FIAP ON

CLOUD FUNDAMENTALS, ADMINISTRATION AND SOLUTION ARCHITECT

# DOCKER

CHRISTIANE DE PAULA REIS

07

## LISTA DE FIGURAS

Figura 7.1 – *Cluster* de computadores ..... 7
Figura 7.2 – Estrutura tradicional x Virtualização ..... 7
Figura 7.3 – Estrutura tradicional x Virtualização ..... 8
Figura 7.4 – Tipos de virtualização ..... 9
Figura 7.5 – Virtualização por VM ..... 10
Figura 7.6 – Virtualização por container ..... 11
Figura 7.7 – Essência do container Docker ..... 14
Figura 7.8 – Componentes da arquitetura do Docker ..... 17
Figura 7.9 – Arquitetura simplificada do Docker para host único ..... 18
Figura 7.10 – Papéis e responsabilidades em container Docker ..... 19
Figura 7.11 – Container Docker e Microsserviços ..... 24
Figura 7.12 – Monolíticos e Microsserviços em container Docker ..... 26
Figura 7.13 – Representação imagem e *layers* ..... 28
Figura 7.14 – Representação container e *layers* ..... 29
Figura 7.15 – Formas de *storage* em um container ..... 30
Figura 7.16 – Volumes de dados e o *driver* de armazenamento ..... 31
Figura 7.17 – Fornecedores x Serviços ..... 39
Figura 7.18 – Arquitetura de uma CDN ..... 40
Figura 7.19 – Antes e depois das CDNs ..... 41

## LISTA DE QUADROS

Quadro 7.1 – Diferenças VM x Container.....12
Quadro 7.2 – Vantagens e desvantagens da arquitetura *Serverless* .....38

## LISTA DE CÓDIGOS-FONTE

Código-fonte 7.1 – Exemplo completo....33
Código-fonte 7.2 – Exemplo primeiro trecho....33
Código-fonte 7.3 – Exemplo segundo trecho....33
Código-fonte 7.4 – Exemplo terceiro trecho....34
Código-fonte 7.5 – Exemplo quarto trecho....34
Código-fonte 7.6 – Exemplo quinto trecho....34
Código-fonte 7.7 – Exemplo sexto trecho....35
Código-fonte 7.8 – Exemplo sétimo trecho....35
Código-fonte 7.9 – Exemplo oitavo trecho....36
Código-fonte 7.10 – Exemplo trecho final....36

**SUMÁRIO**

**7 DOCKER ..... 6**
7.1 Virtualização - recapitulando ..... 6
7.1.1 Sistema virtualizado por máquina virtual (H-based) ..... 9
7.1.2 Sistema virtualizado por container (OS-based) ..... 11
7.1.3 VM ou container? ..... 11
7.2 Container e microcontainers ..... 12
7.3 Docker ..... 13
7.3.1 Vantagens e benefícios no uso do Docker ..... 14
7.3.2 Componentes da arquitetura do container Docker ..... 15
7.3.3 Como funciona o Docker? ..... 17
7.3.4 Boas Práticas para uso do Docker ..... 18
7.4 Automação e implantação em container Docker ..... 19
7.5 Segurança dos containers ..... 21
7.6 Instalando um Docker ..... 22
7.6.1 Instalando um Docker em sistema operacional Windows ..... 23
7.6.2 Instalando um Docker em sistema operacional para Linux, Ubuntu ..... 23
7.6.3 Instalando um Docker em sistema operacional MacOS ..... 23
7.7 Docker container e a relação com os microsserviços ..... 24
7.7.1 Container Docker x Monolíticos e Microsserviços ..... 25
7.7.2 Arquitetura monolítica e o container Docker ..... 26
7.7.3 Arquitetura de microsserviços e o container Docker ..... 27
7.8 *Storage* no container Docker ..... 27
7.8.1 Persistindo dados no ambiente Docker ..... 31
7.9 Docker Compose ..... 32
7.9.1 Entendendo o arquivo docker-compose ..... 33
7.10 *Serverless Computing*: o que é? ..... 36
7.10.1 Como escolher um fornecedor *Serverless* para sua empresa? ..... 38
7.11 CDN ..... 40
**REFERÊNCIAS ..... 42**

# 7 DOCKER

Muitas empresas líderes do segmento da tecnologia já adotaram ou estão mudando para uma infraestrutura baseada em container, seja ela na nuvem ou ela local. Mas você sabe realmente o que é um container?

Neste capítulo, vamos explorar alguns conceitos importantes sobre containers e de outras tecnologias das quais eles dependem (como a virtualização), assim como suas formas de implementação, para que possamos ter condições de fazer uma escolha assertiva de acordo com as necessidades da empresa e do projeto a ser desenvolvido.

## 7.1 Virtualização - recapitulando

Recapitulando o conceito de virtualização que você já conhece, ela surgiu como uma forma de separar melhor o *hardware* do *software*. No modelo tradicional, os *softwares* são instalados em um sistema operacional que, por sua vez, está instalado sobre uma infraestrutura física (o *hardware*). No entanto, sempre que os *softwares* demandavam por mais infraestrutura física para suportar mais usuários ou serviços, o *hardware* precisava ser substituído por outro mais poderoso (processadores mais rápidos, mais memória e por aí vai), o que é chamado de *upgrade* vertical.

Os *clusters* de computadores são computadores ligados entre si dividindo as tarefas e “juntando forças” para resolver problemas computacionais mais complexos, dando início aos *upgrades* horizontais: assim, se fosse necessário atender mais usuários ou trabalhar com um volume maior de dados, bastava adicionar novos computadores a esta força-tarefa. Ainda neste modelo, cada um dos computadores colocados neste *cluster* mantinha sua própria estrutura de sistema operacional e aplicações.

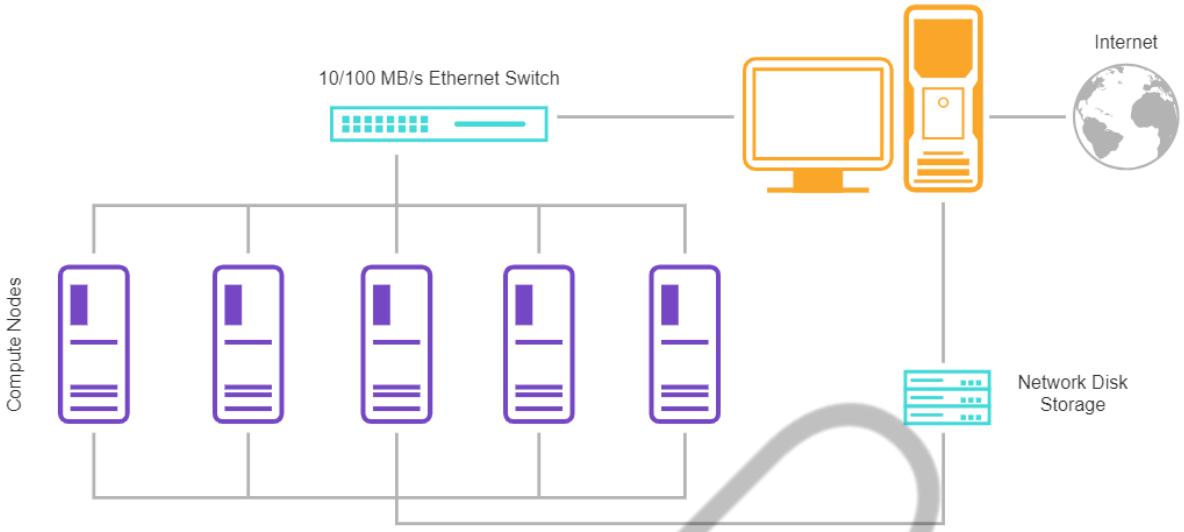


Figura 7.1 – *Cluster* de computadores
Fonte: Google Imagens (2019)

A virtualização possibilitou a abstração do *hardware*, pois uma camada de *software* mimetizando uma infraestrutura física é inserida entre o *hardware* (real) o sistema operacional, possibilitando uma separação mais eficiente destas duas camadas. A virtualização permite, por exemplo, a instalação de vários sistemas operacionais em uma mesma infraestrutura física.

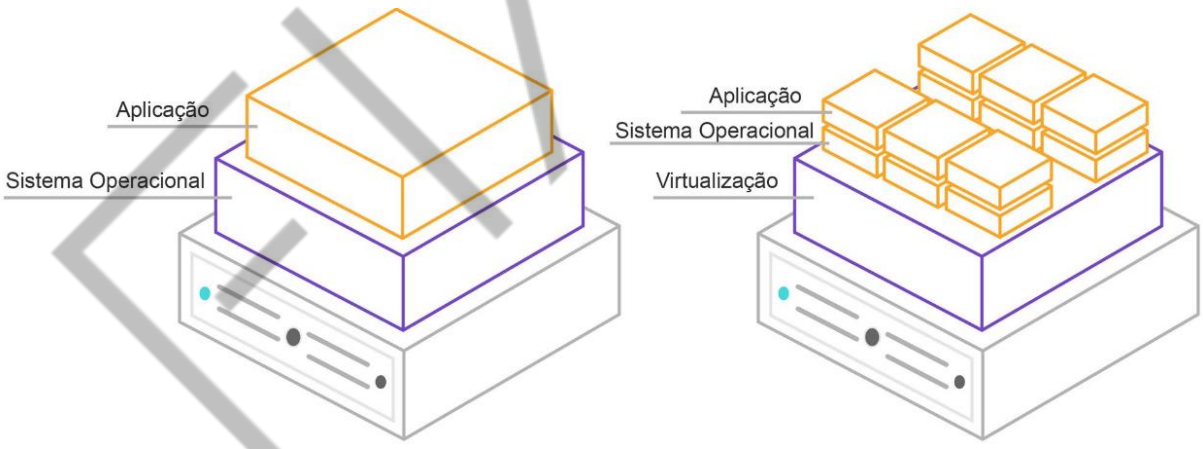


Figura 7.2 – Estrutura tradicional x Virtualização
Fonte: Google Imagens (2019)

Não demorou muito para surgir um inverso: um único sistema operacional (chamado, nesse contexto, de máquina virtual) instalado sob uma única camada de virtualização que compreende diversos computadores abaixo dela. A máquina virtual “enxerga” um único *hardware*, que se comporta como um supercomputador com vários núcleos e, muita memória para armazenamento, permitindo uma grande flexibilidade no aumento destes recursos para a máquina virtual.

Sempre que ela precisar atender mais usuários pode receber rapidamente mais processamento e memória e, por sua vez, sempre que a infraestrutura física for insuficiente, novos computadores são adicionados ao *cluster* em um *upgrade* horizontal. O *Cloud Computing* nada mais é que o compartilhamento desta infraestrutura entre diversas necessidades e clientes, em um modelo de negócio e cobrança diferenciado.

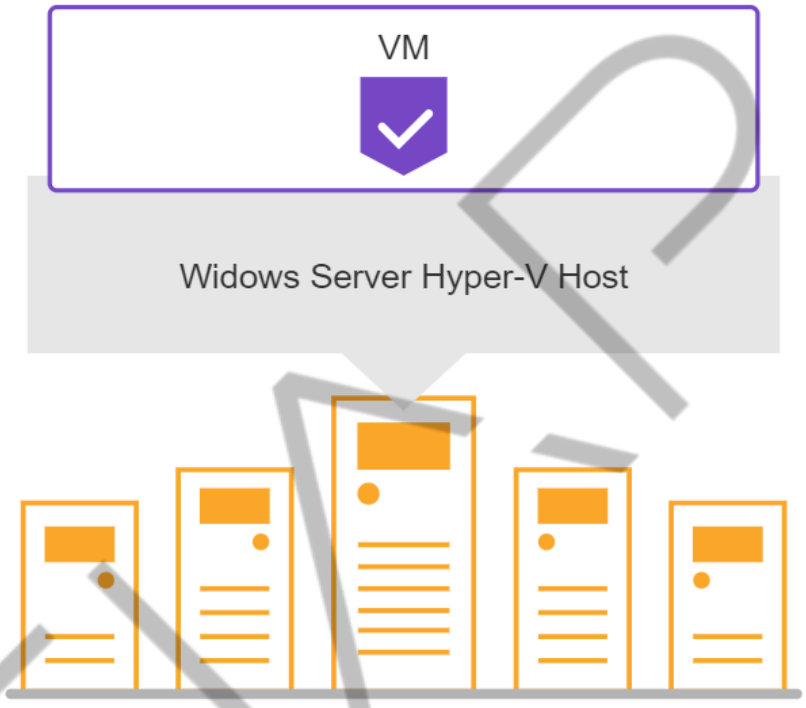


Figura 7.3 – Estrutura tradicional x Virtualização
Fonte: Google Imagens, adaptado por FIAP (2019)

No entanto, com a evolução desta arquitetura em *cluster*, começaram a surgir problemas na criação e manutenção de um Servidor Virtual Privado (*VPS – Virtual Private Server*). Daí surgiram os conceitos de virtualização de máquina e virtualização por containers.

