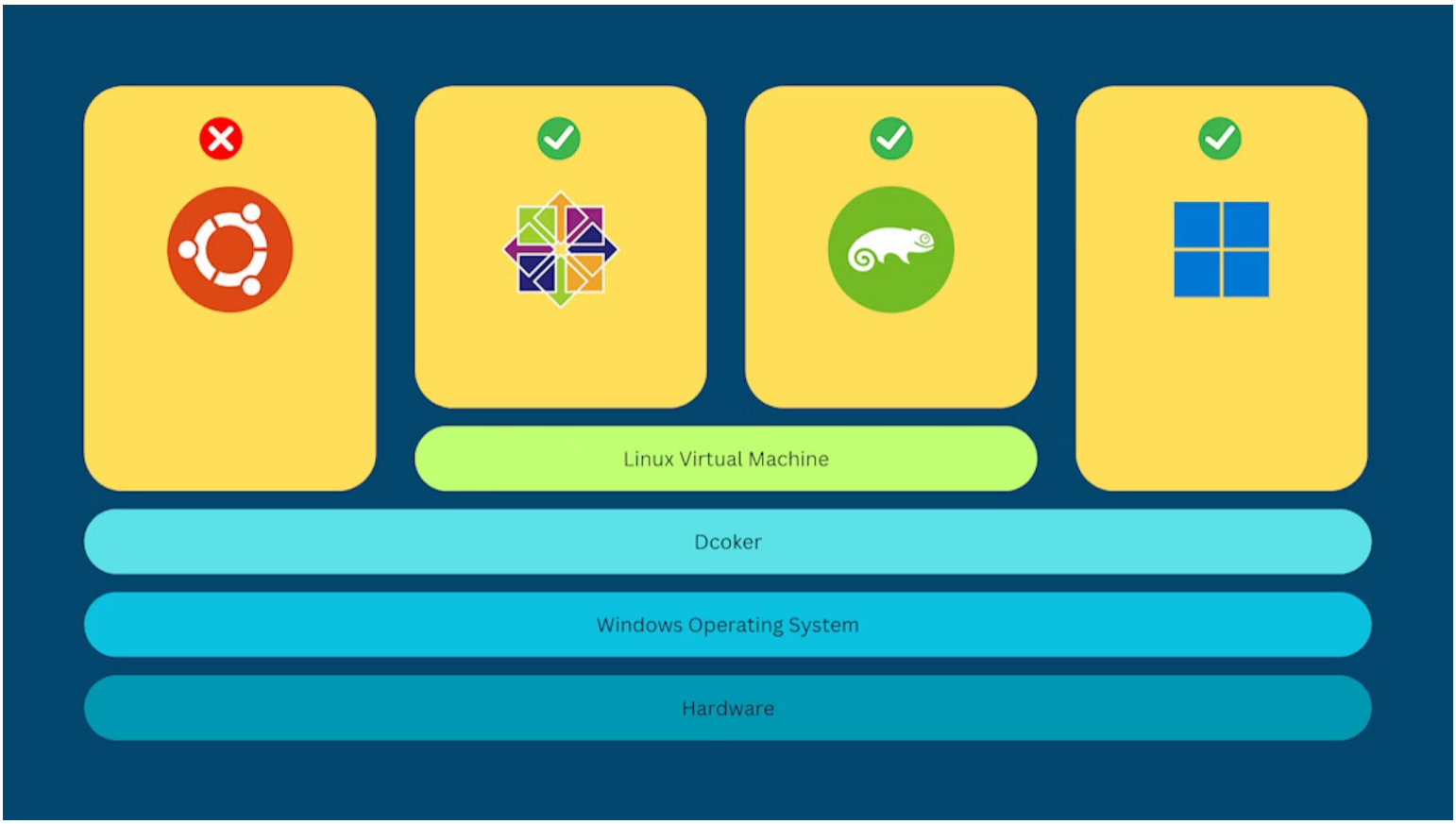
Docker Containers on Windows:

You don't see any difficulties to run Linux Containers on a Linux Docker, that's because Linux-based distributions share a common kernel. However, there are operating systems that do not have the same kernel as Linux, such as Windows. In the context of containerization, running Windows-based containers on a Docker host with a Linux kernel is not possible. To deploy Windows containers, a Docker installation on a Windows server is required.



It’s not uncommon for people to question the notion that Linux containers can run on a Windows environment. Some may even install Docker on Windows, run a Linux container, and claim that it works. However, the reality is that running a Linux container on Windows involves an underlying mechanism.

When Docker is installed on Windows and a Linux container is executed, it is not truly running on Windows. Instead, Windows operates a Linux container within a Linux virtual machine. In essence, it is a Linux container running on a Linux virtual machine within the Windows environment.



