
appimage-builder

Release 0.5.0

Sep 30, 2020

1	Getting help	3
2	First steps	5
2.1	appimage-builder at a glance	5
2.2	Installation guide	7
2.3	Tutorial	8
2.4	Frequent Asked Questions	10
2.5	Shell application (BASH)	11
2.6	Gnome application (gnome-calculator)	11
2.7	Qt/Kde application (kcalc)	13
2.8	Multimedia application (VLC)	14
2.9	PyQt5 application	15
2.10	Path Mapping (GIMP)	17
2.11	AppImage Updates	20
2.12	AppImage Signing	21
2.13	Testing	22
2.14	Troubleshooting	24
2.15	Full AppImage bundle	26
2.16	Producing AppImages on Gitlab CI	29
2.17	Producing AppImages on GitHub	30
2.18	Version: 1 (Draft)	31

appimage-builder is a novel tool for creating AppImages. It uses the system package manager to resolve the application dependencies and creates a complete bundle. It can be used to pack almost any kind of applications including those made using: C/C++, Python, and Java.

Featuring:

- Real GNU/Linux packaging (no more distro packaging)
- Simple recipes
- Simple workflow
- Backward and forward compatibility
- One binary, many target systems.

For information about the AppImage packaging format visit: <https://appimage.org/>

CHAPTER 1

Getting help

Having trouble? We'd like to help!

- Try the [FAQ](#) – it's got answers to some common questions.
- Ask or search questions in [StackOverflow](#) using the [AppImage](#) tag.
- Ask or search questions in the [AppImage](#) subreddit.
- Ask a question in the [#appimage](#) IRC channel.
- Report bugs with [appimage-builder](#) in our [issue tracker](#).

2.1 appimage-builder at a glance

appimage-builder is a tool for packaging other applications into AppImages. Any kind of application can be packaged using this tool and unlike other AppImage creation tools it can be used in modern systems and the resulting bundle will be backward compatible.

NOTICE: Only GNU/Linux distributions that contains the **APT** package manager are supported. In the future other package managers will be added.

2.1.1 Walk-through of an example *appimage-builder* recipe

appimage-builder uses a recipe to configure the AppImage creation process. Here's an example of a recipe for building a Bash AppImage:

```
version: 1

AppDir:
  path: ./AppDir

  app_info:
    id: org.gnu.bash
    name: bash
    icon: utilities-terminal
    version: 4.4.20
    exec: bin/bash
    exec_args: $@

  apt:
    arch: amd64
    sources:
      - sourceline: 'deb [arch=amd64] http://archive.ubuntu.com/ubuntu/ bionic main'
        key_url: 'http://keyserver.ubuntu.com/pks/lookup?op=get&
↪ search=0x3b4fe6acc0b21f32'
```

(continues on next page)

(continued from previous page)

```
include:
  - bash
  - coreutils
exclude:
  - dpkg

test:
  centos:
    image: appimage-builder/test-env:centos-7
    command: "./AppRun -c \"ls\""
    use_host_x: True

AppImage:
  update-information: None
  sign-key: None
  arch: x86_64
```

Put this in a file named *appimage-builder.yml* and run the tool using the following command:

```
appimage-builder --recipe appimage-builder.yml --skip-test
```

When this finishes you will have in current working directory an AppImage file like this: *bash-4.4.20-x86_64.AppImage*

To execute it just do:

```
./bash-4.4.20-x86_64.AppImage
```

2.1.2 What just happened?

When you ran the command `appimage-builder --recipe appimage-builder.yml` the tool read the recipe file and executed the following tasks:

1. APT configuration

An APT configuration will be generated in the `appimage-builder-cache` directory. This directory will hold a cache of the resources used to build the AppImage. In the listed sources will be configured as APT sources and the keys will be added to an internal keyring.

Then `apt update` will be executed using the newly created configuration.

The packages listed in the `AppDir >> apt >> exclude` section will be set as 'Installed' in the APT configuration to prevent their inclusion.

2. Binaries deployment

The packages listed in the `AppDir >> apt >> include` along with their dependencies will be downloaded and deployed into the *AppDir path*. Only the *glibc* packages will be deployed to a special location on *opt/libc* so they can be easily ignored at runtime.

3. Runtime Setup

This step has the purpose of making all the embed resources available to the application at runtime. Therefore it's aid by a set of helpers that are activated depending on whether some binaries are found. Those helpers will add configuration files to the bundle and set the required environment variables to the *.env* file.

By example the Qt helper will be used if *libQt5Core.so.5* is found. This Qt helper will create the required *qt.conf* files to ensure that the Qt plugins are properly resolved.

Finally the AppRun and libapprun_hooks.so files are added. The first one loads the *.env* file and executes the application. The other makes sure that the environment configuration that is required to execute your AppImage doesn't propagate to other applications executed.

4. Tests

Once the binaries and the runtime configuration are in place the AppDir is considered completed and can be executed as follows: `AppDir/AppRun`. This is the same command used by the AppImage runtime to start the application. At this point *appimage-builder* proceeds to run the tests cases described in `AppDir >> test`. In each test case the command specified at `AppDir >> test >> (test name) >> command` is executed inside a container made of the image specified at `AppDir >> test >> (test name) >> image`. This allow us to test how will behave the application in different systems without the need create a virtual machine.

5. Bundling

Finally the whole AppDir is compressed into an squashfs file and appended to a runtime binary. This binary does the function of mounting the bundle at runtime and calling the AppRun in it. It also contains the update information and signature of the AppImage.

To perform this tasks appimageroot is used. If everything went OK, the output should be a nice AppImage file.

2.1.3 What else?

You have seen how to make recipe for Bash and how it's used to build an AppImage. But this is just the surface. With *appimage-builder* you can create recipes for almost any kind of glibc based applications. We invite you to check the examples sections to see other recipes for different frameworks and technologies.

Also it's important to say that contents of your bundle are not limited to those resources available in some APT repository. You can also include self build binaries, check the script section in the recipe specification for more details.

2.1.4 What's next?

The next steps for you is to *install appimage-builder, follow through the tutorial* to learn how to create recipes for more complex applications and join the *appimage community*.

Thanks for your interest!

2.2 Installation guide

The project is built using Python 3 and uses various command-line applications to fulfill its goal. Depending on the host system and the recipe the packages providing such applications may vary.

NOTICE: The only package manager fully supported currently is **APT**. *appimage-builder* will not work on systems without it.

