Desenvolvimento de Aplicações com Arquitetura Baseada em Microservices

Prof. Vinicius Cardoso Garcia
vcg@cin.ufpe.br :: @vinicius3w :: assertlab.com

[IF1007] - Tópicos Avançados em SI 4
https://github.com/vinicius3w/if1007-Microservices

Licença do material

Este Trabalho foi licenciado com uma Licença

Creative Commons - Atribuição-NãoComercial-CompartilhaIgual 3.0 Não Adaptada

Mais informações visite

http://creativecommons.org/licenses/by-nc-sa/3.0/deed.pt

Resources

- There is no textbook required. However, the following are some books that may be recommended:
  - Building Microservices: Designing Fine-Grained Systems
  - Spring Microservices
  - Spring Boot: Acelere o desenvolvimento de microsserviços
  - Microservices for Java Developers A Hands-on Introduction to Frameworks and Containers
  - Migrating to Cloud-Native Application Architectures
  - Continuous Integration
  - Getting started guides from spring.io

Containerizing Microservices with Docker

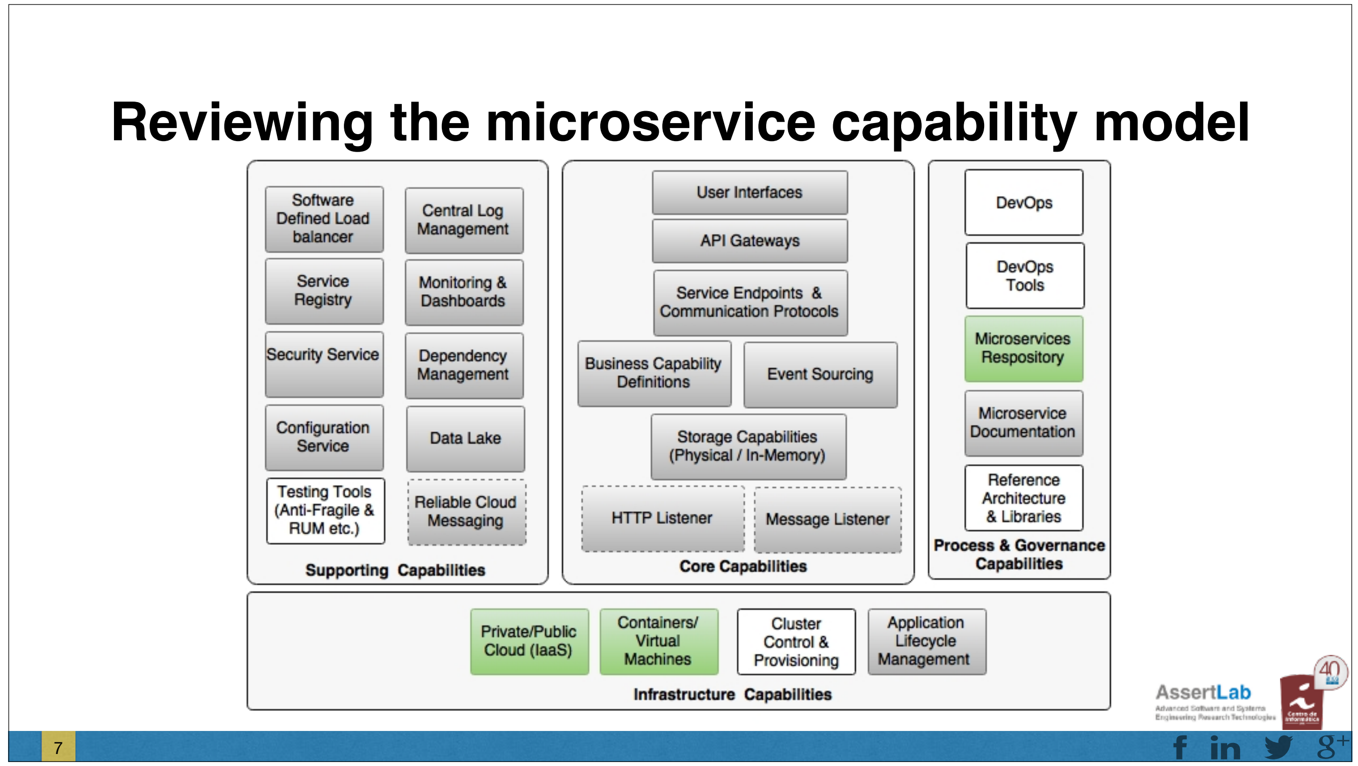
Context

- In the context of microservices, containerized deployment is the icing on the cake
- It helps microservices be more autonomous by self-containing the underlying infrastructure, thereby making the microservices cloud neutral

AssertLab
Advanced Software and Systems Engineering Research Technologies

Topics

- The concept of containerization and its relevance in the context of microservices
- Building and deploying microservices as Docker images and containers
- Using AWS as an example of cloud-based Docker deployments



In this lecture, we will explore the following microservice capabilities from the microservice capability model discussed in lecture 4, Applying Microservices Concepts:
- Containers and virtual machines
- The private/public cloud
- The microservices repository

Understanding the gaps in BrownField PSS microservices

- Lecture 6: Scaling Microservices with Spring Cloud, BrownField PSS microservices were developed using Spring Boot and Spring Cloud. These microservices are deployed as versioned fat JAR files on bare metals, specifically on a local development machine
