

# Your app everywhere – Just in a Snap!

An **interactive** workshop

Till Kamppeter <[till.kamppeter@gmail.com](mailto:till.kamppeter@gmail.com)>

# Introduction

This workshop is based on

**"Snapping like Hell(sworth)"**

from the Ubuntu Summit 2022

Lucy Llewelyn



Heather Ellsworth

Thanks, Heather Ellsworth and Lucy Llewelyn, for your great work!

# Introduction

**Please download these slides!**

You find them on the **Opportunity Open Source web site**, find this workshop on the **timetable**, open its **description page** and go to "**Presentation Materials**" (bottom).

<https://events.canonical.com/event/35/contributions/291/>

This way you can **browse the slides in your own pace** while setting up and doing the exercises.

And you can **copy and paste** command lines and example code.

You can click the numerous links.

You can also read the **advanced topics** which will not get necessarily presented here.

**May the source (and these slides) always be with you!**

# Introduction

## What you will learn

- What the hell are **Snaps**?
- Why package as Snap?
- How to package GNOME applications as Snaps (**snapping** applications)
- Building your application **from source** or packaging **binaries**
- Finding **dependencies**
- Using an **auxiliary Snap with all GNOME libraries**, to save storage

## What you need to know

- Basic installing from source and from classic packages (DEB, RPM, ...)
- Little code tweaks
- Shell scripting

# Introduction

## Why do I organize and host Snap workshops on conferences?

On the conferences there are many free software **app developers** and also **potential snappers** in the community.

- **Upstream should snap:** Free software projects snap their apps by themselves.
  - They know the ins and outs and the quirks of their own apps
  - They can most easily adapt their apps for snapping
  - The one a user could trust most is the creator of the app
- If upstream does not do it we need **volunteers in the community** for app snapping projects, like **snapcrafters**
  - Well-known projects like snapcrafters have a high degree of trustworthiness, too.

Your application everywhere, just in a Snap!

# Setup

# Setup

This is a **workshop**, interactive, so you will try everything on **your laptop!**

You need **Ubuntu 22.04** or later, **snapt**, **snaptcraft**, and **LXD** installed, basic **development/compiling utilities**, activate Debian sources:

```
$ update-manager (Settings -> Ubuntu Software -> Source code)
$ sudo apt install build-essential git
$ sudo apt build-dep gnome-text-editor (Gets all GNOME libraries and headers)

$ sudo snap install lxd
$ sudo lxd init (Simply press <Enter> on each question)
$ sudo adduser `whoami` lxd
$ sudo snap install --classic snapcraft
$ sudo reboot
```

# Setup

-- OR -- use our **virtual machine** image, on **any system!**

- We give you the file **workshop-ubuntu-23.04.qcow2** (5.7 GB) on a USB stick, please make a copy of it, so you can get back to the original state
- Or **download (well!) before the workshop starts** (let link open in browser):  
[https://drive.google.com/file/d/1kkxZ8GE3\\_UtG7orl5v2d4x\\_T4FhMUcbb/view?usp=sharing](https://drive.google.com/file/d/1kkxZ8GE3_UtG7orl5v2d4x_T4FhMUcbb/view?usp=sharing)
- Works with any software which supports **\*.qcow2** images.
- Use **4 GB RAM, 2 CPUs, and 25 GB storage** if possible. 2 GB RAM and 1 CPU should also work.
- Check **BIOS settings** if hardware virtualization support is enabled.
- For example use **Virtual Machine Manager, GNOME Boxes, or qemu-system-x86\_64**



# Setup

## Virtual Machine Manager

- `$ sudo apt install virt-manager`
- Choose connection "**QEMU/KVM**" and NOT "QEMU/KVM User session" so that you can transfer files via "scp" command
- Click button for creating a **new virtual machine**
- Choose "**Import existing disk image**"
- Operating system is **Ubuntu 23.04**
- Choose **4 GB RAM, 2 CPUs, 25 GB storage**
- Get the **VM's IP address** via "ip addr" command
- **Transfer files** via "scp" command on the host, user "ubuntu", password "ubuntu"

