Containers para Aplicações

Introdução Engenharia Informática

Mário Antunes

October 20, 2025

Universidade de Aveiro

Um Olhar Aprofundado sobre AppImage, Snap, e Flatpak

O Problema Principal: “O Inferno das Dependências Linux”

Aplicações Linux tradicionais dependem de **bibliotecas de sistema partilhadas** (ficheiros `.so`).

- **O Conflito:**
  - A Aplicação A precisa da `libXYZ v1.0`
  - A Aplicação B precisa da `libXYZ v2.0`
- **O Resultado:**
  - O seu gestor de pacotes (`apt`, `dnf`) muitas vezes só consegue instalar uma versão.
  - Instalar a Aplicação B quebra a Aplicação A (ou vice-versa).

A Necessidade de Isolamento & Portabilidade

- **Portabilidade:** Uma aplicação empacotada com as suas dependências irá "correr em qualquer lado" (`run anywhere`) em qualquer distribuição Linux, independentemente das suas bibliotecas de sistema.
- **Estabilidade:** Aplicações não podem conflituar com as dependências umas das outras.
- **Segurança:** Se uma aplicação está isolada (`sandboxed`), ela não consegue ler as suas chaves SSH, histórico do navegador, ou outros dados sensíveis.

Como Outros SOs Gerem Isto

Isto não é apenas um problema do Linux.

- **Windows:** Aplicações empacotam quase *todos* os seus ficheiros `.dll` na sua pasta de instalação (ex: `C:\Program Files\App`).
  - **Pró:** Previne conflitos.
  - **Contra:** Muita duplicação; ineficiente.
- **macOS:** "Bundles" `.app` são apenas pastas que contêm o binário da aplicação e todas as suas bibliotecas.
  - **Pró:** Auto-contido e portátil.
  - **Contra:** Também duplica bibliotecas.

Isolamento “Natural”: VMs & Runtimes

Algumas tecnologias fornecem isolamento pela sua própria natureza.

- **Java Virtual Machine (JVM):**
  - O SO corre o processo `java`, não a sua aplicação diretamente.
  - A JVM corre o `bytecode` Java num ambiente gerido e em `sandbox`.
  - Um “Security Manager” controla todo o acesso ao sistema de ficheiros e rede do anfitrião (`host`).

- **Python Virtual Environments (`venv`):**
  - Isto é **isolamento de dependências**, não `sandboxing` de segurança.
  - Cria uma pasta local (`.venv`) com o seu próprio interpretador Python e pacotes (`pygame`, `numpy`).
  - Um ficheiro `requirements.txt` lista todas as dependências, permitindo que `pip install -r requirements.txt` crie um ambiente reprodutível, tal como fizemos no nosso exercício.
  - Isto resolve o problema "Aplicação A vs. Aplicação B" na nossa máquina local, mas não impede a aplicação de ler os nossos ficheiros.

As Soluções Linux Modernas

Três grandes tecnologias emergiram para resolver isto para *qualquer* aplicação, com o objetivo de empacotar a aplicação *e* as suas dependências.

1. **AppImage**
    - **Filosofia:** “Uma aplicação = um ficheiro.” Não é necessária instalação.
2. **Snap**
    - **Filosofia:** “Um pacote seguro e universal.” Apoiado pela Canonical (Ubuntu).
3. **Flatpak**
    - **Filosofia:** “O futuro das aplicações `desktop`.” Apoiado pela Red Hat & comunidade GNOME.

Análise Aprofundada: AppImage

- **Isolamento: Nenhum por defeito.** Foca-se na portabilidade, não na segurança. A aplicação corre como um processo de utilizador normal.
    - *(Pode ser colocada em `sandbox` por ferramentas externas opcionais, como o `firejail`).*
- **Dependências: "Empacotar Tudo."** A aplicação empacota todas as bibliotecas de que precisa, assumindo apenas um sistema base mínimo.
- **Acesso ao Anfitrião: Acesso Total de Utilizador.** A aplicação pode ver e modificar qualquer coisa que o utilizador que a executou pode.

Análise Aprofundada: Snap

- **Isolamento: `Sandbox` Forte.** Usa funcionalidades do `kernel` Linux como `cgroups`, `namespaces`, e **AppArmor** para confinar estritamente a aplicação.
- **Dependências: Empacotadas + `Core Snaps`.** As aplicações empacotam as suas bibliotecas específicas, mas também dependem de um `core snap` partilhado (ex: `core22`) que fornece um `runtime` base do Ubuntu.
- **Acesso ao Anfitrião: "Interfaces."** Negado por defeito. A aplicação tem de declarar o que precisa (ex: `network`, `home`, `camera`).

