Containers

Introdução Engenharia Informática

Mário Antunes

October 13, 2025

Universidade de Aveiro

Uma Forma Moderna de Empacotar e Executar Aplicações

Terminologia 📖

Antes de começarmos, vamos definir alguns termos-chave.

- **Imagem (Image):** Um modelo inerte, apenas de leitura, que contém uma aplicação e as suas dependências. Pense nisto como uma **planta** ou uma classe em programação orientada a objetos.
- **Contentor / Instância (Container / Instance):** Uma **instância** executável de uma imagem. Esta é a aplicação real, a correr (como um objeto criado a partir de uma classe). Os termos são frequentemente usados de forma intercambiável.
- **Registo (Registry):** Um sistema de armazenamento para imagens de contentores. O **Docker Hub** é um registo público popular.
- **Motor Docker (Docker Engine):** A aplicação

O Problema: “Na Minha Máquina Funciona!” 🤔

Todos os programadores já enfrentaram este problema clássico:

- A sua aplicação funciona perfeitamente no seu portátil (que tem Python 3.9, uma versão específica de uma biblioteca, e corre Debian).
- Quando a entrega a um colega (que tem Python 3.8 e corre macOS) ou a implementa num servidor (a correr um SO mais antigo), ela falha.

Estas diferenças nos ambientes criam um enorme desafio para a portabilidade do software.

A Solução: Contentores 📦

Um **contentor** é uma unidade de software padrão e executável que empacota o código de uma aplicação juntamente com todas as suas dependências de tempo de execução. Este pacote é **isolado**, garantindo que a aplicação corre de forma uniforme e consistente em qualquer lugar.

Analogia: Um contentor é como um contentor de transporte padronizado. Não importa o que está lá dentro; pode ser manuseado por qualquer navio compatível (máquina anfitriã).

Como o Isolamento é Alcançado: Namespaces

Os contentores correm à velocidade **máxima do hardware** porque são apenas processos isolados no kernel do anfitrião. O isolamento é fornecido pelos **Namespaces do Linux**.

Os Namespaces virtualizam os recursos do sistema para um processo, fazendo parecer que este tem a sua própria cópia privada. Os namespaces-chave incluem:

- **PID:** Isola os IDs dos processos. Dentro do contentor, a sua aplicação é o PID 1.
- **NET:** Fornece uma pilha de rede isolada (endereços IP, tabelas de encaminhamento).
- **MNT:** Isola os pontos de montagem do sistema de ficheiros.

Analogia: Os Namespaces são como as paredes, caixas de correio privadas e chaves de porta únicas para cada

Como os Recursos são Geridos: Cgroups

Para evitar que um contentor consuma todos os recursos do sistema, o kernel do Linux usa **Control Groups (cgroups)**.

Os Cgroups permitem que o anfitrião limite e monitorize os recursos que um contentor pode usar, tais como:

- Uso de CPU (p. ex., limitar a 1 núcleo de CPU).
- Memória (p. ex., limitar a 512 MB de RAM).
- Largura de banda de I/O de disco.

Analogia: Os Cgroups são como os contadores de serviços públicos e os disjuntores de cada apartamento, garantindo que nenhum inquilino pode usar toda a água ou eletricidade do prédio.

VMs vs. Contentores #1

- **Máquinas Virtuais (VMs)** virtualizam o **hardware**. Cada VM inclui uma cópia completa de um SO convidado e do seu kernel. São pesadas e demoram minutos a arrancar.
- **Contentores** virtualizam o **sistema operativo**. Partilham o kernel do sistema anfitrião e são leves, arrancando em segundos.

VMs vs. Contentores #2

Característica	Máquinas Virtuais (VMs)	Contentores
Analogia	🏡 **Casas:** Totalmente autónomas.	🏢 **Apartamentos:** Partilham a infraestrutura do prédio.
Nível de Abstração	**Virtualização de Hardware**	**Virtualização de SO**
Tamanho	**Gigabytes (GB)**	**Megabytes (MB)**
Tempo de Arranque	**Minutos**	**Segundos ou menos**
Sobrecarga	Baixa a Média	Muito Baixa (Quase nativa)
Uso de Recursos	Mais elevado (SO completo por VM)	Mais baixo (Kernel do SO partilhado)
Isolamento	**Forte** (Nível de hardware)	**Bom** (Nível de processo)
Portabilidade	Portátil (mas grande)	**Extremamente Portátil**

A Imagem do Contentor e as Suas Camadas 📜

Uma **imagem** é um modelo apenas de leitura construído a partir de uma série de **camadas** empilhadas. Cada instrução num `Dockerfile` cria uma nova camada.