2.2.1 Install on Debian/Ubuntu

Installing dependencies

```
sudo apt install -y python3-pip python3-setuptools patchelf desktop-file-utils libgdk-
↳pixbuf2.0-dev fakeroot

# Install appimagetool AppImage
sudo wget https://github.com/AppImage/AppImageKit/releases/download/continuous/
↳appimagetool-x86_64.AppImage -O /usr/local/bin/appimagetool
sudo chmod +x /usr/local/bin/appimagetool
```

Installing latest tagged release:

```
sudo pip3 install appimage-builder
```

Installing development version:

```
sudo pip3 install git+https://github.com/AppImageCrafters/appimage-builder.git
```

2.2.2 Install in a Docker container

There is an issue in the AppImage runtime format that prevents it proper execution inside docker containers. Therefore we must use the following workaround to make *appimagetool* work properly:

```
# Install appimagetool AppImage
sudo wget https://github.com/AppImage/AppImageKit/releases/download/continuous/
↳appimagetool-x86_64.AppImage -O /opt/appimagetool
sudo chmod +x /opt/appimagetool
cd /opt/; sudo /opt/appimagetool --appimage-extract
sudo mv /opt/squashfs-root /opt/appimagetool.AppDir
sudo ln -s /opt/appimagetool.AppDir/AppRun /usr/local/bin/appimagetool
```

2.2.3 Premade Docker image

There is a docker image with *appimage-builder* ready to be used at hub.docker.com. Download it using:

```
docker pull appimagecrafters/appimage-builder:latest
```

2.3 Tutorial

In this page is explained how to build an AppImage for a simple Qt/Qml application. The tutorial is meant to be performed in a Ubuntu system where **appimage-builder** have been installed. Check the *Installation guide* for instructions. The application code can be found [here](#).

2.3.1 Build Configuration

To pack an application as AppImage we need to configure it as it were to be installed in a regular GNU/Linux system. That implies using a Release configuration and setting as /usr as installation prefix. But instead of installing it to our root dir /, it's going to be installed to AppDir. This directory will be used later to also deploy the application dependencies.

It's recommended that the applications install procedure also deploys a desktop entry and a icon for the application, as in a regular installation. These resources will be used to infer the bundle metadata.

```
git clone https://www.opencode.net/azubieta/qt-appimage-template.git
cd qt-appimage-template
cmake . -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=/usr
make install DESTDIR=AppDir
```

2.3.2 Recipe Generation

Once the application binaries are deployed to an AppDir we proceed to run `appimage-builder --generate`. This will proceed to read the application executable from the desktop entry and launch it. The application will show up you can proceed to close it. In complex applications where external plugins are used it's recommended to test all the features before closing, this will make sure that all runtime dependencies are identified.

Now the tool proceeds to confirm the bundle metadata. At this point you can customize the application id, name, icon, version, executable path, execution arguments and the target architecture.

```
? ID [Eg: com.example.app] : QtQuickControls2Application
? Application Name : Qt AppImage Template
? Icon : QtQuickControls2Application
? Version : latest
? Executable path relative to AppDir [usr/bin/app] : usr/bin/qt-appimage-template
? Arguments [Default: $@] : $@
? Architecture : amd64
```

Once done the recipe is printed to the standard output and to a file named `AppImageBuilder.yml`

Filling the gaps

If you open the `AppImageBuilder.yml` file you will find along the apt source line configuration a set of empty `key_url` entries. Those cannot be resolved currently by the tool and must be filled manually.

Apt repositories key urls are usually located in the repository root. In the case of the Ubuntu official repositories and PPAs they can be found in the [Ubuntu keyserver](#).

There is no need to set a `key_url` for each sourceline if it was set before as all the keys are added to the same keyring. Let's remove all the empty `key_url` entries to make apt complain about the missing keys and run `appimage-builder`.

You will see something like this, check the line ends:

```
INFO:apt-get:Err:2 http://archive.ubuntu.com/ubuntu bionic-updates InRelease
INFO:apt-get:The following signatures couldn't be verified because the public key is
↳not available: NO_PUBKEY 3B4FE6ACC0B21F32
```

To make an ubuntu keyring url you can use the following snippet, replace `<KEY ID>` by the value that comes after the `NO_PUBKEY` in the `appimage-builder` output:

```
http://keyserver.ubuntu.com/pks/lookup?op=get&search=0x<KEY ID>  
http://keyserver.ubuntu.com/pks/lookup?op=get&search=0x3B4FE6ACC0B21F32
```

Deploying dependencies

Run `appimage-builder --skip-test --skip-appimage` to deploy the dependencies into to `AppDir`. Notice we all intentionally skipping the test and `AppImage` generation steps. Once the process complete should have a working `AppDir`. To do a quick test you can run `AppDir/AppRun` and the application will show up.

2.3.3 Testing the AppDir

So far we should have a functional `AppDir`, which can be tested by running `AppDir/AppRun`. This will let's know if the bundle will run on our system. To know if it will run on other systems we can make use of docker. Official docker images tend to be minimal so they are great to test our bundle.

`appimage-builder` provides the means to automate the *test* process. All that you have to do is specify the docker image to be used and the command to start the application. In case of an application with a graphical interface set `use_host_x` to `True`. This will share the host X11 server with the docker container.

The generated recipe already comes with a set of tests configured, to run then use:

NOTE: If the docker images are not in your system it may take a while to download.

Once all the tests cases are completed successfully your `AppDir` is ready to be transformed into an `AppImage`.

Two important notes on testing inside docker:

- use docker images that include X11 libraries when testing graphic applications, like the [ones here](#).
- applications with graphical interface will stay running after they are started, therefore you will have to manually close then to proceed with the next test case.

2.3.4 Bundling everything together

You have made and tested and `AppDir` containing your application binaries and it's dependencies. The final step is to generate the `AppImage` as follows:

2.3.5 What's next

The next steps for you is to learn how to do *AppImage Updates* and *AppImage Signing*. You may also want to check the recipe specification *Version: 1 (Draft)* for advanced tuning.

Thanks for your interest!

2.4 Frequent Asked Questions

2.4.1 What kind of application I can pack as AppImage?

In theory every kind of application no matter the technology used to build it. But some of then are a bit complex. Join our *community* to get some help.

2.4.2 What systems are supported?

Currently only Debian or Ubuntu based system are supported. Other will be added in the future.

2.4.3 Where can I ask more about appimage-builder?

In the `appimage-builder` github project or in the *Getting help* spaces.

appimage-builder at a glance Understand what *appimage-builder* is and how it can help you.

Installation guide Get *appimage-builder* installed on your computer.

Tutorial Write your first *appimage-builder* recipe.

