

Virtualization

Operating Systems 2024

Assoc. Prof. Milos Jovanovik, PhD



Универзитет „Св. Кирил и Методиј“ во Скопје
**ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО**



Agenda

- Introduction
- Virtualization
- Containerization
- Docker
 - Architecture
 - Ecosystem
 - Images
 - Containers
- Conclusion



Introduction to Virtualization

- Virtualization is a technique that combines resources from a logical perspective so that their utilization can be optimized.
- It is the process of creating a virtual version of something like computer hardware.
- Virtualization enables computers to be more efficient in a similar fashion.
- Computers that use virtualization optimize the available compute resources.
- Virtualization allows a single computer to host multiple virtual machines (VMs), each potentially running a completely different operating system.
- Virtualization is the “layer” of technology that goes between the physical hardware of a device and the operating system to create one or more copies of the device.



What is a Virtual Machine (VM)?



- Most of the virtualization techniques allow building and running virtual machines (VMs).
- Virtualization creates virtual hardware by cloning physical hardware.
- Each VM runs in an isolated environment on physical hardware and behaves like a full-fledged computer system with dedicated components.
- In a VM, an operating system with applications can run just like on a physical computer and applications running in a VM do not notice this.
- A VM is a set of files.
- With a hypervisor and VMs, one computer can run multiple OS simultaneously.
- Requests from the operating system instances are transparently intercepted by the virtualization software, and converted for the existing physical or emulated hardware.



What is a Hypervisor?

- Software installed on top of hardware that creates a virtualization layer.
- Hosts VMs
- Type 1 Hypervisor – Bare metal hypervisor (VMware ESXi)
- Type 2 Hypervisor – Hosted hypervisor (VMware Workstation)



Virtual Machine Files

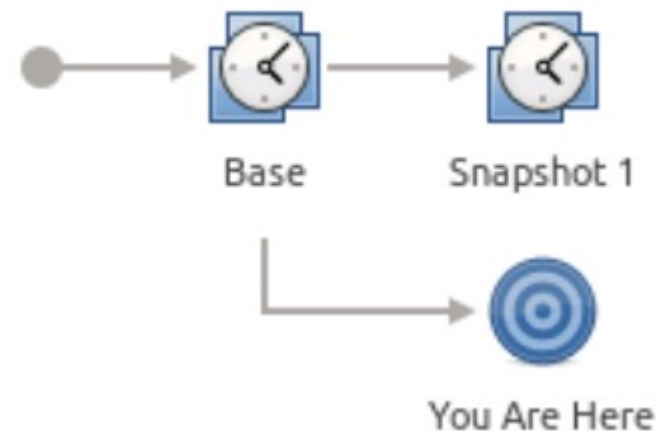
- VMs can be exported and moved to other hosts
- Files are created by the hypervisor and stored in a directory
- Example VM files:

File Type	File Name	Description
Log File	<vmname>.log	Keeps a log of VM activity
Disk File	<vmname>.vmdk	Stores content of VM's disk drive
Snapshot Files	<vmname>.vmsd and <vmname>.vmsn	Stores information about VM snapshots (saved VM state)
Configuration File	<vmname>.vmx	Stores information about VM name, BIOS, guest OS, and memory



What is a Snapshot?

- Working on a VM and need to save progress or state
- Snapshots are saved as files in the VM folder (.vmx)
- What is saved by a snapshot?
 - State of VM disks
 - Contents of VM memory
 - VM settings