Isto torna as construções (builds) rápidas e o uso de disco eficiente, já que múltiplas imagens podem partilhar camadas base comuns.

Dados Persistentes: Volumes 💾

Por defeito, o sistema de ficheiros de um contentor é **efémero** (apagado quando o contentor para).

Para guardar dados permanentemente, usam-se **volumes**, que mapeiam um *diretório* dentro do contentor para um *diretório* na máquina anfitriã.

Rede de Contentores e DNS

O motor de contentores cria uma **rede virtual em modo ponte (bridge)**. Os contentores na mesma rede recebem um IP privado e podem comunicar entre si.

- **Mapeamento de Portas:** Para expor o serviço de um contentor ao mundo exterior, mapeia-se uma porta do anfitrião para uma porta do contentor (p. ex., `-p 8080:80`).
- **DNS Interno:** Ao usar o Docker Compose, cada serviço pode alcançar outro usando o nome do serviço como hostname. O código da sua `webapp` pode simplesmente conectar-se a `http://database` para chegar ao contentor da base de dados.

Apresentando o Docker

O Docker é a plataforma que popularizou os contentores. Fornece um conjunto simples de ferramentas para construir, distribuir e executar qualquer aplicação, em qualquer lugar.

- **Docker Engine:** O serviço de fundo (daemon) que gere os contentores.
- **Docker CLI:** A ferramenta de linha de comandos que usa para interagir com o Docker Engine.
- **Docker Hub:** Um registo público de imagens de contentores pré-construídas.

Comandos Docker Comuns

Comando	Descrição
`docker run [imagem]`	Cria e inicia um novo contentor a partir de uma imagem.
`docker ps`	Lista todos os contentores em execução. `ps -a` lista todos (em execução ou parados).
`docker stop [id/nome]`	Para um contentor em execução de forma controlada.
`docker rm [id/nome]`	Remove um contentor parado.
`docker logs [id/nome]`	Obtém os logs (saída padrão) de um contentor.
`docker pull [imagem]`	Descarrega uma imagem de um registo (como o Docker Hub).
`docker images`	Lista todas as imagens armazenadas localmente.
`docker build -t [nome] .`	Constrói uma nova imagem a partir de um `Dockerfile` no diretório atual.

O `Dockerfile`: Uma Análise Detalhada

Um `Dockerfile` é uma receita para construir uma imagem de contentor. Aqui estão as instruções mais comuns:

- `FROM`: Especifica a imagem base sobre a qual construir (p. ex., `ubuntu:22.04`).
- `WORKDIR`: Define o diretório de trabalho para os comandos seguintes.
- `COPY`: Copia ficheiros ou diretórios do anfitrião para a imagem.

- `RUN`: Executa um comando durante o processo de construção da imagem (p. ex., `RUN apt-get install -y nginx`).
- `CMD`: Fornece o comando padrão a ser executado quando um contentor é iniciado a partir da imagem.
- `ENTRYPOINT`: Configura o contentor para ser executado como um executável.
- `EXPOSE`: Informa o Docker que o contentor escuta nas portas de rede especificadas em tempo de execução.
- `ENV`: Define variáveis de ambiente persistentes.

Exemplo de `Dockerfile`: Um Serviço de Logs

Este `Dockerfile` simples cria um serviço cujo único trabalho é imprimir um carimbo de data/hora a cada 5 segundos. Isto é perfeito para testar o comando `docker logs`.

```
# Usar uma imagem base mínima
FROM alpine:latest

# O comando a executar quando o contentor arranca.
# É um ciclo infinito que imprime a data atual
# e espera 5 segundos.
CMD ["sh", "-c", "while true; do echo \"[LOG] Servidor a correr em $(date)\"; sleep 5; done"
```

Para construir e executar:

```
$ docker build -t logging-service .
$ docker run -d --name logger logging-service
$ docker logs -f logger
```

Docker Compose: Uma Análise Detalhada

Um ficheiro `compose.yml` define uma aplicação multi-serviço. Aqui estão as chaves mais comuns:

- `services`: A chave raiz onde todos os serviços da sua aplicação são definidos.
- `image`: Especifica uma imagem pré-construída de um registo (como o Docker Hub).
- `build`: Especifica o caminho para um `Dockerfile` para construir a imagem do serviço.

- `ports`: Mapeia portas do anfitrião para o contentor (p. ex., `"8080:80"`).
- `volumes`: Monta caminhos do anfitrião ou volumes nomeados no contentor.
- `environment`: Define variáveis de ambiente para o serviço.
- `depends_on`: Define dependências entre serviços, controlando a ordem de arranque.

Exemplo Compose 1: Construir uma Imagem NGINX Personalizada

Este exemplo mostra como empacotar os ficheiros do seu site diretamente numa imagem personalizada.

