Docker Concepts & Formations

Construção e publicação de imagens

`docker image build -t "docker_hub_username"/gsd:ctr2023 .`

Cria uma imagem a partir do `Dockerfile` presente no diretório atual.

- `t` atribui nome e tag à imagem.
- `"docker_hub_username"` é o username no Docker Hub.
- `gsd` é o nome do repositório.
- `ctr2023` é a tag (versão da imagem).
- O `.` (ponto) indica que o contexto de build está na pasta atual.

`docker image push "docker_hub_username"/gsd:ctr2023`

Envia a imagem para o Docker Hub. O nome e tag devem corresponder à imagem criada localmente.

Build multi-plataforma com Buildx

`docker buildx build --platform linux/arm64/v8,linux/amd64 --push --tag gcunha9/gsd:ctr2023 .`

- Usa **Buildx**, que permite compilar imagens para múltiplas arquiteturas.
- `-platform` cria imagens para ARM64 (ex.: Apple Silicon, Raspberry Pi) e AMD64 (x86_64).
- `-push` envia a imagem diretamente para o Docker Hub.
- `-tag` define o nome e a tag da imagem.
- `.` indica o diretório atual como contexto de build.

Gestão de imagens

`docker image rm gcunha9/gsd:ctr2023`

Remove a imagem localmente (não remove do Docker Hub).

Execução de containers

`docker container run -d --name web -p 8000:8080 gcunha9/gsd:ctr2023`

Cria e inicia um container a partir da imagem.

- `d` corre em modo detached (background).
- `-name web` atribui o nome “web” ao container.
- `p 8000:8080` faz o mapeamento da porta 8000 do host para a porta 8080 do container.
- Se a imagem não existir localmente, o Docker faz pull do Docker Hub.

Gestão de containers

`docker container ls`

Lista todos os containers em execução. Com `-a` lista também os containers parados.

`docker container stop "name_of_container"`

Pára o container especificado (pelo nome ou ID).

Container interativo

`docker container run -it --name test alpine sh`

Cria um container com:

- `it` (modo interativo + TTY).
- `-name test` (nome “test”).
- Imagem `alpine` (leve).
- Executa o shell `sh` dentro do container.

`docker container ls`

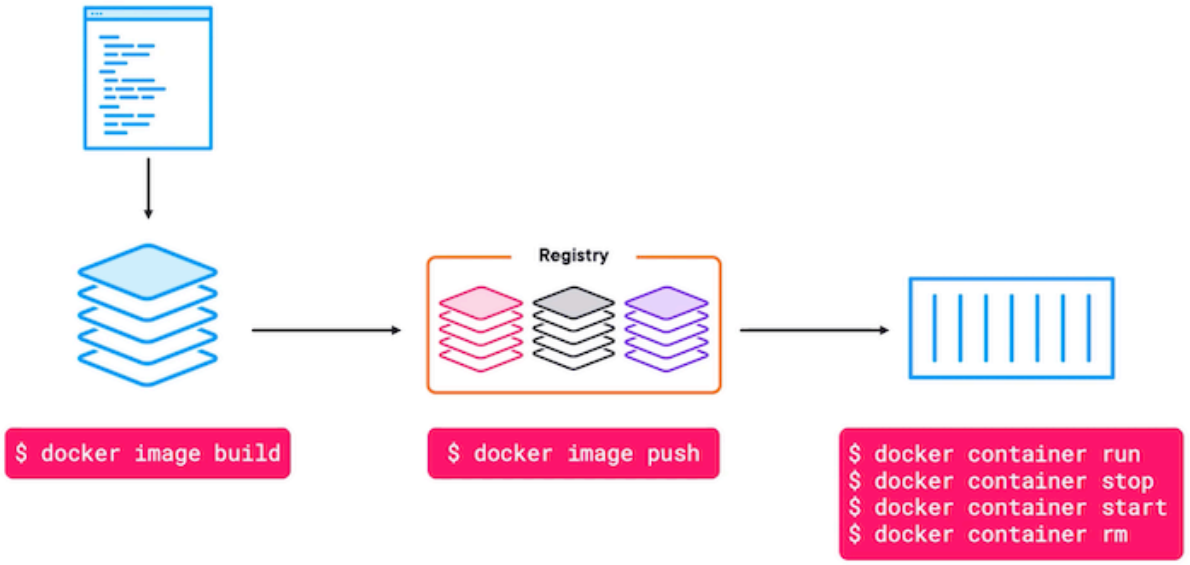
Verifica se o container está em execução.

