Training on

Micro services Spring boot and Docker

MOHAMMED LUQMAN SHAREEF

Day 4

- Application Scalability in Microservice
- Spring Security with Microservice - JWT Authentication
- Introduction to Docker file implementation in microservice.
- Hands-On

Cloud Native Patterns

Deploying multiple services instances

Service Registry & Discovery

Externalized Configuration

Auto-Scaling
- Horizontal
- Vertical

Load balancing and Routing
- Server Side
- Client Side

API Gateway

Distributed tracing

Application Scalability in Microservices

Scalability

What is Scalability?

- The ability of an application to handle increased load by adding resources.

Two types of scalability:

- Vertical
- Horizontal

Why is Scalability Important in Microservices?

- Microservices architecture promotes decentralized, independently scalable services.

Challenges in Scaling Monolithic Applications

Monolithic apps require scaling the entire application, even if only one module needs it.

Resource contention across modules can impact the entire system.

Difficult to isolate performance bottlenecks and failures.

How Microservices Address Scalability

Independent services: Each microservice can be scaled separately based on its demand.

Resource efficiency: Resources can be allocated dynamically where needed.

Technology flexibility: Each service can use the most appropriate tech stack.

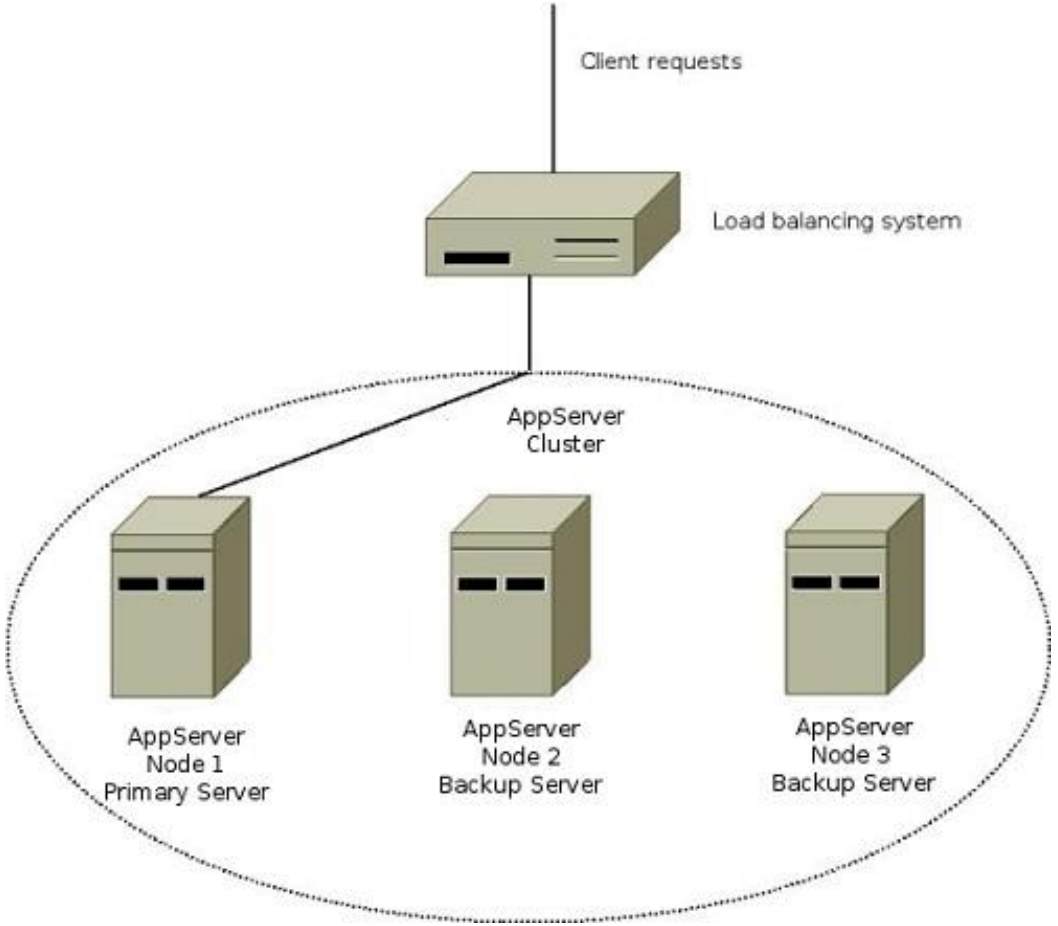
Types of Scaling in Microservices

Horizontal Scaling: Adding more instances of a service to distribute load.