2.5 Shell application (BASH)

This recipe will generate a aarch64 (arm64) AppImage for bash. It's cross-built from a amd64 system.

```

version: 1

AppDir:
  path: ./AppDir

  app_info:
    id: org.gnu.bash
    name: bash
    icon: utilities-terminal
    version: 4.4.20
    exec: bin/bash
    exec_args: $@

  apt:
    arch: arm64
    sources:
      - sourceline: 'deb [arch=arm64] http://ports.ubuntu.com/ubuntu-ports bionic main
↵ '
        key_url: 'http://keyserver.ubuntu.com/pks/lookup?op=get&
↵search=0x3b4fe6acc0b21f32'

    include:
      - bash
      - coreutils
    exclude:
      - libpcre3

AppImage:
  update-information: None
  sign-key: None
  arch: aarch64

```

2.6 Gnome application (gnome-calculator)

NOTE: If your app uses svg images you should bundle `librsvg2-common`

```

version: 1

AppDir:
  path: ./AppDir

  app_info:
    id: org.gnome.Calculator
    name: gnome-calculator
    icon: gnome-calculator
    version: 3.28.0
    exec: usr/bin/gnome-calculator

  apt:
    arch: i386
    sources:
      - sourceline: 'deb [arch=i386] http://mx.archive.ubuntu.com/ubuntu/ bionic main_
↳restricted universe multiverse'
      - key_url: 'http://keyserver.ubuntu.com/pks/lookup?op=get&
↳search=0x3b4fe6acc0b21f32'

    include:
      - gnome-calculator
      - librsvg2-common
    exclude:
      - adwaita-icon-theme
      - humanity-icon-theme

  files:
    exclude:
      - usr/lib/x86_64-linux-gnu/gconv
      - usr/share/man
      - usr/share/doc/*/README.*
      - usr/share/doc/*/changelog.*
      - usr/share/doc/*/NEWS.*
      - usr/share/doc/*/TODO.*

  test:
    debian:
      image: appimagecrafters/tests-env:debian-stable
      command: "./AppRun"
      use_host_x: True
    centos:
      image: appimagecrafters/tests-env:centos-7
      command: "./AppRun"
      use_host_x: True
    arch:
      image: appimagecrafters/tests-env:archlinux-latest
      command: "./AppRun"
      use_host_x: True
    fedora:
      image: appimagecrafters/tests-env:fedora-30
      command: "./AppRun"
      use_host_x: True
    ubuntu:
      image: appimagecrafters/tests-env:ubuntu-xenial
      command: "./AppRun"
      use_host_x: True

```

(continues on next page)

(continued from previous page)

```
AppImage:
  arch: i686
```

2.7 Qt/Kde application (kcalc)

```
version: 1

AppDir:
  path: ./AppDir

  app_info:
    id: org.kde.kcalc
    name: kcalc
    icon: accessories-calculator
    version: 17.12.3
    exec: usr/bin/kcalc

  apt:
    arch: amd64
    sources:
      - sourceline: 'deb [arch=amd64] http://archive.ubuntu.com/ubuntu/ bionic main_
↳restricted universe multiverse'
        key_url: 'http://keyserver.ubuntu.com/pks/lookup?op=get&
↳search=0x3b4fe6acc0b21f32'
      - sourceline: 'deb [arch=amd64] http://archive.ubuntu.com/ubuntu/ bionic-
↳updates main restricted universe multiverse'
      - sourceline: 'deb [arch=amd64] http://archive.ubuntu.com/ubuntu/ bionic-
↳backports main restricted universe multiverse'

    include:
      - kcalc
      - libpulse0
    exclude:
      - core-packages
      - graphics-stack-packages
      - xclient-packages

      - phonon4qt5
      - libkf5service-bin
      - perl
      - perl-base
      - libpam-runtime

  files:
    exclude:
      - usr/lib/x86_64-linux-gnu/gconv
      - usr/share/man
      - usr/share/doc/*/README.*
      - usr/share/doc/*/changelog.*
      - usr/share/doc/*/NEWS.*
      - usr/share/doc/*/TODO.*
  runtime:
```

(continues on next page)

(continued from previous page)

```

env:
  APPDIR_LIBRARY_PATH: $APPDIR/lib/x86_64-linux-gnu:$APPDIR/usr/lib/x86_64-linux-
↳gnu:$APPDIR/usr/lib/x86_64-linux-gnu/pulseaudio

test:
  debian:
    image: appimagecrafters/tests-env:debian-stable
    command: "./AppRun"
    use_host_x: True
    env:
      - QT_DEBUG_PLUGINS=1
  centos:
    image: appimagecrafters/tests-env:centos-7
    command: "./AppRun"
    use_host_x: True
    env:
      - QT_DEBUG_PLUGINS=1
  arch:
    image: appimagecrafters/tests-env:archlinux-latest
    command: "./AppRun"
    use_host_x: True
    env:
      - QT_DEBUG_PLUGINS=1
  fedora:
    image: appimagecrafters/tests-env:fedora-30
    command: "./AppRun"
    use_host_x: True
    env:
      - QT_DEBUG_PLUGINS=1
  ubuntu:
    image: appimagecrafters/tests-env:ubuntu-xenial
    command: "./AppRun"
    use_host_x: True

AppImage:
  update-information: None
  sign-key: None
  arch: x86_64

```

2.8 Multimedia application (VLC)

```

version: 1

script:
  - rm -r ./AppDir || true

AppDir:
  path: ./AppDir

app_info:
  id: vlc
  name: VLC media player
  icon: vlc

```

(continues on next page)

(continued from previous page)

```

version: 3.0.8-0-gf350b6b5a7
exec: usr/bin/vlc

apt:
  arch: amd64
  sources:
    - sourceline: 'deb [arch=amd64] http://archive.ubuntu.com/ubuntu/ bionic main_
↳restricted universe multiverse'
      key_url: 'http://keyserver.ubuntu.com/pks/lookup?op=get&
↳search=0x3b4fe6acc0b21f32'
    - sourceline: 'deb [arch=amd64] http://archive.ubuntu.com/ubuntu/ bionic-
↳updates main restricted universe multiverse'

  include:
    - vlc

test:
  debian:
    image: appimagecrafters/tests-env:debian-stable
    command: "./AppRun"
    use_host_x: True
  centos:
    image: appimagecrafters/tests-env:centos-7
    command: "./AppRun"
    use_host_x: True
  arch:
    image: appimagecrafters/tests-env:archlinux-latest
    command: "./AppRun"
    use_host_x: True
  fedora:
    image: appimagecrafters/tests-env:fedora-30
    command: "./AppRun"
    use_host_x: True
  ubuntu:
    image: appimagecrafters/tests-env:ubuntu-xenial
    command: "./AppRun"
    use_host_x: True

AppImage:
  arch: x86_64

```

2.9 PyQt5 application

Packaging a Python3 application into an AppImage is quite similar to packaging a regular compiled application. The trick consist on embedding the python interpreter along with the application code.

2.9.1 Requirements

- modern Debian/Ubuntu system
- python3 and pip
- appimage-builder installed
- apt-get

2.9.2 Instructions

0. Use the recipe below as template
1. Copy the application code into AppDir/usr/src
2. Copy the application icon to AppDir/usr/share/icons/hicolor/256x256/apps/
3. **Install the application requirements using pip:** `python3 -m pip install --system --ignore-installed --prefix=/usr --root=AppDir -r ./requirements.txt`
4. Setup the **PYTHONHOME** and **PYTHONPATH** environment variables, make sure the
5. Run `appimage-builder`

The complete example source code can be found [here](#).