Sair sem parar o container

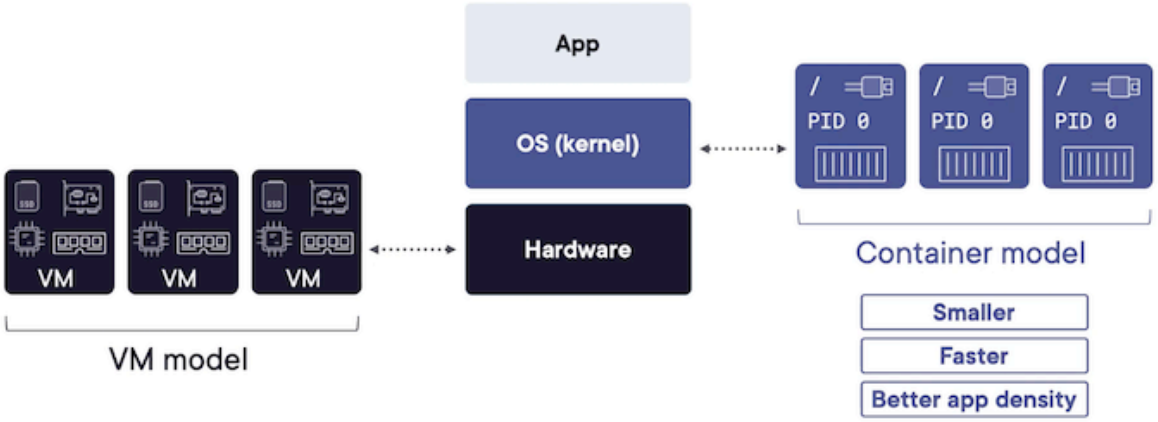
`CTRL+P+Q` → Sai do terminal do container sem o parar (continua em execução).

`exit` → Sai do container e termina a sua execução (o container para).

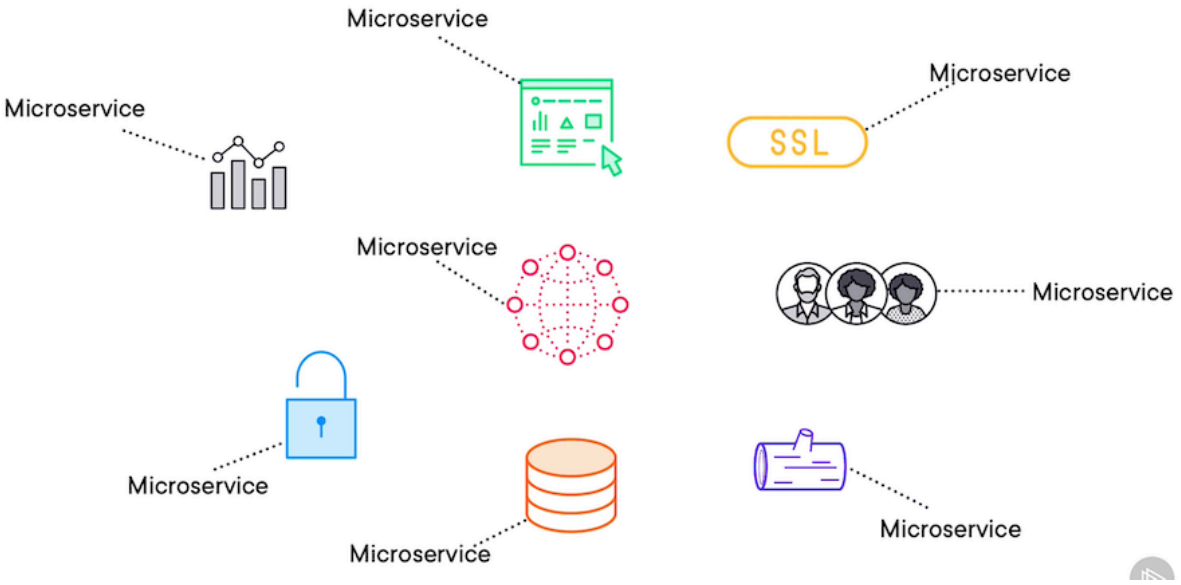
#1


Registry
$ docker image build
$ docker image push
$ docker container run
$ docker container stop
$ docker container start
$ docker container rm

#2


App
OS (kernel)
Hardware
VM
VM
VM
VM model
/
PID 0
/
PID 0
/
PID 0
Container model
Smaller
Faster
Better app density

#3



Microservices

Cada microserviço corre no seu próprio container.

- Exemplo: oito pequenas aplicações (features) → oito microserviços → oito containers.
- Permite escalar ou alterar apenas um microserviço sem impactar os restantes.