Vertical Scaling: Adding more resources (CPU, memory) to a single instance.

Auto-scaling: Automatically increasing or decreasing instances based on traffic.

Horizontal Scaling



Vertical Scaling

Sometimes vertical scaling is right

Buying a bigger box is quick

Redesigning software is not

Running out of MySQL performance?

- Spend months on data federation
- Or, Just buy a ton more RAM

Infrastructure

HA Clusters

Load Balancers

Virtualization

CDN (Content Delivery Network)

Load Balancers

When we have multiple nodes, the load needs to be balanced

- **Hardware Load Balancers**
- **Software Load Balancers**

Hardware LB

A hardware appliance

- Often a pair with heartbeats for HA

Expensive!

- But offers high performance
- Easy to do > 1Gbps

Many brands

- Alteon, Cisco, Netscalar, Foundry, etc
- L7 - web switches, content switches, etc

Software LB

Just some software

- Still needs hardware to run on
- But can run on existing servers

Harder to have HA

- Often people stick hardware LB's in front

Software LB

Lots of options

- Pound
- Perlbal
- Apache with mod_proxy

Scaling Strategies in Microservices

Service Replication:

- Duplicating services across instances.

Partitioning/Sharding:

- Dividing data or tasks among service instances.

Stateless Services:

- Keeping services stateless to enable easier scaling.

Techniques to Achieve Scalability

Load Balancing:

- Distributing traffic among services (e.g., NGINX, HAProxy).

API Gateway:

- Central point to manage service traffic (e.g., Zuul, Spring Cloud Gateway).

Caching:

- Using cache to reduce database and service load (e.g., Redis, Memcached).

Asynchronous Communication:

- Event-driven design to decouple services (e.g., Kafka, RabbitMQ).

Tools for Scaling Microservices

Kubernetes:

- Automates deployment, scaling, and management of containerized apps.

Docker Swarm:

- Simple orchestration tool for Docker containers.

Service Mesh:

- Manages microservice communication (e.g., Istio).

Cloud Platforms:

- AWS, Azure, GCP provide out-of-the-box scaling solutions.

Monitoring and Observability

Key Metrics:

- CPU,
- Memory,
- Request rate,
- Response time,
- Error rates.

Tools:

- Prometheus + Grafana for metrics,
- ELK Stack for logs,
- Zipkin for tracing.

Case Study: Netflix's Microservices Scalability

Auto-scaling with AWS:

- Adjust instances based on traffic.

Service Discovery and Load Balancing:

- Via Eureka and Ribbon.

Chaos Engineering:

- Testing resiliency and scalability with Chaos Monkey.

Microservice Architecture
Best Practices

Each microservice delivers a single complete business capability

- building a microservice such that it does a single thing well without interruptions or waiting time is at the foundation of a robust architecture.

Autonomy over Coordination

- Once you have defined your business capabilities correctly the dependencies between those capabilities are minimized. Therefore, minimal coordination between capabilities is required, leading to optimal autonomy.
- Increased autonomy for a microservice gives it freedom to evolve without impacting other services: the optimal technology can be used, it can scale without having to scale others, etc.

Bounded Context

- How big a Microservice should be is: it should have a well-defined bounded context that will enable us to work without having to consider, or swap, between contexts.

Microservice Architecture
Best Practices

Asynchronous communication over synchronous communication

- Synchronous dependencies between services imply that the calling service is blocked and waiting for a response by the called service before continuing its operation. This is tight coupling, does not scale very well, and the calling service may be impacted by errors in the called service

One team responsible for full life cycle of a microservice

- A single team can deploy and manage a service as well as create new versions and retire obsolete ones. This means that users of the service have a single point of contact for all questions regarding the use of the service.

Data Ownership

- Each service should have its own private data.

API Versioning

- Maintain API versioning

Application Security

Application Security

Access level Security

Transport layer security

Data Security

API Security

Fundamental Principles of Application Security

CIA Triad

AAA

Confidentiality

Confidentiality ensures that sensitive data remains accessible only to authorized individuals or entities.

It prevents unauthorized disclosure or leakage of information that could compromise privacy, intellectual property, or competitive advantage.

