AppImageKit Documentation

Describes the AppImage file format and AppImageKit tools

Version 1.0

Table of Contents

The AppImage format is a standardized format for packaging applications in a way that allows them to run on target systems without further modification. This document describes the AppImage format.

This document is not a formal specification, since the AppImage format is not frozen yet but in the process of being specified more formally. However, this document is intended to describe the philosophy behing the AppImage format and the concrete implementation.

The AppImageKit contains a concrete implementation of the AppImage format and provides tools for conveniently handling AppImages.

Contributors are encouraged to comment on this document and propose formal format descriptions.

Motivation

Historically, UNIX and Linux systems have made it easy to procure source code, however they have made it comparably difficult to use ready-made software in binary form. Especially the Free Software and GNU movements stress the fact that everyone should be free to get the source code. This mode of operation has worked well as long as the user base of these operating systems was largely comprised of technically advanced users. With the widespread adoption of easy-to-use desktop operating systems such as Ubuntu, the user base became less technically-minded and more application-centric.

Package mangers were introduced to mitigate the complexities of dealing with source code by providing libraries of precompiled packages from repositories maintained by distributors or third parties. However, the introduction of package mangers did not drastically reduce compexities or provide robustness - they merely stacked a management layer on top of an already complex system, effectively preventing the user from manipulating installed software directly.

Other systems, most prominently Windows and the legacy Macintosh operating system, have made it relatively simple for independent software publishers (ISPs) to distribute software and for end-users to procure and install software from said ISPs, without any instances (such as distributors or AppStores) between the two parties.

With the introduction of Mac OS X, arguably the first UNIX-based operating system with widespread mass adoption to a non-technical user base, Apple blended traditional UNIX aspects (such as maintaining a traditional filesystem hierarchy, including /bin, /usr, /lib directories) with common "desktop" approaches (such as "installing" an application by dragging it to the hard disk). While Apple uses a package manager-

like approach for managing the base operating system and its updates, it does not do so for the user applications.

Open Source operating systems, such as the most prominent Linux distributions, mostly use package mangers for everything. While this is perceived superior to Windows and the Mac by many Linux enthusiasts, it also creates a number of disadvantages:

1. **Centralization** Some organization decides what is "in" a distribution and what is not. By definition, software "in" a distribution is easier to install and manage that software that is not.

2. **Duplication of effort** In traditional systems, each application is compiled specifically for each target operating system. This means that one piece of software has to be compiled many, many times on many, many systems using much, much power and time

3. **Need to be online** Most package managers are created with connected computers in the mind, making it really cumbersome to "just fetch an app" on an online system, and copy it over to another system that is not connected to the Internet.

A critical distinction between the approach known from Windows and the Mac and the one known from UNIX and Linux is the "platform": While Windows and the Mac are seen as platforms to run software on, most Linux distributions see themselves as the system that includes the applications.

While this leads to a number of advantages that have been frequently reiterated, it also poses a significant number of challenges:

1. **No recent apps on mature operating systems** In most distributions, you get only the version that was recent at the time when the distribution was created. For example, if you use Ubuntu Gutsy then you are stuck forever with the software that was recent at the time when Ubuntu Gutsy was compiled. Even if Firefox might have progressed by several versions in the meantime, you cannot get more recent apps than what was available back when the distribution was put together. That is like if you'd get only software from 2001 when you use Windows XP.

   In the traditional model, the user has to decide: Either use a mature base operating system but be locked out of recent apps (e.g., using Ubuntu LTE), or be forced to update the base operating system to the latest bleeding edge version in order to get the recent apps (e.g., using Debian Sid).

   This situation is clearly not optimal, since the common desktop user would prefer to hardly touch the base operating system (maybe update it every other year or so) but always get the latest apps.

2. **No way to use multiple versions in parallel** Most package managers do not allow you to have more than one version of an app installed in parallel. Hence you have no way to simply try out the latest version of an app without running the risk that it might not be easy to switch back to the older version, especially if the older version is no longer available in your distribution (e.g, old versions get removed from Debian Sid as soon as a newer version appears). This is especially annoying if you would simply like to try out a few things before you decide whether to use the old or the new version.

3. **Not easy to move an app from one machine to another** If you've used an app on one machine and decide that you would like to use the same app either under a different base operating system (say, you want to use OpenOffice on Fedora after having used it on Ubuntu) or if you would simply take the app from one machine to another (say from the desktop computer to the netbook), you have to download and install the app again (if you did not keep around the installation files and if the two operating systems don't share the exact same package format - both of which is rather unlikely).