# Setup

## qemu-system-x86\_64

- **Command line:**

```
$ qemu-system-x86_64 -smp 2 -m 4096 -machine accel=kvm \  
-display gtk,gl=on -net nic,model=virtio \  
-net user,hostfwd=tcp::8022-:22 -drive \  
file=IMAGE.qcow2,cache=none,format=qcow2,id=main,if=none \  
-device virtio-blk-pci,drive=main,bootindex=1 \  
-audiodev pa,id=ac97
```

- **Transfer files**, on host do (password: ubuntu):

```
$ scp -p 8022 FILE ubuntu@localhost:.  
$ scp -p 8022 -r DIR ubuntu@localhost:.
```

# Setup

## Your Virtual Machine

- It is **Ubuntu Desktop 23.04**
- User: **ubuntu**
- Password: **ubuntu**
- Preceed command with **sudo** to run it as root, password **ubuntu**
- Transfer files from your host with **scp**
  - Get the IP address via "ip addr" command
  - Or change host name ("**ubuntu**") via **GNOME Control Center**, section "**About**"
- **All the needed tools are already installed**
- Editors installed: **gnome-text-editor, vim, emacs, nano**
- Install any additional package you need
- Mentioned **GIT repositories** in **examples/**, do exercises in **exercises/**

Your application everywhere, just in a Snap!

# What the hell are Snaps? And why should I use them?

# What the hell are Snaps?

- You are developer of an **application**?
- Already thought about how it **gets distributed** to end users?
  - ⇒ **This could turn people away from Linux!**
- You provide the **source code**
  - Only **tech-savvy users** can use it directly
  - You need **goodwill of distro maintainers** to package your app
  - Distro version released ⇒ No update of your app in this distro version ⇒ User always has to update to **newest distro version**
- **You package** your app, for 10+ distros and have to test on 10+ distros
- **That is a nightmare! Isn't it?**

# What the hell are Snaps?

- You have a **smartphone**? There it is much easier: **Google Play Store, App Store**
  - And remember that Canonical developed a **smartphone OS**?
  - They have **learned** from it!
- ⇒ And now we have ...

# Snap!

# What the hell are Snaps?

- A method of **OS-distribution-independent packaging**
  - **You package and test once**, put your **Snap** into the **Snap Store**, and users of **any distro** (Ubuntu, Debian, SUSE, Red Hat, Windows, ...) can use it.
  - **All libraries and other dependencies** come with your Snap
  - **User experience as with smartphone apps**
- Your app runs in a **security shell** (AppArmor, seccomp, namespaces), isolated from the host system
  - So-called **sandboxed packaging**
  - Communication to outside only via **well-defined interfaces**
  - **Snap Store has control**, has to explicitly permit "dangerous" interfaces
  - This way we can **trust third-party apps**
  - We are not dependent any more on distro maintainers for secure packages

# What the hell are Snaps?

- **Don't fear the daemons, we snap them, too!**
  - Snap is universal, not only desktop apps but also daemons, system utilities, sub-systems, drivers, operating system cores, kernels, ... can get snapped
- **Packaging moves from distros to upstream**
  - 10+ distros, each packaging XXX, inventing the wheel 10+ times
  - So let upstream, XXX.org, snap it, distros take the Snap
  - Distro devs concentrate on distro core or contribute to upstream code
  - Distro version released, app updates continue from upstream
- **Immutable distros, Immutable sub-systems, Immutable apps**
  - Ubuntu Core: Immutable, all-Snap distro, desktop under development
  - Snaps are immutable apps (or immutable sub-systems, like the CUPS Snap)
  - Not that immutable, many system components, like printing stack or GPU drivers in separate Snaps



Your application everywhere, just in a Snap!

