Application layout

Contents

Application bundles in the Apertis system may require several categories of

storage, and to be able to write correct AppArmor profiles, we need to be able to restrict each of those categories of storage to a known directory.

This document is intended to update and partially supersede discussions of storage locations in theapplications[1] andsystem updates and rollback[2] design documents.

The Apertis Application Bundle Specification[3] describes the files that can appear in an application bundle and are expected to remain supported long-term. This document provides rationale for those categories of files, suggested future directions, and details of functionality that is not necessarily long-term stable.

Requirements

Static files

- Most application bundles will contain one or more executable programs[4], in the form of either compiled machine code or scripts. These are read-only and executable, and are updated when the bundle is updated (and at no other time).
  - Some of these programs are designed to be run directly by a user. These are traditionally installed in `/usr/bin` on Unix systems. Other programs are *supporting programs*, designed to be run internally by programs or libraries. These are traditionally installed in `/usr/libexec` (or sometimes `/usr/lib`) on Unix systems. Apertis does not require a technical distinction between these categories of program, but it would be convenient for them to be installed in a layout similar to the traditional one.
- Application bundles that contain compiled executables may contain *private shared libraries*, in addition to those provided by the platform[5], to support the executable. These are read-only ELF shared libraries, and are updated when the bundle is updated.
  - For example, Frampton[6] has a private shared library libframptonagentiface[7] containing GDBus interfaces.
- Application bundles may contain dynamically-loaded *plugins* (also known as loadable modules). These are also read-only ELF shared libraries.
- Application bundles may contain static *resource files* such as `.gresource` resource bundles, icons, fonts, or sample content. This are read-only, and

[1] https://sjoerd.pages.apertis.org/apertis-website/concepts/applications/
[2] https://sjoerd.pages.apertis.org/apertis-website/designs/system-updates-and-rollback/
[3] https://appdev.apertis.org/documentation/bundle-spec.html
[4] https://sjoerd.pages.apertis.org/apertis-website/glossary/#program
[5] https://sjoerd.pages.apertis.org/apertis-website/glossary/#platform
[6] https://gitlab.apertis.org/appfw/frampton
[7] https://gitlab.apertis.org/appfw/frampton/tree/master/src/interface

are updated when the bundle is updated.

  - Where possible, application bundles should embed resources in the executable or library using GResource[8]. However, there are some situations in which this might not be possible, which will result in storing resource files in the filesystem:
    * if the application will load the resource via an API that is not compatible with GResource, but requires a real file
    * if the resource is extremely large
    * if the resource will be read by other programs, such as the icon that will be used by the app-launcher, the .desktop file describing an entry point (used by Canterbury, Didcot etc.), or D-Bus service files (used by dbus-daemon)
  - If a separate `.gresource` file is used, for example for programs written in JavaScript or Python, then that file needs to be stored somewhere.

- The AppArmor profile for an application bundle must allow that application bundle to read, mmap and execute its own static files.

- The AppArmor profile for an application bundle must not allow that application bundle to *write* its own static files, because they are meant to be static. In particular, the AppArmor profile itself must not be modifiable.

Variable files

- The programs in application bundles may save variable data (configuration, state and/or cached files) for each user[9] (Applications design - Data Storage[10]).
  - *Configuration* is any setting or preference for which there is a reasonable default value. If configuration is deleted, the expected result is that the user is annoyed by the preference being reset, but nothing important has been lost.
  - *Cached files* are files that have a canonical version stored elsewhere, and so can be deleted at any time without any effect, other than performance, resource usage, or limited functionality in the absence of an Internet connection. For example, a client for "tile map" services like Google Maps or OpenStreetMap should store map tiles in its cache directory. If cached files are deleted, the expected result is that the system is slower or less featureful until an automated process can refill the cache.
  - Non-configuration, non-cache data includes documents written by the user, database-like content such as a contact list or address book, license keys, and other unrecoverable data. It is usually considered valuable to the user and should not be deleted, except on the

[8] https://developer.gnome.org/gio/stable/GResource.html
[9] https://sjoerd.pages.apertis.org/apertis-website/glossary/#user
[10] https://sjoerd.pages.apertis.org/apertis-website/concepts/applications/#data-storage

user's request. If non-configuration, non-cache data is unintentionally deleted, the expected result is that the user will try to restore it from a backup.