- Lecture 7: Autoscaling Microservices, the autoscaling capability was added with the help of a custom life cycle manager
- Lecture 8: Logging and Monitoring Microservices, challenges around logging and monitoring were addressed using centralized logging and monitoring solutions

Understanding the gaps in BrownField PSS microservices

- There are still a few gaps
- The implementation has not used any cloud infrastructure
- A cloud infrastructure provides all the essential capabilities
- Running multiple microservices on a single bare metal could lead to a "noisy neighbor" problem
- An alternate approach is to run the microservices on VMs. However, VMs are heavyweight in nature
- In the case of Java-based microservices, sharing a VM or bare metal to deploy multiple microservices also results in sharing JRE among microservices
  - The fat JARs created in our BrownField PSS abstract only application code and its dependencies but not JREs

Dedicated machines, as in traditional monolithic application deployments, are not the best solution for deploying microservices. Automation such as automatic provisioning, the ability to scale on demand, self-service, and payment based on usage are essential capabilities required to manage large-scale microservice deployments efficiently

Understanding the gaps in BrownField PSS microservices

- One microservice principle insists that it should be self-contained and autonomous by fully encapsulating its end-to-end runtime environment
- In order to align with this principle, all components, such as the OS, JRE, and microservice binaries, have to be self-contained and isolated
- The only option to achieve this is to follow the approach of deploying one microservice per VM
- However, this will result in underutilized virtual machines, and in many cases, extra cost due to this can nullify benefits of microservices

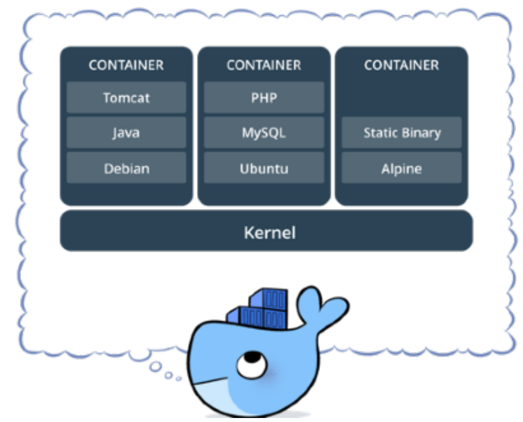
AssertLab

What are containers?