Example #1:

# Hello World!

# Example #1: Hello World! – What are we doing?

## ● What to expect

- Use **GNOME's Gtk Hello World** example as an **example C program with GUI**
- Step-by-step process creating `snapcraft.yaml`
- Iterative build

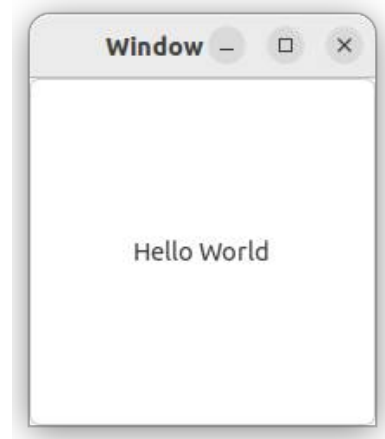
## ● Key concepts

- Basic metadata, apps, parts
- How to include build dependencies (`build-packages:`)
- How to include runtime dependencies (`stage-packages:`)
- "dump" plugin
- Local source (source in same repository as snapping)
- "Manual" building (`override-build:`)
- build and test

# Example #1: Hello World! – Make the app!

- How to make the hello world: <https://www.gtk.org/docs/getting-started/hello-world>
- If you are not using our virtual machine, grab the source code from the site and put it into `~/exercises/hello-world-gtk/src/hello-world-gtk.c`
- Build and run the app:

```
$ cd ~/exercises/hello-world-gtk/src
$ gcc $(pkg-config --cflags gtk4) -o hello-world-gtk \
    hello-world-gtk.c $(pkg-config --libs gtk4)
$ ./hello-world-gtk
```
- Now you know what we will snap ...



# Example #1: Hello World! – Let's snap it!

- We have the **source** and **know how to make the binary**
- Remove the binary, we do not need it in the Snap, it can actually break the Snap:  

```
$ rm hello-world-gtk
```
- Using your favorite editor, create `~/exercises/hello-world-gtk/snapcraft.yaml`  
This is the instruction file for the Snap build
- Ok what do we need?
  - **metadata**: Name, version, summary, description, base, confinement
  - **apps**: List of apps that will be built
  - **parts**: The stuff that builds the apps

(Quick grab: <https://github.com/tillkamppeter/hello-world-gtk>)

# Example #1: Hello World! – Metadata

- Enter these lines into your `snapcraft.yaml` file:

```
name: hello-world-gtk
version: '0.1'
summary: Gtk Hello World example
description: A simple Gtk example
base: core22
confinement: strict
```

- Recommended metadata: <https://snapcraft.io/docs/adding-global-metadata>
- Complete list of metadata keys/values:  
<https://snapcraft.io/docs/snapcraft-top-level-metadata>

(Quick grab: <https://github.com/tillkamppeter/hello-world-gtk>)

# Example #1: Hello World! – Apps

- Add these lines to your `snapcraft.yaml` file:

```
apps:  
  hello-world-gtk:  
    command: src/hello-world-gtk  
    plugs:  
      - x11  
      - wayland
```

(Quick grab: <https://github.com/tillkamppeter/hello-world-gtk>)

# Example #1: Hello World! – Parts

- Add these lines to your `snapcraft.yaml` file:

```
parts:
  hello-world-gtk:
    plugin: dump
    source: .
    override-build: |
      set -eux
      cd src
      gcc $(pkg-config --cflags gtk4) -o hello-world-gtk \
        hello-world-gtk.c $(pkg-config --libs gtk4)
      cd ..
      craftctl default
    build-packages:
      - pkgconf
      - libgtk-4-dev
    stage-packages:
      - libgtk-4-1
```

- For a complete list of parts keys/values: <https://snapcraft.io/docs/adding-parts>

(Quick grab: <https://github.com/tillkamppeter/hello-world-gtk>)

# Example #1: Hello World! – Build it!

- Build command: “snapcraft pack” with options
  - `-v` = verbose output
  - `--debug` = in case of failure, drop us into the build environment
- Output of build process:

```
$ snapcraft pack -v --debug
Starting Snapcraft 7.4.3
Logging execution to
  '/home/ubuntu/.local/state/snapcraft/log/snapcraft-20230729-145410.515361.log'
Running on amd64 for amd64
Launching instance...
[...]
Creating snap package...
Created snap package hello-world-gtk_0.1_amd64.snap
$
```

