Application Containers

Introdução Engenharia Informática

Mário Antunes

October 20, 2025

Exercises

Practical Exercises: Flatpak & AppImage

Objective: This class will guide you through the fundamentals of application packaging. You will start with a simple "Hello World" and progress to packaging a complete Python GUI application with its dependencies.

---

0. Setup: Configure Your Workbench

First, we must install all the tools required for building and testing our application packages.

1. **Update your system:** This command downloads the latest list of available software and upgrades all currently installed packages to their newest versions.

```
$ sudo apt update && sudo apt full-upgrade -y
```

2. **Install tools:** This command installs all the necessary components for our class.

- `curl` & `wget`: Utilities for downloading files from the internet.
- `file`: A utility to identify file types.
- `libfuse2`: A library required by AppImage to "mount" the application package as a virtual filesystem.
- `flatpak`: The command-line tool for running and managing Flatpak applications.
- `flatpak-builder`: The specific tool used to build Flatpak packages from a manifest file.
- `python3`, `python3-pip`, `python3-venv`: The Python interpreter, its package manager (`pip`), and the **virtual environment** tool (`venv`).

```
$ sudo apt install curl wget file libfuse2 flatpak \
flatpak-builder python3 python3-pip python3-venv
```

3. **Add Flathub:** This command adds the **Flathub** repository to your system's Flatpak configuration, but only for your local user (`--user`). A "repository" (or "remote") is a server that hosts Flatpak apps and runtimes. Flathub is the largest and most common repository, and we need it to download the "SDKs" (Software Development Kits) required for building.

```
$ flatpak --user remote-add --if-not-exists \
flathub https://flathub.org/repo/flathub.flatpakrepo
```

4. **Install `appimagetool`:** This downloads the `appimagetool` program, which is what compresses an `AppDir` directory into a single, executable AppImage file. We make it executable (`chmod +x`) and move it to `~/.local/bin`, a standard directory for user-installed programs.

```
$ mkdir -p ~/.local/bin
$ wget -O appimagetool \
"https://github.com/AppImage/AppImageKit/\
releases/download/continuous/appimagetool-x86_64.AppImage"
$ chmod +x appimagetool
$ mv appimagetool ~/.local/bin/
```

5. **Apply the PATH change:** The `PATH` is an environment variable that tells your shell (like `bash`) which directories to search for executable programs. By default, `~/.local/bin` is not always in the `PATH`.

We edit `~/.bashrc` (a file that runs every time you open a new terminal) to add this directory to your `PATH`. This makes `appimagetool` runnable from anywhere.

You can use `nano` to edit the file: `nano ~/.bashrc`. Add the following configuration to the *last* line of the file:

```
export PATH=${HOME}/.local/bin${PATH:+:${PATH}}
```

6. **Log out and log back in.** This re-loads your `~/.bashrc` file and applies the `PATH` change. To verify it's working, open a new terminal and type the following command. You should see the version information for the tool.

```
$ appimagetool --version
```

---

1. "Hello World" 🌍

Let's package a simple shell script.

1.A: Flatpak "Hello World"

Flatpak uses a "manifest" file (in YAML format) to define everything about the application and how to build it.

1. Create a directory for this exercise:

```
$ mkdir ex1-flatpak && cd ex1-flatpak
```

2. Create the application script, named `hello.sh`. This can be created with any editor; one possibility is using `nano`: `nano hello.sh`

```
#!/bin/sh
echo "Hello from a Flatpak Sandbox!"
```