2.9.3 Recipe

```

version: 1
script:
  # Remove any previous build
  - rm -rf AppDir | true
  # Make usr and icons dirs
  - mkdir -p AppDir/usr/src
  # Copy the python application code into the AppDir
  - cp main.py AppDir/usr/src -r
  # Install application dependencies
  - python3 -m pip install --system --ignore-installed --prefix=/usr --root=AppDir -r
↪ ./requirements.txt

AppDir:
  path: ./AppDir

  app_info:
    id: org.appimage-crafters.python-appimage-example
    name: python appimage hello world
    icon: utilities-terminal
    version: 0.1.0
    # Set the python executable as entry point
    exec: usr/bin/python3
    # Set the application main script path as argument. Use '$@' to forward CLI
↪ parameters
    exec_args: "$APPDIR/usr/src/main.py $@"

  apt:
    arch: amd64
    sources:
      - sourceline: 'deb [arch=amd64] http://archive.ubuntu.com/ubuntu/ bionic main
↪ restricted universe multiverse'
      key_url: 'http://keyserver.ubuntu.com/pks/lookup?op=get&
↪ search=0x3b4fe6acc0b21f32'

    include:
      - python3
      - python3-pkg-resources
      - python3-pyqt5

```

(continues on next page)

(continued from previous page)

```

exclude: []

runtime:
  env:
    PATH: '${APPDIR}/usr/bin:${PATH}'
    # Set python home
    # See https://docs.python.org/3/using/cmdline.html#envvar-PYTHONHOME
    PYTHONHOME: '${APPDIR}/usr'
    # Path to the site-packages dir or other modules dirs
    # See https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH
    PYTHONPATH: '${APPDIR}/usr/lib/python3.6/site-packages'

  test:
    fedora:
      image: appimagecrafters/tests-env:fedora-30
      command: ./AppRun
      use_host_x: true
    debian:
      image: appimagecrafters/tests-env:debian-stable
      command: ./AppRun
      use_host_x: true
    arch:
      image: appimagecrafters/tests-env:archlinux-latest
      command: ./AppRun
      use_host_x: true
    centos:
      image: appimagecrafters/tests-env:centos-7
      command: ./AppRun
      use_host_x: true
    ubuntu:
      image: appimagecrafters/tests-env:ubuntu-xenial
      command: ./AppRun
      use_host_x: true

AppImage:
  update-information: 'gh-releases-zsync|AppImageCrafters|python-appimage-
↪example|latest|python-appimage-*x86_64.AppImage.zsync'
  sign-key: None
  arch: x86_64

```

2.10 Path Mapping (GIMP)

In this recipe you will learn how to map file paths in order to work-around fixed paths in compiled binaries. In the process we will create an AppImage for the GNU Image Manipulation Program.

For this example we will use the binaries from the Gimp deb package, but if you can also build your own binaries from source.

2.10.1 Annotations

Gimp is a huge and complex project. It needs python, gtk, BABL, GEGL, fontconfig and many other components therefore we must properly setup everything for behaving well in a portable environment. This setup is made using the environment variables that those pieces of software use in the `runtime > env` section.

- PYTHON: requires PYTHONPATH
- GTK: requires GTK_PATH , GTK_EXE_PREFIX, GTK_DATA_PREFIX
- GDK-Pixbuf: requires the loaders path to be present in the LIBRARY_PATH
- GIMP: requires BABL_PATH, GEGL_PATH, GIMP2_LOCALEDIR

Besides the above mentioned configuration we still need to make Gimp able to find it's configuration and data files. Those files are usually installed to `/etc/gimp/2.0/` and `/usr/share/gimp/2.0` but those paths are hardcoded on build. To make them available we will use the `runtime > path_mappings` feature as follows:

```
runtime:
  path_mappings:
    - /etc/gimp/2.0/:$APPDIR/etc/gimp/2.0/
    - /usr/share/gimp/2.0/:$APPDIR/usr/share/gimp/2.0/
```

At runtime the Gimp binary will be received to access `$APPDIR/etc/gimp/2.0/` instead of `/etc/gimp/2.0/` and `/usr/share/gimp/2.0/` instead of `$APPDIR/usr/share/gimp/2.0/`.

2.10.2 Recipe

```
version: 1

AppDir:
  path: ./AppDir

app_info:
  id: gimp
  name: GNU Image Manipulation Program
  icon: gimp
  version: latest
  exec: usr/bin/gimp

apt:
  arch: amd64
  sources:
    - sourceline: 'deb [arch=amd64] http://archive.ubuntu.com/ubuntu/ bionic main_
↳restricted universe multiverse'
    - sourceline: 'deb [arch=amd64] http://archive.ubuntu.com/ubuntu/ bionic-
↳updates main restricted universe multiverse'
      key_url: 'http://keyserver.ubuntu.com/pks/lookup?op=get&
↳search=0x3b4fe6acc0b21f32'

include:
  - gimp
  - libgtk2.0-0
  - librsvg2-common
  - libjson-glib-1.0-0
  - liblcms2-2
  - libgexiv2-2
  - python2.7
  - libgirepository-1.0-1
  - librsvg2-2
  - librsvg2-bin
  - graphviz
  - libamd2
```

(continues on next page)

(continued from previous page)

```

- libbtf1
- libcamd2
- libccolamd2
- libcholmod3
- libcolamd2
- libcxspase3
- libgexiv2-2
- libgirepository-1.0-1
- libglib2.0-0
- libglul-mesa
- libgs9
- libjpeg8
- libjson-glib-1.0-0
- libklul
- liblapack3
- liblcms2-2
- libld12
- liblensfun1
- libpango-1.0-0
- libpangocairo-1.0-0
- libpng16-16
- librbio2
- librsvg2-2
- libsdl1.2debian
- libspqr2
- libsuitesparseconfig5
- libtiff5
- libumfpack5
- libv4l-0
- libwebp6
- python-gobject-2
- libwebpmux3
- libwebpdemux2
- libpoppler-glib8
- libmng2
- libfreetype6
- libfontconfig1
- libblas3
- libpulse0

```

exclude:

```

- adwaita-icon-theme
- humanity-icon-theme

```

files:**exclude:**

```

- usr/lib/x86_64-linux-gnu/gconv
- usr/share/man
- usr/share/doc/*/README.*
- usr/share/doc/*/changelog.*
- usr/share/doc/*/NEWS.*
- usr/share/doc/*/TODO.*
- usr/include

```

runtime:**path_mappings:**

```

- /etc/gimp/2.0:$APPDIR/etc/gimp/2.0/
- /usr/share/gimp/2.0:$APPDIR/usr/share/gimp/2.0/