- CONTAINERS ARE PROCESSES
- Containers are processes sandboxed by:
  - Kernel namespaces
  - Root privilege management
  - System call restrictions
  - Private network stacks
  - etc

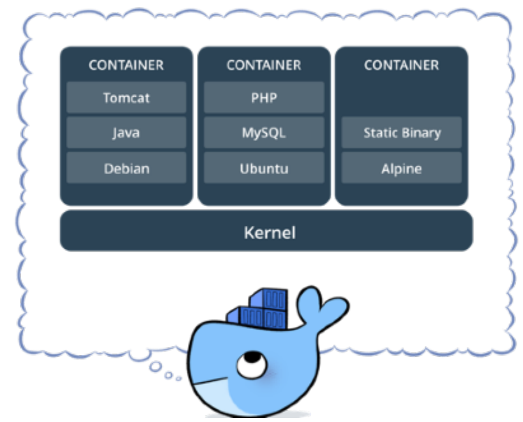
What are containers?

- Containers provide private spaces on top of the operating system
- This technique is also called operating system virtualization
  - The kernel of the operating system provides isolated virtual spaces
  - Each of these virtual spaces is called a container or virtual engine (VE)
- Containers allow processes to run on an isolated environment on top of the host operating system



What are containers?

- Containers are easy mechanisms to build, ship, and run compartmentalized software components
- Generally, containers package all the binaries and libraries that are essential for running an application
- Containers reserve their own filesystem, IP address, network interfaces, internal processes, namespaces, OS libraries, application binaries, dependencies, and other application configurations

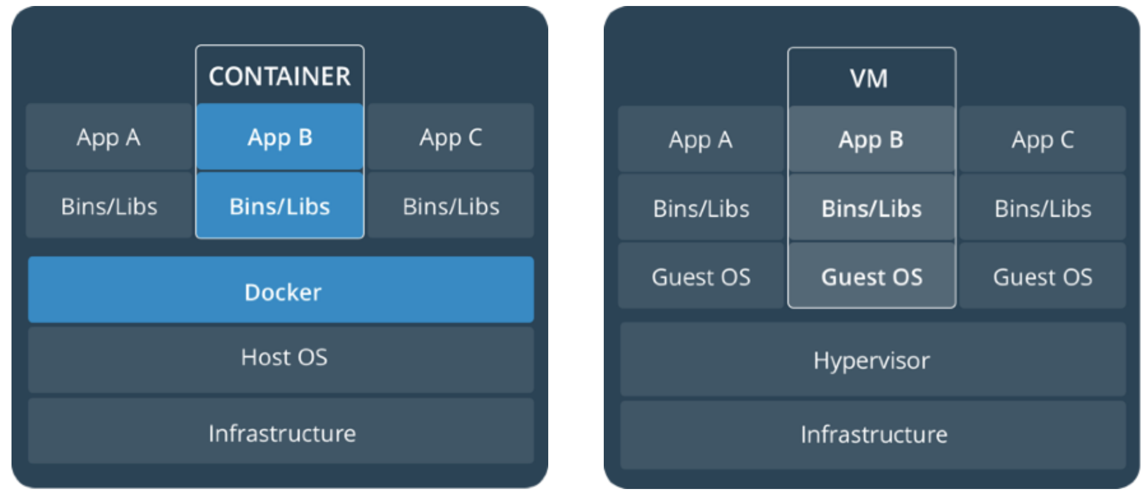


AssertLab

Big white shark?

- There are billions of containers used by organizations
- Moreover, there are many large organizations heavily investing in container technologies
- Docker is far ahead of the competition, supported by many large operating system vendors and cloud providers
- Lmctfy, SystemdNspawn, Rocket, Drawbridge, LXD, Kurma, and Calico are some of the other containerization solutions
- Open container specification is also under development