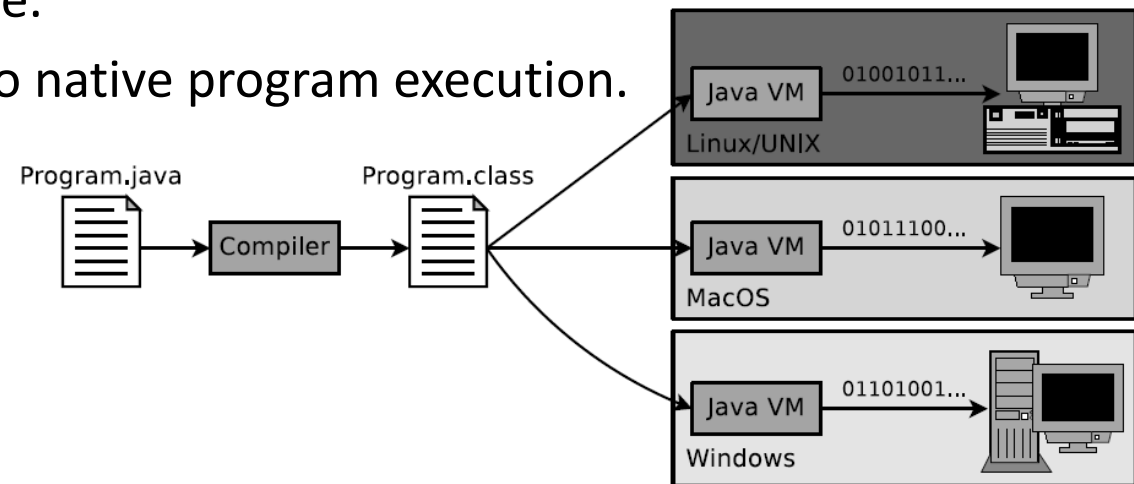
Partitioning

- If *partitioning* is used, the total amount of resources can be split to create subsystems of a computer system.
- Each subsystem can contain an executable operating system instance and can be used in the same way as an independent computer system.
- The resources (CPU, main memory, storage, . . .) are managed by the firmware of the computer that allocates them to the virtual machines. No additional software is required for the implementation of the virtualization functionality.
- Several hundred to thousands of Linux instances can be run simultaneously on a modern mainframe computer.
- Current x86-compatible CPUs with extensions for virtualization applications, such as Intel Vanderpool (VT-x) and AMD Pacifica (AMD-V), support only the partitioning of the CPU itself and not of the entire system.



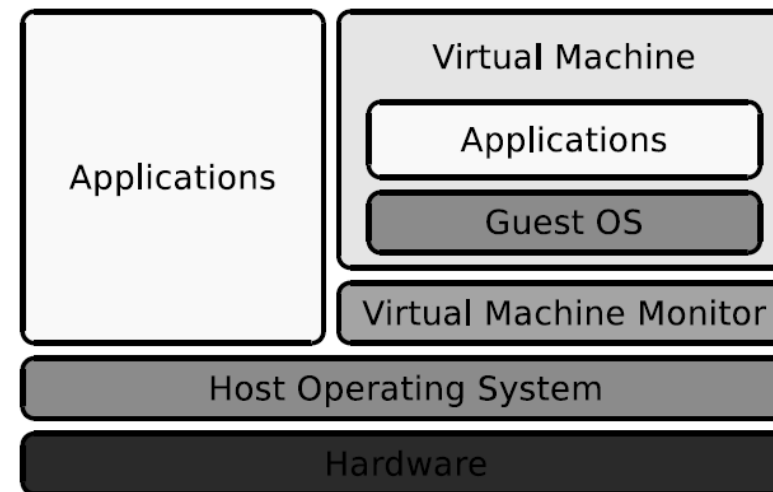
Application Virtualization

- In application virtualization, individual applications run in a virtual environment that provides all the components the application needs.
- The virtual machine is located between the application and the operating system.
 - Java Virtual Machine (JVM).
- Advantage: Provides platform independence.
- Drawback: Lower performance compared to native program execution.



Full Virtualization

- Full virtualization software solutions offer each virtual machine a complete virtual PC environment, including its own BIOS.
- Each guest operating system has its own virtual machine with virtual resources like CPU(s), main memory, storage devices, network adapters, etc.
- The heart of the solution is a so-called Virtual Machine Monitor (VMM) that runs hosted as an application in the host operating system
 - Type-2 hypervisor
- The purpose of the VMM is to allocate hardware resources to virtual machines.
 - Some hardware components are emulated (network adapters).
- Kernel-based Virtual Machine (KVM), Mac-on-Linux, Microsoft Virtual PC, Oracle VirtualBox, VMware Server, VMware Workstation, etc.