```

(continues on next page)

```

env:
  APPDIR_LIBRARY_PATH: '$APPDIR/usr/lib/x86_64-linux-gnu:$APPDIR/lib/x86_64-linux-
↳gnu:$APPDIR/usr/lib:$APPDIR/usr/lib/x86_64-linux-gnu/gdk-pixbuf-2.0/2.10.0/loaders'
  GTK_EXE_PREFIX: $APPDIR/usr
  GIMP2_LOCALEDIR: $APPDIR/usr/share/locale
  PYTHONPATH: $APPDIR/usr/lib/python2.7:$APPDIR/usr/lib/python2.7/site-packages:
↳$PYTHONPATH
  GTK_PATH: $APPDIR/lib/gtk-2.0
  GTK_DATA_PREFIX: $APPDIR
  XDG_DATA_DIRS: $APPDIR/share:$XDG_DATA_DIRS
  BABL_PATH: $APPDIR/usr/lib/x86_64-linux-gnu/babl-0.1
  GEGL_PATH: $APPDIR/usr/lib/x86_64-linux-gnu/gegl-0.4
test:
  debian:
    image: appimagecrafters/tests-env:debian-stable
    command: "./AppRun"
    use_host_x: True
  centos:
    image: appimagecrafters/tests-env:centos-7
    command: "./AppRun"
    use_host_x: True
  fedora:
    image: appimagecrafters/tests-env:fedora-30
    command: "./AppRun"
    use_host_x: True
  ubuntu:
    image: appimagecrafters/tests-env:ubuntu-xenial
    command: "./AppRun"
    use_host_x: True
AppImage:
  arch: x86_64

```

2.11 AppImage Updates

2.11.1 The AppImage Update process

An AppImage can be updated by just downloading the latest whole binary from the author or using delta updates. The second method is much more efficient as it only downloads the parts that changed, therefore, it's faster. Also the `appimageupdatetool` checks the file signature (if present) to ensure that the downloaded file is legit.

The `appimageupdatetool` uses the `zsync` method to do the delta update. Therefore a `.zsync` file is required for the updates to work. The file url is embed into the AppImage, this is known as the update information. There are several ways of specifying this url:

Update information

According to the [specification](#) an AppImage **MAY** have update information embedded for exactly one transport mechanism. Currently three transport mechanisms are available, but only one can be used for each given AppImage. Below we describe them in detail.

zsync

The zsync transport requires a HTTP server that can handle HTTP range requests. Its update information is in the form: `zsync|<zsync file URL>`, by example `zsync|https://server.domain/path/Application-latest-x86_64.AppImage.zsync`

GitHub Releases

The GitHub Releases transport extends the zsync transport in that it uses version information from [GitHub Releases](#). Its update information is in the form:

`gh-releases-zsync|<name space>|<project>|latest|<zsync file name>`, by example `gh-releases-zsync|probono|AppImages|latest|Subsurface-*x86_64.AppImage.zsync`

bintray-zsync

The `bintray-zsync` transport extends the `zsync` transport in that it uses version information from [Bintray](#). Its update information is in the form: `bintray-zsync|<username>|<repository>|<package name>|<zsync file path>`, by example `bintray-zsync|probono|AppImages|Subsurface|Subsurface-_latestVersion-x86_64.AppImage.zsync`

2.11.2 Setting AppImage update information

Before setting the update information make sure that `zsync` is installed in the build system. Then just add the update information line according to the selected method in `AppImage >> update-information` like this

```
AppImage:
  update-information: gh-releases-zsync|probono|AppImages|latest|Subsurface-*x86_64.
  ↪AppImage.zsync
  arch: aarch64
```

Once the build finish there will be a `.zsync` file next to the `AppImage` one. You should publish both of them according to the chosen update protocol.

2.12 AppImage Signing

AppImage can be signed using the Open PGP standard. This ensures that the AppImage comes from the person who pretends to be the author, and ensures that the file has not been tampered with.

2.12.1 Key Generation

To sign an AppImage file you will need a GPG key. Use `gpg2 --full-gen-key` to generate a new one. You can also check the [GnuPG documentation](#) to learn more about it.

2.12.2 Signing

To make appimage-builder use your key to sign the resulting AppImage is enough with specifying the key id in the AppImage >> signature section as follows:

```
AppImage:
  sign-key: 6907M4CCE10E0273853CDA121896X515CC81F0AD
```

NOTE: The private key must be available in the user keyring for GnuPG to find it. Use `gpg --import private.key` to do it.

2.12.3 Reading the signature

To check if the AppImage was properly signed execute it with the following option `--appimage-signature` it will print the signature, if any, to the standard output.

```
$ ./gnome-calculator-3.28.0-x86_64.AppImage --appimage-signature

-----BEGIN PGP SIGNATURE-----

iQIzBAABCgAdFiEEaafEzOEOAnOFPNoSGJb1FcyB8KwFAl6g2B0ACgkQGJb1FcyB
8KzP6g//Wncjb2HLLJ7U3muPb53py8Y5uwes7wE5w8Xbhy+ed42W6jp48cBl4O2G
cMaSJR8xH7yPvaLVWOfIDW6i713QUwtBfknLBdXrd1rhdMNzgXyQiKbwSgSfQcqi
kdaX2xFiXIYUV8e5BBZcfmKoFLNy4Lqfm2q8TICxiNiEdJ4eX5UTjfhWijmGg/pQ
yVNNNGGfhoBoT71DNUiJTeffwgwbDIwHHPiuXvfRyB/h8qfIMTYv0GSM3lWNGUkO
3lN4LRkdZM9t19ZLpvR3uXt5DW1V7i5Q2uIp46pEUGJPnnneO3wM+wzo8r0e9Pur
Nm/KEA9UG7Lfk6ktlq+e1r2pPWtaPwZCk3A//afPBymmGJACzvbN/XitB++hE5nT
RUyRDiFW7BWMx9mWbdXaLEfR1ZOAY5rR/QJA6bKC4IvPSvHWUwYmKJdQV+MTZ0JL
vCao5EtP6FgM0+Hm4dYoSRMK+9IpzIeg8uf8fgcaHa9lMZVryeJzmiR1vx5zu7Z
lDrDuMrqSyQ10wBBIKA7K8oJKOhrc+yNcCK8ldpYcDi4WVnvsb1ffKAKz8SdqI7/
RXwkbISmSkloDXkTRlZKw7Kwkj4spJzUESKsDwif9C4A3lGJw2xj4pAqH1LH1uq9
u07mp5HT1wPtfoBFSXqX3MVLszb6x4Qz1gzVXgWnhx5C5/K0L+8=
=Eruk
-----END PGP SIGNATURE-----
```

2.13 Testing

appimage-builder provides you a simple way of testing the AppImages compatibility with different systems. It uses docker containers to simulate the runtime environments and runs the applications inside.

2.13.1 appimage-builder AppImages

AppImages built using *appimage-builder* include almost every library and resource required by the bundled application to run. This allows to execute it in almost every GNU/Linux with `glibc` system. But there are some libraries that cannot be embed for technical reasons. The most relevant is `libGL` for NVidia, it's a requirement that the client side driver version to be equal to the kernel side. Therefore the graphic stack libraries and others related to then are excluded.

This leads us with a bundle that at runtime uses some libraries from the system and others from the bundle. If the ABI or the implementations of those mixed libraries are not compatible the application will crash. Luckily the libraries developers are careful enough to keep a good backward compatibility and the application works most of the times. But the only way of being 100% sure is by testing.

There are other resources from the system that our applications uses. An incompatibility with them may also lead to a crash. Here is a non-extensive list of those:

- icon themes
- fonts
- widgets themes (GTK/QT)
- ALSA / PulseAudio server
- X.Org server
- Wayland server

2.13.2 Tests in Docker container

appimage-builder provides a way of easily configuring a set of test using Docker containers to make sure your bundle will work on a given system. Therefore you will need to have a working docker image with the system resources listed above to make it work.