Análise Aprofundada: Flatpak

- **Isolamento: `Sandbox` Forte.** Usa `namespaces` do `kernel` e uma ferramenta chamada **Bubblewrap (`bwrap`)** para criar um ambiente privado para a aplicação.
- **Dependências: `Runtimes` Partilhados.** Uma aplicação requisita um "Runtime" (ex: `org.gnome.Platform`). Este é descarregado *uma vez* e partilhado por todas as aplicações que precisam dele. Muito eficiente.
- **Acesso ao Anfitrião: "Portals."** Negado por defeito. Quando uma aplicação precisa de um ficheiro, ela pede a um "Portal", que abre um seletor de ficheiros *fora* da `sandbox`. O utilizador escolhe um ficheiro, e *apenas* esse ficheiro é dado à aplicação.

Comparação: Sandboxing & Dependências

Funcionalidade	AppImage	Snap	Flatpak
Sandboxing	❌ Nenhum (por defeito)	✅ Forte (AppArmor)	✅ Forte (Bubblewrap)
Permissões	Acesso total de utilizador	Interfaces (Declarativas)	Portals (Interativos)
Modelo de Dependências	Tudo empacotado no ficheiro	Empacotadas + `Core snaps`	`Runtimes` Partilhados

Comparação: Distribuição & Apoio

Funcionalidade	AppImage	Snap	Flatpak
Distribuição	Descentralizada (qualquer URL)	Centralizada (Snap Store)	Descentralizada (Repositórios)
Apoio Central	Comunidade	Canonical (Ubuntu)	Red Hat / GNOME
Precisa de um `Daemon`?	❌ Não	✅ Sim (`snapd`)	✅ Sim (`flatpak-daemon`)
Integração com o `Desktop`	Opcional (`appimaged`)	Automática	Automática

Limitações: Os Compromissos

- **Espaço em Disco:**
  - **AppImage/Snap:** Empacotar pode ser ineficiente. Uma aplicação de 10MB pode tornar-se num pacote de 150MB.
  - **Flatpak:** `Runtimes` são grandes (muitas vezes 500MB+), mas isto é um `download` **único**.
- **Tempo de Arranque:**
  - **AppImage:** Tem de "montar" o sistema de ficheiros comprimido em cada arranque (pode ser lento).
  - **Snap:** Notoriamente lento no *primeiro arranque* enquanto configura a `sandbox`.

Limitações: O Problema da “Prisão”

- **Segurança vs. Usabilidade:**
  - A `sandbox` é uma “prisão”. Isto é ótimo para a segurança, mas pode ser frustrante.
  - “Porque é que a minha aplicação não vê o meu tema do `desktop`?” (Maioria resolvido agora).
  - “Porque é que a minha aplicação não vê a minha pasta pessoal?” Isto é uma **funcionalidade**, não um `bug`, mas requer que as aplicações sejam reescritas para usar `Portals` corretamente.
- **Não serve para Tudo:**
  - Pouco adequado para ferramentas de linha de comandos (`command-line tools`) que precisam de integração profunda com o sistema (ex: `docker`, `htop`, `drivers` de sistema).

Prática: A Estrutura `AppDir` do AppImage

Um AppImage é apenas um diretório comprimido. Este diretório é chamado de **`AppDir`**.

`MyGame.AppDir/` (A pasta raiz)

- **`AppRun` (Obrigatório):** O `script` de `entrypoint`. É isto que corre quando dá um duplo clique no AppImage. É nosso trabalho escrever este `script` para configurar o ambiente (como o `PYTHONPATH` para o Pygame) e lançar o binário principal.

- `my-game.desktop` **(Obrigatório):** O ficheiro de integração com o `desktop`. Diz ao menu de aplicações do sistema:
  - `Name=My Game`
  - `Exec=AppRun` (Sempre `AppRun`)
  - `Icon=my-game` (O nome do ícone, sem extensão)
- `my-game.png` **(Obrigatório):** O ficheiro de ícone nomeado no ficheiro `.desktop`.
- `usr/...`**:** Uma estrutura Linux padrão contendo os seus binários, bibliotecas, e o interpretador Python portátil.

Prática: AppImage “Hello World”

Aqui, criamos a estrutura `AppDir` *mínima*.