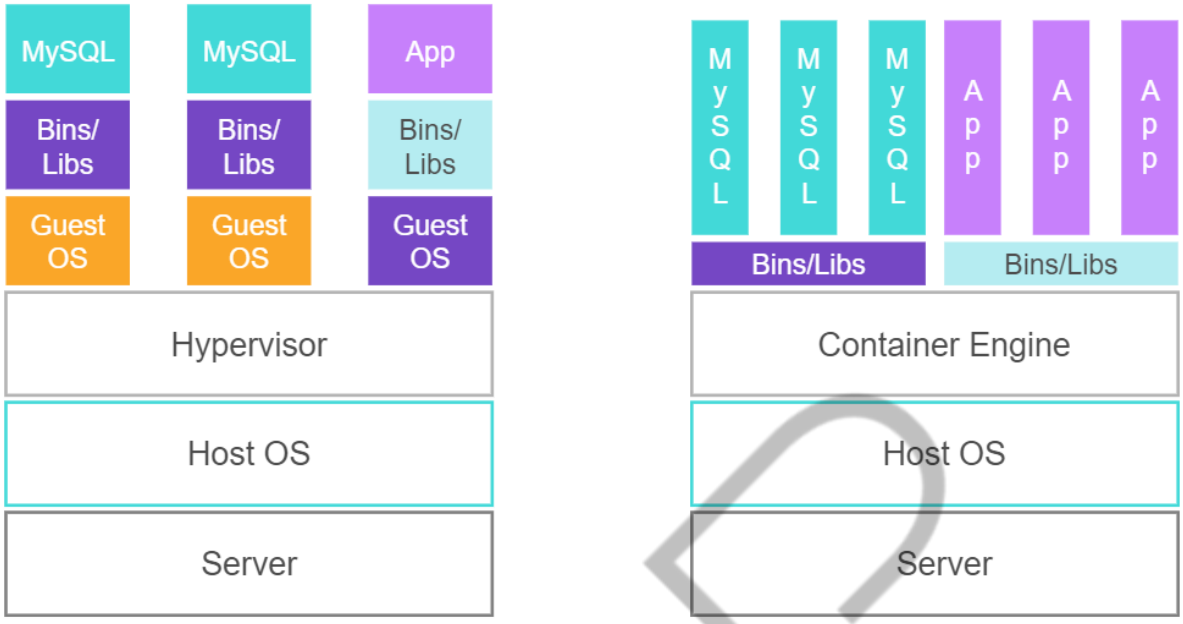


Figura 7.4 – Tipos de virtualização
Fonte: Elaborado pela autora (2019)

A virtualização de máquina trouxe uma série de benefícios, entre eles temos a escalabilidade e a elasticidade, que podem ocorrer de duas formas diferentes: com hipervisor (H-based) ou em nível de sistema operacional (SO) (OS-based).

Vamos conhecer, então, como usar e quais as principais características e benefícios de cada um deles.

### 7.1.1 Sistema virtualizado por máquina virtual (H-based)

H-based é o tipo mais comum. Uma máquina virtual (VM, do inglês *Virtual Machine*) requer um Sistema Operacional (SO) completo e exclusivo, além de *kernel* próprio, binários, aplicativos e bibliotecas, o que exige dimensionar espaço grande no servidor e custo de manutenção. Diversas VMs com SOs diferentes podem ser executadas e uma mesma infraestrutura física.

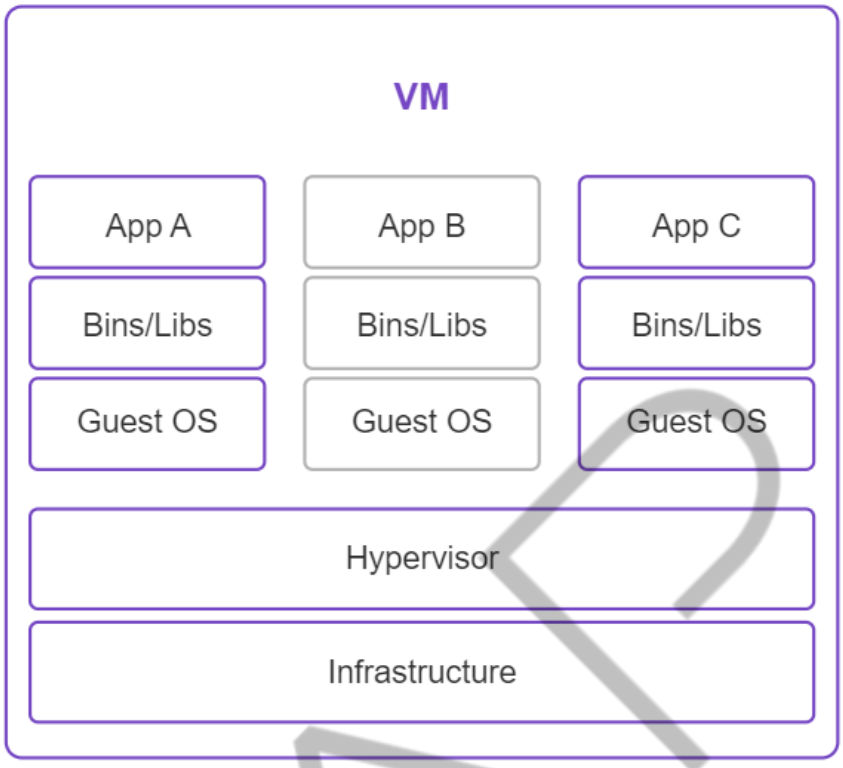


Figura 7.5 – Virtualização por VM
Fonte: Docker Get Started (2019)

Virtualização por VM necessita de uma camada intermediária chamada *hypervisor,* para gerenciar a comunicação de cada VM com o SO hospedeiro (exemplo: VMWare, Hyper-V e o VirtualBox). O hypervisor é um elemento fundamental para virtualizar o servidor, pois é responsável por criar e executar VMs com o objetivo de possibilitar que um determinado *software* seja executado sobre um servidor físico para emular um sistema de *hardware.*

Os benefícios encontrados na criação de virtualização por VM são:

- Capacidade de consolidar aplicações em um único sistema.
- Possibilidade de recuperação de desastres (disaster recover).
- Redução de custos com rápido provisionamento.
- Possibilidade de usar os servidores liberados para montar ambientes de testes.

Apesar dos benefícios listados, uma VM contém vários GBs e demora minutos para ser executada, assim, foi necessário criar uma alternativa mais simples e leve, surgindo, então, a virtualização com container.

### 7.1.2 Sistema virtualizado por container (OS-based)

Implementações de virtualização de containers estão fazendo sucesso na *Cloud Computing* devido à facilidade de uso, além de possibilitar melhor gerenciamento de serviços e melhor gestão dos recursos computacionais da nuvem. “A conteinerização (OS-based) passou a ser amplamente difundida com o surgimento do Docker em 2013” (DOCKER, 2016).

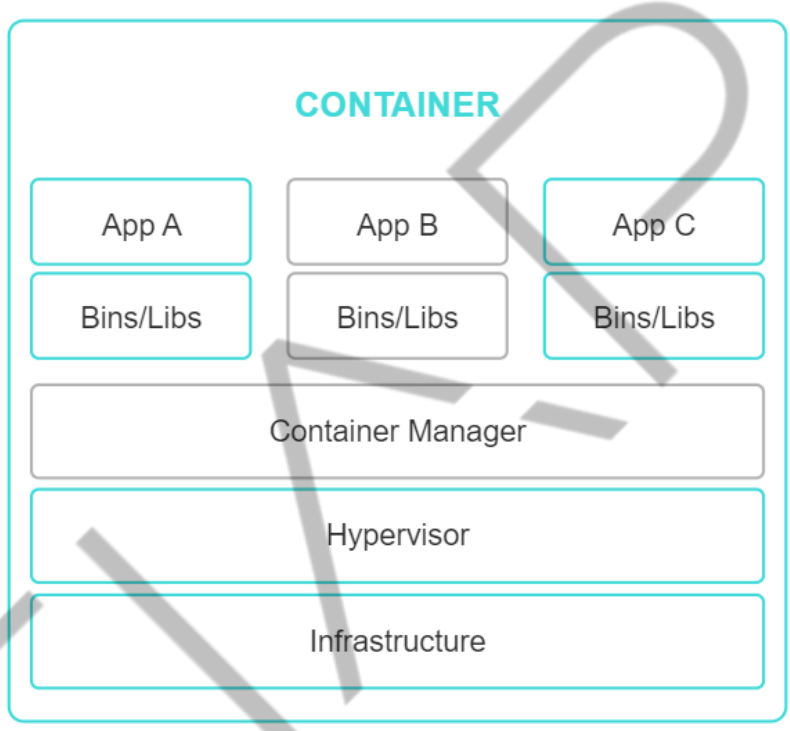


Figura 7.6 – Virtualização por container
Fonte: Docker Get Started (2019)

Virtualizar por *container* possibilita, por exemplo, portar sua aplicação diretamente do seu notebook para o servidor de produção ou para uma instância virtual em uma nuvem pública.

### 7.1.3 VM ou container?

Em suma, a grande diferença entre os tipos de virtualizadores que você viu neste capítulo é que os containers são executados como um processo isolado dentro do *host* e compartilham o *kernel* (SO), enquanto a VM possui um SO completo para cada máquina virtual.

| Virtual Machine | Container |
|---|---|
| Desempenho limitado | Desempenho nativo |
| Virtualização em nível de *hardware* | Virtualização de SO |
| Aloca memória necessária | Requer menos espaço de memória |
| Cada VM é executada em seu próprio SO | Todos os containers compartilham o SO |
| Tempo de inicialização em minutos | Tempo de inicialização em milissegundos |

Quadro 7.1 – Diferenças VM x Container
Fonte: Elaborado pela autora (2019)

- **VM:** é a melhor opção quando se tem uma grande variedade de instâncias de SOs para gerenciar ou quando você precisa executar vários aplicativos em servidores diferentes.
- **Container**: é a melhor opção quando sua a prioridade for executar o maior o número de aplicativos em um menor número de servidores.

Comparando a virtualização por container (*OS-based*) com máquinas virtuais (*H-based*), é possível afirmar que a *OS-based* é mais vulnerável nos quesitos isolamento e segurança, mas apresentam melhor desempenho e flexibilidade (ALLES; CARISSIMI; SCHNORR, 2018).

Pode acontecer de a sua empresa necessitar da configuração de ambos os tipos de virtualização que podem ser utilizados concomitantemente para fornecer ambientes com funcionalidade máxima. Esses tipos também trazem benefícios e desvantagens e a decisão final vai depender das necessidades específicas de cada organização.

## 7.2 Container e microcontainers

O conceito de container surgiu no contexto da virtualização com o propósito de minimizar os investimentos em *hardware* redundantes e de facilitar a gestão de recursos computacionais.