Examples:

- Encryption
- Access Control
- Data Masking

Integrity

Integrity guarantees the accuracy, consistency, and trustworthiness of data throughout its lifecycle. It prevents unauthorized modification or alteration of information, ensuring that data remains reliable and unaltered.

Examples:

- Hashing
- Digital Signatures

Availability

Availability ensures that authorized users have timely and reliable access to information and resources when needed. It protects against disruptions, outages, or denial-of-service attacks that could hinder operations or productivity.

Examples:

- Redundancy
- Load Balancing
- Disaster Recovery

Authentication

Authentication is the process of verifying the identity of a user or entity attempting to access a system or resource. It ensures that individuals are who they claim to be, preventing unauthorized individuals from gaining entry.

Authentication Factors

- Something You Know
  - Password, PIN etc.,
- Something You Have
  - Smart Card, Mobile device etc.,
- Something You Are
  - Fingerprints, Iris etc.,

Muti-Factor Authentication

Authorization

Authorization is the process of determining what actions or resources an authenticated user or entity is permitted to access. It ensures that individuals have the appropriate level of access based on their roles, responsibilities, or specific permissions.

Authorization Models:

- Role-based access control (RBAC):
- Attribute-based access control (ABAC):

Accountability

Accountability in application security refers to ensuring that all actions, events, and decisions related to the security of an application can be traced back to an individual or a system entity responsible for those actions.

It ensures that every security event, from user interactions to system modifications, is logged, attributed, and traceable to the person or process that performed it.

This is critical for maintaining control over access and actions within an application, ensuring compliance with security policies, and facilitating auditing and forensics.

How is it achieved?

- Logging and Monitoring
- Audit trails
- Non-repudiation

API Authentication Mechanisms

Basic Authentication

- Uses HTTP headers to transmit credentials (username and password) encoded in Base64.

JSON Web Tokens (JWT)

- JWT is an open standard for securely transmitting information between parties as a JSON object.
- Contains encoded JSON objects, including claims and signatures to verify authenticity.

OAuth 2.0

- OAuth 2.0 is not an authentication protocol but an authorization framework.
- An authorization framework that enables applications to obtain limited access to user accounts on an HTTP service.
- It allows third-party applications to obtain limited access to an HTTP service.

API Keys

- A unique identifier used to authenticate a user, developer, or calling program to an API.
- Often used in scenarios where simple authentication is sufficient.

Spring Security

Spring Security

Spring Security is a powerful and highly customizable authentication and access-control framework for Java applications.

It is the de-facto standard for securing Spring-based applications.

Features

- Comprehensive Security: Supports authentication, authorization, and protection against common attacks.
- Integration: Seamlessly integrates with Spring Boot applications.
- Customization: Highly extensible to meet custom security requirements.
- Ease of Use: Provides default security configurations to get started quickly.

Spring Security Architecture

Security Filters

- Spring Security uses a chain of servlet filters to intercept HTTP requests.
- Filter Chain: Each filter performs specific security checks.
- DelegatingFilterProxy: Integrates Spring-managed beans into the servlet filter chain.