- The programs in application bundles may save variable data (configuration, state and/or cached files) that are shared between all users[11] (Applications design - Data storage[12]).
- Newport needs to be able to write downloaded files[13] to a designated directory owned by the application bundle.
  - Because Newport is a platform service, its AppArmor profile will need to be allowed to write to *all* apps' directories.
  - Because downloads might contain private information, Newport must download to a user- and bundle-specific location.
- The AppArmor profile for an application bundle must allow that application bundle to read and write its own variable files.
- The AppArmor profile for an application bundle should not allow that application bundle to execute its own variable files ("write xor execute"), making a broad class of arbitrary-code-execution vulnerabilities considerably more difficult to exploit.
- Large media files such as music and videos should normally be shared between all users[14] and all multimedia application bundles. (Multi-user design - Requirements[15])

Upgrade, rollback, reset and uninstall

Store applications

Suppose we have a store application bundle[16], Shopping List version 23, which stores each user's grocery list in a flat file. A new version 24 becomes available; this version stores each user's grocery list in a SQLite database.

- Shopping List can be installed and upgraded. This must be relatively rapid.
- Before upgrade from version 23 to version 24, the system should make version 23 save its state and exit, terminating it forcibly if necessary, so that processes from version 23 do not observe version 24 files or any intermediate state, which would be likely to break their assumptions and cause a crash.
  - This matches the user experience seen on Android: graphical and background processes from an upgraded `.apk` are terminated during upgrade.

[11] https://sjoerd.pages.apertis.org/apertis-website/glossary/#user
[12] https://sjoerd.pages.apertis.org/apertis-website/concepts/applications/#data-storage
[13] https://bugs.apertis.org/show_bug.cgi?id=283
[14] https://sjoerd.pages.apertis.org/apertis-website/glossary/#user
[15] https://sjoerd.pages.apertis.org/apertis-website/concepts/multiuser/#requirements
[16] https://sjoerd.pages.apertis.org/apertis-website/glossary/#store-application-bundle

- Before upgrade from version 23 to version 24, the system must take a copy (snapshot) of each user's data for this application bundle.
- After upgrade from version 23 to version 24, the current data will still be in the version 23 format (a flat file).
- When a user runs version 24, the application bundle may convert the data to version 24 format if desired. This is the application author's choice.
- If a user rolls back Shopping List from version 24 to version 23, the system must restore the saved data from version 23 for each user. (Applications design[17] §4.1.5, "Store Applications — Roll-back")
  - This is because the application author might have chosen to use an incompatible format for version 24, as we have assumed here.
  - For simplicity, we do not require a way for application authors to avoid the data being rolled back.
- Shopping List can be uninstalled. This must be relatively rapid. (Applications design[18] §4.1.4, "Store Applications — Removal")
- When Shopping List is uninstalled from the system, the system must remove all associated data, for all users.
  - If a multi-user system emulates a per-user choice of apps by hiding or showing apps separately on a per-user basis, it should delete user data at the expected time: if user 1 "uninstalls" Shopping List, but user 2 still wants it installed, the system may delete user 1's data immediately.
- To save space, *cache files* (defined to mean files that can easily be re-created, for example by downloading them) should not be included in snapshots. Instead of being rolled back, these files should be deleted during a rollback. (System Update and Rollback design[19] §6.3, "Update and Rollback Procedure")
- **Unresolved:** Are downloads rolled back?

Built-in applications

By definition, built-in application bundles[20] are part of the same filesystem image as the platform. They are upgraded and/or rolled back with the platform. Suppose platform version 2 has a built-in application bundle, Browser version 17. A new platform version 3 becomes available, containing Browser version 18.