3. Create the manifest file, `pt.ua.deti.iei.HelloWorld.yml`. This file defines:

    - `app-id`: A unique, reverse-DNS name for your app.
    - `runtime` / `sdk`: The base system your app will run on and be built with.
    - `command`: The program to run when the app starts.
    - `modules`: The list of build steps. Here, we define one module that installs our `hello.sh` script into the sandbox's executable path (`/app/bin/`).

```
app-id: pt.ua.deti.iei.HelloWorld
runtime: org.freedesktop.Platform
runtime-version: '25.08'
sdk: org.freedesktop.Sdk
command: hello.sh

modules:
  - name: hello-module
    buildsystem: simple
    build-commands:
      # Installs the script into the sandbox's /app/bin/ folder
      - install -Dm755 hello.sh /app/bin/hello.sh
    sources:
      # Tells the builder to find 'hello.sh' in our project dir
      - type: file
        path: hello.sh
```

4. **Build the package:** This command runs `flatpak-builder` with several important options.

    - `--user`: Builds and installs the app just for your user, without needing `sudo`.
    - `--install`: Automatically installs the app after a successful build.
    - `--install-deps-from=flathub`: Automatically finds and installs any missing SDKs or runtimes from Flathub.
    - `--force-clean`: Deletes the `build-dir` to ensure a fresh build.

- build-dir: The name of the temporary directory to use for building.

```
$ flatpak-builder --user --install --install-deps-from=flathub \
--force-clean build-dir pt.ua.deti.iei.HelloWorld.yml
```

5. **Run and Cleanup:** `flatpak run` executes your application inside its sandbox. After use `cd ..` to exit the directory.

```
$ flatpak run pt.ua.deti.iei.HelloWorld
$ flatpak uninstall --user pt.ua.deti.iei.HelloWorld
```

1.B: AppImage "Hello World"

AppImage works by bundling an entire directory (named `AppDir`).

1. Create a directory for this exercise:

```
$ mkdir ex1-appimage && cd ex1-appimage
```

2. Create the `AppDir` and the main `AppRun` script. The **`AppRun`** file is a special script that acts as the entrypoint. It is the *first* thing that runs when you execute the AppImage. We also create a dummy `icon.png` file.

```
$ mkdir -p HelloWorld.AppDir
$ echo '#!/bin/sh' > HelloWorld.AppDir/AppRun
$ echo 'echo "Hello from an AppImage!"' >> HelloWorld.AppDir/AppRun
$ chmod +x HelloWorld.AppDir/AppRun
$ touch HelloWorld.AppDir/icon.png
```

3. Create a file named `HelloWorld.AppDir/hello.desktop`. This is a **`.desktop` file**, a standard way to tell the Linux desktop environment about your application. It defines the app's `Name`, what command to `Exec` (our `AppRun` script), and what `Icon` to use. `appimagetool` *requires* this file.

```
[Desktop Entry]
Name=Hello
Exec=AppRun
Icon=icon
Type=Application
Categories=Utility;
```

4. **Build the package:** We run `appimagetool` on our `AppDir`. We must also specify `ARCH=x86_64` because the tool cannot "guess" the architecture from a simple shell script. It needs this to name the final file correctly.If necessary change the `ARCH` varible to `arm64`. This will create `Hello-x86_64.AppImage` or `Hello-arm64.AppImage` on sucess.

```
$ ARCH=x86_64 appimagetool HelloWorld.AppDir
```

5. **Run and Cleanup:** After use `cd ..` to exit the directory.

```
$ chmod +x Hello-x86_64.AppImage
$ ./Hello-x86_64.AppImage

# Cleanup
$ rm -rf Hello-x86_64.AppImage
```

---

2. Python CLI App: ASCII Tree 🌳

Let's package a simple Python CLI app. We'll create a `pytree.py` script that recursively lists directories in a tree format.

2.A: Run with Virtual Environment (Venv)

First, let's run the app natively to confirm it works. We will use a **Python virtual environment** to manage dependencies, even though this simple script has none.

A virtual environment (`venv`) is an isolated "bubble" for a Python project. It keeps its *own* Python interpreter and installed packages, so this project's packages (e.g., `pygame`) won't conflict with another project's packages.