Containers são uma unidade única de *software* que aglomera o código e as dependências de uma aplicação, de maneira que esta seja executada fácil e eficazmente (FELTER *et al*, 2015). Ou seja, você pode executar sua aplicação de forma independente do sistema, pois o container é o próprio pacote executável que carrega dentro de si tudo o que a sua aplicação necessita para ser executada, como

as definições do sistema, bibliotecas, códigos e ferramentas, que são tratados como microcontainers dentro de um container.

Um container é agnóstico da infraestrutura que o hospeda, o que contribui para a confiabilidade ao executar a sua aplicação nos diversos ambientes de implantação, reduzindo, assim, possíveis conflitos. Cada uma das aplicações de um container possui uma área reservada de recursos computacionais e é executada de forma isolada umas das outras.

O container Docker surgiu como uma alternativa mais leve às máquinas virtuais. Docker é uma plataforma *open source* da DotCloud, uma empresa de PaaS (*Platform as a Service*) que utiliza o conceito de container para “empacotar” a aplicação que poderá ser reproduzida em qualquer plataforma após ser transformada em uma imagem Docker.

### 7.3 Docker

Container Docker é uma plataforma aberta para criação, execução e publicação (*deploy*) de containers. Usa o Linux Container (LXC) como *back-end* e foi desenvolvido na linguagem de programação Go (criada pela empresa Google), possuindo alto desempenho e tendo como objetivo facilitar o desenvolvimento, implantação e execução de aplicações em ambientes isolados da forma mais rápida possível. Por exemplo, a implantação dos microsserviços ocorre de forma completamente automatizada através da Automação e do Docker.

Quando se fala em Docker, as palavras-chave que remetem a este tipo de container são “Desenvolver”, “Entregar” e “Rodar em qualquer ambiente”.

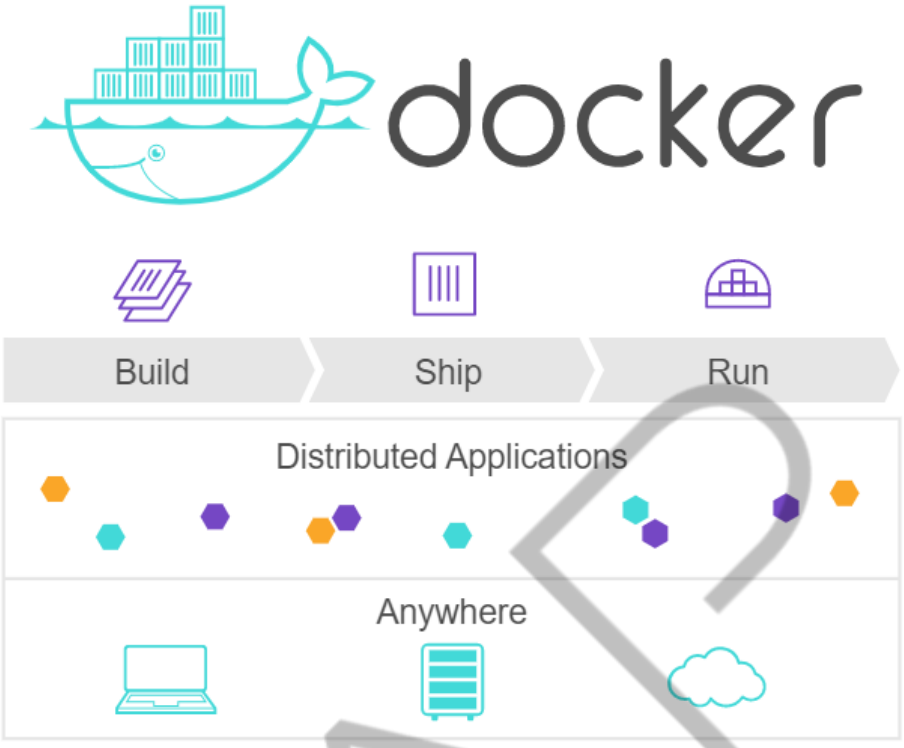


Figura 7.7 – Essência do container Docker
Fonte: Elaborado pela autora (2019)

Veja quais são as vantagens e benefícios do virtualizador de containers Docker.

### 7.3.1 Vantagens e benefícios no uso do Docker

As vantagens propostas por esse tipo de virtualização são:

- Portabilidade.
- Flexibilidade.
- Escalonamento.
- Disponibilidade.
- Velocidade.

E os inúmeros benefícios são:

- Rapidez no *boot*.
- Redução de custos.
- Implantação simplificada.

- Padronização: as imagens são construídas por meio de arquivos de definição.
- Garante performance consistente em todos os ambientes que a aplicação executará (exemplo: ambiente de testes, ambiente virtual e ambiente de produção).
- Possibilita compartilhamento de arquivos entre *host* e *container*.
- Execução de *deploys.*
- Gerenciamento da infraestrutura da aplicação.
- Agilidade no processo de criação, manutenção e evolução da aplicação.
- Aumento de produtividade com a integração das equipes de desenvolvimento e s*ysadmin.*
- Economia de recursos, inclusive recursos físicos, simplificando a aplicação da metodologia DevOps e facilitando o desenvolvimento ágil.
- Possibilidade de fazer *upload* de vários containers simultaneamente, consumindo menos recursos do *hardware* virtual ou físico.
- Controle de versão de imagens que facilitam o processo de *rollback* de forma rápida em caso de problemas.
- Suporte à implementação de CI/CD (Continuous Integration / Continuous Deployment).

Diversas soluções de hospedagem adotaram esta tecnologia, levando à disseminação da plataforma por centenas de empresas, como Uber, PayPal, eBay, Spotify etc.

Você vai ver, em seguida, algumas dicas de como trabalhar com um container Docker.

### 7.3.2 Componentes da arquitetura do container Docker

Os principais componentes da arquitetura do Docker envolvem:

- **Docker para Mac, Linux e Windows:** versões que permitem instalar e executar containers nos sistemas operacionais de forma isolada.
- **Docker Daemon:** *software* que roda na máquina onde o Docker está instalado. Usuário não interage diretamente com o Daemon.
- **Docker Client:** CLI ou REST API que aceita comandos do usuário e repassa estes comandos ao Docker Daemon.
- **Docker Image:** é um *template* – uma imagem contém todos os dados e metadados necessários para executar containers a partir de uma imagem.
- **Docker Container:** detém tudo o que é necessário para uma aplicação ser executada. Cada container é criado a partir de uma imagem Docker. Cada container é uma aplicação isolada independente.
- **Docker Engine:** usado para criar imagens e containers no Docker.
- **Docker Dockerfile:** arquivo-texto que contém uma sintaxe simples para a criação de novas imagens Docker a partir de uma imagem base. Todo container começa com um Dockerfile.
- **Docker Registry:** coleção de imagens hospedadas e rotuladas que juntas permitem a criação do sistema de arquivos de um container. Um registro pode ser público ou privado.
- **Docker Hub:** registro usado para hospedar e baixar diversas imagens Docker. Pode ser visto como uma plataforma SaaS (*Software as a Service*) de compartilhamento e gerenciamento de imagens Docker.
- **Docker Compose:** componente para definir aplicações usando diversos containers no Docker.
- **Docker Swarm**: ferramenta que permite o agrupamento (*clustering*) de Containers Docker.

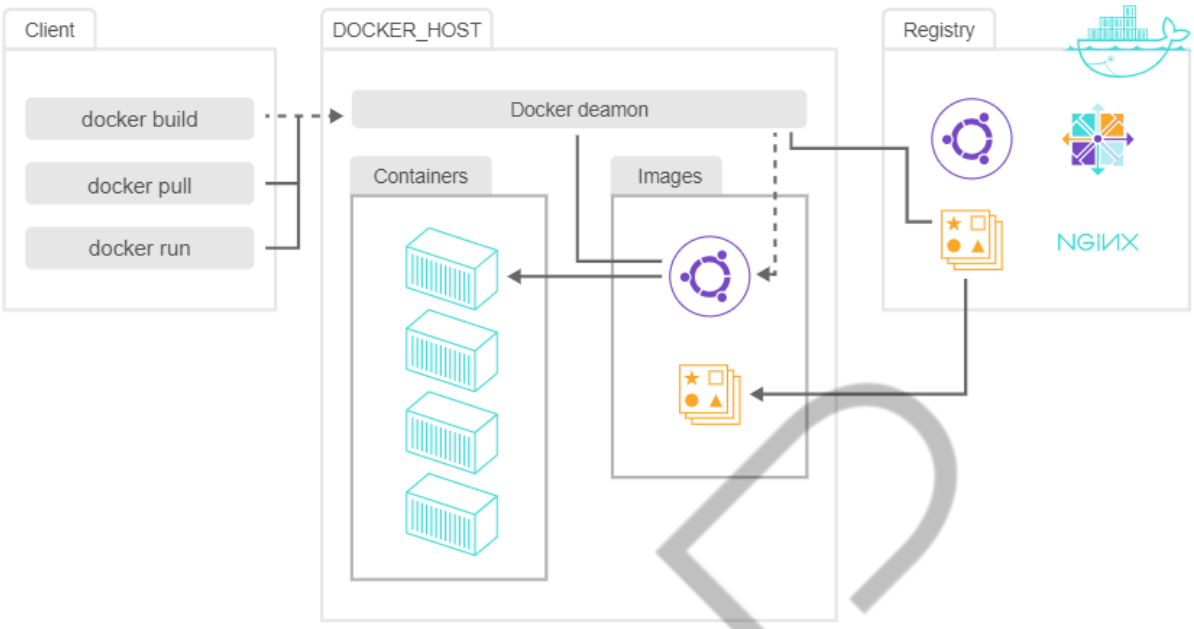


Figura 7.8 – Componentes da arquitetura do Docker
Fonte: Docker Docs (2019)

O container Docker faz a comunicação entre cliente e servidor por meio de uma API e conta com o *docker engine* para resolver o problema de dependência de infraestrutura, possibilitando que os aplicativos em container sejam executados em qualquer lugar de forma consistente.

### 7.3.3 Como funciona o Docker?

Os containers têm um *microkernel*, ou seja, acessam diretamente o SO hospedeiro e representam uma unidade isolada de disco, memória, processamento e rede próprios para prover um ambiente virtual, contendo apenas os ativos encapsulados da aplicação, o que reduz o tempo de *deploy*.

Essa separação permite uma grande flexibilidade e possibilita que ambientes distintos coexistam na mesma máquina hospedeira (*host*), sem causar problemas. Por exemplo, o ambiente de container permite instalar os requisitos (*softwares*, arquivos etc.) que um microsserviço necessita para executar sem se preocupar com as instalações de outro SO.

O container é construído com recursos de kernel do Linux, como "**namespaces**", "**cgroups**", "**chroot**" entre outras funcionalidades, para isolar uma área específica para sua aplicação.

O Docker Registry (registro) é um repositório provido pelo Docker que se encontra na nuvem e disponibiliza uma área para envio, download, pesquisa e compartilhamento de imagens (*snapshots*) de containers.