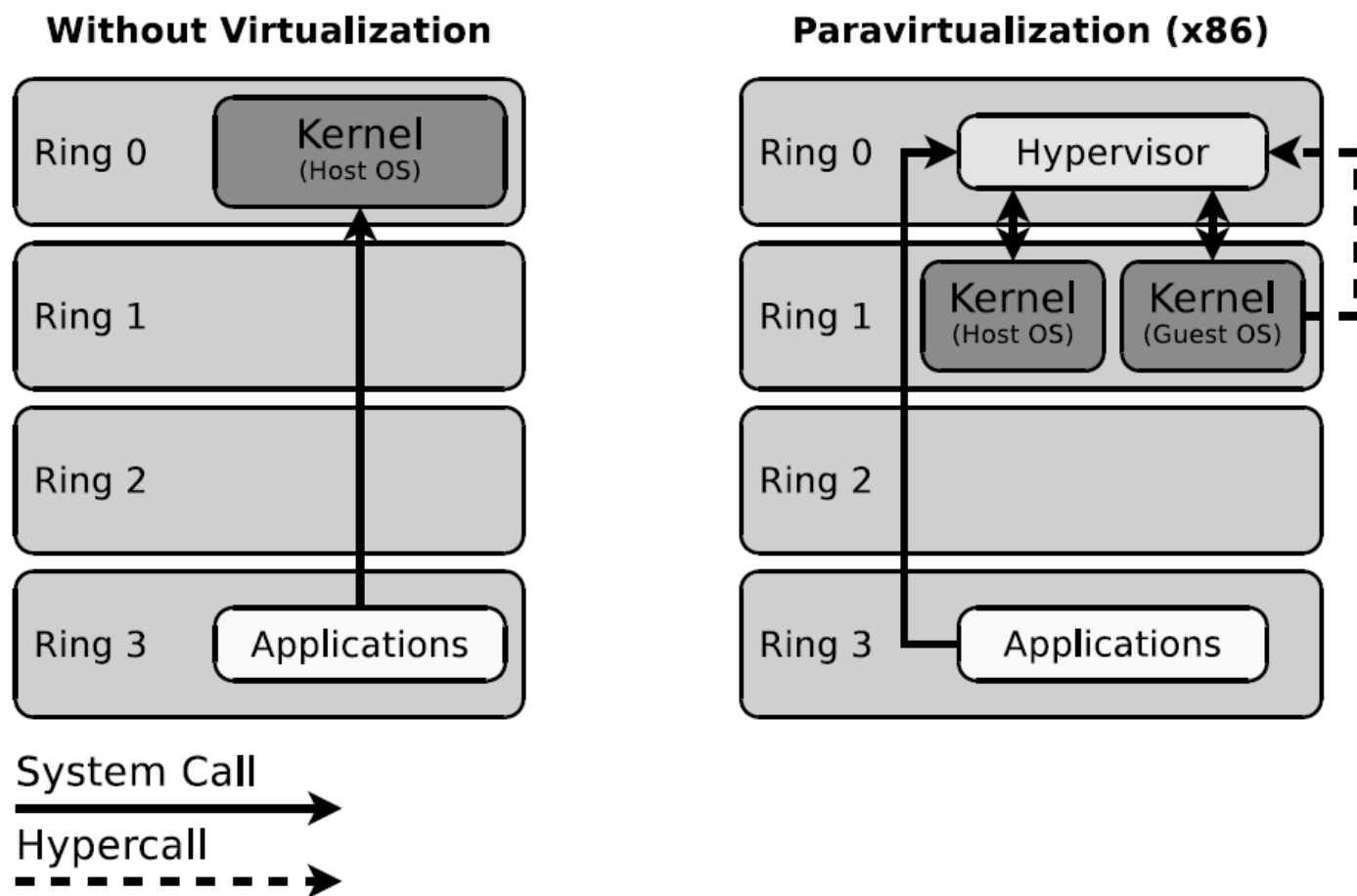


Paravirtualization

- In paravirtualization, the guest operating systems use an abstract management layer, the hypervisor, to access the physical resources.
 - Type-1 hypervisor
- The hypervisor distributes the hardware resources among the guest systems in the same way as an operating system does it among the running processes.
- The hypervisor runs in the privileged Ring 0.
- A host operating system is mandatory because of the device drivers and runs in less privileged Ring 1.
- Since the kernel of the host operating system can no longer execute privileged instructions due to its position in ring 1, the hypervisor provides hypercalls.
 - When an application requests a system call, a replacement function in the hypervisor is called.
 - The hypervisor then orders the execution of the corresponding system call at the kernel of the host operating system.
 - Requires kernel modifications in the guest OS (only possible for free software OS)!
- Xen, Citrix Xenserver, Virtual Iron, and VMware ESX Server.



Paravirtualization

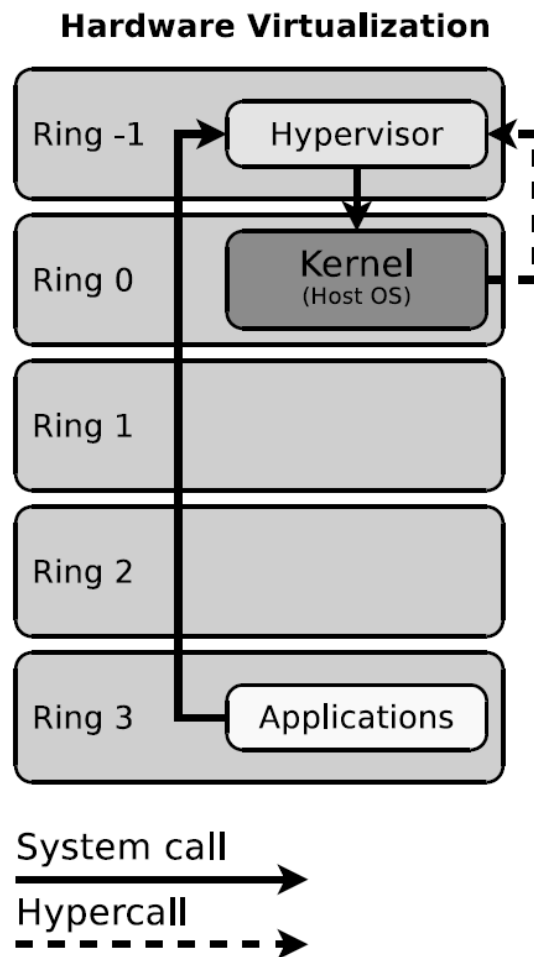


Hardware Virtualization

- Up-to-date x86-compatible CPUs from Intel and AMD implement extensions for hardware virtualization.
- Unmodified operating systems can be used as guest operating systems.
- The extension resulted in a modification of the privilege levels by adding ring -1 for the hypervisor.
- The hypervisor or VMM runs in ring -1, and at all times has total control over the CPU and the other hardware resources, because ring -1 has a higher privilege than ring 0.
- The virtual machines run in ring 0 and are called Hardware Virtual Machine (HVM) in such a scenario.
- An advantage of hardware virtualization is that guest operating systems do not need to be modified.
 - Proprietary operating systems, such as Windows, can be run as guest operating systems.
- Some examples of virtualization products that support hardware virtualization are Xen since version 3, Windows Server since version 2008 (Hyper-V), VirtualBox and KVM.