[17] https://sjoerd.pages.apertis.org/apertis-website/concepts/applications/
[18] https://sjoerd.pages.apertis.org/apertis-website/concepts/applications/
[19] https://sjoerd.pages.apertis.org/apertis-website/designs/system-updates-and-rollback/
[20] https://sjoerd.pages.apertis.org/apertis-website/glossary/#built-in-application-bundle

- The platform can be upgraded. This does not need to be particularly rapid: a platform upgrade is a major operation which requires rebooting, etc. anyway.
- Before upgrade from version 2 to version 3, the system must take a copy (snapshot) of each user's data for each built-in application bundle.
- Immediately after upgrade, the data is still in the format used by Browser version 17.
- If the platform is rolled back from version 3 to version 2, the system must restore the saved data from platform version 2 for every built-in application, across all users. (Applications design[21] §4.2.4, "Built-in Applications — Rollback"; System Update and Rollback design[22] §6.3, "Update and Rollback Procedure")
- Uninstalling a built-in application bundle is not possible (Applications design[23] §4.2.3, "Built-in Applications — Removal") but it should be possible to delete all of its variable data, with the same practical result as if an equivalent store application bundle had been uninstalled and immediately reinstalled.
- Cache files for built-in applications are treated the same as cache files for Store applications, above.

Global operations

User accounts can be created and/or deleted.

- Deleting a user account does not need to be as rapid as uninstalling an application bundle. It should delete that user's per-user data in all application bundles.

A "data reset" operation affects the entire system. It clears everything.

- A "data reset" does not need to be as rapid as uninstalling an application bundle. It should delete all variable data in each application bundle, and all variable data that is shared by application bundles.

Unresolved: Does data reset uninstall apps?

System extensions

Bundles with sufficient store curator approval[24] and permissions flags may install *system extensions* which will be loaded automatically by platform components. The required permissions may vary according to the type of system extension. For example, a privileged system-wide systemd unit should be a "red flag" which is normally only allowed in built-in applications, whereas a `.desktop` file for a menu entry[25] should normally be allowed in store bundles, provided

[21] https://sjoerd.pages.apertis.org/apertis-website/concepts/applications/
[22] https://sjoerd.pages.apertis.org/apertis-website/designs/system-updates-and-rollback/
[23] https://sjoerd.pages.apertis.org/apertis-website/concepts/applications/
[24] https://sjoerd.pages.apertis.org/apertis-website/concepts/app_store_approval/
[25] https://sjoerd.pages.apertis.org/apertis-website/concepts/application-entry-points/

that its name matches the relevant ISV's reversed domain name.

Public system extensions

Depending on the type of system extension, an extension might also be intended to be loaded directly by store applications. For example, every store application should normally load the current user interface theme, and the set of icons associated with that theme (although each store application bundle may augment these with its own private theming and icon data if desired). We refer to extensions of this type as *public system extensions*, analogous to the *public interfaces* defined by the Interface discovery[26] design.

Security and privacy considerations

- Given an AppArmor profile name, it must be easy to determine (for example via a library API provided by Canterbury) whether the program is part of a built-in application bundle, a store application bundle, or the platform. For application bundles, it must be easy to determine the bundle ID. This is because the uid and the AppArmor profile name are the only information available to services like Newport that receive requests via D-Bus.
- Similarly, given a bundle ID and whether the program is part of a built-in or store application, it must be easy to determine where it may write. Again, this is for services like Newport.
- If existing open source software is included in an application bundle, it may read configuration from `$prefix/etc` with the assumption that this path is trusted. Accordingly, we should not normally allow writing to `$prefix/etc`.
- The set of installed store application bundles is considered to be confidential, therefore typical application bundles (with no special permissions) must not be able to enumerate the entry points, systemd units, D-Bus services, icons etc. provided by store application bundles. A permission flag could be provided to make an exception to this rule, for example for an application-launcher application like Android's Trebuchet.
  - **Unresolved:** Are inactive themes visible to all?
- **Unresolved:** Are built-in bundles visible to all?

Miscellaneous

- Directory names should be namespaced by reversed domain names[27], so that it is not a problem if two different vendors produce an app-bundle with a generic name like "Navigation".
- Because we recommend the GNU Autotools (autoconf, automake, libtool), the desired layout should be easy to arrange by using configure options

[26] https://sjoerd.pages.apertis.org/apertis-website/concepts/interface_discovery/
[27] https://sjoerd.pages.apertis.org/apertis-website/glossary/#reversed-domain-name

such as `--prefix`, in a way that can be standardized by build and packaging tools.