(Quick grab: <https://github.com/tillkamppeter/hello-world-gtk>)



# Example #1: Hello World! - Install and run it!

- Command: "snap install" with options

- --dangerous = If not installed from store, circumvent store signature

```
$ sudo snap install hello-world-gtk_0.1_amd64.snap --dangerous
hello-world-gtk 0.1 installed
```

```
$ snap run hello-world-gtk
```

```
(hello-world-gtk:109667): Gtk-WARNING **: 14:50:30.500: Locale not supported by C library.
    Using the fallback 'C' locale.
```

```
Gsk-Message: 14:50:30.710: Failed to realize renderer of type 'GskGLRenderer' for surface
'GdkX11Toplevel': libEGL not available
```

```
(hello-world-gtk:109667): Gtk-CRITICAL **: 14:50:30.759: Unable to connect to the accessibility bus
```

```
at 'unix:path=/run/user/1000/at-spi/bus_1,guid=65774ea9c81c70feb5c5007163669b8e': Could not
connect: Permission denied
```

```
Gdk-Message: 14:50:30.760: Failed to get file transfer portal: Could not connect: Permission denied
```

# Example #1: Hello World! – Important Remark

- You probably have seen that **we have compiled the source before snapping** and, instead of including that binary, **let the Snap build process compile it again.**
- This is very important for **sandboxed packaging.**
- Our virtual machine is **Ubuntu 23.04**, so our manually compiled executable is based on the libraries of this distribution
- The executables encapsulated in the Snaps get their libraries from a base distro, in our case **Ubuntu 22.04** (`base: core22` in `snapcraft.yaml`), also the ``stage-packages:`` are from this distro version.
- The binary built under Ubuntu 23.04 does not necessarily work under 22.04, making the Snap fail.

Your application everywhere, just in a Snap!

Example #2:

# GNOME Text Editor

# Example #2: GNOME Text Editor – A GNOME app!

- Example for an **actual GNOME application**
- "extensions: [gnome]" in "apps:" entry: This includes the **GNOME snapcraft extension**, saves work and resources
  - Adding **GNOME, GTK, and desktop** resources
  - Adding all needed plugs: "x11", "wayland", "desktop", "gsettings", ...
  - Connects to the **GNOME content provider Snaps** which provide **all libraries, icons, themes, ...** → **Shared resources save storage space!**
  - A **snapcraft extension** is for snapcraft.yaml like a \*.inc for C files.
- "desktop: FILE.desktop" in "apps:" entry: **\*.desktop for launcher icon**
- D-Bus (session bus) slot for GtkApplication registration
- **When app prints** and plugs "cups", we connect to **CUPS Snap** as content provider to **dependency-install** it, like the GNOME content provider Snaps
- "layout : ": To assure that auxiliary directories of the app can get accessed

# Example #2: GNOME Text Editor – Metadata

- Metadata in ~/examples/gnome-text-editor-workshop-examples/local-deb/snapcraft.yaml (<https://github.com/tillkampeter/gnome-text-editor-workshop-examples>):

```
name: gnome-text-editor
base: core22
grade: stable
confinement: strict
Summary: GNOME's Text Editor
Description: bla bla ...
version: "42.2"
# Trigger automatic installation of the CUPS Snap via "default-provider: cups"
plugs:
  foo-install-cups:
    interface: content
    content: foo
    default-provider: cups
    target: $SNAP_DATA/foo
slots:
  # for GtkApplication registration
  gnome-text-editor:
    interface: dbus
    bus: session
    name: org.gnome.TextEditor
layout:
  /usr/share/gnome-text-editor:
    bind: $SNAP/usr/share/gnome-text-editor
```

# Example #2: GNOME Text Editor – Apps

- Apps entry in `~/examples/gnome-text-editor-workshop-examples/local-deb/snapcraft.yaml` (<https://github.com/tillkamppeter/gnome-text-editor-workshop-examples>):

```
apps:  
  gnome-text-editor:  
    extensions: [gnome]  
    command: usr/bin/gnome-text-editor  
    desktop: usr/share/applications/org.gnome.TextEditor.desktop  
  plugs:  
    - gsettings  
    - mount-observe  
    - cups  
    - home
```