Hardware Virtualization

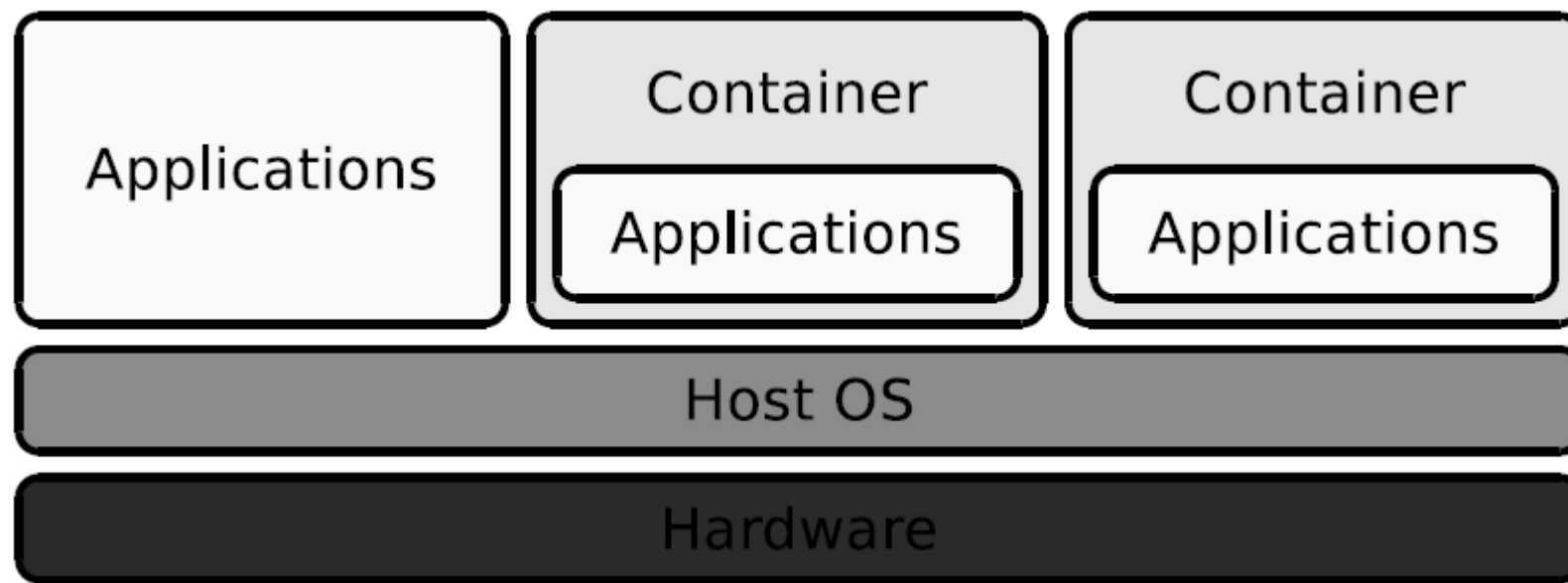


Operating System-level Virtualization: Containerization

- With operating system virtualization, several identical system environments can be run under the same kernel, isolated from each other.
 - Containers or jails
- When launching a virtual machine, in contrast to full virtualization, paravirtualization, and emulation, no additional operating system is started. Instead, an isolated runtime environment is created.
 - All containers use the same kernel.
- Applications that run in a container can only get in touch with applications within the same container.
- One advantage of this virtualization concept is the reduced overhead because the kernel manages the hardware as usual.
- This virtualization concept is particularly useful in situations where applications need to run in isolated environments with high security.
 - Or, automated installation of sophisticated application software, such as a web server or a database, without having to consider package dependencies.
- **Docker**, Solaris, OpenVZ, FreeBSD, Virtuozzo...



Operating System-level Virtualization: Containerization





Containerization

- Segregation of Duties in DevOps
 - **Developers**: care about their apps running inside containers;
 - **Operators**: care about managing the containers;
- Efficient Development Life Cycle
 - Reducing the time between code being written, and code being tested, deployed and used;
 - Makes apps portable and easy to build;
- Encourages SOA
 - Encourages service-oriented & microservices architectures;
 - Easier distribution, scaling and debugging;