- Where possible, functions in standard open-source libraries in our stack, such as GLib, Gtk, Clutter should "do the right thing". For example, `g_get_cache_dir()` should continue to be the correct function to call to get a parent directory for an application's cache.
- Where possible, functions in other standard open-source libraries such as Qt and SDL should generally also behave as we would want. This can be achieved by making use of common Linux conventions such as the XDG Base Directory specification[28] where possible. However, these other libraries are likely to have less strong integration with the Apertis platform in general, so there may be pragmatic exceptions to this principle: full compatibility with these libraries is a low priority.

Provisional recommendations

The overall structure of these recommendations is believed to be valid, but the exact paths used may be subject to change, depending on the answers to the Unresolved design questions and comparison with containerization technologies such as Flatpak.

Writing application bundles

Application bundle authors should refer to the Apertis Application Bundle Specification[29] instead of this section. This section might describe functionality that is outdated or has not yet been implemented.

Static data

For system-wide static data, programs in application bundles should:

- link against private shared libraries in the Automake `$libdir` or `$pkglibdir` via the `DT_RPATH` (libtool will do this automatically)
- link against public shared libraries provided by the platform in the compiler's default search path, without a `DT_RPATH` (again, libtool will do this automatically)
- run executables from the platform, if required, using the normal `$PATH` search
- run other executables from the same bundle using paths in the Automake `$bindir`, `$libexecdir` or `$pkglibexecdir`
- load static data from the Automake `$datadir`, `$pkgdatadir`, `$libdir` and/or `$pkglibdir` (using the data directories for architecture-independent data, and the library directories for data that may be architecture-specific)

[28] http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html
[29] https://appdev.apertis.org/documentation/bundle-spec.html

    - where possible, resource files should be embedded in the executable or library using GResource; if that is not possible, they can be included in a `.gresource` resource bundle in the `$datadir` or `$pkgdatadir`; if that is not possible either, they can be ordinary files in the `$datadir` or `$pkgdatadir`
    - load plugins from the Automake `$pkglibdir` or a subdirectory
- install system extensions to the appropriate subdirectories of `$datadir` and `$prefix/lib`, if used:
    - `.desktop` files describing entry points (applications and agents) in `$datadir/applications`
    - D-Bus session services in `$datadir/dbus-1/services`
    - D-Bus system services in `$datadir/dbus-1/system-services`
    - systemd user units in `$prefix/lib/systemd/user`
    - systemd system units in `$prefix/lib/systemd/system`
    - icons in subdirectories of `$datadir/icons` according to the freedesktop.org Icon Theme Specification[30]

All of these paths will be part of the application bundle. For store applications, they will be installed, upgraded, rolled back and removed as a unit. For built-in applications, all of these paths will be part of the platform image.

Icons and themes

This section might be split out into a separate design document as more requirements become available.

Icons should be installed according to the freedesktop.org Icon Theme specification[31].

If an application bundle installs a general-purpose icon that should represent an included application throughout the Apertis system, it should be installed in the `hicolor` fallback theme, i.e. `$datadir/icons/hicolor/$size/apps/$app_id.$format`, where `$size` is either a pixel-size or `scalable`, and `$format` is `png` or `svg`.

> The reserved icon theme name `hicolor` is used as the fallback whenever a specific theme does not have the required icon, as specified in the freedesktop.org Icon Theme specification[32]. The name `hicolor` was chosen for historical reasons.

If an application author knows about specific icon themes and wishes to install additional icons styled to coordinate with those themes, they may create `$datadir/icons/$theme_name/$size/apps/$app_id.$format` for that purpose. This should not be done for themes where the desired icon is simply a copy of the `hicolor` icon.

[30] http://standards.freedesktop.org/icon-theme-spec/icon-theme-spec-latest.html
[31] http://standards.freedesktop.org/icon-theme-spec/icon-theme-spec-latest.html
[32] http://standards.freedesktop.org/icon-theme-spec/icon-theme-spec-latest.html

Rationale: Suppose there is a popular theme named `org.example.metallic`, and a popular application named `com.example.ShoppingList`. If the author of Shopping List has designed an icon that matches the metallic theme, we would like the application launcher to use that icon. If not, the author of the metallic theme might have included an icon in their theme that matches this popular application; we would like to use that icon as our second preference. Finally, if there is no metallic-styled icon available, the launcher should use the application's theme-agnostic icon from the `hicolor` fallback directory. We can achieve this result by placing icons from each app bundle's `$datadir` in an early position in the launcher's `XDG_DATA_DIRS`, and placing icons from the theme itself in a later position in `XDG_DATA_DIRS`: the freedesktop Icon Theme lookup algorithm will look for a metallic icon in all the directories listed in `XDG_DATA_DIRS` before it falls back to the `hicolor` theme.