# Example #2: GNOME Text Editor – Get the app

There are several methods to get hold on the app for snapping it  
(~/examples/gnome-text-editor-workshop-examples/  
<https://github.com/tillkamppeter/gnome-text-editor-workshop-examples>):

- `local-deb/`: **Local binary Debian package**, no compiling or source needed, can for example be a **proprietary package to install on Ubuntu Core**. To try, `COPY ~/exercises/gnome-text-editor/gnome-text-editor_42.2-0ubuntu1_amd64.deb`

```
parts:  
  gnome-text-editor:  
    plugin: dump  
    source: gnome-text-editor_42.2-0ubuntu1_amd64.deb  
    source-type: deb
```

We must take care of (non-GNOME) dependencies, via `stage-packages`: (or adding them in additional parts)

# Example #2: GNOME Text Editor – Get the app

- `stage-packages/`: **App which is in Ubuntu as Debian package**, simply **pulled in as a dependency**, always for the correct architecture, but note that the **dependencies of the staged packages are also staged** (and the GNOME ones are not needed, they are in the content provider Snap).

```
parts:  
  gnome-text-editor:  
    plugin: nil  
    source: .  
    stage-packages:  
      - gnome-text-editor
```

To do it correctly we will have to **remove unwished files**, so this method is better for simple command line utilities.

- The 2 previous methods were for quick-&-dirty snapping, let's do the real thing ...



# Example #2: GNOME Text Editor – Build the app

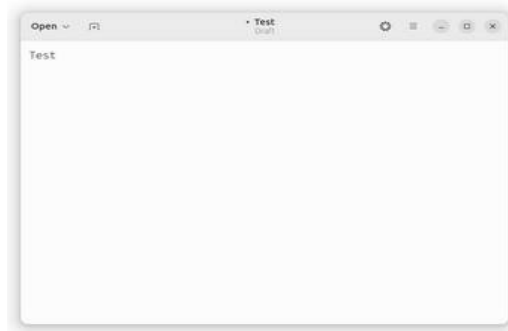
- snap/: Build the app **from source**
  - **The way for "official" Snaps** (especially if you are upstream).
  - No (binary) Debian packages of the app involved.
  - Get **official upstream source**, ideally directly from its **GIT repository** using release tags
    - When hosting the snapping on GitHub we can auto-update the Snap on new upstream versions: <https://ubuntu.com/blog/improving-snap-maintenance-with-automation>
  - **Snapcraft plugins support common build systems:**
    - autotools, make, cmake, flutter, rust, go, ... Usually **meson** for **GNOME apps**
    - You only specify build configuration parameters, all commands are called automatically.
      - meson-parameters:
        - --prefix=/usr
        - --buildtype=release

# Example #2: GNOME Text Editor – Build the app

- **Extract upstream version number, summary, and description** and use it for the Snap (from GIT, from code, from AppStream XML file, ...)
  - At top level: `"adopt-info: gnome-text-editor"`
  - In `gnome-text-editor` part: `"parse-info: <XML file>"`
  - In `gnome-text-editor` app: `"common-id: <App ID from XML file>"`
  - In `gnome-text-editor` part scriptlet: `"craftctl set version=..."`
- **Provide each app's icon** as `$SNAP/meta/gui/icons/hicolor/scalable/apps/NAME.svg` and refer to it using `"Icon=NAME"` in the `*.desktop` file
- **Caching of icon theme directories:** If caches for them are created with `gtk-update-icon-cache`, less system calls and disk seeks. In `override-prime`: find directories with `index.theme` file

# Example #2: GNOME Text Editor – Snap and run!

- **Build the Snap** with the usual  
`$ snapcraft pack -v --debug`
- Make sure that there is **no classically installed GNOME Text Editor** running  
`$ ps aux | grep gnome-text-editor`  
Close any still running instance.
- **Start your snapped GNOME Text Editor** via  
`$ snap run gnome-text-editor`  
or  
`$ /snap/bin/gnome-text-editor`  
Both commands override the `$PATH` priority of `/usr/bin`
- If you click "**Activities**" in the upper left and **search**, the **second result should be the Snap**
- **All versions here generally work** but spill different amounts of warnings into the terminal



Your application everywhere, just in a Snap!