Authentication and Authorization

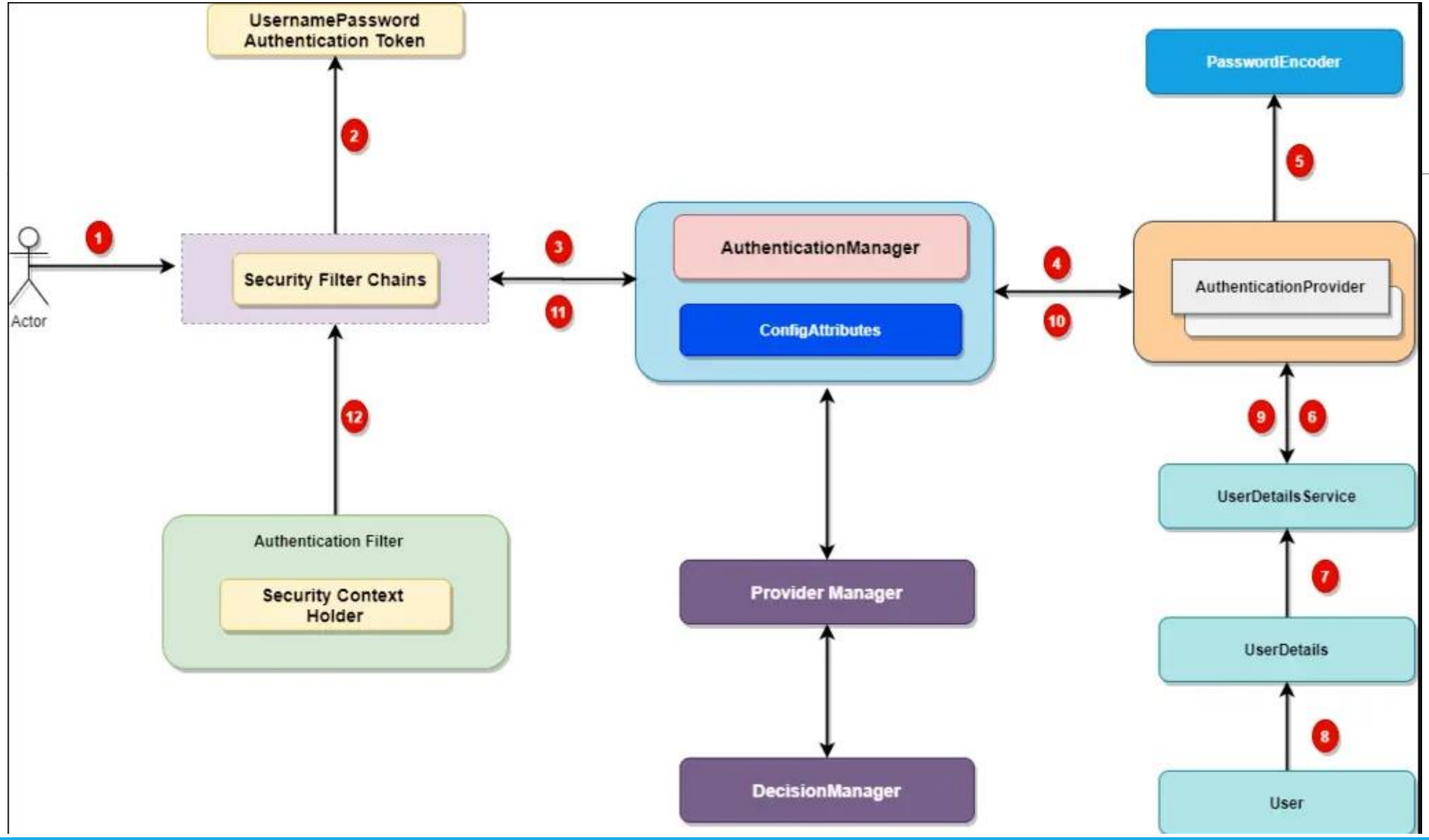
- Authentication: Verifying the identity of a user or system.
- Authorization: Determining whether an authenticated user has access to a specific resource.

Security Context

- Stores security information in a thread-local SecurityContextHolder.
- Contains the Authentication object representing the principal.

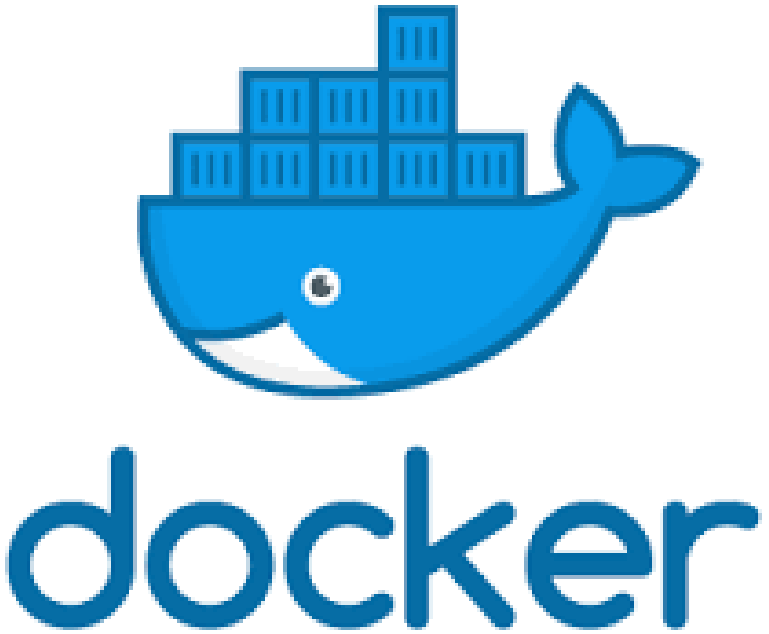
Authentication Providers and Tokens

- AuthenticationManager: Delegates authentication requests to a list of AuthenticationProviders.
- AuthenticationProvider: Performs authentication logic.
- Authentication Token: Represents the authentication request or result.