The application may install additional icons representing actions, file types, emoticons, status indications and so on into its `$datadir/icons`. For example, a web browser might require an icon representing "incognito mode", which is probably not present in all icon themes. Similar to the application icon, the browser may install variants of that icon for themes other than `hicolor`, if its author is aware of particular themes and intends the icon to coordinate with those themes.

Unresolved: Standard icon sizes?

Per-user, per-bundle data

For *cached files* that are specific to the application and also specific to a user, programs in application bundles may read and write the directory given by `g_get_user_cache_dir()` or by the environment variable `XDG_CACHE_HOME`. This location is kept intact during upgrades, but is not included in the snapshot made during upgrade, so it is effectively emptied during rollback. It is also removed by uninstallation or a data reset.

For *configuration* that is specific to the application and also specific to a user, the preferred API is the `GSettings` abstraction described in the Preferences and Persistence design document[33]. As an alternative to that API, programs in application bundles may read and write the directory given by `g_get_user_config_dir()`, or equivalently by the environment variable `XDG_CONFIG_HOME`. This locations is kept intact and also backed up during upgrades, restored to its old contents during a rollback, and removed by uninstallation of the bundle, deletion of the user account, or a data reset.

For other variable data that is specific to the application and also specific to a user, programs in application bundles may read and write the directory given by `g_get_user_data_dir()`, or equivalently by the environment variable `XDG_DATA_HOME`. This location has the same upgrade, rollback and removal be-

[33] https://sjoerd.pages.apertis.org/apertis-website/designs/preferences-and-persistence/

haviours as `g_get_user_config_dir()`. Applications may distinguish between configuration and other variable data, but we do not anticipate that this will be necessary in Apertis.

For downloads, programs in application bundles may read and write the result of `g_get_user_special_dir (G_USER_DIRECTORY_DOWNLOADS)`. Each application bundle may assume that it has a download directory per user, shared by all separate from other users and other application bundles. The download service, Newport, may also write to this location. Uninstalling the application bundle or removing the user account causes the download directory to be deleted.

Unresolved: Are downloads rolled back?

Per-user, bundle-independent data

For variable data that is shared between all applications but specific to a user, programs in application bundles may read and write locations in the user's subdirectory of `/home` if they have appropriate permissions flags for their AppArmor profiles to allow it. We should restrict this capability, because it may affect the behaviour of other applications.

These locations should not be what is returned by `g_get_config_home()`, because we want the default to be that app bundles are self-contained. We could potentially provide a way to arrange for specific directories to be symlinked or bind-mounted into the normally-app-specific `g_get_user_config_dir()` and so on.

These locations are not subject to upgrade or rollback, and are never cleared or removed by uninstalling an app-bundle. They are cleared when the user account is deleted, or when a data-reset is performed on the entire device.

Unresolved: How do bundles discover the per-user, bundle-independent location?

Unresolved: Is `g_get_home_dir()` bundle-independent?

User-independent, per-bundle data

As of Apertis 16.12, this feature has not yet been implemented.

For variable data that is specific to the application but shared between all users, programs in application bundles may read and write `/var/Applications/$bundle_id/cache`, `/var/Applications/$bundle_id/config` and/or `/var/Applications/$bundle_id/data`. Convenience APIs to construct these paths should be provided in libcanterbury. Ribchester should create and chmod these directories if and only if the app has a permissions flag saying it uses them, so that the system will deny access otherwise.

These locations have the same upgrade and rollback behaviour as the per-user, per-bundle data areas. They are deleted by a whole-device data reset, but are not deleted if an individual user account is removed.

Shared data

For media files, programs in application bundles may read and write the result of `g_get_user_special_dir (G_USER_DIRECTORY_MUSIC)` and/or `g_get_user_special_dir (G_USER_DIRECTORY_VIDEOS)`. These locations are shared between users and between bundles. The platform may deny access to these locations to bundles that do not have a special permissions flag.