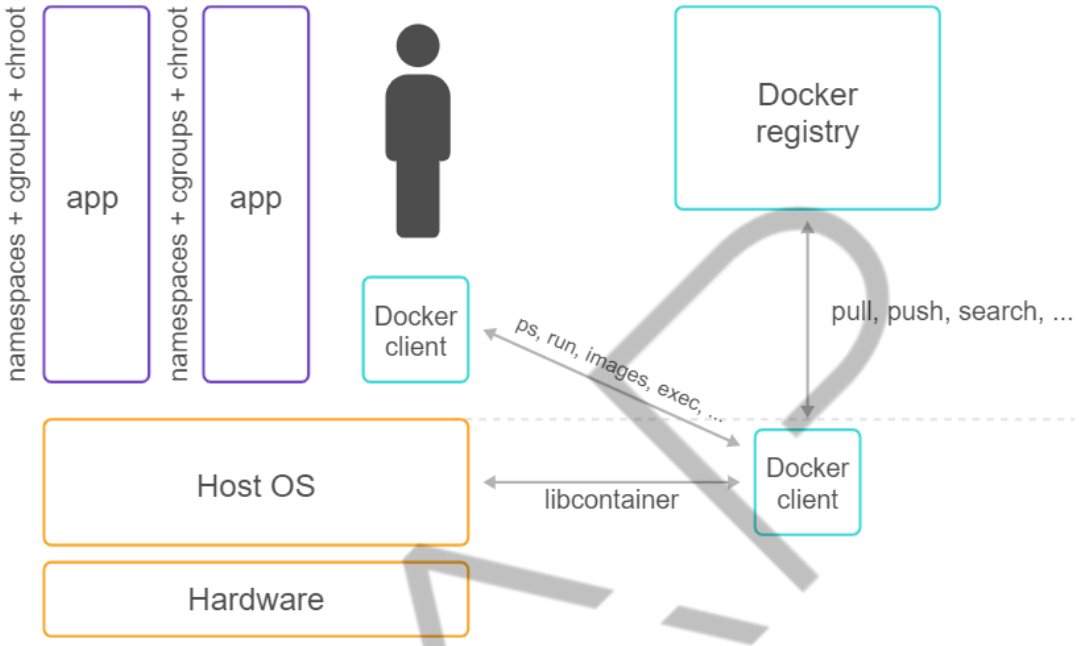


Figura 7.9 – Arquitetura simplificada do Docker para host único
Fonte: Google Imagens, adaptado por FIAP (2019)

Uma aplicação necessita de diversos containers para funcionar. Gerenciar múltiplos containers envolve certa complexidade e orquestrar todos esses componentes de forma manual pode levar a ocorrências de falha humana. Foi aí que surgiu o **Docker Compose**, com a proposta de facilitar a vida dos DevOps (mais especificamente dos TechOps) na tarefa de gerenciamento de diversos containers.

### 7.3.4 Boas Práticas para uso do Docker

Como você já viu, o container Docker tem diversas vantagens e benefícios, e como qualquer outra tecnologia, deve ser bem empregado para se atingir o máximo de seu potencial.

Como boas práticas, listamos:

- Um container Docker é apenas um serviço (um processo do *host* hospedeiro) e não deve ser tratado como uma máquina virtual.

- Por ser um processo do *host* hospedeiro, não deve ter vida longa (*uptime*), precisa ser iniciado e encerrado.
- Não se deve armazenar dados dentro do container Docker. Para armazenar dados dinâmicos, faça uso do recurso de criar e montar um container como um volume de dados.
- Para acesso aos containers Docker, o *host* hospedeiro deve prover recursos de segurança essenciais.

### 7.4 Automação e implantação em container Docker

A plataforma Docker possibilita o uso de ferramentas no ambiente de aplicativos que tem como objetivo reduzir o atrito, acelerar a taxa de mudança e melhorar eficiências.

Devido à característica de suportar a implementação de CI/CD, adotar um fluxo de trabalho de DevOps implementando “Container as a Service” baseado no container Docker possibilita ao time DevOps o desenvolvimento de código de forma colaborativa através do compartilhamento de imagens.

Com um *pipeline* unificado, *softwares* e dependências podem ser facilmente compartilhados com as operações de TI e com ambientes de produção, minimizando conflitos de aplicativos entre diferentes ambientes, garantindo segurança e uma cadeia de confiança para seu conteúdo. Como resultado, os *pipelines* construir/testar/implantar ficam mais simples e facilitam a implantação para diferentes infraestruturas.

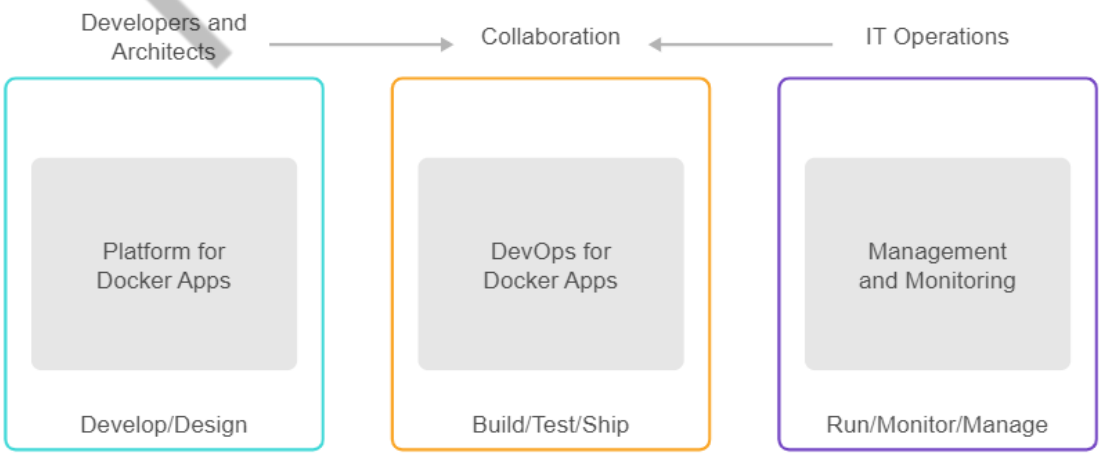


Figura 7.10 – Papéis e responsabilidades em container Docker
Fonte: Google Imagens, adaptado por FIAP (2019)

- **Desenvolvedores e Arquitetos**: responsáveis por gravar e executar código localmente em containers do Docker, fazendo uso do Docker para Windows ou para Mac.
- **DevOps**: responsáveis por construir/testar/implantar. É o time DevOps que opera o *pipeline* da integração contínua (*Continuous Integration – CI*), fazendo uso do Dockerfile fornecido no repositório de código.
- **Operações de TI**: responsáveis pelo gerenciamento do ambiente de produção, monitoramento, escalabilidade, infraestrutura e também por garantir que os aplicativos operem da forma esperada para os usuários finais.

Como benefícios importantes para a empresa, por um sólido fluxo de trabalho de DevOps para aplicativos em container Docker, temos:

- Vantagem competitiva devido a fluxos de trabalho contínuos.
- Entrega do produto final com maior qualidade, mais agilidade e melhor conformidade.
- Melhor rastreabilidade.
- Inovação mais rápida e com menor custo de investimento.
- Economia de custos ao realizar uma combinação de recursos locais e em nuvem.
- Gerenciamento de ambientes de forma mais eficiente.
- Colaboração entre desenvolvedores e operadores de TI.
- Maior transparência entre as áreas envolvidas.
- Ferramentas totalmente integradas.
- Utilização dos recursos provisionados com mais eficiência.

A prática da colaboração entre os pilares de “Desenvolvedores e Arquitetos”, “DevOps” e “Operações de TI” promove um ciclo de aprendizagem contínua dos DevOps e é capaz de substituir pela automação os processos manuais suscetíveis a erros. Nenhuma das áreas envolvidas precisa conhecer o conteúdo dos demais

containers. Cada uma delas é responsável por desempenhar suas atividades sem se preocupar com os mecanismos e funcionamentos das demais áreas.

### 7.5 Segurança dos containers

A virtualização por container baseia-se em propriedades semelhantes às das máquinas virtuais, porém, a adoção desta tecnologia no ambiente de produção ainda tem como maior barreira a segurança (CLUSTERHQ, 2016). Quando se fala em compartilhamento de recursos através de um mesmo núcleo (container), vêm à tona questões sobre riscos e vulnerabilidades.

Para garantir que apenas o container comprometido seja impactado em uma situação de ataque, o ideal é implementar uma camada de separação e isolamento. (SANS.ORG, 2015). Segurança de container envolve segurança dos recursos, problema com código legado e complexidade na orquestração em escala.

Implementar técnicas de Controle de Acesso pode auxiliar a contornar possíveis ameaças. Neste caso, estamos nos referindo ao controle sobre acesso aos recursos do sistema após autenticação das credenciais e da identidade da conta de um usuário que teve acesso concedido com sucesso.

Por exemplo, é possível restringir acesso a alguns recursos específicos para um determinado usuário ou grupo de usuários, enquanto outros podem ter acesso permitido após o login no sistema. Podem-se aplicar técnicas de Controle de Acesso Discricionário (*Discretionary Access Control – DAC)* ou Controle de Acesso Obrigatório (*Mandatory Access Control – MAC)*.

- **Controle de Acesso Discricionário (DAC)**: normalmente, é o mecanismo de controle de acesso padrão para a maioria dos SOs. Oferece ao usuário a possibilidade de definir permissões de acesso para recursos próprios, ou seja, somente para os recursos que estão associados a ele através de uma lista de controle de acesso (ACL, do inglês Access Control List).
- **Controle de Acesso Obrigatório (MAC)**: é o mais rigoroso de todos os níveis de controle. O acesso a qualquer recurso (como os arquivos de dados) é estritamente controlado pelo SO com base nas configurações que

foram definidas pelo administrador do sistema. O usuário não tem permissão para alterar um controle de acesso ao recurso.

Existem outras preocupações que devem ser consideradas, e para identificar qual tipo de segurança para containers se faz necessária, considere os seguintes critérios:

- **Segurança do container em si**: analisar que tipos de mecanismos podem ser aplicados entre o núcleo do SO e os demais containers que garantam o isolamento e possibilitem o controle de acesso aos dados e limitação de recursos por usuário. Os critérios normalmente são configurados pelo administrador do sistema e podem ser ativados por padrão no momento da inicialização do container.
- **Segurança dos recursos na comunicação entre containers:** reduzir a área de ataque restringindo o acesso e limitando a comunicação apenas entre containers que foram previamente especificados para garantir integridade e confidencialidade das informações que estão sendo trafegadas na rede.
- **Segurança das imagens do container**: é preciso estabelecer um processo de auditoria para analisar possíveis vulnerabilidades por falta de atualização nas imagens que estão hospedadas na comunidade do Docker e no repositório oficial Docker Hub. O repositório de vulnerabilidades chamado Banco de Dados Nacional de Vulnerabilidades (*National Vulnerability Database – NVD)* mantém informações sobre vulnerabilidades e exposições de segurança que estão catalogadas com ID único; e publica uma lista de Vulnerabilidades e Exposições Comuns (*Common Vulnerabilities and Exposures – CVE*) com uma análise de segurança correspondente.

### 7.6 Instalando um Docker

Você deve seguir os passos elencados abaixo, de acordo com o sistema operacional.

### 7.6.1 Instalando um Docker em sistema operacional Windows

- Baixe o Docker para Windows pelo link: <https://download.docker.com/win/beta/InstallDocker.msi>
- Verifique se o suporte à virtualização está habilitado em seu computador.
- Seu Windows deve ser 64 bits versão 1607 e build: 14393.0.
- Você deve habilitar o recurso do hyper-v.

### 7.6.2 Instalando um Docker em sistema operacional para Linux, Ubuntu

- Abra o terminal com o atalho Ctrl + Alt + T. Baixe as últimas atualizações do sistema:

```
$ sudo apt update && sudo apt upgrade
```

- Instale o Docker utilizando o repositório do Ubuntu 17.04:

```
$ sudo apt install docker.io
```

- Inicie o Docker:

```
$ sudo systemctl start docker
```

- Entretanto, garanta que ele seja iniciado após a reinicialização:

```
$ sudo systemctl enable docker
```

- Caso queira verificar a versão instalada:

```
$ docker -v
```

### 7.6.3 Instalando um Docker em sistema operacional MacOS

Certifique-se de estar utilizando MAC OS X Sierra 10.12 ou superior.

- Baixe o Docker para Mac através do link: https://download.docker.com/mac/beta/Docker.dmg

- Dê duplo clique no arquivo DMG, autorize a instalação e informe sua senha de administrador; depois dê um duplo clique em Docker.app para iniciar o Docker.
- Após baixar, rode em uma janela do terminal os seguintes comandos para verificar se está tudo ok:

```
$ docker -version
$ docker-compose --version
$ docker-machine --version
```

## 7.7 Docker container e a relação com os microsserviços

Containers são áreas definidas pelo Docker no servidor, onde as estruturas de microsserviços podem ser implementadas compartilhando o mesmo sistema operacional (SO). Desta forma, um microsserviço pode ser implementado dentro de um container de forma modular e de acordo com as necessidades de funcionamento de um container. Cada microsserviço pode executar um processo único e se comunicar com outros.