The AppImage format has been created with specific objectives in mind.

1. **Be Simple** AppImage is intended to be a very simple format that is easy to understand, create, and manage.

2. **Maintain binary compatibility** AppImage is a format for binary software distribution. Software packaged as AppImage is intended to be as binary-compatible as possible with as many systems as possible. The need for (re-)compilation of software should be greatly reduced.

3. **Be distribution-agostic** An AppImage should run on all base operating systems (distributions) that it was created for (and later versions). For example, you could target Ubuntu 9.10, openSUSE 11.2, and Fedora 13 (and later versoins) at the same time, without having to create and maintain separate packages for each target system.

4. **Remove the need for installation** AppImages contain the app in a format that allows it to run directly from the archive, without having to be installed first. This is comparable to a Live CD. Before Live CDs, operating systems had to be installed first before they could be used.

5. **Keep apps compressed all the time** Since the application remains packaged all the time, it is never uncompressed on the hard disk. The computer uncompresses the application on-the-fly while accessing it. Since decompression is faster than reading from hard disk on most systems, this has a speed advantage in addition to saving space. Also, the time needed for installation is entirely removed.

6. **Allow to put apps anywhere** AppImages are "relocateable", this allowing the user to store and exectute them from any location (including CD-ROMs, DVDs, removable disks, USB sticks).

7. **Make applications read-only** Since AppImages are read-only by design, the user can be reasonably sure that an app does not modify itself during operation.

8. **Do not require recompilation** AppImages must be possible to create from already-existing binaries, without the need for recompilation. This greatly speeds up the AppImage creation process, since no compiler is usually involved. It also allows third parties to package closed-source applications as AppImages.

9. **Keep base operating system untouched** Since AppImages are intended to run on plain systems that have not been specially prepared by an administrator, AppImages may not require any unusual preparation of the base operating system. Hence, they cannot rely on special kernel patches, kernel modules, or any applications that do not come with the targeted distributions by default.

10. **Do not require root** Since AppImages are intended to be run by end users, they should not reqiure an administrative account (root) to be installed or used. They may, however, be installed by an administrator (e.g., in multi-user scenarios) if so desired.

The key idea of the AppImage format is "one app = one file". Every AppImage contains an app and all the files the app needs to run. In other words, each AppImage has no dependencies other what is included in the targeted base operating system(s). While it would theoretically be possible to create rpm or deb packages in the same way, it is hardly ever done. In contrast, doing so is strongly encouraged when dealing with AppImages and is the default use case of the AppImage format.

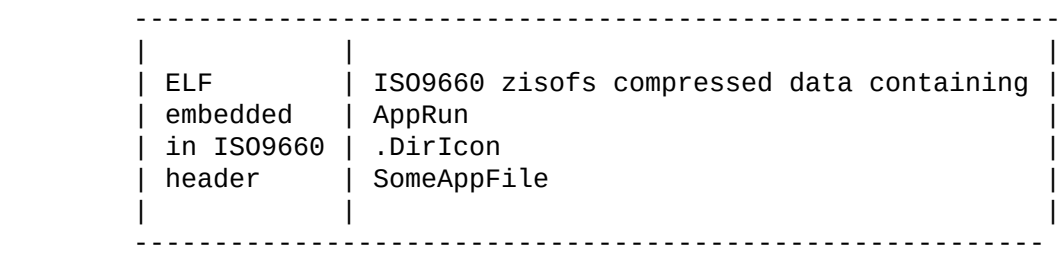
In short: An AppImage is for an app what a Live CD is for an operating system.

Format overview

An AppImage is an ISO 9660 file with zisofs compression containing a minimal AppDir (a directory that contains the app and all the files that it requires to run which are not part of the targeted base operating systems) and a tiny runtime executable embedded into its header. Hence, an AppImage is both an ISO 9660 file (that you can mount and examine) and an ELF executable (that you can execute).