1. Create a project directory:

```
$ mkdir ex2-pytree && cd ex2-pytree
```

2. Create the `pytree.py` script, and make it executable: `chmod +x pytree.py`.

```python
#!/usr/bin/env python3
import os
import sys

def tree(startpath):
    """Prints a directory tree."""
    for root, dirs, files in os.walk(startpath):
        # Don't visit .venv or __pycache__
        if '.venv' in dirs:
            dirs.remove('.venv')
        if '__pycache__' in dirs:
            dirs.remove('__pycache__')

        level = root.replace(startpath, '').count(os.sep)
        indent = '    ' * (level - 1) + '   ' if level > 0 else ''

        print(f'{indent} {os.path.basename(root)}/')

        subindent = '    ' * level + '   '
        for f in files:
            print(f'{subindent} {f}')

if __name__ == "__main__":
    # Use current directory or a specified path
    path = sys.argv[1] if len(sys.argv) > 1 else '.'
    tree(os.path.abspath(path))
```

3. **Run the app:** Since this app has no dependencies, we can run it directly. After use `cd ..` to exit the directory.

```
$ ./pytree.py
# Try it on another directory
$ ./pytree.py /tmp
```

2.B: Package `pytree` as a Flatpak

1. Create a project directory:

```
$ mkdir ex2-flatpak && cd ex2-flatpak
```

2. Copy the `pytree.py` file from the previous exercise:

```
$ cp ../ex2-pytree/pytree.py .
```

3. Create the manifest `pt.ua.deti.iei.pytree.yml`. We use `org.gnome.Platform` as our runtime because it conveniently includes a Python 3 interpreter, so we don't have to build Python ourselves.

```yaml
app-id: pt.ua.deti.iei.pytree
runtime: org.gnome.Platform
runtime-version: '48'
sdk: org.gnome.Sdk
command: pytree.py

modules:
  - name: pytree
```

```yaml
    buildsystem: simple
    build-commands:
    - install -Dm755 pytree.py /app/bin/pytree.py
    sources:
    - type: file
      path: pytree.py
```

4. **Build and Install:**

```
$ flatpak-builder --user --install --install-deps-from=flathub \
--force-clean build-dir pt.ua.deti.iei.pytree.yml
```

5. **Run and Cleanup:** When you run it the first time, it only lists the files *inside its own sandbox*. To make it useful, we must grant it permission to see our host files. `--filesystem=home` is a "portal" that pokes a hole in the sandbox, giving the app access to our home directory. After use `cd ..` to exit the directory.

```
$ flatpak run pt.ua.deti.iei.pytree

# It runs inside a sandbox, so it only sees itself!
# Let's give it access to our home directory to test it:
$ flatpak run --filesystem=home pt.ua.deti.iei.pytree ~/

$ flatpak uninstall pt.ua.deti.iei.pytree
```

2.C: Package `pytree` as an AppImage

1. Create a project directory:

   ```
   $ mkdir ex2-appimage && cd ex2-appimage
   ```

2. Create the `AppDir`:

   ```
   $ mkdir -p Pytree.AppDir && cd Pytree.AppDir
   ```

3. **Download and extract portable Python:** Here, we use `wget` to download a pre-built, portable version of Python. An AppImage is just a compressed filesystem, so we use `--appimage-extract` to unpack it. We then move its contents (`mv squashfs-root/* .`) into the root of our `AppDir`. Change the python URL if you use another architecture (such as arm or amr64).

```
$ wget "https://github.com/niess/python-appimage/releases/\
download/python3.10/python3.10.19-cp310-cp310-manylinux_2_28_x86_64.AppImage" \
-O python.AppImage
$ chmod +x python.AppImage
$ ./python.AppImage --appimage-extract
$ mv squashfs-root/* .
$ rm -rf python* squashfs-root/
```

3. **Copy your script:** We copy our script into the `usr/bin` directory provided by the portable Python we just extracted.

```
$ cp ../../ex2-pytree/pytree.py usr/bin/
```