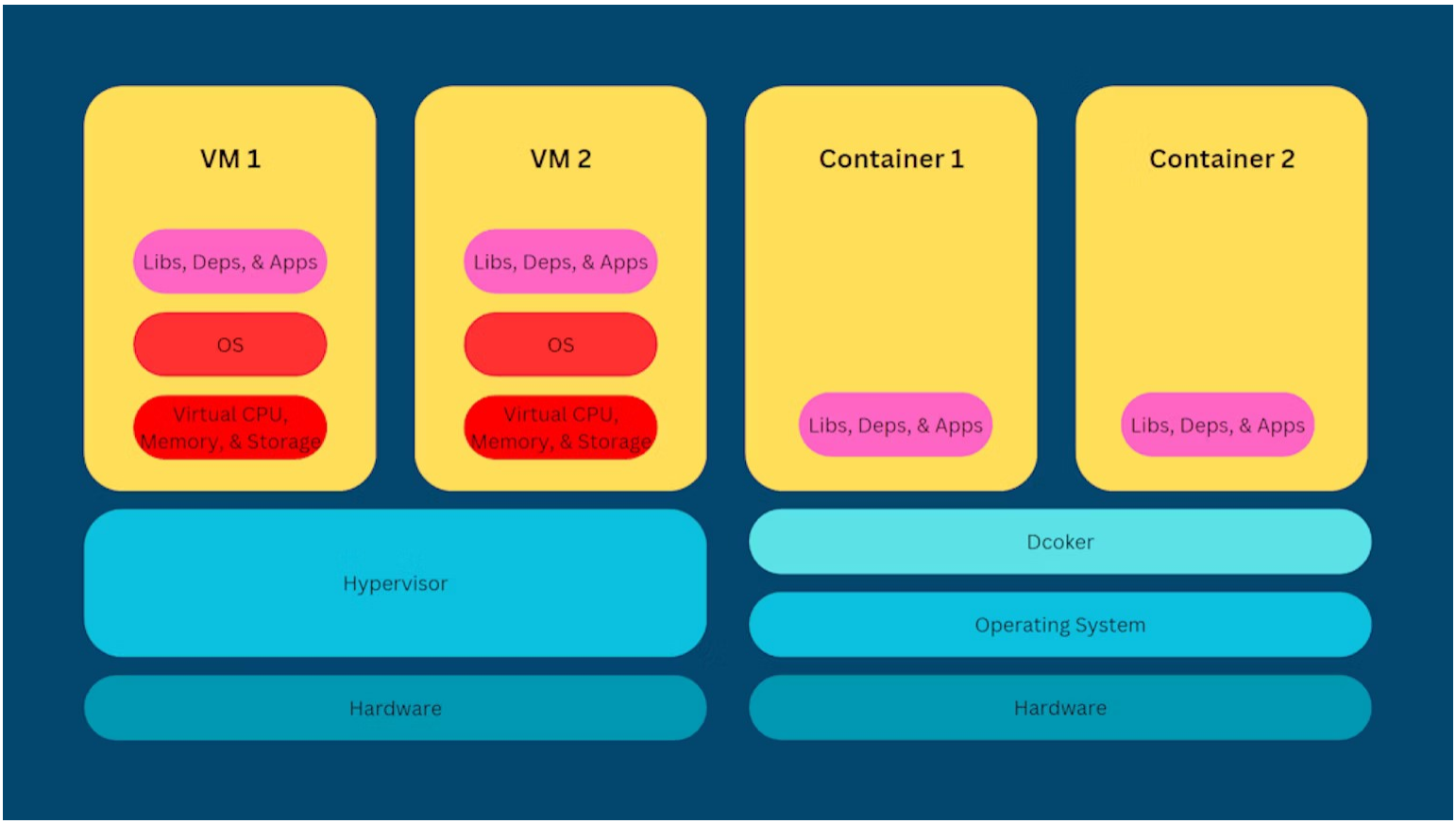
While it may seem like a disadvantage that Docker does not enable running different operating systems and kernels on the same hardware, it is essential to understand Docker’s primary purpose and its unique advantages.

Docker is not designed as a hypervisor to virtualize and execute diverse operating systems and kernels simultaneously. Instead, its core objective is to package and containerize applications, facilitating their seamless deployment and execution across various environments.

The true strength of Docker lies in its ability to encapsulate an application, along with its dependencies and configurations, into a container. This container can then be transported and executed anywhere, anytime, and in any quantity.

Containers vs Virtual Machines

People from Virtualization backgrounds may often end up in confusion. The problem is not with them, the problem lies in some similarities between both technologies. Containers and virtual machines (VMs) are both technologies used for the isolation and deployment of applications, but they differ in their approach and resource utilization. Let's do a comparison between containers and virtual machines:



Containers:

- Containers are lightweight and utilize operating system-level virtualization.
- They share the host system's operating system kernel, which makes them more efficient in terms of resource utilization.
- Containers have faster startup times and require less memory compared to virtual machines.
- They offer faster deployment and scaling capabilities.
- Containers provide process-level isolation, meaning each container runs in its own isolated environment while sharing the host's kernel.