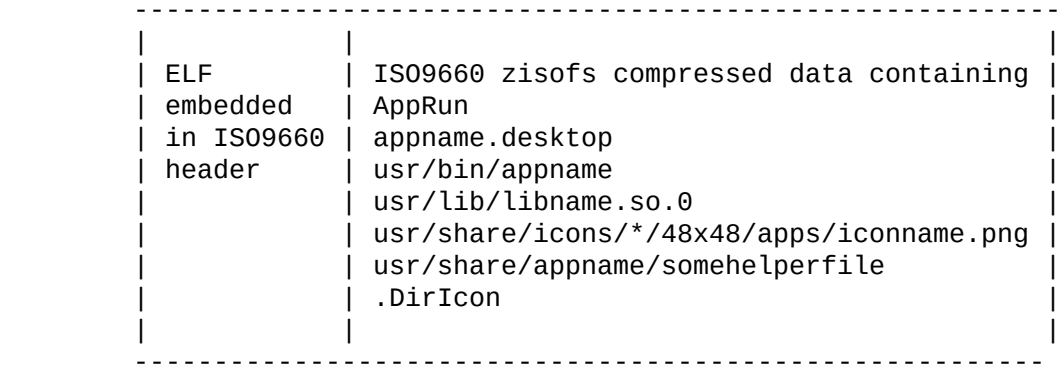
When you execute an AppImage, the tiny embedded runtime mounts the ISO file, and executes the app contained therein.

A minimal AppImage could potentially look like this:



1. `AppRun` is the binary hat is executed when the AppImage is run
2. `.DirIcon` contains a 48x48 pixel PNG icon that is used for the AppImage
3. `SomeAppFile` could be some random file that the app reqires to run

However, in order to allow for automated generation, processing, and richer metadata, the AppImage format follows a somewhat more elaborate convention:



1. The ELF embedded in the ISO9660 header always executes the file called `AppRun` inside the ISO9660 file.
2. The file `AppRun` inside the ISO9660 file is not the actual executable, but instead a tiny helper binary that finds and exectues the actual app. Generic AppRun files have been implemented in bash and C as parts of AppImageKit. The C version is generally preferred as it is faster and more portable.
3. `AppRun` usually does not contain hardcoded information about the app, but instead retrieves it from the file `appname.desktop` that follows the Desktop File Specification.

   A minimal `appname.desktop` file that would be sufficient for AppImage would need to contain

   ```
   [Desktop Entry]
   Name=AppName
   Exec=appname
   Icon=iconname
   ```

   This desktop file would tell the AppRun executable to run the executable called `appname`, and would specify `AppName` as the name for the AppImage, and `iconname.png` as its icon.

   However, it does not hurt if the desktop file contains additional information. Should it be desired to provide additional metadata with an AppImage, the desktop file could be extended with `X-AppImage-...` fields as per the Desktop File Specification. Usually, desktop files provided in deb

or rpm archives are suitable to be used in AppImages. However, abolute paths in the Exec statement are not supported by the AppImage format.

4. The AppImage contains the usual `usr/` hierarchy (following the File Hierarchy Standard). In the concrete example from the desktop file above, the AppRun executable would look for `usr/bin/appname` and would execute that. Also, the **AppImageKitAssistant** (a tool used to create AppImages easily) would look for `usr/share/icons/*/48x48/iconname.png` and use that as the `.DirIcon` file, effectively making it the icon of the AppImage.

5. The app must be programmed in a way that allows for relocation. In other words, the app must not have hardcoded paths such as `/usr/bin`, `/usr/share`, `/etc` inside the binary. Instead, it must use relative paths, such as ./bin.

   Since most binaries contained in deb and rpm archives generally are not created in a way that allows for relocation, they need to be either changed and recompiled (e.g., using the binreloc framework), or the binaries need to be patched. As recompiling is not convenient in most cases, AppRun changes to the `usr/` directory prior to executing the app, enabling the app to specify all paths relative to the AppImage's `usr/` directory. This allows one to use patched binaries (where the string `/usr` has been replaced with the same-length string `././`, which means "current directory"). AppImageKit comes with a tool that does this automatically. Note that if you use the `././` patch, then your app must not use **chdir**, or otherwise it will break.

6. The ELF embedded in the ISO9660 header contains an icon embedded into the ELF executable following the elficon specification. AppImageKitAssistant would automatically embed the specified icon.

Note that the AppImage format has been conceived to facilitate the conversion of deb and rpm packages into the AppImage format with minimal manual effort. Hence, it contains some conventions in addition to those specified by the AppDir format, to which it is compatible to the extent that an unpacked AppImage can be used as an AppDir with the ROX Filer.

The AppImageKit

The AppImage format is complemented by a suite of tools called AppImageKit that provide a concrete sample implementation of the ideas expressed in the format, and that can greatly simplify dealing with AppImages.