Estrutura de Ficheiros Necessária:

```
.
├── docker-compose.yml
├── Dockerfile
└── my-website/
    └── index.html
```

Dockerfile

```
# Usar a imagem oficial do NGINX como base
FROM nginx:alpine

# Copiar a nossa página web personalizada para o diretório raiz da web da imagem
COPY ./my-website /usr/share/nginx/html
```

docker-compose.yml

```
services:
  webserver:
    build: .
    ports:
      - "8080:80"
```

Exemplo 1: Explicação

Neste método, criamos uma **imagem autónoma e portátil** que inclui o código da nossa aplicação.

1. Quando executa `docker-compose up`, a diretiva `build: .` diz ao Compose para procurar um `Dockerfile` no diretório atual.
2. O `Dockerfile` começa a partir de uma imagem base padrão do `nginx`.
3. A instrução `COPY` pega na sua pasta local `./my-website` e copia o seu conteúdo diretamente para o sistema de ficheiros da imagem em `/usr/share/nginx/html`.
4. É criada uma nova imagem personalizada contendo tanto o NGINX como a sua página web.
5. Um contentor é iniciado a partir desta nova imagem.

Conceito-Chave: A aplicação e o seu código são empacotados juntos. Isto é ideal para **implementações de produção**, já que a imagem resultante é um artefacto consistente e imutável que pode ser executado em qualquer lugar.

Exemplo Compose 2: Usar um Volume para Servir Conteúdo

Este exemplo usa uma imagem NGINX padrão e injeta o conteúdo do site usando um volume.

Estrutura de Ficheiros Necessária:

```
.
├── docker-compose.yml
└── my-website/
    └── index.html
```

docker-compose.yml

```yaml
services:
  webserver:
    image: nginx:alpine
    ports:
      - "8080:80"
    volumes:
      - ./my-website:/usr/share/nginx/html
```

(Não é necessário Dockerfile para este método)

Exemplo 2: Explicação

Este método mantém o seu código na máquina anfitriã e liga-o dinamicamente ao contentor.

1. Quando executa `docker-compose up`, a diretiva `image: nginx:alpine` diz ao Compose para ir buscar a imagem padrão do NGINX ao Docker Hub. Nenhuma imagem personalizada é construída.
2. Um contentor é iniciado a partir desta imagem padrão.
3. A diretiva `volumes` cria uma ligação em tempo real entre a pasta `./my-website` no seu anfitrião e a pasta `/usr/share/nginx/html` dentro do contentor.
4. Quando o NGINX dentro do contentor procura ficheiros para servir, está na verdade a lê-los diretamente do disco da sua máquina anfitriã.

Conceito-Chave: O contentor não tem estado (stateless), e o código vive no anfitrião. Se alterar o seu ficheiro `index.html` no anfitrião, a alteração é refletida **instantaneamente** sem reconstruir ou reiniciar o contentor. Isto é ideal para **desenvolvimento local**.

Exemplo Compose 3: NGINX com uma Cache Varnish

Este exemplo avançado orquestra dois serviços: um servidor web NGINX e uma cache Varnish que se posiciona à sua frente para acelerar a entrega de conteúdo.

Estrutura de Ficheiros Necessária:

```
.
├── docker-compose.yml
└── varnish/
    └── default.vcl
```

`varnish/default.vcl` (Configuração do Varnish)

```
vcl 4.1;

// Definir o servidor de backend de onde o Varnish irá obter o conteúdo.
// 'nginx' é o nome do nosso outro serviço no docker-compose.yml.
backend default {
    .host = "nginx";
    .port = "80";
}
```

`docker-compose.yml`

```yaml
services:
  # A cache Varnish que é exposta ao mundo exterior
  cache:
    image: varnish:stable
    volumes:
      # Montar a nossa configuração personalizada do Varnish
      - ./varnish:/etc/varnish
    ports:
      # Mapear a porta 8080 do anfitrião para a porta 80 da cache
      - "8080:80"
    depends_on:
      - nginx

  # O servidor web NGINX, que NÃO é exposto ao anfitrião
  nginx:
    image: nginx:alpine
    # Nenhuma secção de portas significa que só é acessível a partir da rede Docker
```

Exemplo 3: Explicação

Esta configuração demonstra uma arquitetura multi-camada realista e de alto desempenho, onde os serviços comunicam internamente.