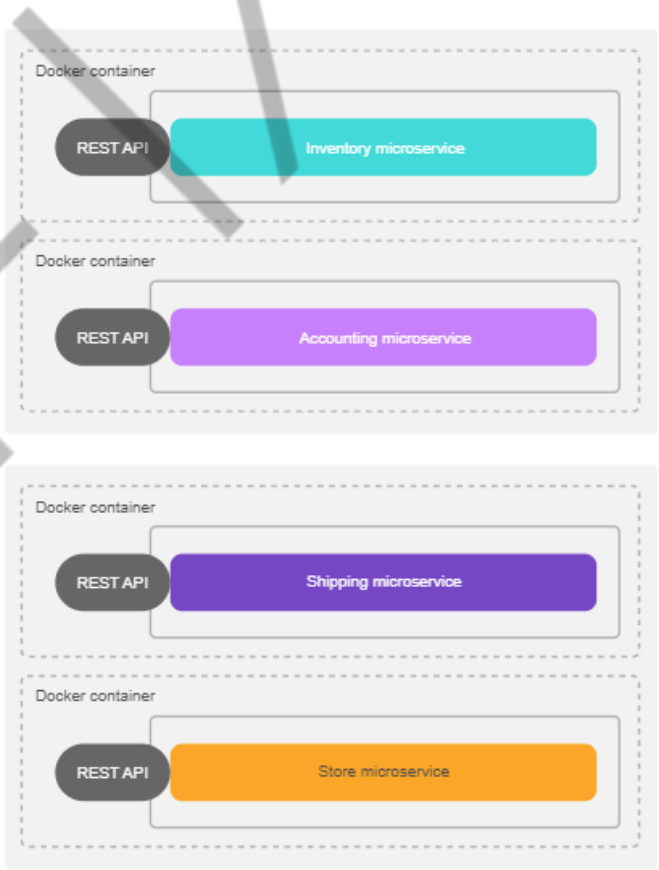


Figura 7.11 – Container Docker e Microsserviços
Fonte: Google Imagens, adaptado por FIAP (2019)

O pacote de características de um container, que você viu no capítulo anterior, foi o que possibilitou o surgimento dos microsserviços que compõem uma determinada aplicação. Considerando a possibilidade de desenvolver e realizar o *deploy* dos aplicativos de forma muito rápida e independente e a possibilidade de gerenciamento de cada microsserviço.

### 7.7.1 Container Docker x Monolíticos e Microsserviços

O que você já sabe sobre os monolíticos é que são arquiteturas tradicionais que, devido ao seu tipo funcionamento, são difíceis de escalar, porque implementam todo o aplicativo (serviços diversos, bibliotecas etc.) em uma única unidade, apresentando complexidade para manter e atualizar. Contudo, são menos complexos para gerenciamento do contexto de transações e segurança, pois possuem apenas uma entrada para o aplicativo.

Em contrapartida, temos a arquitetura de microsserviços e os aplicativos baseados em container. Na arquitetura de microsserviços, cada componente do aplicativo realiza uma única função e executa de forma independente. A arquitetura baseada em container abriga microsserviços, facilitando a implantação de um aplicativo que ocorre de forma “modular” e individual, causando o mínimo impacto possível na disponibilidade geral do sistema.

E qual a relação do container Docker com os Monolíticos e os Microsserviços?

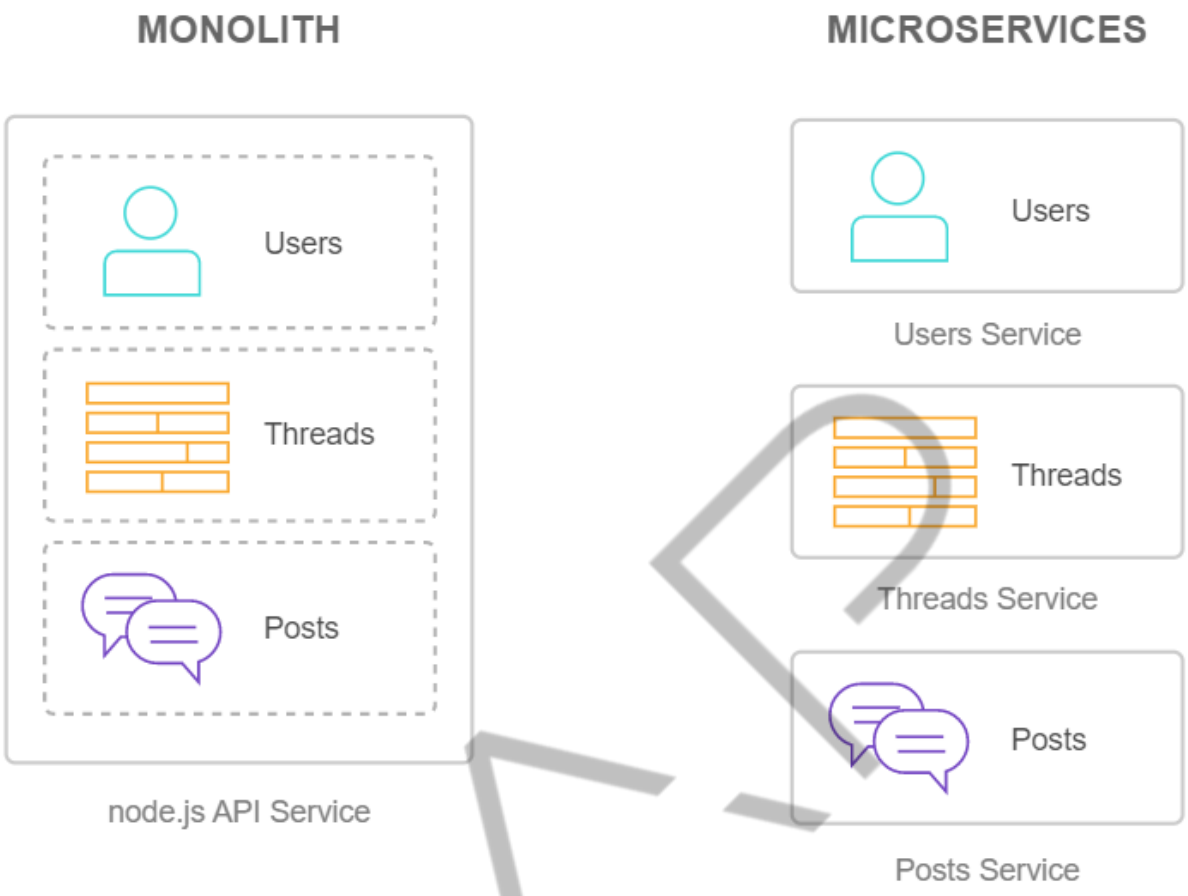


Figura 7.12 – Monolíticos e Microsserviços em container Docker
Fonte: Google Imagens, adaptado por FIAP (2019)

A seguir, veja o comportamento de cada um deles.

### 7.7.2 Arquitetura monolítica e o container Docker

Dentro de um **Container Docker**, o aplicativo node.js de uma arquitetura monolítica se comporta como um “bloco” para implantação e é executado como um único serviço, compartilhando os recursos disponíveis para todos os demais containers. Em uma situação de pico de demanda, é preciso escalar toda a arquitetura, porque um monólito não permite dimensionar partes individuais. A função do *cluster* principal do node.js é distribuir o tráfego para os operadores que compõem o aplicativo monolítico.

### 7.7.3 Arquitetura de microsserviços e o container Docker

O aplicativo node.js de uma arquitetura de microsserviços em um container Docker é composto por recursos individuais que rodam como serviço independente dentro do seu próprio container. Cada recurso pode ser atualizado e escalado separadamente dos demais. No *cluster* principal do node.js, um microsserviço se comunica com outros serviços através de uma API.

## 7.8 *Storage* no container Docker

É possível persistir dados no container Docker?

Para responder a esta pergunta, precisamos previamente entender como o Docker gerencia seu sistema de arquivos nas imagens e nos containers em execução. O Docker é totalmente baseado no conceito de *layers* (camadas).

- **Imagem e *layers:***

Uma imagem do Docker é composta por diversos *layers*. Cada *layer* representa uma instrução no Dockerfile da imagem. As camadas são empilhadas umas sobre as outras e quando você cria um novo container, adiciona uma nova camada *writable* sobre as camadas subjacentes.

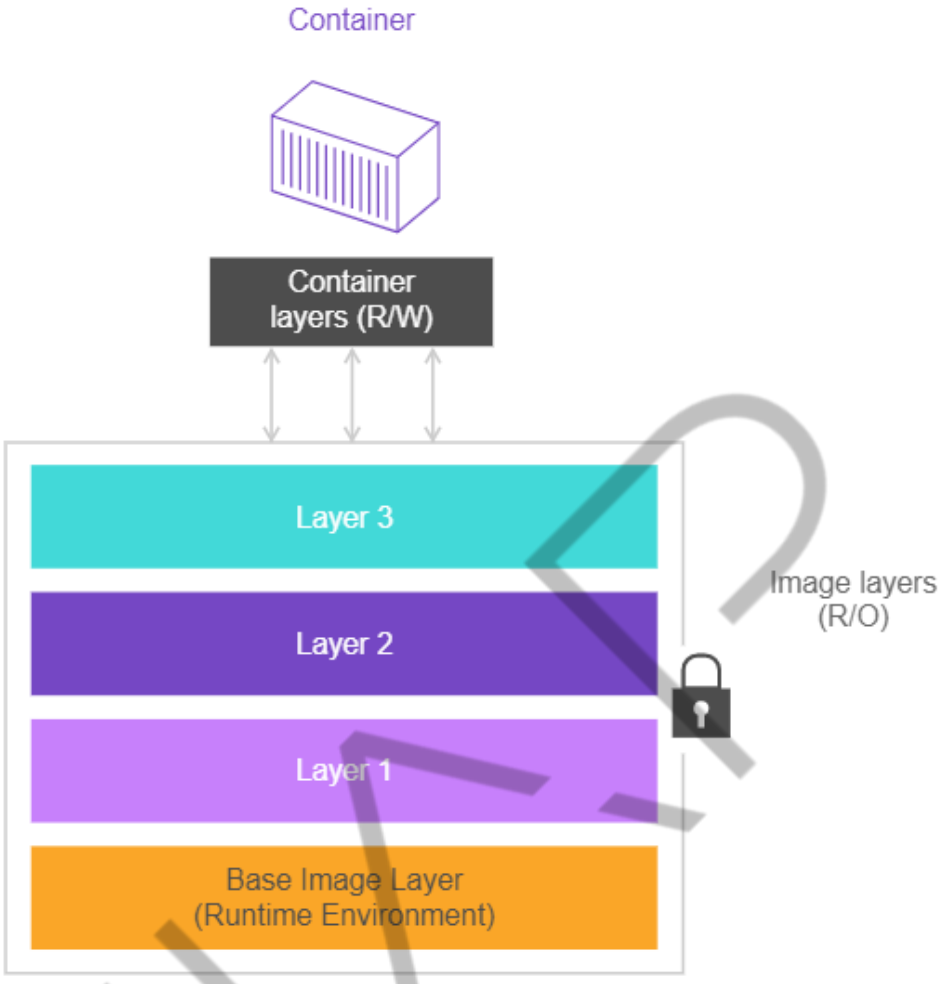


Figura 7.13 – Representação imagem e *layers*
Fonte: Google Imagens, adaptado por FIAP (2019)

- **Container e *layers:***

A principal diferença entre um container e uma imagem é a camada *writable* superior. Todas as gravações no container, que adicionam novos dados ou modificam dados existentes, são armazenadas nessa camada *writable*. Quando o container é excluído, a camada *writable* também é. A imagem subjacente permanece inalterada.