1. **Criar o diretório, `script`, e metadados:**

```
mkdir -p HelloWorld.AppDir
cd HelloWorld.AppDir

# Criar o entrypoint AppRun
echo '#!/bin/bash' > AppRun
echo 'echo "Hello from an AppImage!"' >> AppRun
chmod +x AppRun

# Criar o ficheiro .desktop
echo '[Desktop Entry]' > hello.desktop
echo 'Name=Hello' >> hello.desktop
echo 'Exec=AppRun' >> hello.desktop
echo 'Icon=hello' >> hello.desktop
echo 'Type=Application' >> hello.desktop

# Adicionar um ícone vazio (dummy)
touch hello.png
```

Prática: Empacotar o AppImage

1. **Empacotar!**

```
# Voltar ao diretório pai
cd ..

# Descarregar o appimagetool (só precisa de o fazer uma vez)
wget https://github.com/AppImage/AppImageKit/releases/download/continuous/appimagetool-x86_64.AppIm
chmod +x appimagetool-x86_64.AppImage

# Executar a ferramenta no seu diretório
# Temos de definir ARCH para aplicações baseadas em scripts
ARCH=x86_64 ./appimagetool-x86_64.AppImage HelloWorld.AppDir
```

Resultado: Agora tem o `Hello-x86_64.AppImage`.

Execute-o: `./Hello-x86_64.AppImage`

Prática: O `Manifest` do Flatpak (`.yml`)

Um Flatpak é construído a partir de um ficheiro "manifest" que atua como uma "receita".

- `app-id`: O nome único (ex: `com.example.HelloWorld`).
- `runtime` / `sdk`: O sistema base sobre o qual construir (ex: `org.gnome.Platform`). Não empacotamos o Python; usamos o que vem no `runtime`.
- `command`: O executável a correr.
- `modules`: A lista de "partes" a construir. É aqui que listamos o código da nossa aplicação e as suas dependências (como o `pygame` do PyPI ou o nosso jogo de um URL `git`).

Prática: Flatpak “Hello World”

1. **Criar o `script`:**

```
# Criar um ficheiro chamado hello.sh
echo '#!/bin/sh' > hello.sh
echo 'echo "Hello from a Flatpak Sandbox!"' >> hello.sh
```

2. Criar o `manifest` (`com.example.HelloWorld.yml`):

```yaml
app-id: com.example.HelloWorld
runtime: org.freedesktop.Platform
runtime-version: '23.08'
sdk: org.freedesktop.Sdk
command: hello.sh
modules:
  - name: hello-module
    buildsystem: simple
    build-commands:
      # Instalar o script na sandbox
      - install -Dm755 hello.sh /app/bin/hello.sh
    sources:
      # Dizer ao builder para encontrar o hello.sh no dir do projeto
      - type: file
        path: hello.sh
```

Prática: A Ferramenta `flatpak-builder`

O comando `flatpak-builder` lê o seu `manifest .yml` e realiza a compilação dentro de um ambiente limpo e em `sandbox`.

```
# 1. Construir e instalar a aplicação
flatpak-builder --user --install --force-clean \
  build-dir com.example.HelloWorld.yml
```

- **`--user`**: Instala para o utilizador atual (sem `sudo`).
- **`--install`**: Instala a aplicação assim que é construída.
- **`--force-clean`**: Apaga o diretório de compilação antigo para um começo limpo.
- **`build-dir`**: Uma pasta temporária para o processo de compilação.

```
# 2. Execute a sua nova aplicação!
flatpak run com.example.HelloWorld
```

Prática: Repositórios Flatpak

O Flatpak é descentralizado, como o `git`. Não existe uma "loja" (`store`) única.

- **O que é um Repositório?**
  - Um servidor (ou pasta local) que aloja aplicações, gerido pelo `ostree`.
  - Pode ter múltiplos "remotes" (repositórios) configurados.
- **Flathub: O Repositório "Principal"**
  - `flathub.org` é o repositório central *de facto* para a maioria das aplicações `desktop` (Spotify, VS Code, GIMP, Steam).
  - `flatpak remote-add --if-not-exists flathub https://flathub.org/repo/flathub.flatpakrepo`

- **Como Publicar:**
  - Para colocar a sua aplicação no Flathub, submete o seu ficheiro `manifest .yml` ao repositório GitHub deles como um `pull request`.
  - O sistema de compilação deles constrói, assina e publica automaticamente a sua aplicação por si.

Conclusão

- O **Isolamento** resolve o "Inferno das Dependências" e adiciona **segurança**.
- **AppImage:** Melhor para **portabilidade** simples. "Correr a partir de uma `pen` USB."
  - *Foco:* Estrutura de ficheiros (`AppDir`) e `script` `AppRun`.
- **Snap:** Forte em **IoT/Servidores** e no Ubuntu. Apoiado por uma corporação.
  - *Foco:* Loja central, segurança forte.
- **Flatpak:** O líder no espaço `desktop`. Apoiado pela comunidade (GNOME/KDE) e Red Hat.
  - *Foco:* Escrever "receitas" declarativas (`manifests` `.yml`) e deixar o `flatpak-builder` e os `runtimes` partilhados fazer o trabalho pesado.