4. **Update the `AppRun` entrypoint:** The portable Python package comes with its own `AppRun` script. We just need to edit its *last line* to call our `pytree.py` script instead of starting a Python shell. Finally, make it executable: `chmod +x AppRun`

```bash
#! /bin/bash
# If running from an extracted image, then export ARGV0 and APPDIR
if [ -z "${APPIMAGE}" ]; then
    export ARGV0="$0"

    self=$(readlink -f -- "$0") # Protect spaces (issue 55)
    here="${self%/*}"
    tmp="${here%/*}"
    export APPDIR="${tmp%/*}"
```

```
fi

# Resolve the calling command (preserving symbolic links).
export APPIMAGE_COMMAND=$(command -v -- "$ARGV0")

# Export TCl/Tk
export TCL_LIBRARY="${APPDIR}/usr/share/tcltk/tcl8.6"
export TK_LIBRARY="${APPDIR}/usr/share/tcltk/tk8.6"
export TKPATH="${TK_LIBRARY}"

# Export SSL certificate
export SSL_CERT_FILE="${APPDIR}/opt/_internal/certs.pem"

# Call Python
"$APPDIR/opt/python3.10/bin/python3.10" "$APPDIR/usr/bin/pytree.py" "$@"
```

5. Create a file named `pytree.desktop` and fill it. We also create a dummy `icon.png` file to satisfy `appimagetool`.

```
[Desktop Entry]
Name=PyTree
Exec=AppRun
Icon=icon
Type=Application
Categories=Utility;
```

```
$ touch icon.png
```

6. **Build, Run, and Cleanup:** After use `cd ..` to exit the directory.

```
$ cd ..  # Go back to ex2-appimage directory
$ ARCH=x86_64 appimagetool Pytree.AppDir

$ chmod +x PyTree-x86_64.AppImage
$ ./PyTree-x86_64.AppImage

# Test it on your home directory
$ ./PyTree-x86_64.AppImage ~/

$ rm -rf PyTree-x86_64.AppImage
```

---

3. Python GUI App: Tic-Tac-Toe 🎮

3.A: Run with Virtual Environment (venv)

This step simulates what a user would do: download the source, extract it, and run it locally.

1. Create a directory and download the source: This archive (a `.tar.gz`) contains a top-level folder. `tar --strip-components=1` is a useful command to extract the *contents* of that folder directly into our current directory, ignoring the top-level folder itself.

```
$ mkdir ex3-tictactoe && cd ex3-tictactoe

$ wget "https://github.com/mariolpantunes/tictactoe/archive/refs/tags/tictactoe-1.0.tar.gz"\
-O tictactoe-1.0.tar.gz

# Extract the downloaded source
$ tar --strip-components=1 -zxvf tictactoe-1.0.tar.gz
```

2. **Create and activate the `venv`:** This time, creating a virtual environment is crucial because we have dependencies. You should see `(venv)` at the beginning of your terminal prompt. This means your shell is now using the Python and `pip` from inside the `./venv` directory.

```
$ python3 -m venv ./venv
$ source venv/bin/activate
```

3. **Install dependencies from the file:** A `requirements.txt` file lists all the Python packages a project needs. `pip install -r` reads this file and installs them (like `pygame`) into the *active virtual environment*.

```
$ pip install -r requirements.txt
```

4. **Run the game:**

```
$ python main.py
```

5. **Deactivate the venv:** This command restores your shell to use the system's default Python. After use `cd ..` to exit the directory.

```
$ deactivate
```