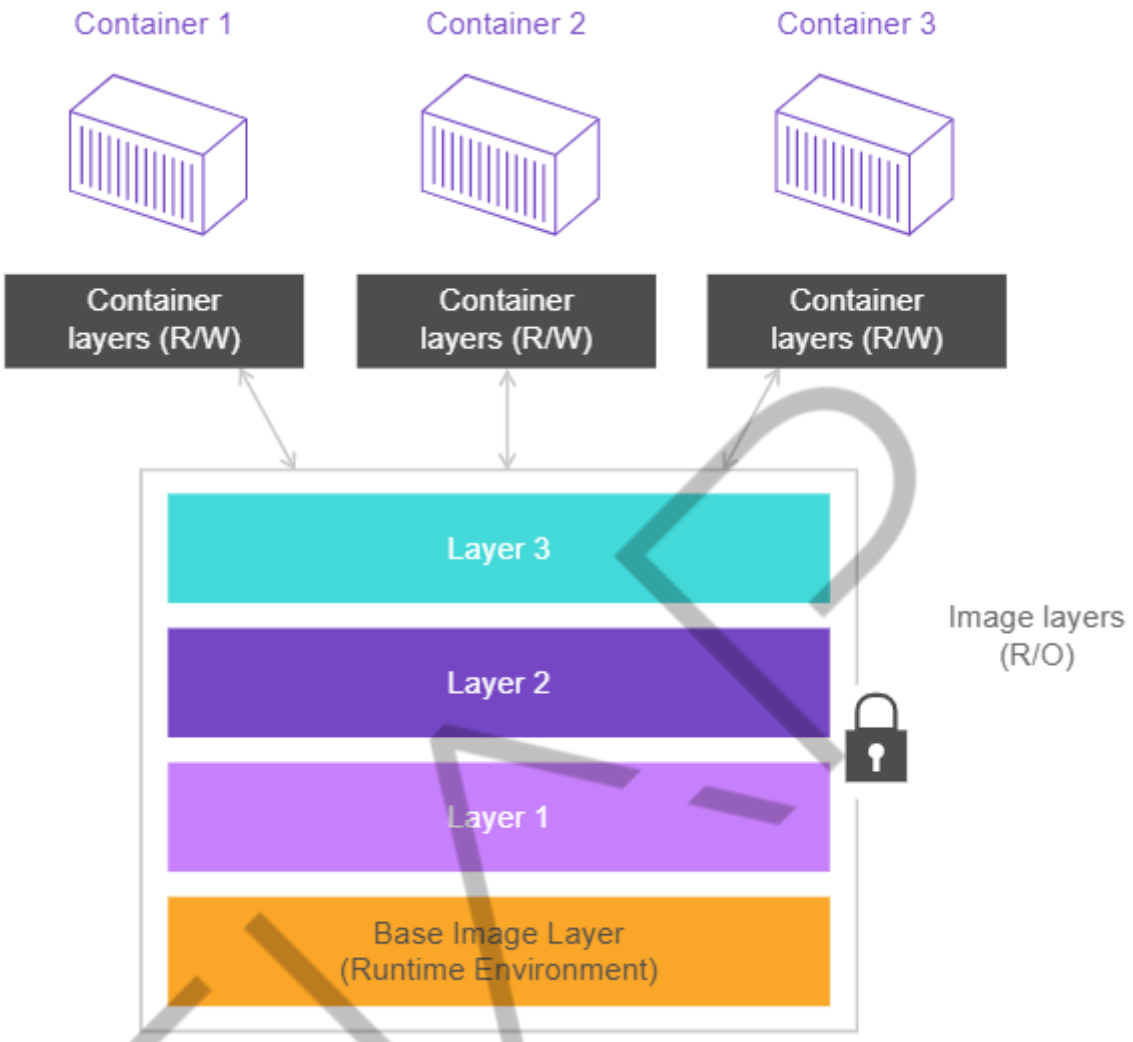


Figura 7.14 – Representação container e *layers*
Fonte: Google Imagens, adaptado por FIAP (2019)

Para que seja possível gerenciar as interações e os conteúdos dos *layers* de imagem e dos *layers* de container *writable*, é necessário fazer uso de *storage drivers* (*drivers* de armazenamento). O *storage driver* é responsável por escrever na camada *writable* de um container. Cada *storage driver* trata a implementação de forma diferente, podendo ser por **volume**, por **montagem de ligação** ou por **montagem tmpfs** (se você estiver executando o Docker no Linux), porém todas elas usam a tecnologia de cópia na gravação (*CoW - Copy-on-Write*) e *layers* de imagens empilháveis.

**DICA**: Containers devem ser leves.
Adicionar dados desnecessários os tornam pesados para criar e executar.

Veja quais são as maneiras de persistência de dados que o Docker fornece ao realizar o *storage* da máquina *host* em containers:

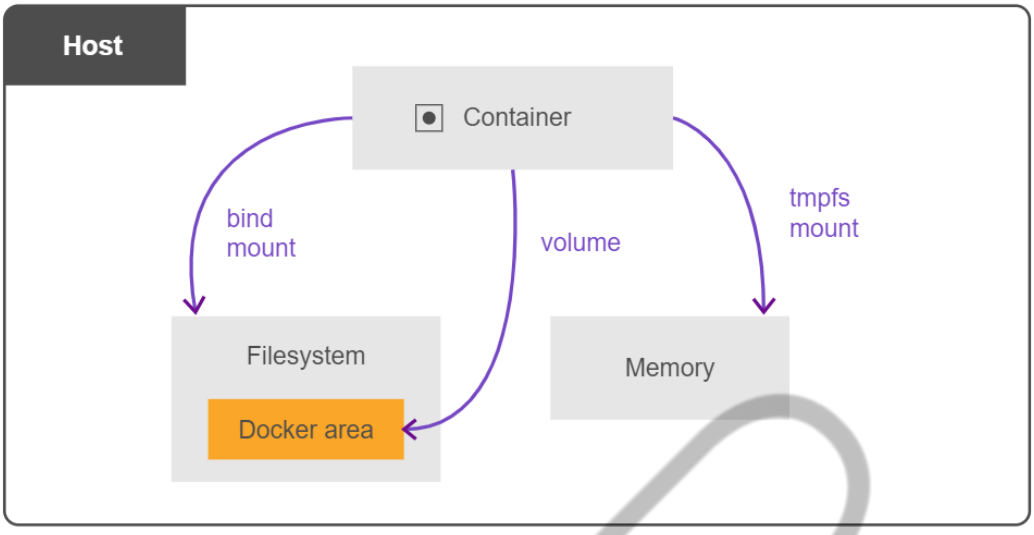


Figura 7.15 – Formas de *storage* em um container
Fonte: Docker Docs (2019)

- **Montagens de ligação (*bind mounts*):** estão em qualquer lugar no *host*, mas podem ser modificadas por outros aplicativos.

As montagens de ligação nasceram com o Docker e são muito eficientes, mas dependentes do sistema de arquivos da máquina *host* que possui uma estrutura de diretórios específica. Um arquivo ou diretório precisa existir na máquina *host* do container Docker.

- **Montagens tmpfs (*temporary filesystem mounts*)**: estão no espaço da memória do *host* e nunca são gravadas no sistema de arquivos da máquina *host*.

Considere o uso de uma montagem tmpfs no caso de o container gerar dados de estado não persistentes. Este tipo de *storage* evita gravar dados na camada *writable* do container e aumenta seu desempenho.

- **Volumes**: fazem parte do sistema de arquivos do *host*, mas são gerenciados pelo Docker em um *path* específico e não devem ser modificados por outros aplicativos. Usar volumes para persistir dados da sua aplicação é garantir maior disponibilidade para seu ambiente.

### 7.8.1 Persistindo dados no ambiente Docker

Trabalhar com volumes foi considerada a melhor forma para resolver grande parte dos problemas de persistência de dados em um container, pois o conteúdo do volume fica fora do ciclo de vida do container; é também a melhor maneira de evitar problemas de desempenho ao longo do caminho, devido ao fato de que os volumes não aumentam o tamanho dos containers que os utilizam.

Os volumes são completamente gerenciados pelo Docker, por isso, é o mecanismo preferido para persistir dados gerados e usados pelos containers do Docker.

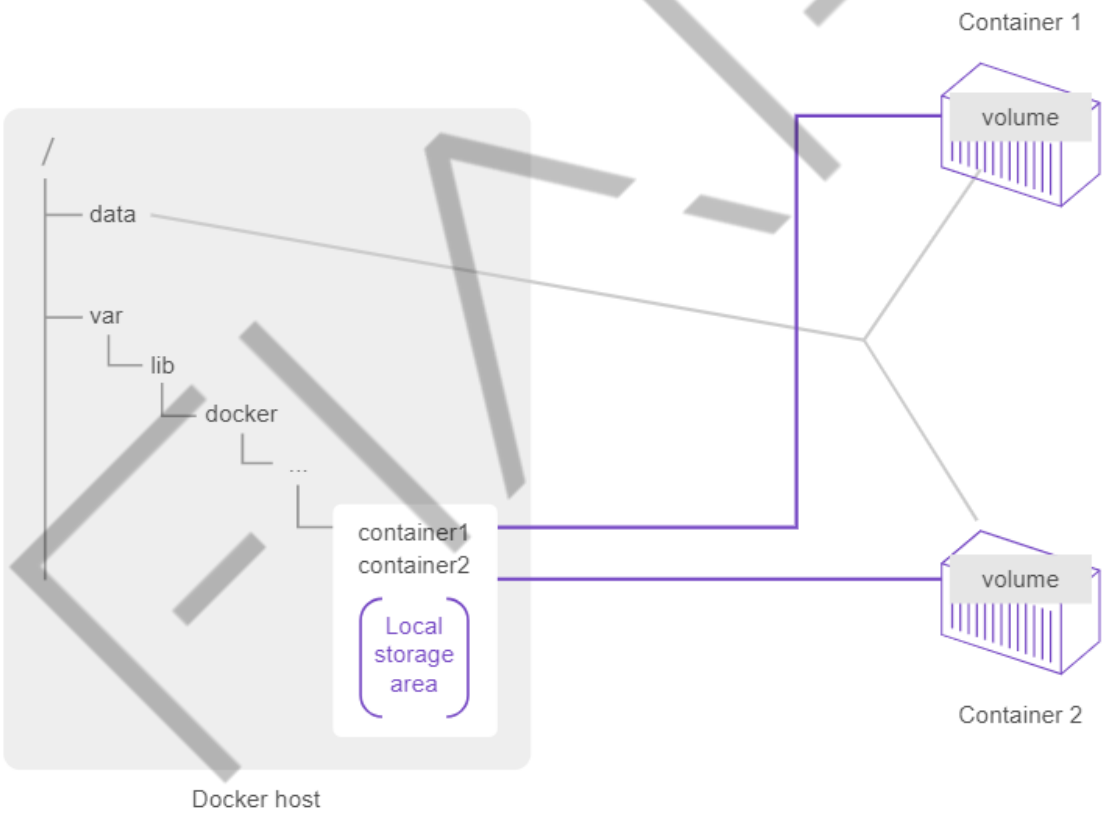


Figura 7.16 – Volumes de dados e o *driver* de armazenamento
Fonte: Docker Docs (2019)

São diversas as vantagens do uso de volumes em relação a montagens de ligação:

- Funcionam nos containers do Linux e do Windows.
- São mais fáceis de fazer backup ou migrar.
- Podem ser compartilhados com mais segurança entre containers.

- O conteúdo de um novo volume pode ser preenchido previamente por um container.
- Os *drivers* de volume permitem armazenar volumes em *hosts* remotos ou provedores de nuvem, criptografar o conteúdo de volumes ou adicionar outras funcionalidades.
- É possível gerenciar volumes fora do escopo de qualquer container usando os comandos do Docker CLI ou a API do Docker.
- Maior disponibilidade para seu ambiente.