1. O `docker-compose.yml` define dois serviços: `cache` (Varnish) e `nginx`.
2. Apenas o serviço `cache` expõe uma porta (`8080`) à máquina anfitriã. O serviço `nginx` está completamente isolado do mundo exterior.
3. O ficheiro de configuração personalizado do Varnish (`default.vcl`) é montado no contentor `cache`. Este ficheiro diz ao Varnish que o seu "backend" (o servidor web real) está localizado no hostname `nginx`.
4. Graças ao **DNS interno** do Docker, o nome do serviço `nginx` resolve automaticamente para o endereço IP privado do contentor `nginx`, permitindo que o Varnish se conecte a ele.

O Fluxo do Pedido: `Browser do Utilizador` → `Máquina Anfitriã (Porta 8080)` → `Contentor Varnish (Cache)` → `Contentor NGINX (Servidor de Origem)`

A Magia do Caching: No primeiro pedido de uma página web, o Varnish vai buscá-la ao contentor `nginx` e armazena uma cópia na sua memória. Para todos os pedidos subsequentes da mesma página, o Varnish serve a cópia diretamente da sua cache, o que é incrivelmente rápido e evita que o servidor NGINX tenha de fazer qualquer trabalho.

Conceito-Chave: Isto demonstra uma poderosa **descoberta de serviços (service discovery)** e a criação de um **proxy reverso**, um padrão fundamental na arquitetura web.

A Origem: Linux Containers (LXC)

Antes do Docker, havia o **LXC**.

- O LXC é uma interface de espaço de utilizador para as funcionalidades de contenção do kernel Linux (namespaces e cgroups).
- Fornece um conjunto de ferramentas de mais baixo nível para criar e gerir contentores.
- Os contentores LXC são frequentemente descritos como sendo mais parecidos com máquinas virtuais muito leves e de arranque rápido do que com contentores de aplicação. Eles tipicamente executam um sistema `init` completo e são usados para isolar sistemas operativos inteiros.

O Padrão: Docker

O Docker pegou na tecnologia subjacente do LXC e construiu um ecossistema de alto nível e amigável ao utilizador à sua volta.

- Introduziu o conceito de imagens portáteis através do `Dockerfile`.
- Criou um registo centralizado (Docker Hub) para partilhar imagens.
- O seu foco são os contentores **centrados na aplicação**, empacotando uma única aplicação ou processo por contentor. Esta filosofia é uma pedra angular da arquitetura de microsserviços.

A Alternativa Moderna: Podman

O Podman é uma alternativa popular e moderna ao Docker, desenvolvida pela Red Hat.

- **Sem Daemon:** Ao contrário do Docker, o Podman não requer um daemon central sempre em execução, o que é frequentemente citado como um benefício de segurança.
- **Sem Root (Rootless):** O Podman foi projetado para executar contentores como um utilizador regular, sem necessitar de privilégios de root.
- **Compatível com CLI:** A interface de linha de comandos do Podman é intencionalmente idêntica à do Docker. Em muitos sistemas, pode simplesmente executar `alias docker=podman` e usar os mesmos comandos que já conhece.

Conclusão e Pontos-Chave

- Os contentores resolvem o problema do “na minha máquina funciona”, empacotando uma aplicação com as suas dependências numa unidade **portátil**.
- Eles alcançam isolamento e gestão de recursos através de funcionalidades do kernel Linux como **namespaces** e **cgroups**.
- O `Dockerfile` fornece uma receita para construir imagens, e o **Docker Compose** ajuda a gerir aplicações multi-serviço.
- Os contentores revolucionaram o desenvolvimento de software, formando a base das práticas modernas de **DevOps e cloud-native**.

Recursos Adicionais e Links Úteis 📚

Para continuar a sua jornada de aprendizagem com contentores, aqui estão alguns excelentes recursos:

- **Folha de Consulta Oficial do Docker:** Uma referência oficial e concisa para os comandos mais comuns.
  - https://docs.docker.com/get-started/docker_cheatsheet.pdf
- **Folha de Consulta Definitiva do Docker (Collabnix):** Uma folha de consulta mais abrangente com exemplos detalhados e explicações.
  - https://dockerlabs.collabnix.com/docker/cheatsheet/

- **Como Otimizar Imagens Docker (GeeksforGeeks):** Aprenda técnicas como builds multi-estágio para tornar as suas imagens mais pequenas, rápidas e seguras.
    - https://www.geeksforgeeks.org/devops/how-to-optimize-docker-image/
- **Otimizar Dockerfiles para Builds Rápidas (WarpBuild):** Entenda como estruturar o seu `Dockerfile` para tirar o máximo partido do cache de camadas e acelerar significativamente o seu processo de construção.
    - https://www.warpbuild.com/blog/optimizing-docker-builds
- **LinuxServer.io:** Um projeto comunitário que fornece e mantém imagens de contentores de alta qualidade e fáceis de usar para muitas aplicações auto-hospedadas populares (como servidores de multimédia, clientes de