UsernamePassword Authentication Token
PasswordEncoder
2
5
1
Actor
Security Filter Chains
3
11
AuthenticationManager
ConfigAttributes
4
10
AuthenticationProvider
12
9
6
Authentication Filter
Security Context Holder
Provider Manager
UserDetails Service
7
UserDetails
8
DecisionManager
User

Spring Security Demo

Containerization



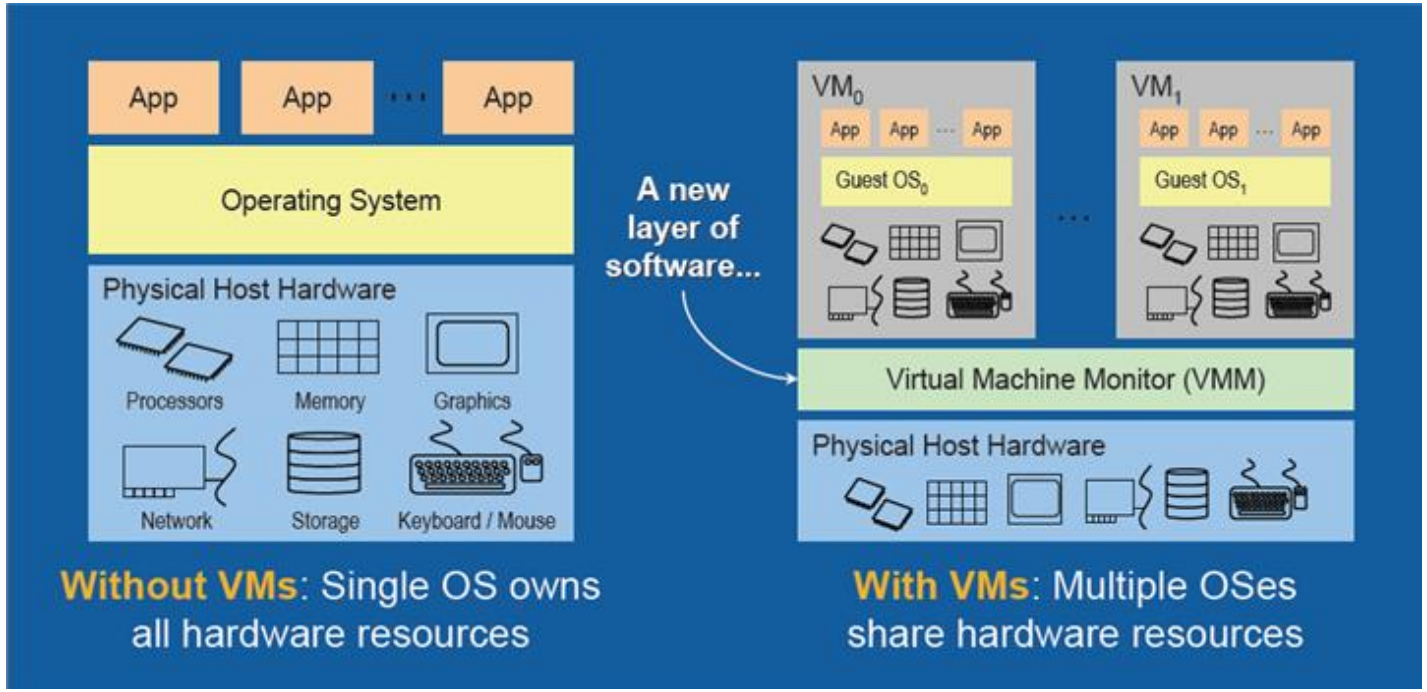
What is Virtualization?

Virtualization is an abstraction layer that decouples the physical hardware from the operating system to deliver greater IT resource utilization and flexibility.

- It allows you to:

  - Consolidate workloads to reduce hardware, power, and space requirements.