3.B: Package Tic-Tac-Toe as a Flatpak

1. Create a new directory for this build:

```
$ mkdir ex3-flatpak && cd ex3-flatpak
```

2. **Create the manifest `pt.ua.deti.iei.tictactoe.yml`:** This manifest is more complex.
    - `finish-args`: Sets `PYTHONPATH` so the Python interpreter inside the sandbox can find our `minMaxAgent.py` module, which we install in `/app/lib/game`.
    - `python-deps` module: Manually specifies the URL and checksum (`sha256`) for the `pygame` source code. `flatpak-builder` downloads this and builds it from scratch.
    - `game` module: Downloads the game's source code from its URL (just like `wget` did). The `build-commands` then install all the game's parts: the Python scripts, the `assets` folder, and the `.desktop` and `icon` files for application menu integration.

```yaml
app-id: pt.ua.deti.iei.tictactoe
runtime: org.gnome.Platform
runtime-version: "48"
sdk: org.gnome.Sdk
command: game
finish-args:
  - --share=ipc
  - --socket=x11
  - --socket=wayland
  - --device=dri
  - --env=PYTHONPATH=/app/lib/game
modules:
  - name: python-deps
    buildsystem: simple
    build-options:
      env:
        MAKEFLAGS: -j$(nproc)
    build-commands:
      - pip3 install --isolated --no-index --find-links="file://${PWD}" --prefix=/app pygame
    sources:
      - type: file
        url: https://pypi.io/packages/source/p/pygame/pygame-2.6.1.tar.gz
        sha256: 56fb02ead529cee00d415c3e007f75e0780c655909aaa8e8bf616ee09c9feb1f
  - name: game
    buildsystem: simple
    build-commands:
      - install -d /app/lib/game/
      - install -Dm644 minMaxAgent.py /app/lib/game/minMaxAgent.py
      - install -d /app/share/game/
      - cp -r assets /app/share/game/
      - install -Dm755 main.py /app/bin/game
```

```yaml
      - install -Dm644 pt.ua.deti.iei.tictactoe.desktop
        /app/share/applications/pt.ua.deti.iei.tictactoe.desktop
      - install -Dm644 assets/icon.png
        /app/share/icons/hicolor/128x128/apps/pt.ua.deti.iei.tictactoe.png
    sources:
      - type: archive
        url: https://github.com/mariolpantunes/tictactoe/archive/refs/tags/tictactoe-1.0.zip
        sha256: 4210c04451ae8520770b0a7ab61e8b72f0ca46fbf2d65504d7d98646fda79b5a
```

4. **Build and Install:** After installing, your game should appear in your desktop's application menu!

```
$ flatpak-builder --user --install --install-deps-from=flathub \
--force-clean build-dir pt.ua.deti.iei.tictactoe.yml
```

5. **Run and Cleanup:** After use `cd ..` to exit the directory.

```
$ flatpak run pt.ua.deti.iei.tictactoe
$ flatpak uninstall pt.ua.deti.iei.tictactoe
```

3.C: Package Tic-Tac-Toe as an AppImage

1. Create a build directory:

```
$ mkdir ex3-appimage && cd ex3-appimage
```

2. **Download the game source:**

```
$ wget "https://github.com/mariolpantunes/tictactoe/archive/refs/tags/tictactoe-1.0.tar.gz" \
-O tictactoe-1.0.tar.gz
```

3. Create the `AppDir`:

```
$ mkdir -p TTT.AppDir && cd TTT.AppDir
```

4. **Download and extract portable Python:** This is the same as in Exercise 2.C.

```
$ wget "https://github.com/niess/python-appimage/releases/\
download/python3.10/python3.10.19-cp310-cp310-manylinux_2_28_x86_64.AppImage" \
-O python.AppImage
$ chmod +x python.AppImage
$ ./python.AppImage --appimage-extract
$ mv squashfs-root/* .
$ rm -rf python* squashfs-root/
```

5. **Extract your game source:**

```
$ tar --strip-components=1 -zxvf ../tictactoe-1.0.tar.gz
```