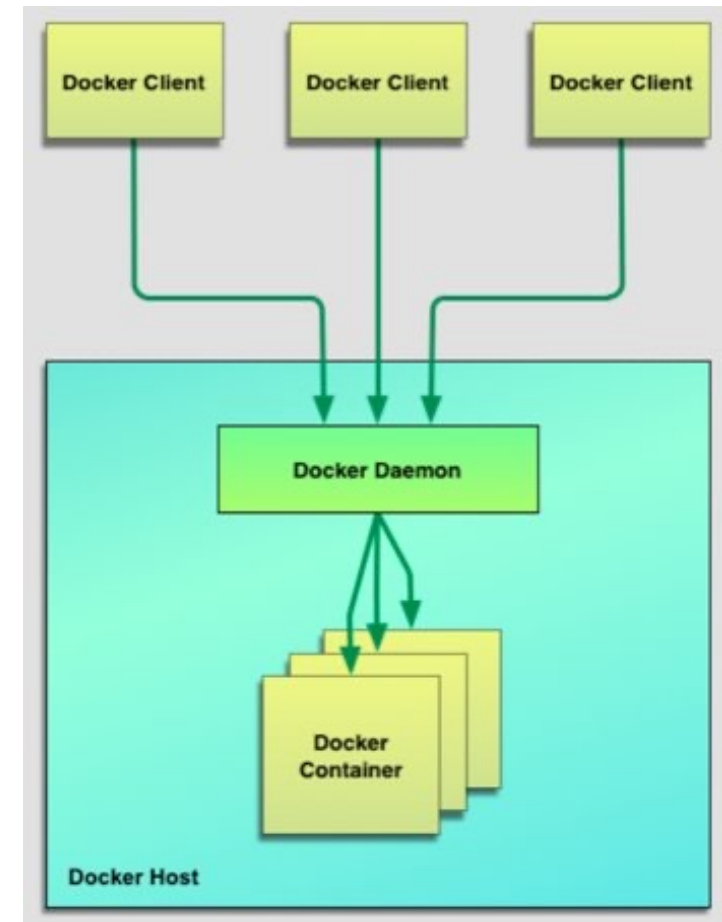
Containerization

- Containers
 - Run in user space in the OS --> OS-level virtualization;
 - Lean technology, limited overhead – they use the OS's system call interface;
- Docker
 - Open-source engine that automates the deployment of applications into containers;
 - Provides strong container isolation, their own network, storage stacks and resource management capabilities;
 - Adds an application deployment engine on top of a virtualized container execution environment;
 - Lightweight, simple and fast;
 - Does not require an emulator or hypervisor layer – uses the OS's normal system call interface;
 - Reduces the overhead required to run containers and allows a greater density of containers to run on a host.



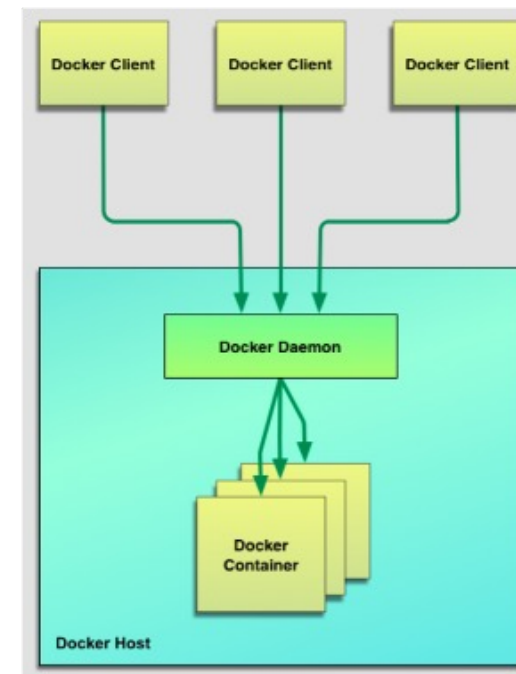
Docker Architecture

- Docker Components:
 - Docker Engine (Docker client and server);
 - Docker Images;
 - Registries;
 - Docker Containers;
- Docker Engine
 - Docker client talks to the Docker server (client-server app);
 - Docker server a.k.a. Docker Engine, or Docker daemon;
 - CLI binary: docker;



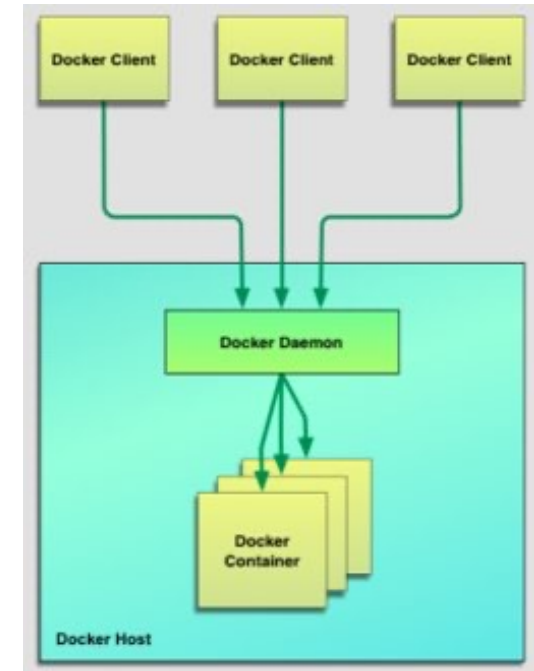
Docker Architecture (2)

- Docker Images
 - The building blocks of the Docker world;
 - Layered format, built step-by-step, e.g.:
 - Add a file;
 - Run a command;
 - Open a port;
 - The “source code” of Docker containers;
- Registries
 - The place Docker stores the Docker images;
 - Public and private;
 - Docker Hub: <https://hub.docker.com>;



Docker Architecture (3)

- Docker Containers
 - Container = package for your app / service;
 - You launch a container from a Docker image;
 - Image = building / packing aspect;
 - Container = running / execution aspect;
 - It can be created, started, stopped, restarted, destroyed;
 - Docker doesn't care about what's inside a container;
 - Docker doesn't care about where you ship your container;
- Docker network
 - Docker networks are virtual networks used to connect multiple containers.
 - They allow containers to communicate with each other and with the host system.