Example #3:

# GNOME Calculator

# Example #3: GNOME Calculator – Stepping it up!

- Have a look at another example
  - <https://github.com/ubuntu/gnome-calculator.git>
- Many things are similar to GNOME Text Editor, but there is also
  - **Summary and description are static**, not taken from AppStream XML
  - Calculator **does not print**, so we do not drag in the CUPS Snap
  - Instead of using "layout:" we use "--prefix=/snap/gnome-calculator/current/usr" and "organize:"
  - In "override-build:" We patch out the **documentation building** in `meson.build`, documentation **does not make sense in a Snap**
  - "cleanup:" part: **Core Snap** and **GNOME content provider Snaps** contain a lot of files of which we get **duplicates** by build-packages and stage-packages and their dependencies. Remove them systematically (Check linter in build process!!).

Your application everywhere, just in a Snap!

Your Turn!

Let's snap **Your** app!

Your application everywhere, just in a Snap!

Get the **perfect snapper** –  
More info/Links

# More info/links:

- **Example Snaps** used here:
  - <https://github.com/ubuntu/gnome-text-editor>
  - <https://github.com/ubuntu/gnome-calculator>
  - <https://github.com/OpenPrinting/ghostscript-printer-app>
- More Snap magic, not only for daemons, in my "**Daemon Snapper's workshop**" (links to slides and exercises/examples):
  - <https://events.canonical.com/event/2/contributions/42/>
- Workshop **GNOME app Snap example from Olivier Tilloy**, each commit in this GIT repository is one step of the `snapcraft.yaml` development:
  - <https://git.launchpad.net/~osomon/+git/secrets-snap/log/?h=main>
- Want to snap something cute? **Qt/KDE** apps? Jesús' talk from Akademy 2023:
  - <https://github.com/jssotomdz/qt-snaps>



# More info/links:

- GitHub workflow to **auto-update your Snap on each upstream release**:
  - <https://ubuntu.com/blog/improving-snap-maintenance-with-automation>
- Ubuntu blogs from Oliver Smith about **optimizing performance of Snaps**:
  - <https://ubuntu.com/blog/how-are-we-improving-firefox-snap-performance-part-1>
  - <https://ubuntu.com/blog/how-are-we-improving-firefox-snap-performance-part-2>
  - <https://ubuntu.com/blog/improving-firefox-snap-performance-part-3>
  - <https://ubuntu.com/blog/firefox-snap-updates-and-upgrades>
- And to know why we all are snapping like hell (**all-Snap Desktop**):
  - <https://ubuntu.com/blog/ubuntu-core-an-immutable-linux-desktop>
- Want to watch some **snappy videos**? Here we go:
  - <https://www.youtube.com/watch?v=TfB6QwR2GYg>
  - <https://www.youtube.com/watch?v=ido6kGmSHWI>

# More info/links:

- And at **OpenPrinting** we are also snappy:
  - <http://www.openprinting.org/>
  - <https://openprinting.github.io/about-us/>
  - <https://openprinting.github.io/news/>
  - <https://snapcraft.io/publisher/openprinting>
  - <https://github.com/OpenPrinting>

Your application everywhere, just in a Snap!

# Advanced Topics

Your application everywhere, just in a Snap!

# Finding Dependencies

# Finding Dependencies

A Snap needs to include all dependencies of the application: Libraries, fonts, icons, utilities, ... There are several ways to find them:

- **Desktop apps:** Use **snappycraft extensions** to cover most: "gnome", "kde-neon", "flutter-...", ...
- Have a look at **classic Debian (or RPM) packages**. Overtake their dependencies for the Snap
- Investigate executables (and libraries) with the "ldd" command
- Read **upstream source documentation**
- Check the **linter** output in the end of the Snap build

Your application everywhere, just in a Snap!