There is a list of pre-built docker images that you can use for your tests including the following GNU/Linux distributions:

- Arch `appimagecrafters/tests-env:archlinux-latest`
- Fedora `appimagecrafters/tests-env:fedora-30`
- Debian `appimagecrafters/tests-env:debian-stable`
- Ubuntu `appimagecrafters/tests-env:ubuntu-bionic`
- CentOS `appimagecrafters/tests-env:centos-7`

Those distributions are between the most populars or are base for others so if your app work there it has a high provability to work on derivatives.

The whole docker images list can be found: <https://hub.docker.com/repository/docker/appimagecrafters/tests-env>

For details on how to setup the tests cases check the *test* specification.

2.13.3 Recipe tests setup

Tests cases can be described in the recipe file. Those are placed inside the `AppDir >> test` section. Bellow you will find an example of a test case for Debian. To know more about this section check the *test* specification:

```
AppDir:
  test:
    debian:
      image: appimagecrafters/tests-env:debian-stable
      command: "./AppRun"
      use_host_x: True
      env:
        - QT_DEBUG_PLUGINS=1
```

2.13.4 Manual test running

Any AppImage/AppDir can be also manually tested using the `appimage-tester` command. This is part of *appimage-builder* since v0.5.2.

```
usage: appimage-tester [-h] [--log LOGLEVEL] --docker-images DOCKER_IMAGE
                        [DOCKER_IMAGE ...] [--test] [--static-test]
                        target
```

NOTE: Type 1 AppImages need to be extracted or mounted manually before running the tests.

Regular Docker tests

A regular test will try to run the target application inside the specified docker containers. A running X11 server is required if the app has a GUI.

```
appimage-tester --test ~/MyApp-1.8.4.AppImage --docker-images 'appimagecrafters/
↳tests-env:debian-stable'
```

Static Docker tests

Static test will lockup the external dependencies of the given target and will check if all of them are present in the system contained in the docker image. This does not execute the application.

```
appimage-tester --static-test ~/MyApp-1.8.4.AppImage --docker-images
↳'appimagecrafters/tests-env:debian-stable'
```

NOTE: Optional plugins can have runtime dependencies that may not be present in the test system but as they are optional the app will run properly.

2.14 Troubleshooting

Resulting AppImage can be defective for several reasons here we will explore then and explain the possible solutions.

2.14.1 Bundle information

The first thing to check when Appimage is created is the `.bundle.yml` file. It's located in the AppDir root. This file contain a resume of the packages included in the bundle and the libraries it expect to be present in the system. You can inspect then too look for missing packages or undesired external dependencies.

NOTE: This file is only generated for AppImages built using `appimage-builder >= v0.5.3`.

appimage-builder

`appimage-builder` may be used with two different log levels in its arguments. The default `--log` level is `INFO`, and the more informative `--log` level is `DEBUG`. Users may specify log arguments in the `appimage-builder` command using the `--log LOGLEVEL` argument where “LOGLEVEL” is either “INFO” or “DEBUG”. e.g. `appimage-builder --log DEBUG --generate`

```
usage: appimage-builder [-h] [--recipe RECIPE] [--log LOGLEVEL < INFO | DEBUG> ]
                        [--skip-script] [--skip-build] [--skip-tests]
                        [--skip-appimage] [--generate]
```

```
AppImage crafting tool
optional arguments:
```

(continues on next page)

(continued from previous page)

```

-h, --help          show this help message and exit
--recipe RECIPE    recipe file path (default: $PWD/AppImageBuilder.yml)
--log LOGLEVEL     logging level (default: INFO, debug: DEBUG, e.g. appimage-builder --
↳log DEBUG --generate)
--skip-script      Skip script execution
--skip-build       Skip AppDir building
--skip-tests       Skip AppDir testing
--skip-appimage    Skip AppImage generation
--generate         Try to generate recipe from an AppDir

```

appimage-inspector

appimage-inspector is a tool for inspecting AppImages and AppDirs. It's shipped along with appimage-builder since v0.5.3. This tool allow us to query information about a given target bundle.

```

usage: appimage-inspector [-h] [--log LOGLEVEL] [--print-needed]
                          [--print-runtime-needed]
                          [--print-dependants DO_PRINT_DEPENDANTS]
                          target

AppImage/AppDir analysis tool

positional arguments:
  target                AppImage or AppDir to be inspected

optional arguments:
  -h, --help            show this help message and exit
  --log LOGLEVEL        logging level (default: INFO)
  --print-needed        Print bundle needed libraries
  --print-runtime-needed
                        Print bundle needed libraries for the current system
  --print-dependants DO_PRINT_DEPENDANTS
                        Print bundle libraries that depends on

```

2.14.2 Issues

Non portable application

Many applications are coded to find their resources in fixed locations. Every time an AppImage runs it's mounted in a different location, something like /tmp/.mountXXXXXX where the exes are replaced by random alpha-numeric characters. This means that the app resource path will change every time it's ran.

Therefore app developers should make their apps configurable at runtime. This can be done by using environment variables or a configuration file next to the main binary.

Missing libraries

In some scenarios your application may crash on a certain system. This is usually happens because a required library is not being embed. To identify the culprit run your application using `LD_DEBUG=libs`. This will print to the standard output the information related to the shared libraries loading an unloading.

The output will look like this:

```
5491: find library=libpthread.so.0 [0]; searching
5491: search cache=/etc/ld.so.cache
5491: trying file=/lib/x86_64-linux-gnu/libpthread.so.0
5491:
5491:
5491: calling init: /lib/x86_64-linux-gnu/libpthread.so.0
```

In this case `libpthread.so.0` is found and initialized. As we will have a missing library we have to look for those output blocks where there is a `find library` with out a `init:`. To do it in a test inside docker use the following snippet:

```
test:
  debian:
    image: appimagecrafters/tests-env:fedora-30
    command: "./AppRun"
    use_host_x: True
    env:
      - LD_DEBUG=libs
```

More information about the glibc loader debug information can be found on the tool [manual pages](#).

To fix this issue just add to your bundle the package that provides this library.

Missing resources

To detect which resource files (settings files, icons, database files or others) are being used by the application we can use `strace`. Specifically you can trace `openat` calls like this:

```
$strace -e trace=openat ls

openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpcre.so.3", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/proc/filesystems", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, ".", O_RDONLY|O_NONBLOCK|O_CLOEXEC|O_DIRECTORY) = 3
appimage-appsdir AppImageServices builder builder-tests-env libappimage
↪ TheAppImageWay
appimage-firststrun apprun builder-docs cli-tool plasma-appimage-
↪ integration
```

Fixing this kind of issues is a bit more complicated as the path to the resources are sometime fixed in the source code. If it's possible you can try patching the binaries but the recommended solution is to modify the source code to resolve the resource files from a relative location. For this purpose you can use a configuration file next to the main binary or environment variables.