  - Run multiple Operating Systems simultaneously on the same hardware



Application Configuration Challenges

Application behavior changes when moved form one ENV to other due to

- Multiple Configurations
- Library versions
- Different underlying components
- Etc.,

- And not just the software,
  - *The network topology*
  - *Security policies*
  - *Storage etc.,*

  Might be different where the software has to run

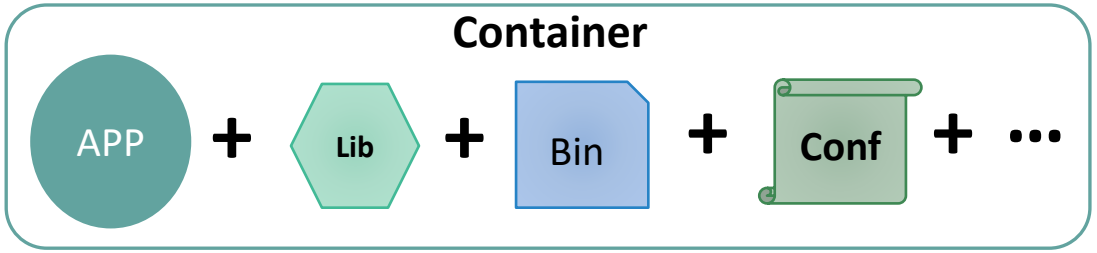
What are Containers?

Containers are a solution to the problem of how to get software to run reliably when moved from one computing environment to another.

This could be from a DEV Box to QA, from a Staging to PROD and perhaps from a physical machine in a data center to a virtual machine in a private or public cloud.

- A container consists of an entire runtime environment:
  - an application, plus
  - all its dependencies,
  - Libraries,
  - binaries, and
  - configuration files

  needed to run it, bundled into one package.



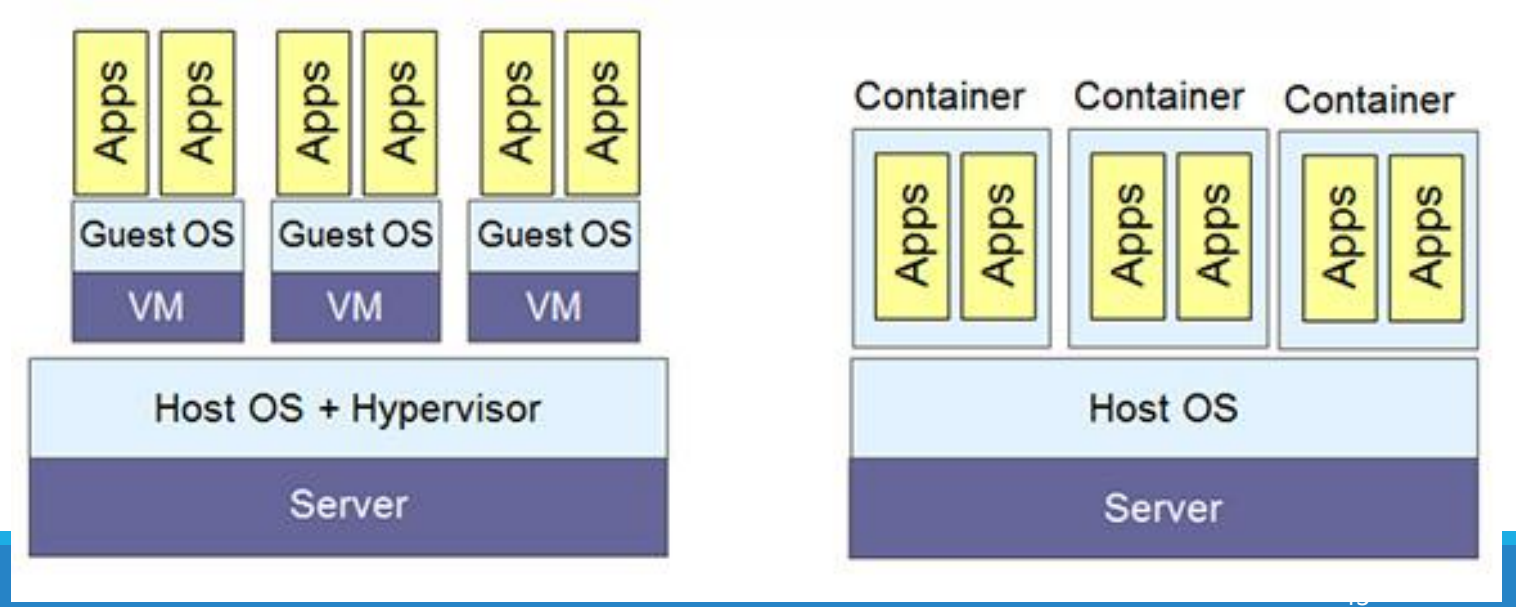
Are Containers different from Virtualization?