Docker Architecture (4)

- Dockerfile
 - The Dockerfile is a script with instructions on how to make a Docker image.
 - Contains information about the operating system, languages, environment variables, file locations, network ports, and other details needed to run the image.
 - The commands in the file are placed in groups, and those groups are automatically executed.
- Docker Compose
 - Docker Compose is a tool used to define and run multi-container Docker applications.
 - It allows users to define the services that make up their application in a single file.
 - Docker Compose makes constructing, running and managing entire applications very easy, regardless of the number of containers they consist of.
 - You can start them at once, stop them, restart them, view their logs in a single place, etc.
- Docker Swarm
 - Docker Swarm is an optional orchestration service that can be used to schedule Docker containers and operate them across a cluster of servers.
 - Kubernetes - well-known alternative to Docker Swarm





The Docker Ecosystem

- Docker (Docker Engine)
 - The core technology behind Docker;
 - Open-source software that runs on Linux, as a daemon;
 - Makes it possible to run containers on top of Linux kernel;
- Docker CLI
 - The CLI that developers usually use to interact with the Docker Engine, via `docker` and `docker-compose` commands;
 - It is open-source software, as well;



The Docker Ecosystem (2)

- Docker Desktop
 - Closed-source software, which provides a full environment for working with Docker technologies: containers, images, volumes, etc.;
 - Provides Windows and macOS users with a Linux VM to run Docker on, out-of-the-box;
- Docker Hub
 - A public hosted repository service, for storing, finding and sharing Docker images;
- Docker, Inc.
 - USA-based company, which owns Docker Desktop and Docker Hub;



Docker Images

- A Docker image is a read-only template containing a set of instructions for creating a container that can run on the Docker platform.
- It provides a convenient way to package up applications and preconfigured server environments, which you can use for your own private use or share publicly with other Docker users.
- Made up of a collection of files, such as **installations**, **application code**, and **dependencies**.
- Docker Image can be created by using one of two methods:
 - Interactive - By running a container from an existing Docker image, manually changing that container environment through a series of live steps, and saving the resulting state as a new image.
 - Dockerfile - By constructing a plain-text file, known as a Dockerfile, which provides the specifications for creating a Docker.