The difference between VMs and containers

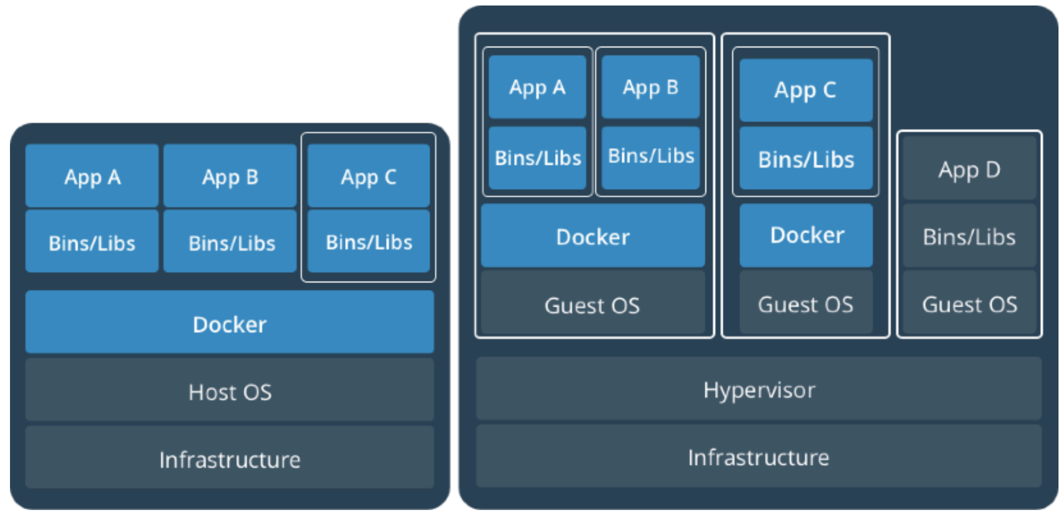


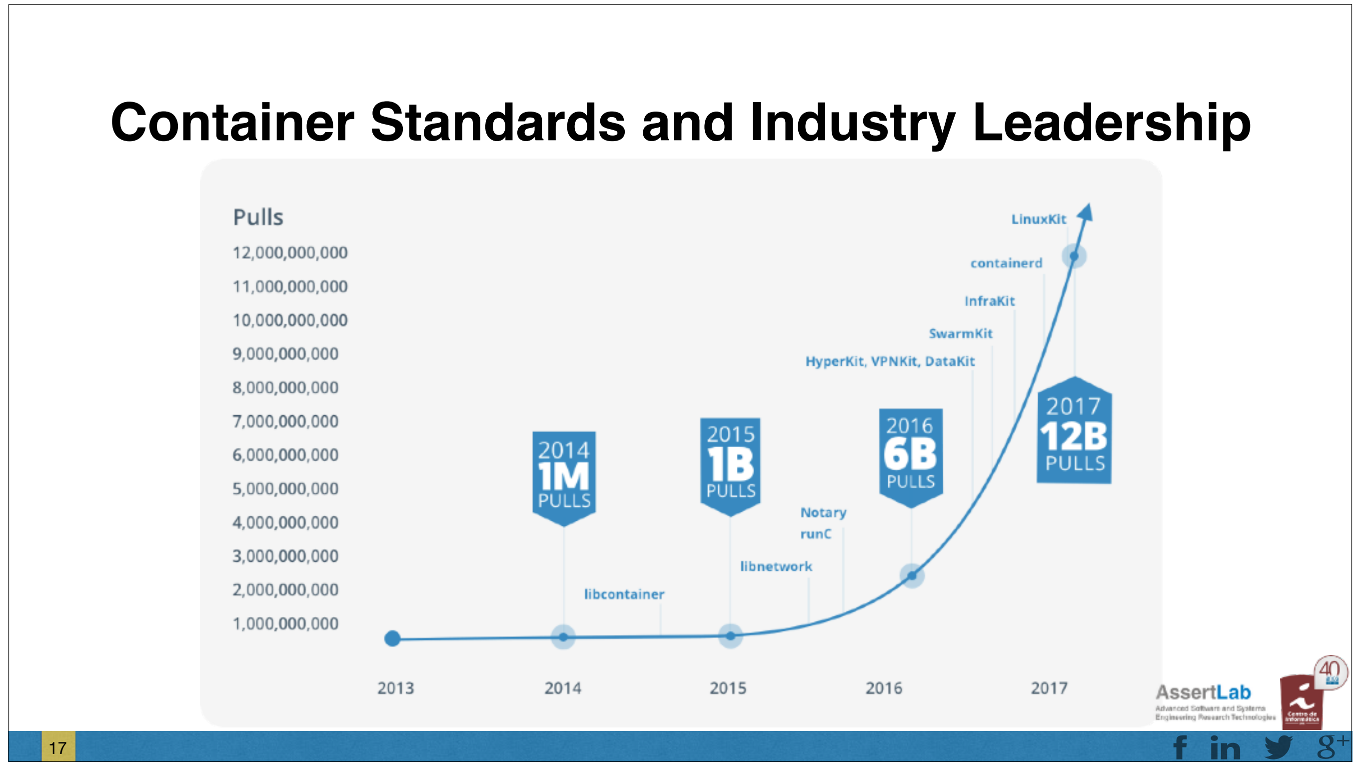
AssertLab

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), and start almost instantly.

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, one or more apps, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.

Containers and Virtual Machines Together





The launch of Docker in 2013 jump started a revolution in app development - by bringing software containers to the masses. In just a few years, Docker has transformed the industry with a new lexicon, framework and standards in app development, packaging and management.

Starting with the pivot of dotCloud and turning a piece of dotCloud into a better Linux container technology - one that is portable, flexible, and easy to deploy. Docker open sourced the libcontainer and partnered with a worldwide community of contributors to further its development. By June 2015, just a couple short years later, Docker donated the specification and runtime code now known as runC, to the Open Container Initiative (OCI) to help establish standardization as the container ecosystem grows and matures.

Following this evolution, Docker continues to give back with the containerd project. Containerd is the core container runtime of the Docker engine daemon, an industry-standard container runtime with an emphasis on simplicity, robustness and portability, designed as an embeddable component for higher level systems. Docker engine is built on runC and containerd. This collaborative project enables portability for developers, container standardization through the OCI, a growing ecosystem of tools, and a free marketplace of images - pre-made apps that the community can share and build on.