Cloud Native

Aplicações **cloud native** aproveitam capacidades como:

- *Self-healing* (recuperação automática em caso de falhas).
- *Auto-scaling* (escala automática consoante a carga).
- *Rolling updates* (atualizações contínuas sem downtime).
- Entre outras funcionalidades típicas de ambientes distribuídos e elásticos.

Aplicações descritas em ficheiros de configuração

Em vez de uma série de comandos, descrevemos a aplicação num ficheiro de configuração e damos esse ficheiro ao Docker para que faça o deploy e a

gestão.

- Facilita a reprodutibilidade.
- Permite partilhar facilmente a configuração com a equipa.

Compose.yaml

O ficheiro `compose.yaml` (ou `docker-compose.yml`) define:

- Os microserviços da aplicação.
- Redes (networks) de comunicação entre serviços.
- Serviços auxiliares, como `redis`.
- Volumes e outras configurações persistentes.

Docker Compose: subir e descer ambientes

`docker compose up --detach`

- Inicia todos os serviços definidos no `compose.yaml`.
- `-detach` corre em background.
- Ao executar `docker container ls`, vemos os containers definidos (ex.: `app` e `redis`).

`docker compose down --volumes`

- Encerra e remove todos os containers criados pelo `docker compose`.
- `-volumes` remove também os volumes (dados persistentes).

Após isso, se fizermos `docker container ls` e `docker volume ls`, verificamos que não há containers nem volumes em execução: tudo foi removido.

Cluster

Um **cluster** é composto por uma ou mais máquinas (chamadas **nós** ou *nodes*).

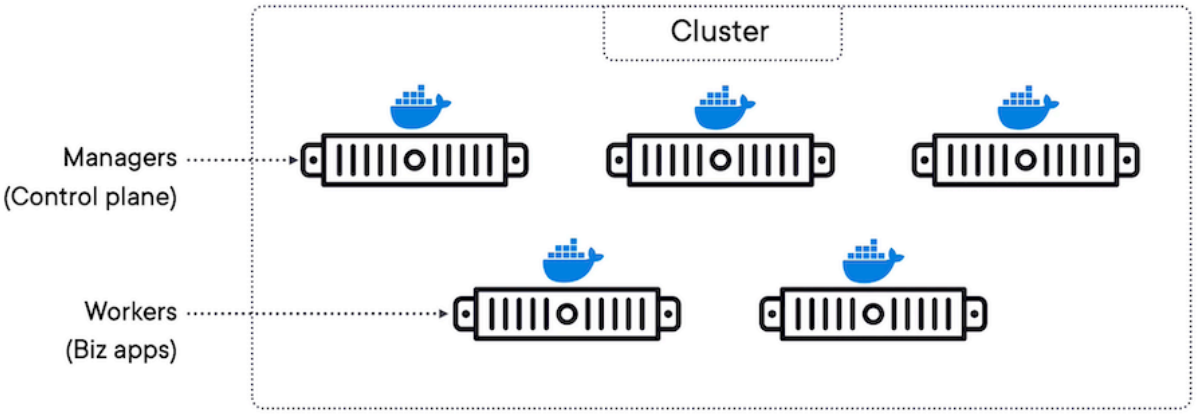
- Os nodes podem ser servidores físicos, VMs, ou instâncias em cloud.
- Juntos, funcionam como um sistema distribuído para correr containers.

Nodes com Multipass

No exemplo, os nodes são **VMs leves criadas com Multipass**, uma ferramenta da Canonical que permite criar e gerir VMs Linux de forma simples e rápida

(ideal para testes ou demos).

#4



Building a Swarm

Um **Swarm** é um cluster de Docker nodes que trabalham em conjunto.

- Estrutura típica: **3 managers + 2 workers** (boa prática → sempre ter número ímpar de managers para garantir *quorum* e evitar falhas de consenso).
- Managers coordenam o cluster e tomam decisões.
- Workers apenas executam containers conforme instruções do manager.

Multipass: gestão de VMs

`multipass version`

Mostra a versão instalada do Multipass.

`multipass find`

Lista imagens disponíveis para criar VMs (ex.: Ubuntu, Docker-ready, etc).

`multipass launch docker --name node1`

Cria e inicia uma VM chamada `node1`, já preparada com Docker.

`multipass list`

Lista todas as VMs criadas e o seu estado.

`multipass delete node4`

Remove a VM `node4` (mas mantém ficheiros temporários).

`multipass purge`

Liberta os recursos de todas as VMs removidas (limpa os ficheiros).