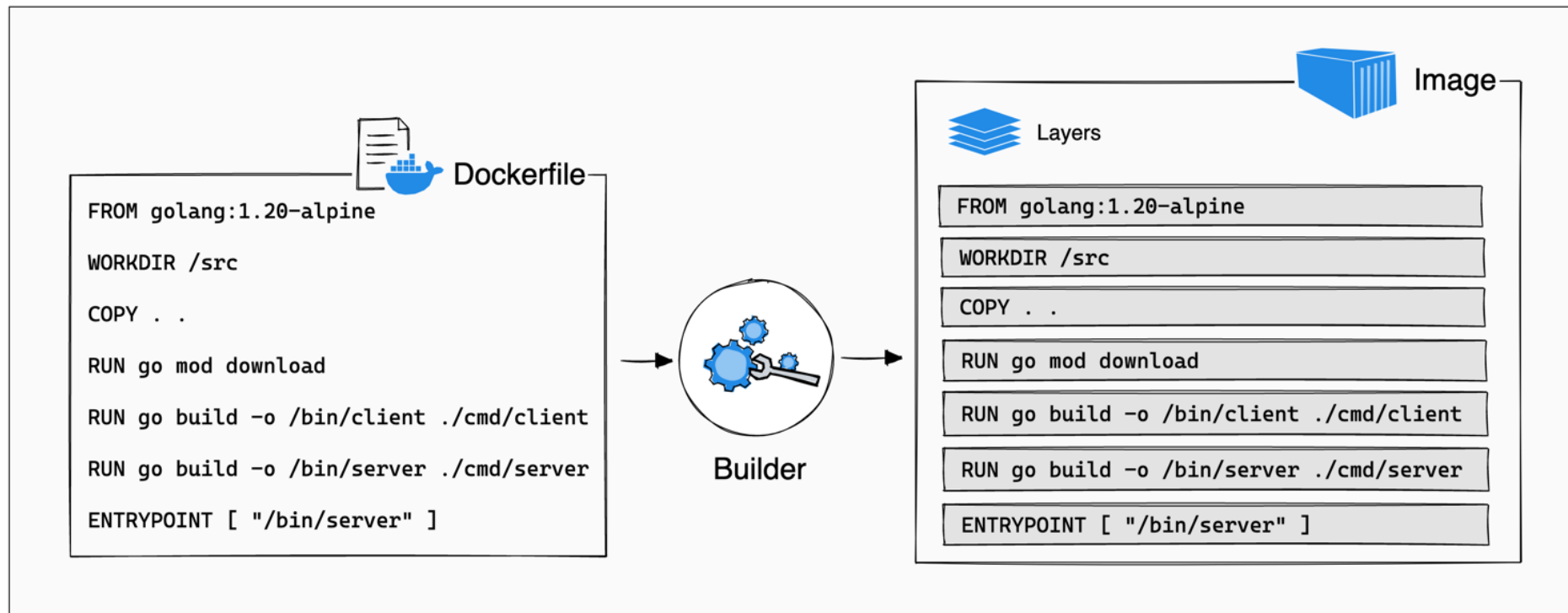


Docker Layers


- Each of the files that make up a Docker image is known as a layer.
- These layers form a series of intermediate images, built one on top of the other in stages, where each layer is dependent on the layer immediately below it.
- The hierarchy of your layers is key to efficient lifecycle management of your Docker images.
 - Best practice: Organize layers that change most often as high up the stack as possible. Saves time during rebuilding.
- Container layer
 - Each time Docker launches a container from an image, it adds a thin writable layer, known as the container layer, which stores all changes to the container throughout its runtime.
- Parent Image
 - The first layer of a Docker image is known as the “parent image”.




Docker Layers



Docker Layer: Optimization

 Layers	Cache?
FROM golang:1.20-alpine	✓
WORKDIR /src	✓
COPY . .	✗
RUN go mod download	✗
RUN go build -o /bin/client ./cmd/client	✗
RUN go build -o /bin/server ./cmd/server	✗
ENTRYPOINT ["/bin/server"]	✗



 Layers	Cache?
FROM golang:1.20-alpine	✓
WORKDIR /src	✓
* COPY go.mod go.sum .	✓
RUN go mod download	⊙✓
COPY . .	✗
RUN go build -o /bin/client ./cmd/client	✗
RUN go build -o /bin/server ./cmd/server	✗
ENTRYPOINT ["/bin/server"]	✗

Cached Layers

- When you run a build, the builder attempts to reuse layers from earlier builds.
- If a layer of an image is unchanged, then the builder picks it up from the build cache.
- If a layer has changed since the last build, that layer, and all layers that follow, must be rebuilt.

Layers	Cache?
FROM golang:1.20-alpine	✓
WORKDIR /src	✓
COPY . .	✗
RUN go mod download	✗
RUN go build -o /bin/client ./cmd/client	✗
RUN go build -o /bin/server ./cmd/server	✗
ENTRYPOINT ["/bin/server"]	✗



Docker Commands: Container Management

<code>docker ps</code>	List the running containers
<code>docker ps -a</code>	List all the containers, both running and stopped.
<code>docker create [image]</code>	Create a container without starting it.
<code>docker create -it [image]</code>	Create an interactive container with pseudo-TTY
<code>docker rename [container] [new-name]</code>	Rename a container
<code>docker rm [container]</code>	Remove a stopped container.
<code>docker rm -f [container]</code>	Force remove a container, even if it is running
<code>docker logs [container]</code>	View logs for a running container



Docker Commands: Running a Container

<code>docker run [image] [command]</code>	Run a command in a container based on an image.
<code>docker run -name [container-name] [image]</code>	Create, start and name a container.
<code>docker run -p [host]:[container-port] [image]</code>	Map a host port to a container port.
<code>docker run -rm [image]</code>	Run a container and remove it after it stops.
<code>docker run -d [image]</code>	Run a detached (background) container.
<code>docker run -it [image]</code>	Run an interactive process, e.g. a shell in a container.
<code>docker start [container]</code>	Start a container.
<code>docker stop [container]</code>	Stop a container
<code>Docker restart [container]</code>	Restart a container.
<code>docker exec -it [container] [shell]</code>	Run a shell inside a running container.



Docker Commands: Image Management

<code>docker build [dockerfile-path]</code>	Create an image from a Dockerfile
<code>docker build .</code>	Build an image using the files from the current path.
<code>docker build -t [name]:[tag] [location]</code>	Create an image from a Dockerfile and tag it.
<code>docker build -f [file]</code>	Specify a file to build from.
<code>docker pull [image]</code>	Pull an image from registry.
<code>docker push [image]</code>	Push an image to a registry.
<code>docker commit [container] [new-image]</code>	Create an image from a container.
<code>docker images</code>	Show all locally stored top level images.
<code>docker rmi [image]</code>	Remove an image.
<code>docker images prune</code>	Remove unused images.