The benefits of containers

- Self-contained: package the essential application binaries and their dependencies together
- Lightweight: containers, in general, are smaller in size with a lighter footprint.
  - The simplest Spring Boot microservice packaged with an Alpine container with Java 8 would only come to around 170 MB in size
- Scalable: container images are smaller in size and there is no OS booting at startup
- Portable: are built with all the dependencies, they can be ported across multiple machines or across multiple cloud providers
- Lower license cost: Many software license terms are based on the physical core

The benefits of containers

- DevOps: the lightweight footprint of containers makes it easy to automate builds and publish and download containers from remote repositories
- Version controlled: support versions by default
- Reusable: container images are reusable artifacts
- Immutable containers: containers are created and disposed of after usage, they are never updated or patched
  - Immutable containers are used in many environments to avoid complexities in patching deployment units
  - Patching results in a lack of traceability and an inability to recreate environments consistently

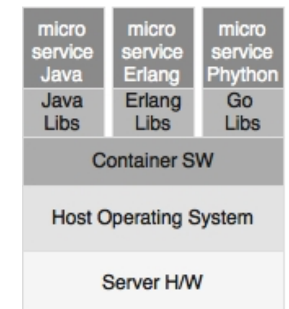
Microservices and containers

- Microservices can run without containers, and containers can run monolithic applications
- Containers are good for monolithic applications
  - but the complexities and the size of the monolith application may kill some of the benefits of the containers

There is no direct relationship between microservices and containers. Microservices can run without containers, and containers can run monolithic applications. However, there is a sweet spot between microservices and containers.