### 7.9 Docker Compose

Docker Compose é um gerenciador de containers e sua principal função é executar vários containers como um único serviço. Como exemplo prático, imagine uma situação: seu aplicativo necessita do WordPress e do banco de dados MySQL e, para executá-los, você pode criar um único arquivo que inicia os dois containers como um serviço, sem precisar iniciar cada um deles separadamente.

E como isso é possível? A resposta é: através de um arquivo de configuração!

Esse arquivo de configuração é o docker-compose.yml (ou .yaml), semelhante ao Dockerfile, escrito em YAML (*Ain't Markup Language*). Dentro dele, você define cada container como um serviço e configura todos os parâmetros necessários, tais como variáveis de ambiente, portas expostas e outros mais, para executar cada container. É possível também especificar quais volumes e rede serão criados para serem utilizados nos parâmetros dos serviços.

A grande vantagem de trabalhar com o arquivo compose é diminuir a responsabilidade do desenvolvedor ou do *sysadmin* no gerenciamento do *deploy*, pois o compose encapsula os comandos e todas as dependências que a sua aplicação necessita para executar em qualquer ambiente.

### 7.9.1 Entendendo o arquivo docker-compose

Uma informação muito importante sobre o padrão YAML é que ele é baseado na indentação com espaços como separador dos blocos de códigos das definições, não sendo permitida a utilização do TAB. Caso a indentação não seja feita corretamente, o docker-compose apresentará falha em sua execução.

**DICA**: O arquivo docker-compose pode ser armazenado com extensão yml ou yaml e funciona perfeitamente com ambas.

O caminho-padrão para um arquivo Compose é ./docker-compose.yml.

Vamos, então, a um exemplo de funcionamento do compose:

```
version: "3.7"
services:
  fiap-web:
    build: .
      context: ./dir
      dockerfile: Dockerfile-alternate
      args:
        versao: 1
    ports:
      - "8080:80"
  fiap-image1:
    image: fiap-image1
```

Código-fonte 7.1 – Exemplo completo
Fonte: Elaborado pela autora (2019)

No arquivo acima, a primeira linha define a versão do docker-compose.yml que, no nosso caso, usaremos a versão 3.7.

```
version: "3.7"
```

Código-fonte 7.2 – Exemplo primeiro trecho
Fonte: Elaborado pela autora (2019)

Na segunda linha, no mesmo nível de indentação, temos o início do bloco de **serviços** com o comando **services**.

```
version: "3.7"
services:
```

Código-fonte 7.3 – Exemplo segundo trecho
Fonte: Elaborado pela autora (2019)

Como segundo nível de indentação (*), linha três, temos o nome do primeiro **serviço** desse arquivo, o **fiap-web**, abrindo o bloco de definições do **serviço**, ou seja, tudo que for definido a partir do próximo nível de indentação faz parte desse serviço.

```
version: "3.7"
services:
  fiap-web:
```

Código-fonte 7.4 – Exemplo terceiro trecho
Fonte: Elaborado pela autora (2019)

Seguindo para o próximo nível de indentação (*), temos a primeira definição do **serviço fiap-web**, o build, abrindo um novo bloco de código para parametrizar o funcionamento da construção da sua imagem antes de sua execução.

```
version: "3.7"
services:
  fiap-web:
    build: .
```

Código-fonte 7.5 – Exemplo quarto trecho
Fonte: Elaborado pela autora (2019)

No próximo nível de indentação (*), linha seis, temos o **context** como um parâmetro do **build**. O valor **"./dir"** fornecido no parâmetro **context** é um caminho relativo, indicando que uma pasta chamada **dir** se encontra no mesmo nível de sistema de arquivo do **docker-compose.yml** ou no local onde esse comando será executado. O contexto **"./dir"** mostra que apenas os arquivos que existirem dentro desta pasta poderão ser usados para construir sua imagem.

```
version: "3.7"
services:
  fiap-web:
    build: .
      context: ./dir
```

Código-fonte 7.6 – Exemplo quinto trecho
Fonte: Elaborado pela autora (2019)

Na linha sete, ainda no mesmo nível de indentação da definição **context** e dentro do bloco **build**, temos o **Dockerfile.** Este parâmetro indica o nome do arquivo que será usado para a construção da imagem em questão.

Por padrão, o **docker-compose** procura por um arquivo chamado **Dockerfile** dentro da pasta informada no **context**. No nosso caso, especificamos que deverá ser usado o **Dockerfile-alternate** (arquivo alternativo) para construir a nova imagem.

```
version: "3.7"
services:
  fiap-web:
    build: .
      context: ./dir
      dockerfile: Dockerfile-alternate
```

Código-fonte 7.7 – Exemplo sexto trecho
Fonte: Elaborado pela autora (2019)

Na linha oito, temos o **args**, abrindo um novo bloco de código para definir os argumentos que serão usados pelo **Dockerfile**. Os argumentos de construção são variáveis de ambiente, acessíveis apenas durante o processo de construção, e devem ser especificadas primeiramente no seu Dockerfile e depois no seu arquivo Docker-compose.

O próximo nível de indentação (*), dentro do bloco **args**, contém a chave **"versao"** e o valor **"1"** como argumento que será passado para o **Dockerfile**.

```
version: "3.7"
services:
  fiap-web:
    build: .
      context: ./dir
      dockerfile: Dockerfile-alternate
      args:
        versao: 1
```

Código-fonte 7.8 – Exemplo sétimo trecho
Fonte: Elaborado pela autora (2019)

A próxima instrução, na linha nove, mostra a segunda definição do **serviço fiap-web**, no mesmo nível de indentação do build, que é **ports**. Esta definição indica qual porta do container será exposta no **Docker host**. No nosso caso, usaremos a porta 8080 do container, com a 80 do Docker host.

```
version: "3.7"
services:
  fiap-web:
    build: .
      context: ./dir
      dockerfile: Dockerfile-alternate
      args:
        versao: 1
    ports:
```

```
      - " 8080:80"
```

Código-fonte 7.9 – Exemplo oitavo trecho
Fonte: Elaborado pela autora (2019)

Finalizando nosso exemplo, o bloco de código do serviço **fiap-web** já está completo, e iniciaremos, então, um outro bloco de código de serviço novo, no nível de indentação do primeiro serviço (*), que nomeamos de **fiap-image1**.

O próximo nível de indentação (*), na linha doze, mostra a primeira definição do serviço **fiap-image1** que, nesse caso, é o **image**. Essa definição é responsável por informar qual imagem será usada para iniciar esse container. Essa imagem será obtida do repositório configurado no **Docker host** que, por padrão, é o hub.docker.com.

```
version: "3.7"
services:
  fiap-web:
    build: .
      context: ./dir
      dockerfile: Dockerfile-alternate
      args:
        versao: 1
    ports:
      - "8080:80"
  fiap-image1:
    image: fiap-image1
```

Código-fonte 7.10 – Exemplo trecho final
Fonte: Elaborado pela autora (2019)

* Notação: usamos dois espaços para indentação como separador de bloco.

## 7.10 *Serverless Computing*: o que é?

***Serverless Computing*** ou ***Serverless Framework*** é uma extensão natural do **conceito de nuvem**, no qual é possível criar softwares e executar aplicativos sem a necessidade de investir recursos para a administração de máquinas virtuais. O *Serverless Framework* é um modelo de serviço de *Cloud Computing*, mas não é um serviço na nuvem. Trata-se de uma **ferramenta poderosa** de linha de comando em Node.js que simplifica o **uso e o gerenciamento dos recursos que estão na nuvem**.

A computação sem servidor incorpora serviços **BaaS** (*Back-end as a Service*) de terceiros, também pode adotar uma arquitetura **FaaS** (*Functions as a Service*) para executar código personalizado em containers gerenciados, fazendo uso de software de provedores **SaaS** (*Service as a Service*), com o GitHub, Twilio, Auth0, Stripe, entre outros.

Um fato importante é que a *Cloud Computing* e a virtualização de servidores trouxeram mais flexibilidade para o mundo de TI (Tecnologia da Informação) e facilitaram a criação da arquitetura de containers, da arquitetura *Serverless* e também o desenvolvimento de novas tecnologias para a abstração de recursos. Os benefícios alcançados ao adotar essa arquitetura são:

- Redução de recursos computacionais.
- Redução de custos com gestão e operação de infraestrutura.
- Escalabilidade.
- Alta disponibilidade.
- Tolerância a falhas.

E como implementar *Serverless* na sua empresa?

A realidade é que devemos contar com **fornecedores especializados** que proveem este tipo de serviço para gerenciamento de diversos ambientes e operação de infraestrutura, além da capacidade de lidar com o processo de implantação. O grande benefício da parceria com fornecedores externos é o **ganho de tempo** para se concentrar no seu aplicativo e em processos críticos, sem a necessidade de se preocupar com a configuração de servidores ou com provisionamento.

Abaixo, temos um descritivo sobre as vantagens e desvantagens de se trabalhar com *Serverless Framework*:

| Vantagens | Desvantagens |
|---|---|
| É extremamente barato (US$ 0,20/milhão de requisições na AWS): a utilização de recursos é cobrada por uso (pay-per-use) e é rentável principalmente quando o tráfego é irregular ou inesperado. | Dificuldades de debugging e teste. |
| Não é preciso configurar o servidor (exemplo Node, NGINX, Apache etc.). | As functions devem executar no tempo limite (timeout) de 300 segundos, para garantir que sejam rápidas e escaláveis. |

| | |
|---|---|
| | Se extrapolarem esse limite, o custo aumenta e pode até acarretar em outras consequências, como lentidão ou queda do serviço que está em execução. |
| O deploy é simples, basta encapsular todo o código em um ZIP e subir para o servidor. | Dependência de fornecedor: em conjunto com as functions, normalmente, utilizamos outros serviços, por exemplo, armazenamento de arquivos (como o Amazon S3), proxy de API (como o AWS API Gateway), entre outros, o que leva a ter uma certa amarração em caso de migração de fornecedor. |
| A escalabilidade ocorre de forma automática: os servidores são autoescaláveis e tudo isso é gerenciado pela empresa provedora. | Perda de performance momentânea devido ao maior tempo de resposta, no caso de ocorrerem execuções espaçadas de cold-starts (funções inativas). |

Quadro 7.2 – Vantagens e desvantagens da arquitetura *Serverless*
Fonte: Elaborado pela autora (2019)

Como **boas práticas** de implementação de serviços que serão hospedados no *Framework Serverless*, destacamos a importância de desenvolver um **serviço extremamente simples** e com **objetivos bem definidos** devido às dificuldades para *debbugar* ou testar. No geral, a arquitetura *Serverless* é utilizada para processar *functions* que não necessitam de execução imediata (**funções assíncronas**), sendo importante mencionar que esse tipo de função possui um limite de tempo de execução (*timeout*) e deve ser ***stateless*** (não armazenar estado entre cada execução).

E qual é o melhor fornecedor de *Framework Serverless*? Você vai descobrir em seguida.

Vimos um exemplo prático sobre isso em nosso sexto capítulo sobre Google Cloud Computing, o App Engine, você se lembra? Para funções, procure por Google Cloud Functions também!

### 7.10.1 Como escolher um fornecedor *Serverless* para sua empresa?

Atualmente, você pode contar com três principais fornecedores de soluções *Serverless*: Amazon AWS, Microsoft Azure e Google Cloud. Porém, além da solução *Serverless*, você vai necessitar de diversos outros serviços que também abordam a capacidade de executar código sem um servidor e que formam um conjunto de funções que são essenciais para melhor funcionamento e execução do seu aplicativo,

como o serviço de mensageria, a capacidade ou a forma de armazenamento dos dados, entre outros. Veja um comparativo por fornecedores e serviços:

| | Amazon AWS | Azure | Google Cloud |
|---|---|---|---|
| **Computação** | AWS Lambda | Azure Functions | Cloud Functions |
| **Armazenamento** | Amazon S3 | Azure Storage | Cloud Storage |
| **Mensageria** | Amazon SQS<br>Amazon SNS | Service Bus | Cloud Pub/Sup |
| **Proxy de API** | AWS API Gateway | API Management | Cloud Endpoints |

Figura 7.17 – Fornecedores x Serviços
Fonte: adaptado por FIAP (2019)

Cada empresa disponibiliza um serviço com características únicas. Desta forma, o ideal é analisar o que cada fornecedor pode oferecer e se a forma de execução do serviço ofertado é adequada e essencial para o devido funcionamento do seu aplicativo. O pioneiro do movimento *Serverless* foi o AWS Lambda, lançado em 2014. Posteriormente, vieram o Azure Functions e o Cloud Functions, respectivamente. Vamos trabalhar com o AWS Lambda no nosso conteúdo.