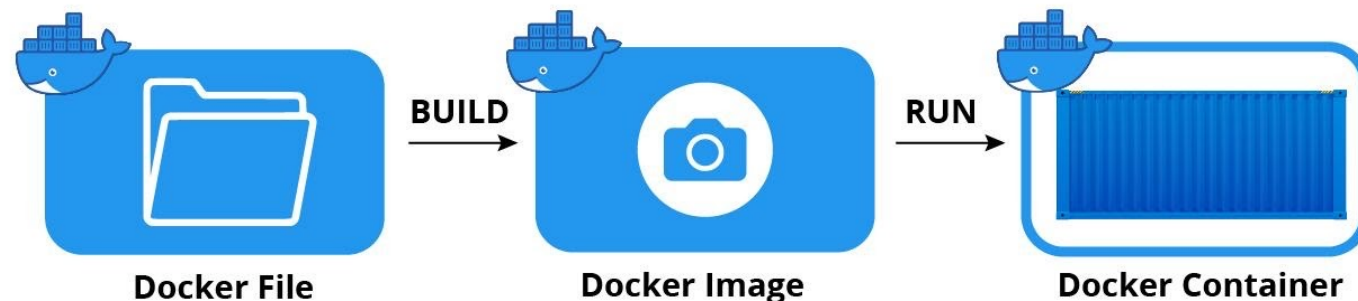
How to Build a Docker Image

- Interactive

- Advantages: Quickest and simplest way to create Docker images. Ideal for testing, troubleshooting, determining dependencies, and validating processes.
- Disadvantages: Difficult lifecycle management, requiring error-prone manual reconfiguration of live interactive processes. Easier to create unoptimized images with unnecessary layers.

- Dockerfile

- Advantages: Clean, compact and repeatable recipe-based images. Easier lifecycle management and easier integration into continuous integration (CI) and continuous delivery (CD) processes. Clear self-documented record of steps taken to assemble the image.
- Disadvantages: More difficult for beginners and more time consuming to create from scratch.



Interactive Method: Example

- Use the following Docker run command to start an interactive shell session with a container launched from the Docker Hub debian:11-slim image:

- `$ docker run -it debian:11-slim bash`

- Install the nginx server:

- `$ apt-get update && apt-get install -y nginx`

- Exit the interactive mode:

- `Ctrl + D`

- List all docker containers (including stopped):

- `$ docker ps -a`

output:

CONTAINER ID	IMAGE	COMMAND NAMES	CREATED	STATUS	PORTS
61d961dfdf7b	debian:11-slim	"bash"	About a minute ago	Exited (0)	6 seconds ago

- Save the image using the docker commit command, specifying either the ID or name of the container from which the image should be created:

- `$ docker commit 61d961dfdf7b debian_nginx`

- docker images command to see the image you've just created:

- `$ docker images`

output:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
debian_nginx	latest	538a64303a41	8 seconds ago	164MB

Dockerfile Method: Dockerfile Statements

Command	Purpose
FROM	To specify the parent image.
WORKDIR	To set the working directory for any commands that follow in the Dockerfile.
RUN	To install any applications and packages required for your container.
COPY	To copy over files or directories from a specific location.
ADD	As COPY, but also able to handle remote URLs and unpack compressed files.
ENTRYPOINT	Command that will always be executed when the container starts. If not specified, the default is <code>/bin/sh -c</code>
CMD	Arguments passed to the entrypoint. If ENTRYPOINT is not set (defaults to <code>/bin/sh -c</code>), the CMD will be the commands the container executes.
EXPOSE	To define which port through which to access your container application.
LABEL	To add metadata to the image.

Dockerfile Method: Example

Dockerfile content

```
# Use the official debian:11-slim as base
FROM debian:11-slim
# Install nginx and curl
RUN apt-get update && apt-get upgrade -y && apt-get install -y nginx curl
&& rm -rf /var/lib/apt/lists/*
```

Use the docker build command to create the image from the Dockerfile.

Use the -t flag to set an image name and tag:

```
$ docker build -t my-nginx:0.1 .
```

Use the docker images command to see the new image:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
my-nginx	0.1	f95ae2e1344b	10 seconds ago	164MB



Conclusion (Virtualization) ...

- **Enhanced Resource Utilization:** Enables more efficient use of hardware resources by allowing multiple virtual machines to run on a single physical server.
- **Isolation and Security:** Provides strong isolation between virtual machines, enhancing security by preventing the spread of vulnerabilities and ensuring better containment of potential threats.
- **Flexibility and Scalability:** Allows for easy scaling of resources, provides flexibility in adapting to changing workloads.
- **Cost Savings:** By consolidating multiple workloads on a single server, hardware costs, power consumption, and overall infrastructure expenses are reduced.
- **Disaster Recovery and Backup:** Facilitates efficient disaster recovery strategies by enabling the creation of snapshots and backup copies of virtual machines.
- **Improved Maintenance and Upgrades:** Simplifies maintenance tasks and system upgrades by allowing for the easy migration of VMs between physical servers, minimizing downtime and disruption.



... Conclusion (Virtualization)

- **Compatibility and Legacy Support:** Enables the coexistence of different operating systems and legacy applications on the same physical hardware.
- **Resource Partitioning:** Allows for the partitioning of resources, such as CPU, memory, and storage, ensuring fair and efficient allocation among VMs for optimal performance.
- **Cloud Integration:** Serves as a foundational technology for cloud computing, enabling the creation and management of virtual instances in cloud environments for enhanced scalability and flexibility.
- **Enhanced Development and Testing:** Provides a controlled and replicable environment for software development and testing, without the need for additional physical hardware.





Conclusion (Docker)

- Docker is a powerful tool for creating, deploying, and managing containerized applications in a wide range of environments.
- Improves consistency, portability, and scalability in software development and deployment processes.
- Provides a flexible and efficient way to package applications and their dependencies.
 - Easier to move code between development, testing, and production environments;
- Large and active community of users and contributors.
- A highly important tool for both DevOps engineers and software developers.





Questions?