6. **Install dependencies from `requirements.txt`:** We use the *bundled* Python's `pip` to install packages. The `--target` flag tells `pip` to install `pygame` *inside* our `AppDir`'s `site-packages` folder, not on the host system.

```
$ ./usr/bin/python3.10 -m pip install -r \
$ ./requirements.txt --target ./usr/lib/python3.10/site-packages/
```

7. **Copy your game files:** We move the game's scripts and assets into the `AppDir`.

```
$ mv main.py minMaxAgent.py assets usr/bin/
```

8. **Update the `AppRun` entrypoint:** This `AppRun` script is updated to set the `PYTHONPATH` variable. This tells the Python interpreter to look for modules in *two* places: our `site-packages` directory (to find `pygame`) and our `usr/bin` directory (to find `minMaxAgent.py`). Finally, make it executable: `chmod +x AppRun`.

```bash
#! /bin/bash
# If running from an extracted image, then export ARGV0 and APPDIR
if [ -z "${APPIMAGE}" ]; then
    export ARGV0="$0"
```

```
    self=$(readlink -f -- "$0") # Protect spaces (issue 55)
    here="${self%/*}"
    tmp="${here%/*}"
    export APPDIR="${tmp%/*}"
fi

# Resolve the calling command (preserving symbolic links).
export APPIMAGE_COMMAND=$(command -v -- "$ARGV0")

# Export TCl/Tk
export TCL_LIBRARY="${APPDIR}/usr/share/tcltk/tcl8.6"
export TK_LIBRARY="${APPDIR}/usr/share/tcltk/tk8.6"
export TKPATH="${TK_LIBRARY}"

# Export SSL certificate
export SSL_CERT_FILE="${APPDIR}/opt/_internal/certs.pem"

# Export PyGame
export PYTHONPATH="$APPDIR/usr/lib/python3.10/site-packages:$APPDIR/usr/bin"

# Call Python
"$APPDIR/opt/python3.10/bin/python3.10" "$APPDIR/usr/bin/main.py" "$@"
```

9. **Add metadata:** We move the `.desktop` file and copy the icon into the root of the `AppDir` so `appimagetool` can find them.

```
$ mv pt.ua.deti.iei.tictactoe.desktop ./tictactoe.desktop
$ cp usr/bin/assets/icon.png ./pt.ua.deti.iei.tictactoe.png
```

10. **Build, Run, and Cleanup:**

```
$ cd .. # Go back to ex3-appimage directory
$ appimagetool TTT.AppDir

$ chmod +x TicTacToe-x86_64.AppImage
$ ./TicTacToe-x86_64.AppImage

$ rm -rf *.AppImage tictactoe-v1.0.tar.xz
```

Conclusion

In these exercises, you packaged a Python application in two distinct ways: as a self-contained **AppImage** and as a sandboxed **Flatpak**.

While both methods achieve portability, this workshop highlights the significant advantages of the Flatpak ecosystem, especially for complex applications.

The **AppImage** process required us to *manually* create a bundle. We had to:

1. Download a portable Python interpreter.
2. Manually install dependencies into a specific `site-packages` folder.
3. Write a custom `AppRun` script to set environment variables like `PYTHONPATH`.

The **Flatpak** process, in contrast, is **declarative** and **reproducible**.

1. **The Manifest is the Recipe:** We simply *declared* all our needs in a single `.yml` manifest file. This one file defines the app, its sources (like the GitHub URL), its Python dependencies, and its sandbox permissions.
2. **Runtimes are Efficient:** Instead of bundling a 100MB+ Python interpreter, we simply requested the `org.gnome.Platform`. This runtime is downloaded *once* by the user and shared across all their Flatpak apps, making our game package itself incredibly small and fast to build.
3. **The Build is Easier:** We didn't need to write any complex shell scripts. `flatpak-builder` handled all the work of downloading the SDK, building `pygame`, and placing files in the correct directories based

on our simple `install` commands.

Overall, Flatpak's use of manifests and shared runtimes results in a build process that is far more automated, maintainable, and efficient for both developers and end-users.