O Lambda é um serviço da Amazon que permite executar códigos sem provisionar ou gerenciar servidores. É um modelo de programação extremamente leve e orientado a eventos, o que facilita a reutilização de ideias e estilo de codificação. As vantagens adquiridas com o uso do serviço da Amazon são:

- É possível executar o código do seu microsserviço no AWS para praticamente qualquer tipo de aplicativo ou serviço de *back-end,* sem nenhuma administração.
- Oferece suporte para diversas linguagens de programação e pode ser acionado de outros serviços da AWS ou ser chamado diretamente de qualquer site Web ou aplicativo de celular.
- É altamente integrado ao API Gateway. A possibilidade de fazer chamadas síncronas do API Gateway para o AWS Lambda permite a criação de aplicativos totalmente sem servidor.
- É responsável por executar e fazer o ajuste de escala de seu código com alta disponibilidade.

## 7.11 CDN

Rede de Distribuição de Conteúdo (*CDN – Content Delivery Network*) refere-se a um grupo de servidores distribuídos geograficamente que trabalham juntos para fornecer uma entrega rápida de conteúdo da Internet. Hoje, a maioria do tráfego da web é atendida por meio de CDNs, incluindo o tráfego de sites importantes, como Facebook, Netflix e Amazon.

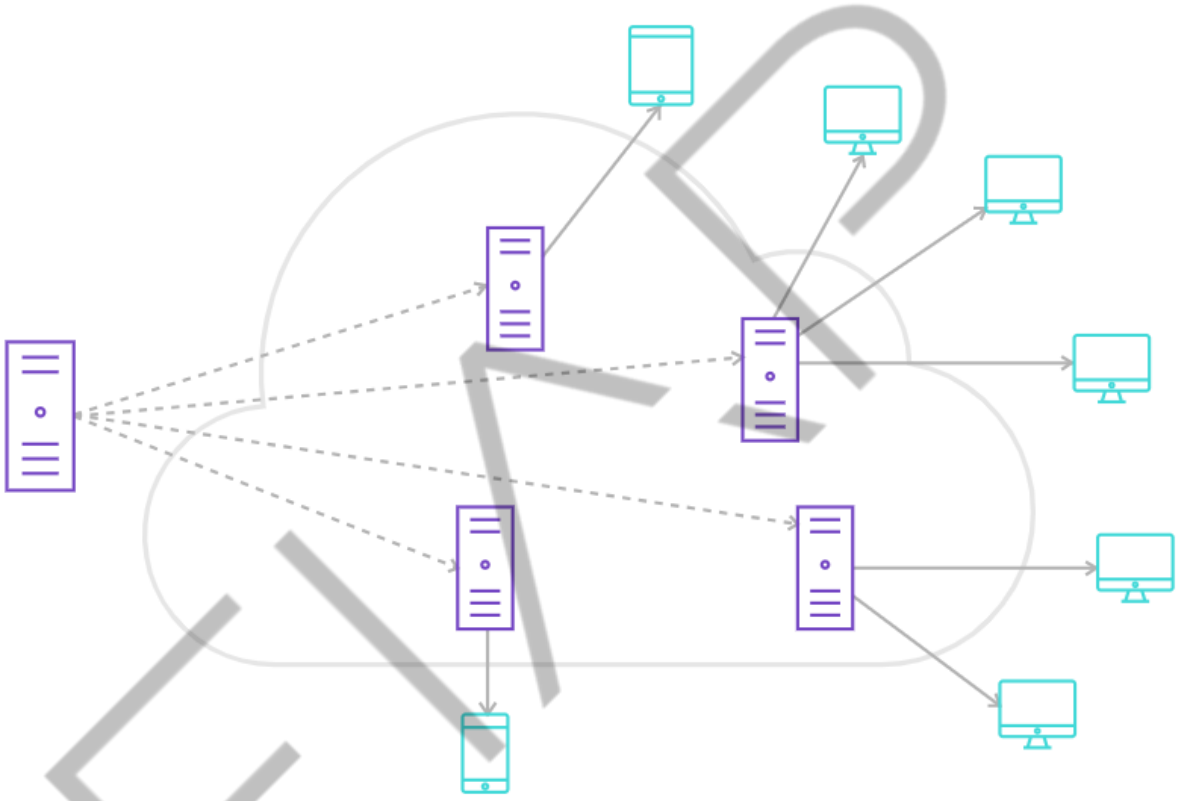

Figura 7.18 – Arquitetura de uma CDN
Fonte: adaptado por FIAP (2019)

Como benefícios com o uso dos CDNs, temos:

- Armazenamento em cache para reduzir custos de largura de banda e para evitar interrupções no serviço.
- Aumento da segurança do site e *e-commerces,* fornecendo melhorias nos certificados de segurança, mitigação de "Negação de Serviço Distribuída" (*DDoS - Distributed Denial of Service*) e outras otimizações.
- Melhoria no tempo de carregamento do site ao distribuir o conteúdo mais próximo de seus visitantes, usando um servidor CDN localizado em um

determinado Ponto de Presença (*POP - Point Of Presence*) que esteja geograficamente próximo do usuário.

- Aumento da disponibilidade de conteúdo e redundância: devido à natureza distribuída, os CDNs podem lidar com mais tráfego e resistir a falhas de hardware.

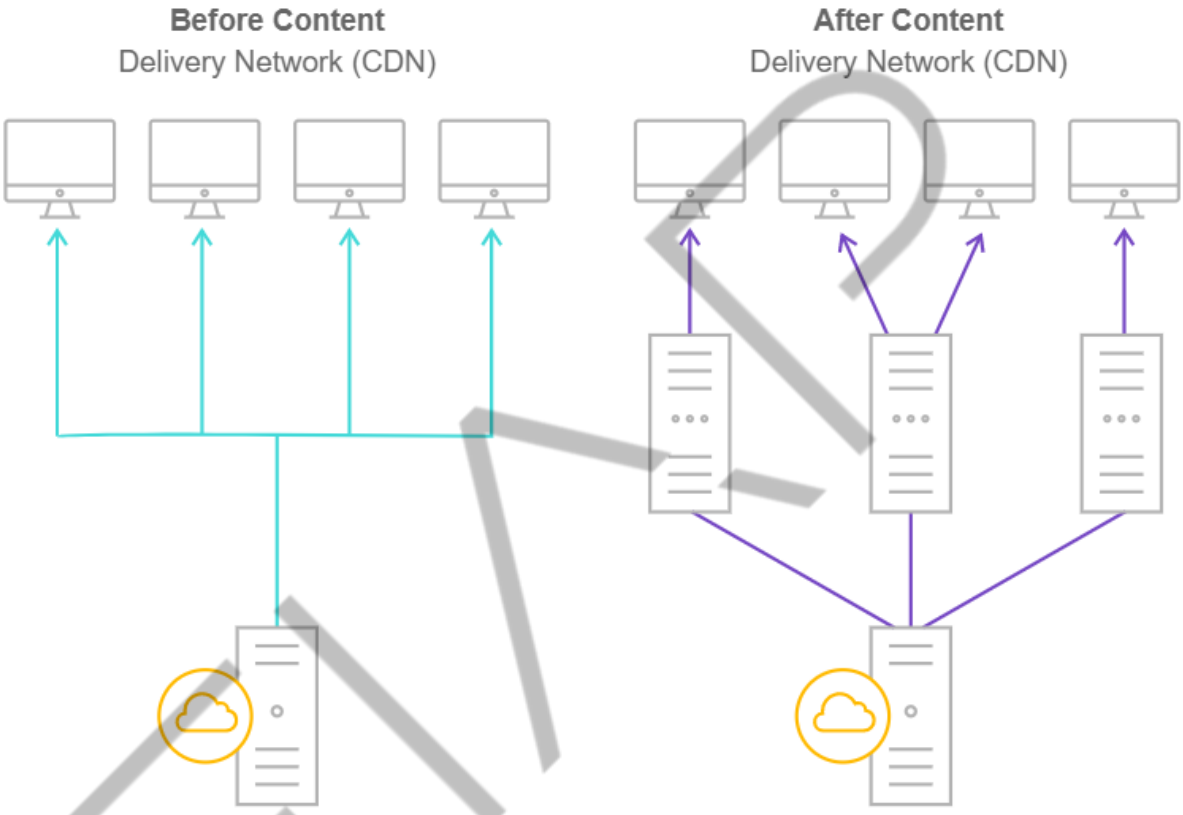


Figura 7.19 – Antes e depois das CDNs
Fonte: adaptado por FIAP (2019)

Para atender à alta demanda no consumo de conteúdo web pelo consumidor final, as empresas estão usando cada vez mais as CDNs para acelerar conteúdo estático, conteúdo dinâmico, conteúdo móvel, transações de comércio eletrônico, vídeo, voz, jogos etc.

# REFERÊNCIAS

ALLES, G. R.; CARISSIMI, A.; SCHNORR, L. M. Assessing the computation and communication overhead of Linux Containers for HPC applications. **Anais do Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD),** 19 jan. 2018. Disponível em: <https://www.researchgate.net/publication/334168150_Assessing_the_Computation_and_Communication_Overhead_of_Linux_Containers_for_HPC_Applications/link/5d3617d24585153e59196410/download>. Acesso em: 11 set. 2019.

CLUSTERHQ. **Container market adoption.** 2016. Disponível em: <https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2015.pdf>. Acesso em: 11 set. 2019.

CVE. **Common vulnerabilities and exposures website.** [s.d.]. Disponível em: <https://cve.mitre.org/>. Acesso em: 11 set. 2019.

DOCKER. **Docker website**. [s.d.]. Disponível em: <https://www.docker.com/>. Acesso em: 11 set. 2019.

DOCKER. **Modern app architecture for the enterprise.** 2016. Disponível em: <https://www.docker.com/sites/default/files/WP_ModernAppArchitecture_04.12.2016_1.pdf>. Acesso em: 11 set. 2019.

FELTER, W. *et al*. An updated performance comparison of virtual machines and Linux Containers**.** In: Performance Analysis of Systems and Software (ISPASS), 2015. **International Symposium On**. IEEE, 2015, p. 171-172.

HAYDEN, M. **Securing Linux Containers.** 2015. Disponível em: <https://www.sans.org/reading-room/whitepapers/linux/securing-linux-containers-36142>. Acesso em: 8 fev. 2021.

NEWMAN, S. **Building Microservices**. O'Reilly Media, 2015.

NIST – National Vulnerability Database. [s.d.]. Disponível em: <https://nvd.nist.gov/>. Acesso em: 8 fev. 2021.

## LISTA DE ABREVIATURAS

| | |
|---|---|
| API | Application Programming Interface |
| BPM | Business Process Management |
| EAI | Enterprise Application Integration |
| HTTP | Hiper Text Transfer Protocol |
| IDE | Integrated Development Environment |
| Java EE | Java Enterprise Edition |
| RPC | Remote Procedure Call |
| SOA | Service Oriented Architecture |
| SOAP | Simple Object Access Protocol |
| TI | Tecnologia da Informação |
| WSDL | Web Service Description Language |
| XML | Extensible Markup Language |
| URL | Uniform Resource Locators |
| URI | Uniform Resource Identifiers |
| DDD | Domain-Driven Design |
| P&D | Pesquisa e Desenvolvimento |
| VM | Virtual Machine |