`multipass info node1`

Mostra detalhes da VM `node1` (IP, memória, CPU, disco, etc).

`multipass shell node1`

Abre um shell dentro da VM `node1` (entramos diretamente no sistema).

Comandos Swarm

`docker swarm leave --force`

- Força o node atual a sair do cluster Swarm.
- Normalmente usado em VMs de teste/demos quando queremos reconfigurar.

`docker swarm init --advertise-addr <ip_address>`

- Inicializa um novo cluster Swarm.
- O parâmetro `-advertise-addr` define o IP pelo qual os outros nodes irão contactar este manager.

`docker swarm join-token worker`

- Mostra o comando necessário para que outro node entre no cluster como **worker**.
- Inclui o token de autenticação e o IP/porta do manager.

`docker node ls` (executado no manager)

- Lista todos os nodes que pertencem ao cluster (managers e workers).
- Mostra estado, funções e disponibilidade de cada node.

Gestão de nodes no Swarm

`docker node update --availability drain node1`

- Marca o node `node1` como indisponível para receber novos containers (*drain mode*).

- O Swarm migra automaticamente os containers que estavam nesse node para outros disponíveis.

Serviços no Swarm

`docker service create --name web -p 8080:8080 --replicas 3 gcunha9/gsd:web2023`

- Cria um novo serviço chamado `web`.
- Publica a porta 8080 do host → 8080 do container.
- Inicia 3 réplicas (containers) a partir da imagem `gcunha9/gsd:web2023`.

`docker service rm web`

- Remove o serviço `web` (mata todos os containers associados).

`docker service ls`

- Lista todos os serviços ativos no cluster Swarm.

`docker container ls` (dentro de um node)

- Lista os containers que **aquele node** está a correr.
- Importante: nem todos os nodes correm todos os containers → depende da distribuição feita pelo Swarm.

`docker service ps <service_name>`

- Lista as tasks (containers) de um serviço, mostrando em que node cada réplica está a correr e o estado de cada uma.

Escalar serviços

`docker service scale web=10` (executado no manager)

- Ajusta o número de réplicas do serviço `web` para 10.
- O Swarm distribui automaticamente estas réplicas pelos nodes disponíveis.

`docker service ps <service_name>`

- Mostra a nova distribuição das réplicas pelos nodes após o scaling.

Remover containers manualmente

`docker container rm <container_id> -f`

- Força a remoção de um container específico.

- No entanto, se esse container pertencer a um serviço do Swarm, o **orquestrador vai recriá-lo** para garantir o estado desejado (replicas sempre ativas).

Stacks no Swarm

`docker stack deploy -c compose.yml counter`

- Faz deploy de uma stack chamada `counter`, baseada no ficheiro `compose.yml`.
- Uma stack é um conjunto de serviços (definidos via Docker Compose) geridos pelo Swarm.

`docker stack ls`

- Lista as stacks ativas no cluster.

`docker stack services counter`

- Mostra os serviços pertencentes à stack `counter`.

`docker stack ps counter`

- Mostra todas as tasks (containers) da stack `counter`, com estado e em que node estão a correr.

Edição de ficheiros com vi (dentro de uma VM/node)

- `vi compose.yml` → Abre o ficheiro `compose.yml` no editor vi.
- `cw` → Substituir valores (change word).
- `:w` → Guardar alterações.
- `:wq` → Guardar e sair.
- `:q!` → Sair sem guardar.

Depois de editar:

`docker stack deploy -c compose.yml counter`

- Reaplica o deploy da stack com as alterações feitas no ficheiro.

Docker Compose: definição de redes, volumes e serviços

No `compose.yml` podemos definir **redes**, **volumes** e os serviços que compõem a aplicação.

- **Networks**: se não existir, o Compose cria automaticamente a rede definida (ex.: `counter-net`).

- **Volumes**: também são criados se não existirem (ex.: `counter-vol`), usados para persistência de dados.

Serviços definidos no Compose

Container web-fe

- Criado a partir da imagem definida no `app.py` (normalmente referida no `Dockerfile`).
- O Compose faz o **build** da imagem com base nas instruções do ficheiro:
  - `build: .` → o contexto de build é o diretório atual, onde está o `Dockerfile`.
- Configurações típicas:
  - **target: listening** → refere-se ao estado de saúde/endpoint que o serviço expõe.
  - **published** → mapeia uma porta do host para a porta interna do container (*port mapping*).

Container redis

- Serviço adicional definido no `compose.yml`.
- Baseado na imagem oficial `redis`, usado como cache ou base de dados em memória.