# The GNOME extension and library/resource Snaps

# The GNOME extension and library/resource Snaps

- With "extensions: [gnome]" in an "apps:" entry we include the **GNOME snapcraft extension**
- A **snapcraft extension** is for snapcraft.yaml like a \*.inc for C files
- **Saves a lot of work for the snapper**, to not have to repeat GNOME-specific stuff and update it separately in each Snap
  - Adds **GNOME, GTK, and desktop** resources
  - Adds all **needed plugs**: "x11", "wayland", "desktop", "gsettings", ...
  - Connects to the **GNOME content provider Snaps** which provide **all libraries, icons, themes, ...**
- **Content provider Snaps** contain files to be shared between other Snaps, typically libraries, icons, themes for **GUI apps** for **GNOME, KDE, Flutter, ...**
  - **Snaps bringing all their dependencies get huge! Sharing helps.**

# The GNOME extension and library/resource Snaps

- **TODO: More snapcraft extensions:**

- **CUPS:** Do both plug "cups" and add the content-provider blob to install the CUPS Snap by a simple "extension: [cups]". See snapcraft.yaml of the GNOME Text Editor
- **Printer Applications:** A lot of repetition in the snapcraft.yaml and a lot of work to apply updates in each of them ... Imagine simply having parts to build a classic CUPS driver plus an "extension: [retro-printer-app]" and that's it ... See <https://github.com/OpenPrinting/ps-printer-app>



Your application everywhere, just in a Snap!

# App from source or binary?

# Apps from source or binary?

With the GNOME Text Editor we have seen that one can build Snaps from **binary files** or **compile the source**, what to use?

- **Source code**, usually from **upstream project (is that you?)**
  - **Free software/open-source** only
  - **Security**: Auditable, choice of more secure compiler options, quick fixes
  - **Adaptable** to Snap file system and encapsulation
  - **More frequent updates**
    - Does not depend on goodwill of distro packagers, low frequency of Ubuntu LTS, ...
    - Immediately grab upstream releases, even development snapshots
    - GIT-based update-automation:  
<https://ubuntu.com/blog/improving-snap-maintenance-with-automation>
    - Especially important for hardware enablement (drivers)

# Apps from source or binary?

## ● Binary files

- Allows snapping **proprietary, closed-source** software
  - **Do not put into the Snap Store without permission!**
  - For private use, like **running the software on Ubuntu Core**
- Quick addition of software pieces as "stage - packages : "
  - **Update only every 2 years with Ubuntu LTS**
  - Pulls in all the package's dependencies, can be good, but also bad (desktop apps with content provider Snap).
- **Less/no knowledge in compiling/patching/coding needed**
- **Not adaptable**, for file system paths, system users, chown, ... **So snapping could get tricky**

Your application everywhere, just in a Snap!

# Patching the code

# Patching the code

- Why do we need to **modify (patch) the upstream code**?
  - Complex applications often get **tricky to snap**
  - **Upstream's design** did not take Snap into account
  - Upstream **does not always accept the changes**, or it takes time until the next release
  - Even **classic Debian/RPM** packaging often needs patches
- **Examples** for patching needs
  - Directory and file **locations used are hard-coded**, no options, config, ...
  - A snapped service (with a slot) has to determine **what the client Snap plugs**, to accept/deny inquiry
  - Application does **operations which are not allowed under confinement** (like chown/chmod) but are also not needed under confinement

# Patching the code

- In `snapcraft.yaml` (only relevant lines, patch is in `snap/local/` of your project repository):

```
parts:
```

```
  my-application:
```

```
    override-build: |
```

```
      patch -p1 < $CRAFT_PROJECT_DIR/snap/local/log.patch
```

```
      craftctl default
```

# Patching the code

- Alternative: **Command-line editing** with "sed" or "perl":
- In `snapcraft.yaml` (only relevant lines):

```
parts:
```

```
  my-application:
```

```
    override-build: |
```

```
      sed -i.bak -e 's|Icon=@app_id@$|Icon=app.icon|g' \
        app.desktop.in.in
```

```
      perl -p -i -e 's/chown lp.lp file//' script.sh
```

```
      craftctl default
```

Your application everywhere, just in a Snap!