Currently the AppImageKit contains (among others)

- **create-appdir**, a command line tool running on Ubuntu that turns packaged software into AppDirs. This tool can be used to semi-automatically prepare AppDirs that can be used as the input for AppImageAssistant. Note that while create-appdir has been written for Ubuntu, it should also run on debian and could be ported to other distributions as well, then using the respective package managers insted of apt-get.

- **AppImageAssistant**, a GUI app that turns an AppDir into an AppImage.

- **AppRun**, the executable that finds and executes the app contained in the AppImage. create-appdir automatically embeds AppRun into the AppDirs it creates.

- **runtime**, the tiny ELF binary that is embedded into the header of each AppImage. AppImageKit automatically embeds the runtime into the AppImages it creates.

AppImageKit also contains additoinal tools and helpers.

Creating AppImages

The general workflow for creating an AppImage involves the following steps:

1. Gather suitable binaries If the application has already been compiled, you can use existing binaries (for example, contained in .tar.gz, deb, or rpm archives). Note that the binaries must not be compiled on newer distributions than the ones you are targeting. In other words, if you are targeting Ubuntu 9.10, you should not use binaries compiled on Ubuntu 10.04.

2. Gather suitable binaries of all dependencies that are not part of the base operating systems you are targeting. For example, if you are targeting Ubuntu, Fedora, and openSUSE, then you need to gather all libraries and other dependencies that your app requires to run that are not part of Ubuntu, Fedora, and openSUSE.

3. Create a working AppDir from your binaries. A working AppImage runs your app when you execute its `AppRun` file.

4. Turn your AppDir into an AppImage. This compresses the contents of your AppDir into a single, self-mounting and self-executable file.

5. Test your AppImage on all base operating systems you are targeting. This is an important step which you should **not** skip. Subtle differences in distributions make this a must. While it is possible in most cases to create AppImages that run on various distributions, this does not come automatically, but requires careful hand-tuning.

While it would theoretically be possible to do all these steps by hand, AppImageKit contains tools that greatly simplify the tasks.

Most of the time, the software you would like to package as an AppImage has already been packaged for one of the commonly used distributions. At the time of this writing, most software was available for Ubuntu (including software in personal package archives, PPAs) and other repositories. If your app is already available there (and chances are), then it is especially straightforward to create an AppImage using the tools contained in the AppImageKit.

1. Run the version of Ubuntu that you are targeting, or an older version. Do not run a newer version of Ubuntu than the one you are targeting, or your AppImage will likely not run on the targeted version. It is recommended to run from the official Ubuntu Live CD (or Live USB) system, since this is guaranteed to be in a known default state.

2. If your software is available in a PPA or other third-party repository, add the repository to the system (by editing `/etc/apt/sources.list` or by using **add-apt-repository**).

3. Run **sudo apt-get update** to refresh the package information.

4. Run **create-appimage**. For example, if you would like to create an AppImage of **gnubik**, run **`create-appimage gnubik`**. This is similar to running **`apt-get install gnubik`**; however, instead of installing the app to your system it creates an AppImage of your app. Note that the AppImage contains `gnubik.desktop` and `AppRun`.

   ### Note

   create-appimage automatically fetches dependencies that are not part of your current operating system installation. This is possibly not enough to satisfy all dependencies that are not part of all the distributions that you are targeting. Hence, you might have to copy some additional dependencies into your AppDir manually, until it runs on all the distributions that you are targeting.

Note

create-appimage automatically patches the binaries in your AppImage so that the absolute path `/usr` is replaced with the relative path `././` (which simply means "here"), effectively making the binaries relocateable. While this is sufficient for most C/C++ apps, it might not be sufficient for apps using some higher-level frameworks such as Perl, Python, C#, etc. In this case, you have to adjust your app manually until it runs from any location ("relocateable").

5. Double-click `AppRun`. The gnubik app should run. The `gnubik.desktop` file tells `AppRun` which executable should be executed. You could also try coping your AppDir to the other systems targeted, like Fedora and openSUSE, and check whether your app runs there as well. However, this step will be done (with great simplification) later on.

Note

If your app does not run when you double-click the `AppRun` file, then you need to fix this first before you can progress to making an AppImage out of the AppDir. The **strace** command can be helpful in determining files that cannot be found by the app (e.g., **`strace -eopen -f ./AppRun 2>&1 | grep ENOENT`** gives you all the files that are tried to be opened but not found).

6. Run **AppImageAssistant** and select the `gnubik.AppDir/` that you just created. AppImageAssistant will guide you through the process of creating the AppImage. In the process, it checks whether your `gnubik.desktop` file has all the necessary information, whether the icon is available, and whether the executable specified in the `gnubik.desktop` file can be found in the `gnubik.AppDir/usr/bin` directory.