Containers and virtual machines have similar resource isolation and allocation benefits -- but a different architectural approach allows containers to be more portable and efficient.

VM hypervisors are "based on emulating virtual hardware. That means they're fat in terms of system requirements."

Containers, however, use shared operating systems. That means they are much more efficient than hypervisors in system resource terms.

- Instead of virtualizing hardware, containers rest on top of a single OS instance.



Why Containers?

By containerizing the application platform and its dependencies, differences in OS distributions and underlying infrastructure are abstracted away.

Containers gives you instant application portability.

Developers can use Docker to pack, ship, and run any application as a lightweight, portable, self sufficient LXC container that can run virtually anywhere

Why Containers?

It makes it possible to get far more apps running on the same old servers and it also makes it very easy to package and ship programs.

This in turn means you can “leave behind the useless 99.9% VM junk, leaving you with a small, neat capsule containing your application”

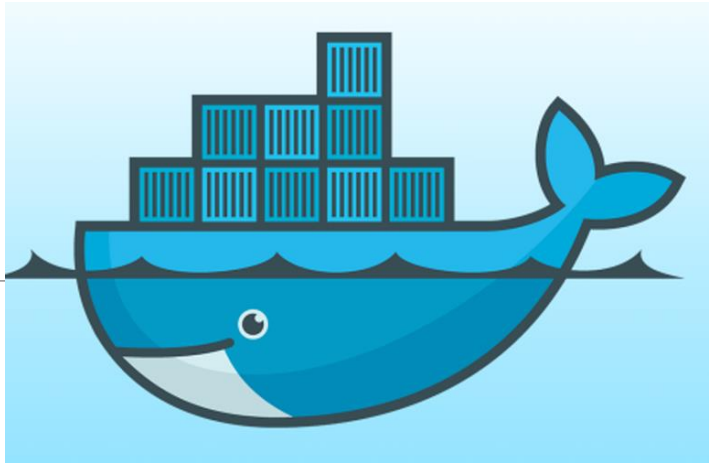
Can save a data center or cloud provider tens-of-millions of dollars annually in power and hardware costs.

Benefits of Containerization

Containerization is increasingly popular because containers are:

- Flexible: Even the most complex applications can be containerized.
- Lightweight: Containers leverage and share the host kernel.
- Interchangeable: You can deploy updates and upgrades on-the-fly.
- Portable: You can build locally, deploy to the cloud, and run anywhere.
- Scalable: You can increase and automatically distribute container replicas.
- Stackable: You can stack services vertically and on-the-fly.

Docker



Companies are adopting Docker at a remarkable rate.

Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools, system libraries – anything that can be installed on a server. This guarantees that the software will always run the same, regardless of its environment.

What is Docker?

Docker is an open platform for developing, shipping, and running applications.

It 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 quickly, you can significantly reduce the delay between writing code and running it in production.

Docker Image

A Docker image is a file, comprised of multiple layers, used to execute code in a Docker container. An image is essentially built from the instructions for a complete and executable version of an application, which relies on the host OS kernel. When the Docker user runs an image, it becomes one or multiple instances of that container.

Docker Use Cases

CI/CD

Enable developers to develop and test applications more quickly and within any environment

DEVOPS

Break down barriers between Dev and Ops teams to improve the app development process

INFRASTRUCTURE OPTIMIZATION

Decrease infrastructure costs while increasing its efficiency

Docker Commands

Usage: docker COMMAND

Commands:

version	Show the Docker version information
info	Display system-wide information
ps	List containers
run	Run a command in a new container
pull	Pull an image or a repository from a registry
start	Start one or more stopped containers
stop	Stop one or more running containers
restart	Restart one or more containers
kill	Kill one or more running containers
inspect	Return low-level information on Docker objects
images	List images
build	Build an image from a Dockerfile
rename	Rename a container
rm	Remove one or more containers

Container Orchestration Tools

With the new Application Packaging and Delivery approach a lot of new tools are coming up in the market adopting this. For Example

Docker Swarm

- Docker Cluster management

Kubernetes

- Open source container cluster manager originally designed by Google

Mesosphere

- Container and Data Services Management tool

Thank You