# Interfaces: Safe vs. Dangerous



# Interfaces: Safe vs. Dangerous

- Snapped applications are **completely encapsulated** (AppArmor, seccomp, namespaces)
- By default, they cannot communicate with the host system or with other Snaps
- Communication is possible via **well-defined interfaces**: "network", "cups", "dbus", ...
- A "**plug**" has to be connected with a "**slot**" of the system or of another Snap in order to communicate
  - "**Safe**" interfaces
    - Ex.: "cups" which allows listing available printers and printing
    - **are auto-connected** when installing from Snap Store
  - "**Dangerous**" interfaces
    - Ex.: "cups-control" which allows creating/removing printers, delete all jobs ...
    - **need manual connection** or **permission** from Snap Store team for auto-connection

Your application everywhere, just in a Snap!

# Cleaning up ...

# Cleaning up ...

- Snaps containing all the app's dependencies **can get very large**
- So we should take care not to pack too much
- **We should not include (or remove):**
  - **Header files of libraries** (\* .inc), pkgconfig files (\* .pc), static libraries (\* .a), \* .la, lintian
  - **Development utilities**
  - **Man pages** and other **documentation, examples**
  - /var
  - Library binaries which are **reported unused** by the linter
  - Files which the desktop's/GUI toolkit's content provider Snap already contains
  - Parts of the upstream package we do not need in this Snap (if we need libcups of the CUPS package, remove cupsd)

# Cleaning up ...

- How to remove/clean up:
  - "Negative" entries in "prime:"  
prime:
    - -usr/include
    - -usr/lib/pkgconfig
    - -usr/share/fonts
    - -usr/share/man
    - -usr/share/doc
    - -usr/share/doc-base
    - -usr/share/lintian

# Cleaning up ...

- **Do not build the whole source code** (via "override-build:") or build ("./configure") parameters
- **Skip "make install", manually copy selected files** (via "override-build:")

```
override-build: |
  set -eux
  ./configure --sysconfdir=/var/snap/...
  cd cups
  make
  cd ..
  cd filter
  make rastertoepson
  cd ..
  mkdir -p $CRAFT_PART_INSTALL/usr/lib
  cp cups/libcups*.a $CRAFT_PART_INSTALL/usr/lib/
  cp -P cups/libcups*.so* $CRAFT_PART_INSTALL/usr/lib/
  [...]
  mkdir -p $CRAFT_PART_INSTALL/usr/lib/ghostscript-printer-app/filter
  cp filter/rastertoepson $CRAFT_PART_INSTALL/usr/lib/ghostscript-printer-app/filter
  #craftctl default # DO NOT do the default action here!!
```

# Cleaning up ...

- **Build from source instead of pulling it as "stage-packages:",** then adapt/customize
  - The Ghostscript Printer Application Snap contains many printer drivers which Debian provides as "printer-driver-..." Debian packages
  - The packages depend on CUPS (the daemon), Ghostscript, QPDF, and a lot of other things
  - As we have the CUPS daemon in the CUPS Snap and build the other dependencies by ourselves from source for newer versions (than Ubuntu LTS) and updates, we avoid "stage-packages:" here and build from source
  - But we use the source of Debian's packaging repos to get it with Debian's patches

# Cleaning up ...

- **"cleanup:" part applied in the very end of Snap build**

- Spot duplicate files (here libraries), especially also duplicates with the content provider Snaps for the dektops/GUIs

```
cleanup:
```

```
  after: [gnome-calculator]
```

```
  plugin: nil
```

```
  build-snaps: [core22, gtk-common-themes, gnome-42-2204]
```

```
  override-prime: |
```

```
    set -eux
```

```
    for snap in "core22" "gtk-common-themes" "gnome-42-2204"; do
```

```
      cd "/snap/$snap/current" && \
```

```
        find . -type f,l -name *.so.* \
```

```
          -exec rm -f "$CRAFT_PRIME/{" \;
```

```
    done
```