Note

If the icon is not found, make sure that you have a 48x48 pixel png file called gnubik.png either in `gnubik.AppDir/` or in the appropriate `gnubik.AppDir/usr/share/icons/...` subdirectory (as per the Desktop File Specification).

7. At the end of the AppImageAssistant, you will be prompted to run your AppImage in various test environments, such as the targeted versions of Ubuntu, Fedora, and openSUSE. AppImageAssistant runs your AppImage in a special chroot environment based on each distribution's Live CD image.

   Be sure not to release an AppImage that you have not tested on each of the distributions you are targeting. While it is reasonably safe to assume that later versions of the distributions you have tested will still run your AppImage (at least if they are carefully designed), it is not safe to assume that other distributions will do so, too.

Note

For this to work, you need to prepare squashfs images of the targeted base operating systems. Instructions on how to do this can be found at http://portablelinuxapps.org/forum.

8. Optionally, if you would like to distribute your AppImage to your users, rename it to "AppName Version", e.g., "GNUbik 1.0" and put it up for download. Please do **not** put the AppImage inside a compressed archive, since it is already compressed by itself.

9. Optionally, if you would like to get comments on your AppImage and/or suggest it for inclusion on PortableLinuxApps, a website dedicated to AppImages that run on Ubuntu, Fedora, and openSUSE, then create a posting in the "Labs" area of http://portablelinuxapps.org/forum

If your software has not yet been packaged for Ubuntu (or if you simply like doing things "from scratch"), then you can create your AppDir manually. In that case, replace steps 1-4 above with:

- Create an AppDir structure that looks (as a minimum) like this:

  ```
  MyApp.AppDir/
  MyApp.AppDir/AppRun
  MyApp.AppDir/myapp.desktop
  MyApp.AppDir/myapp.png
  MyApp.AppDir/usr/bin/myapp
  MyApp.AppDir/usr/lib/libfoo.so.0
  ```

  Of course you can leave out the library if your app does not need one, or if all libraries your app needs are already contained in every base operating system you are targeting.

- Your binary, `myapp`, must not contain any hardcoded paths that would prevent it from being relocateable. You can check this by running **`strings MyApp.AppDir/usr/bin/myapp | grep /usr`**. Should this return something, then you need to modify your app programmatically (e.g., by using relative path or by using binreloc). If you prefer not to change the source code of your app and/or would not like to recompile your app, you can also patch the binary, for example using the command **`sed -i -e 's|/usr|./././|g' MyApp.AppDir/usr/bin/myapp`**.

  Note

  The same is true for any helper binaries and/or libraries that your app depends on. You can do so with **`cd MyApp.AppDir/usr/ ; find -type f . -exec sed -i -e 's|/usr|././|g' {} \; ; cd -`** which replaces all occurences of `/usr` with `././` which simply means "here".

- myapp.desktop should contain (as a minimum):

  ```
  [Desktop Entry]
  Name=MyApp
  Exec=myapp
  Icon=myapp
  ```

- Then, proceed with step 5 from above.

Contributing

You are invited to contribute to the AppImage format, the AppImageKit tools, and the PortableLinuxApps website (which is a showcase of AppImage).

The preferred channel of communication is http://portablelinuxapps.org/forum - please make sure to register, as registered users have access to the developer section of the forum.

Acknowledgements

This work stands on the shoulders of giants. The following persons and organizations should specifically be thanked (in no particular order), even though this list can never be exhaustive:

- Apple Inc. for popularizing the notion of application bundles (even tough others have used this concept before). AppImages would not be understood by people as easily if it wasn't for Apple's .app bundles and .dmg disk images.

- The contributors of the ROX project, which introduced the AppDir format that the AppImage format is conceptually based on. AppImages improve on ROX AppDirs in that they encapsulate the AppDirs in a compressed container file which adds robustness and ease of administration.

- The contributors of the klik project, which AppImageKit is conceptually based on (with the principal author of AppImageKit being the founder of the klik project). AppImages imporve on klik in that they need no runtime to be installed on the base operating system before they can be used.
- Alexander Larsson for his work on Glick, which AppImageKit is technically based on. AppImageKit improves on Glick in that it uses a compressed filesystem and in that it provides additional tools which simpify creating AppImages.
- The contributors of the Python project, which gives developers a powerful tool to turn ideas into reality rapidly. AppImageKit would have been much more cumbersome to create if Python would not exist.