2.15 Full AppImage bundle

A full AppImage bundle is an AppImage that uses absolutely no libraries from the system. This allows to run it in weird GNU/Linux or even in FreeBSD. A full bundle includes software components that are considered present in every GNU/Linux distribution such as fonts config, libGL, libEGL and other related pieces of software.

2.15.1 Strong points & use cases

A full bundle is the best choice when we want to freeze a piece of software in time, as the only missing dependency will be the Linux Kernel. It maximizes the portability/compatibility of the bundle allowing it to run in non GNU/Linux system such as FreeBSD.

2.15.2 Draw backs

Adding more binaries to the bundle means that it will be bigger, usually about 30 Mb more. Also there is an issue with the NVidia drivers, they require the client and the kernel modules to have the same version. So if your software requires graphic acceleration and your target user may have an NVidia card making a full bundle may not be a good idea.

2.15.3 Instructions

To make a full bundle you have to explicitly include those packages that are excluded by default. The recipe below show how to create an full AppImage bundle for kcalc.

```

version: 1

AppDir:
  path: ./AppDir

  app_info:
    id: org.kde.kcalc
    name: kcalc
    icon: accessories-calculator
    version: 17.12.3
    exec: usr/bin/kcalc

  apt:
    arch: amd64
    sources:
      - sourceline: 'deb [arch=amd64] http://archive.ubuntu.com/ubuntu/ bionic main_
↳restricted universe multiverse'
        key_url: 'http://keyserver.ubuntu.com/pks/lookup?op=get&
↳search=0x3b4fe6acc0b21f32'
      - sourceline: 'deb [arch=amd64] http://archive.ubuntu.com/ubuntu/ bionic-
↳updates main restricted universe multiverse'
      - sourceline: 'deb [arch=amd64] http://archive.ubuntu.com/ubuntu/ bionic-
↳backports main restricted universe multiverse'

    include:
      - kcalc
      - libpulse0

      # Full bundle requirements
      - libx11-6
      - libgl1
      - libglapi-mesa
      - libdrm2
      - libegl1
      - libxcb-shape0
      - libxcb1
      - libx11-xcb1

```

(continues on next page)

```

- fontconfig-config
- libfontconfig1
- libfreetype6
- libglx0
- libxcb-xfixes0
- libxcb-render0
- libxcb-glx0
- libxcb-shm0
- libglvnd0
- libxcb-dri3-0
- libxcb-dri2-0
- libxcb-present0

exclude:
- phonon4qt5
- libkf5service-bin
- perl
- perl-base
- libpam-runtime

files:
exclude:
- usr/lib/x86_64-linux-gnu/gconv
- usr/share/man
- usr/share/doc/*/README.*
- usr/share/doc/*/changelog.*
- usr/share/doc/*/NEWS.*
- usr/share/doc/*/TODO.*

runtime:
env:
APPDIR_LIBRARY_PATH: $APPDIR/lib/x86_64-linux-gnu:$APPDIR/usr/lib/x86_64-linux-
↳gnu:$APPDIR/usr/lib/x86_64-linux-gnu/pulseaudio

test:
debian:
image: appimagecrafters/tests-env:debian-stable
command: "./AppRun"
use_host_x: True
centos:
image: appimagecrafters/tests-env:centos-7
command: "./AppRun"
use_host_x: True
arch:
image: appimagecrafters/tests-env:archlinux-latest
command: "./AppRun"
use_host_x: True
fedora:
image: appimagecrafters/tests-env:fedora-30
command: "./AppRun"
use_host_x: True
ubuntu:
image: appimagecrafters/tests-env:ubuntu-xenial
command: "./AppRun"
use_host_x: True

AppImage:

```

(continues on next page)

(continued from previous page)

```

update-information: None
sign-key: None
arch: x86_64

```

2.16 Producing AppImages on Gitlab CI

appimage-builder can be easily integrated into a gitlab-ci recipe. Here you will learn how to do it.

2.16.1 Docker Images

There is an appimage-builder docker image ready to be used on gitlab-ci you can find it [here](#). It's based on Ubuntu 18.04 and brings all the dependencies required by the tool. The docker image name is: appimagecrafters/appimage-builder.

2.16.2 Recipe

The usual approach while writing a gitlab-ci recipe for building AppImages is to use the `before_script` section to install the application dependencies. In the `script` section we will do the project configuration, binary building and the AppImage generation.

In the code snippet below you can find a complete `gitlab-ci.yml` recipe for building an AppImage for a [Hello World Qt](#) project using the latest Qt from the [KDE Neon repositories](#)

```

appimage-amd64:
  image: appimagecrafters/appimage-builder
  before_script:
    # update appimage-builder (optional)
    - apt-get update
    - apt-get install -y git wget
    - pip3 install --upgrade git+https://www.opencode.net/azubieta/appimagecraft.git

    # app build requirements
    - echo 'deb http://archive.neon.kde.org/user/ bionic main' > /etc/apt/sources.
↪list.d/neon.list
    - wget -qO - https://archive.neon.kde.org/public.key | apt-key add -
    - apt-get update
    - apt-get install -y qt5-default qtdeclarative5-dev cmake
  script:
    - cmake . -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=/usr
    - make install DESTDIR=AppDir
    - appimage-builder --skip-test
  artifacts:
    paths:
      - '*.AppImage*'
    expire_in: 1 week

```

2.16.3 OpenCode

Any Gitlab instance can be used to host your AppImage builds but it's recommended to do it on [opencode.net](#). This instance part of the OpenDesktop ecosystem were the [AppImageHub](#) project lives. This will allow to mark published

binaries as “OFFICIAL”.

2.17 Producing AppImages on GitHub

2.17.1 Build AppImage Action

To produce an AppImage on GitHub use the [build AppImage Action](#). This will run `appimage-builder` in an Ubuntu container and will output the AppImage file in the current working directory.

It takes for input the path to the `appimage-builder` recipe file and outputs the paths to the AppImage and the `zsync` file.

NOTE: Use the same sources lists for the recipe and the build system, otherwise resulting bundle may be faulty.

2.17.2 Update Information

AppImage Update support fetching updates from GitHub releases. To enable it se the following update information in your recipe file:

```
gh-releases-zsync|<user>|<project>|latest|*.AppImage.zsync`.
```

Replace `<user>` by the GitHub user or organization name and `project` buy the project name.

2.17.3 Workflow example

A complete example project can be found at: <https://github.com/AppImageCrafters/qt-hello-world>

```
name: C/C++ AppImage
on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]
jobs:
  build-appimage:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: install dependencies
        run: |
          sudo apt-get update
          sudo apt-get install -y qt5-default qtdeclarative5-dev cmake
      - name: configure
        run: cmake . -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=/usr
      - name: build
        run: make -j`nproc` install DESTDIR=AppDir
      - name: Build AppImage
        uses: AppImageCrafters/build-appimage-action@master
        env:
          UPDATE_INFO: gh-releases-zsync|AppImageCrafters|qt-hello-world|latest|*x86_
          64.AppImage.zsync
```

(continues on next page)

(continued from previous page)

```

with:
  recipe: AppImageBuilder.yml
- uses: actions/upload-artifact@v2
with:
  name: AppImage

```

2.18 Version: 1 (Draft)

In this section is described the recipe specification and all the components that affects its behaviour.