Microservices and containers

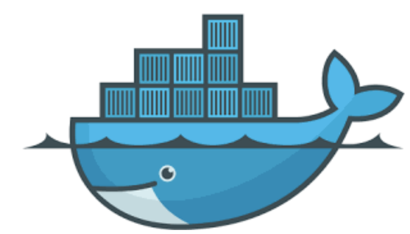
- The real advantage of containers can be seen when managing many polyglot microservices
- Eliminate the need to have different deployment management tools to handle polyglot microservices
- Not only abstract the execution environment but also how to access the services
- Irrespective of the technologies used, containerized microservices expose REST APIs
- Once the container is up and running, it binds to certain ports and exposes its APIs
- As containers are self-contained and provide full stack isolation among services, in a single VM or bare metal, one can run multiple heterogeneous microservices and handle them in a uniform way



Containers help developers package microservices written in any language or technology in a platform- and technology-agnostic fashion and uniformly distribute them across multiple environments

Introduction to Docker

- Containers have been in the business for years, but the popularity of Docker has given containers a new outlook
- As a result, many container definitions and perspectives emerged from the Docker architecture
- Docker is so popular that even containerization is referred to as dockerization
- Docker is a platform to build, ship, and run lightweight containers based on Linux kernels



AssertLab

Security

"Gartner asserts that applications deployed in containers are more secure than applications deployed on the bare OS."

http://blogs.gartner.com/joerg-fritsch/can-you-operationalize-docker-containers/

Introduction to Docker

- The most basic thing Docker provides is a
- FRAMEWORK FOR SERVICE ENCAPSULATION
- But what are the implications of this for developers, ops, and orgs?

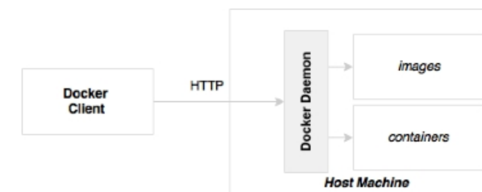
Docker Product Offerings

	Community Edition	Enterprise Edition
Add Ons	Cloud: Private repos as a service Cloud: Autobuilds as a service Cloud: Security scanning as a service	DDC: On-prem integrated container management, registry and security DSS: On-prem image scanning
Platform	docker	docker
Infrastructure	amazon web services, Microsoft Azure, debian, fedora, CentOS	CERTIFIED amazon web services, Microsoft Azure, ubuntu, SUSE Linux Enterprise Server, CentOS, ORACLE LINUX, redhat

AssertLab

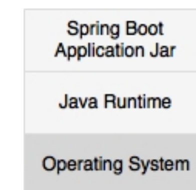
The key components of Docker

- The Docker daemon is a server-side component that runs on the host machine responsible for building, running, and distributing Docker containers
  - exposes APIs for the Docker client to interact with the daemon. These APIs are primarily REST-based endpoints
- The Docker client is a remote command-line program that interacts with the Docker daemon through either a socket or REST APIs
  - Docker users use the CLI to build, ship, and run Docker containers



Docker concepts

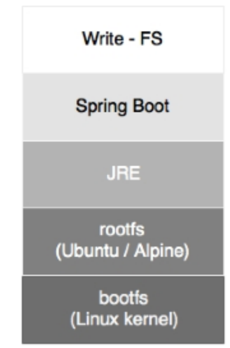
- A Docker image is the read-only copy of the operating system libraries, the application, and its libraries
- Once an image is created, it is guaranteed to run on any Docker platform without alterations
- In Spring Boot microservices, a Docker image packages operating systems such as Ubuntu, Alpine, JRE, and the Spring Boot fat application JAR file
  - Docker images are based on a layered architecture in which the base image is one of the flavors of Linux
  - Each layer, as shown in the preceding diagram, gets added to the base image layer with the previous image as the parent layer
  - Docker uses the concept of a union filesystem to combine all these layers into a single image, forming a single filesystem



In typical cases, developers do not build Docker images from scratch. Images of an operating system, or other common libraries, such as Java 8 images, are publicly available from trusted sources. Developers can start building on top of these base images. The base image in Spring microservices can be JRE 8 rather than starting from a Linux distribution image such as Ubuntu