- Container images contain only the necessary dependencies and libraries required for the application, making them smaller in size.
- Containers are well-suited for running microservices architectures and modern, cloud-native applications.
- They enable easier and more consistent application deployment across different environments.

Virtual Machines:

- Virtual machines, on the other hand, simulate the entire hardware stack and run a complete operating system.
- They require a hypervisor to emulate the virtual hardware and manage the guest operating systems.
- VMs are more resource-intensive as they allocate dedicated resources (CPU, memory, storage) to each instance.
- They have slower startup times compared to containers and require more memory due to the overhead of running a separate operating system.
- VMs provide stronger isolation between instances as they run independent operating systems.
- They are useful for running legacy applications, applications with specific OS requirements, or when complete isolation between instances is necessary.
- VMs offer more flexibility in terms of running different operating systems and configurations on the same physical hardware.

When and Why to Use Docker Containers?

Both Virtualization and Containers are powerful technologies, we should compare them with each other. They have their own pros and cons. It is good to utilize both in combine to reap the maximum output. Well, since this post is dedicated to Docker Container technology, let’s see when and why to use Docker Containers.

Docker containers are widely used in various scenarios and offer several benefits that make them a popular choice for application deployment. Here are some situations where using Docker containers is advantageous:

1. **Application Portability:** Docker containers provide a consistent runtime environment that can be easily replicated across different computing environments. This portability allows applications to run consistently on developer machines, testing environments, production servers, and even cloud platforms. Docker’s containerization approach ensures that the application and its dependencies are bundled together, reducing compatibility issues and the “it works on my machine” problem.
2. **Scalability and Resource Efficiency:** Docker containers offer efficient resource utilization and enable horizontal scaling. They are lightweight, start quickly, and require minimal memory, allowing for efficient utilization of system resources. Containers can be easily replicated and orchestrated using container orchestration platforms like Kubernetes or Docker Swarm, enabling applications to scale up or down based on demand.
3. **Rapid Deployment and Continuous Integration:** Docker containers facilitate rapid and consistent

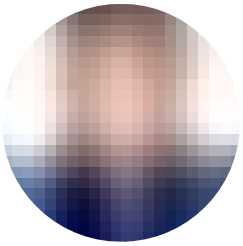
deployment of applications. The containerized application, along with its dependencies, is packaged into a single container image, which can be quickly deployed across different environments. Docker's standardized format for container images simplifies the integration of container deployment into continuous integration and continuous deployment (CI/CD) pipelines, enabling faster and automated release cycles.

4. **Microservices Architecture:** Docker is well-suited for building and deploying microservices-based architectures. Containers allow for modular application design, where each microservice runs in its own isolated container. This approach offers flexibility, scalability, and independent development and deployment of microservices. Docker containers make it easier to manage and orchestrate complex distributed systems composed of multiple interconnected services.
5. **Development Environment Consistency:** Docker containers ensure consistency between development, testing, and production environments. Developers can package their applications and dependencies into containers, ensuring that the runtime environment remains the same across different stages of the development lifecycle. This reduces configuration discrepancies, streamlines troubleshooting, and minimizes the risk of deployment issues due to environment differences.
6. **Collaboration and Reproducibility**: Docker simplifies collaboration between developers by providing a standardized container image format. Developers can share container images through registries like Docker Hub, making it easy to distribute and reproduce development environments. This enhances collaboration, facilitates knowledge sharing, and improves the reproducibility of software builds.

We hope this article helps Understand Docker Containers. We are going to end this post for now, we will cover more details in the upcoming sessions. Please keep visiting thesecmaster.com for more such technical information. Visit our social media page on Facebook, Instagram, LinkedIn, Twitter, Telegram, Tumblr, & Medium and subscribe to receive information like this.

You may also like these articles:

- Understand the Docker Architecture with TheSecMaster
- Where You Should Get Started with Docker- Community Edition (CE) vs Enterprise Edition (EE)
- Discover the Easiest Way to Install Docker on Ubuntu With This Step-by-Step Guide!
- Step-by-step Procedure to Install Docker Desktop on Linux
- Step-by-Step Guide: Installing Docker Desktop on macOS



Arun KL

Arun KL is a cybersecurity professional with 15+ years of experience in IT infrastructure, cloud security, vulnerability management, Penetration Testing, security operations, and incident response. He is adept at designing and implementing robust security solutions to safeguard systems and data. Arun holds multiple industry certifications including CCNA, CCNA Security, RHCE, CEH, and AWS Security.

Like it? Share it!

Login

Add a comment

M↓ MARKDOWN

ADD COMMENT

Powered by **Commento**

Recently added

Explore

View All