THIS DOCUMENT IS UNDER CONSTRUCTION AND MAY CHANGE UNTIL THE 1.0 RELEASE OF APPIMAGE-BUILDER

2.18.1 Environment variables

Environment variables can be placed anywhere in the configuration file; must have `!ENV` before them and specified in this format to be parsed: `${VAR_NAME}`.

```

AppDir:
  app_info:
    version: !ENV ${APP_VERSION}
    exec: !ENV 'lib/${GNU_ARCH_TRIPLET}/qt5/bin/qmlscene'
AppImage:
  arch: !ENV '${TARGET_ARCH}'
  file_name: !ENV 'myapp-${APP_VERSION}_${TIMESTAMP}-${ARCH}.AppImage'

```

NOTE: To mix variables that must be parsed with other that not use the following syntax: `!ENV '${PARSED_VAR}-${NON_PARSED_VAR}'`

2.18.2 script

The script section consists of a list of shell instructions. It should be used to deploy your application binaries and resources or other resources that cannot be resolved using the package manager.

Example of how to deploy a regular `cmake` application binaries.

```

script:
- cmake .
- make DESTDIR=Appdir install

```

In the cases where you don't use a build tool or it doesn't have an install feature, you can run any type of command in this section. In the example below a QML file is deployed to be used as part of a pure QML application.

```

script:
- mkdir -p AppDir
- cp -f main.qml AppDir/

```

2.18.3 AppDir

The *AppDir* section is the heart of the recipe. It will contain information about the software being packed, its dependencies, the runtime configuration, and the tests. See the example below corresponding to a pure QML application.

path

Path to the AppDir.

```
AppDir:  
  path: ./AppDir
```

app_info

- **id:** application id. Is a mandatory field and must match the application desktop entry name without the .desktop extensions. It's recommended to use reverse domain notation like *org.goodcoders.app*.
- **name:** Application name, feel free here.
- **icon:** Application icon. It will be used as the bundle icon. The icon will be copied from \$APPDIR/usr/share/icons or from your system folder /usr/share/icons.
- **version:** application version.
- **exec:** path to the application binary. In the case of interpreted programming languages such as Java, Python or QML, it should point to the interpreter binary.
- **exec_args:** arguments to be passed when starting the application. You can make use of environment variables such as \$APPDIR to refer to the bundle root and/or \$@ to pass arguments to the binary.

```
app_info:  
  id: org.appimagecrafters.hello_qml  
  name: Hello QML  
  icon: text-x-qml  
  version: 1.0  
  exec: usr/lib/qt5/bin/qmlscene  
  exec_args: $@ ${APPDIR}/main.qml
```

apt

The apt section is used to list the packages on which the app depends and the sources to fetch them.

- **arch:** Binaries architecture. It must match the one used in the sources configuration.
- **sources:** apt sources to be used to retrieve the packages.
 - **sourceline:** apt configuration source line. It's recommended to include the Debian architecture on it to speed up builds.
 - **key_url:** apt key to validate the packages in the repository. An URL to the actual key is expected.
- **include:** List of packages to be included in the bundle. Package dependencies will also be bundled.
- **exclude:** List of packages to *not* bundle. Use it to exclude packages that aren't required by the application.

```
apt:  
  arch: i386  
  sources:  
    - sourceline: 'deb [arch=i386] http://mx.archive.ubuntu.com/ubuntu/ bionic main_  
↳ restricted universe multiverse'  
      key_url: 'http://keyserver.ubuntu.com/pks/lookup?op=get&search=0x3b4fe6acc0b21f32  
↳ '
```

(continues on next page)

(continued from previous page)

```
include:
- qmlscene
- qml-module-qtquick2
exclude:
- qtchooser
```

The tool generates a cache where the downloaded packages and other auxiliary files are stored, it will be located in the current work dir with the name **appimage-builder-cache**. It's safe to erase it and should not be included in your VCS tree.

files

The files section is used to manipulate (include/exclude) files directly. [Globing expressions](#) can be used to match multiple files at once.

- **include:** List of absolute paths to files. The file will be copied under the same name inside the AppDir. i.e.: `/usr/bin/xrandr` will end at `$APPDIR/usr/bin/xrandr`.
- **exclude:** List of relative globing shell expressions to the files that will not be included in the *AppDir*. Expressions will be evaluated relative to the *AppDir*. Use it to exclude unrequired files such as *man* pages or development resources.

```
files:
exclude:
- usr/share/man
- usr/share/doc/*/README.*
- usr/share/doc/*/changelog.*
- usr/share/doc/*/NEWS.*
- usr/share/doc/*/TODO.*
```

test

The *test* section is used to describe test cases for your final AppImage. The AppDir as it's can be already executed. Therefore it can be placed inside a Docker container and executed. This section eases the process. Notice that you will have to test that the application is properly bundled and isolated, therefore it's recommended to use minimal Docker images (i.e.: with no desktop environment installed).

IMPORTANT: Docker is required to be installed and running to execute the tests.

Each test case has a name, which could be any alphanumeric string and the following parameters:

- **image:** Docker image to be used.
- **command:** command to execute.
- **use_host_x:** whether to share or not the host X11 session with the container. *This feature may not be supported by some containers as it depends on X11.*
- **env:** list of environment variables to be passed to the Docker container.

```
test:
fedora:
image: fedora:26
command: "./AppRun main.qml"
use_host_x: True
ubuntu:
```

(continues on next page)

(continued from previous page)

```
image: ubuntu:xenial
command: "./AppRun main.qml"
use_host_x: True
```

runtime

Advanced runtime configuration.

- **env**: Environment variables to be set at runtime.
- **path_mappings** Setup path mappings to workaround binaries containing fixed paths. The mapping is performed at runtime by intercepting every system call that contains a file path and patching it. Environment variables are supported as part of the file path.

Paths are specified as follows: <source>:<target>

Use the `$APPDIR` environment variable to specify paths relative to it.

```
runtime:
  path_mappings:
    - /etc/gimp/2.0/:$APPDIR/etc/gimp/2.0/
  env:
    PATH: '${APPDIR}/usr/bin:${PATH}'
```

2.18.4 AppImage

The AppImage section refers to the final bundle creation. It's basically a wrapper over `appimagetool`

- **arch**: AppImage runtime architecture. Usually, it should match the embed binaries architecture, but a different —compatible one— could be used. For example, i386 binaries can be used in an AMD64 architecture.
- **update-info**: AppImage update information. See [Making AppImages updateable](#).
- **sign-key**: The key to sign the AppImage. See [Signing AppImage](#).
- **file_name**: Use it to rename your final AppImage. By default it will be named as follows: `${AppDir.app_info.name}-${AppDir.app_info.version}-${AppImage.arch}.AppImage`. Variables are not supported yet and are used only for illustrative purposes.