Docker image

- Every time we rebuild the application, only the changed layer gets rebuilt, and the remaining layers are kept intact
- Multiple containers running on the same machine with the same type of base images would reuse the base image, thus reducing the size of the deployment
- For instance, in a host, if there are multiple containers running with Ubuntu as the base image, they all reuse the same base image



When a container is initiated, a writable filesystem is placed on top of all the other filesystems for the processes to run. Any changes made by the process to the underlying filesystem are not reflected in the actual container. Instead, these are written to the writable filesystem. This writable filesystem is volatile. Hence, the data is lost once the container is stopped. Due to this reason, Docker containers are ephemeral in nature

Docker image

- IMAGES ARE LAYERED FILESYSTEMS
- Provide filesystem for container process
- Image = stack of immutable layers
- Start with a base image
- Add layer for each change

AssertLab

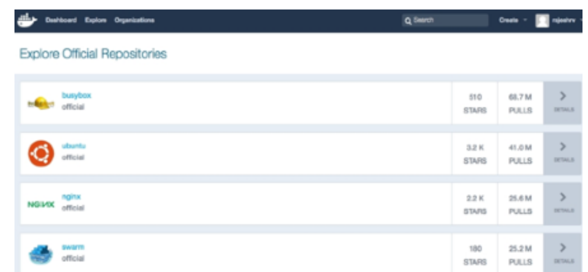
Docker containers

- Docker containers are the running instances of a Docker image
- Containers use the kernel of the host operating system when running
  - Hence, they share the host kernel with other containers running on the same host
- The Docker runtime ensures that the container processes are allocated with their own isolated process space using kernel features such as **cgroups** and the kernel **namespace** of the operating system
-

In addition to the resource fencing, containers get their own filesystem and network configurations as well.
The containers, when instantiated, can have specific resource allocations, such as the memory and CPU. Containers, when initiated from the same image, can have different resource allocations. The Docker container, by default, gets an isolated subnet and gateway to the network. The network has three modes.

The Docker registry

- The Docker registry is a central place where Docker images are published and downloaded from
- The URL https://hub.docker.com is the central registry provided by Docker
- The Docker registry has public images that one can download and use as the base registry
- Docker also has private images that are specific to the accounts created in the Docker registry



AssertLab
Advanced Software and Systems Engineering Research Technologies

Dockerfile

- A Dockerfile is a build or scripting file that contains instructions to build a Docker image
- There can be multiple steps documented in the Dockerfile, starting from getting a base image
- The docker build command looks up Dockerfile for instructions to build
- One can compare a Dockerfile to a pom.xml file used in a Maven build

Dockerfile

- FROM command defines base image
- Each subsequent command adds a layer
- docker image build ... builds image from Dockerfile

```
# Comments begin with the pound sign
FROM ubuntu:16.04
RUN apt-get update
ADD /data /myapp/data
...
```

AssertLab
Advanced Software and Systems Engineering Research Technologies

Homework 11.1

- A guided tour to deploying microservices in Docker

The future of containerization – unikernels and hardened security

- Containerization is still evolving, but the number of organizations adopting containerization techniques has gone up in recent times
- Currently, Docker images are generally heavy
  - In an elastic automated environment, where containers are created and destroyed quite frequently, size is still an issue
  - A larger size indicates more code, and more code means that it is more prone to security vulnerabilities

The future of containerization – unikernels and hardened security

- The future is definitely in small footprint containers
- Docker is working on unikernels, lightweight kernels that can run Docker even on low-powered IoT devices
  - Unikernels are not full-fledged operating systems, but they provide the basic necessary libraries to support the deployed applications

The future of containerization – unikernels and hardened security

- The security issues of containers are much discussed and debated
- The key security issues are around the user namespace segregation or user ID isolation
- If the container is on root, then it can by default gain the root privilege of the host
- Using container images from untrusted sources is another security concern
- Docker is bridging these gaps as quickly as possible, but there are many organizations that use a combination of VMs and Docker to circumvent some of the security concerns

Homework 11.2

- Work through the exercises 1 to 18 in the Docker Fundamentals Exercises book