For other variable data that is shared between all applications and all users, programs in application bundles may read and write the result of `g_get_user_special_dir (G_USER_DIRECTORY_PUBLIC_SHARE)`. The platform may deny access to this location to bundles that do not have a special permissions flag. This location is shared between users and between bundles.

These locations are unaffected by upgrade or rollback, but will be cleared by a data reset.

Other well-known directories

Unresolved: Is `PICTURES` per-user?

Unresolved: What is the scope of `DESKTOP`, `DOCUMENTS`, `TEMPLATES`?

Implementation

Application bundles should be installed according to the Apertis Application Bundle Specification[34]. This document does not duplicate the information provided in that specification, but only gives rationale.

The split between `/Applications` or `/usr/Applications` for static data, and `/var/Applications` for variable data, makes it easy for developers and AppArmor profiles to distinguish between static and variable data. It also results in the two different algorithms used during upgrade for store apps being applied to different directories.

The additional split between `/Applications` for store application bundles, and `/usr/Applications` for built-in application bundles, serves two purposes:

- `/usr` is part of the *system partition*, which is read-only at runtime (for robustness), contains the platform and built-in application bundles, and has a limited storage quota because the safe upgrade/rollback mechanism means it appears on-disk twice. `/Applications` is part of the *general storage partition*, which has a more generous storage quota and is read/write at runtime.
- Using a distinctive prefix for built-in application bundles makes it trivial to identify built-in applications from their AppArmor profile names, which are conventionally linked to the programs' filenames.

[34] https://appdev.apertis.org/documentation/bundle-spec.html

The specified layout was chosen so that the static files in `share/` and `lib/` could be organised in the way that would be conventional for Automake installation with a `--prefix=/Applications/$bundle_id` or `--prefix=/usr/Applications/$bundle_id` option. For example, because the app icon in a store app bundle is named something like `/Applications/$bundle_id/share/icons/hicolor/$size/apps/$entry_point_id.png`, it can be installed to `${datadir}/icons/hicolor/$size/apps/$entry_point_id.png` in the usual way.

If there are any non-Automake-based application bundles, they should be configured to install in the same GNU-style directory hierarchy that we would use with Automake, with the analogous parameter corresponding to ${prefix}. We do not recommend distributing non-Automake-based application bundles.

The top-level `config`, `cache`, `data` directories within the bundle's variable data should only be created if the application bundle has special permissions flags. `config`, `cache`, `data` should be considered to be a minor "red flag" by [app-store curators]App Store Approval[35]: because they share data across user boundaries, they come with some risk.

System integration links for built-in applications

The `.deb` package for built-in applications should also include symbolic links for the following system integration files:

- Entry points: link `/usr/share/applications/*.service` points to `/usr/Applications/$bundle_id/share/applications/*.service`
- Icons: `/usr/share/icons/*` → `/usr/Applications/$bundle_id/share/icons/*`
- Other theme files: `/usr/share/themes/*` → `/usr/Applications/$bundle_id/share/themes/*`

Store applications must not contain these links: similar links are created at install-time instead. See Store application system integration links for details.

Special directory configuration

Programs in store application bundles should be run with these environment variables, so that they automatically use appropriate directories:

- `XDG_DATA_HOME=/var/Applications/$bundle_id/users/$uid/data` (used by `g_get_user_data_dir`)
- `XDG_DATA_DIRS=/Applications/$bundle_id/share:/var/lib/apertis_extensions/public:/usr/share` (used by `g_get_system_data_dirs`)
  - See Store application system integration links for the rationale for `/var/lib/apertis_extensions/public`
- `XDG_CONFIG_HOME=/var/Applications/$bundle_id/users/$uid/config` (used by `g_get_user_config_dir`)

[35] https://sjoerd.pages.apertis.org/apertis-website/concepts/app_store_approval/

- `XDG_CONFIG_DIRS=/var/Applications/$bundle_id/etc/xdg:/Applications/$bundle_id/etc/xdg:/etc/xdg` (used by `g_get_system_config_dirs`)
- `XDG_CACHE_HOME=/var/Applications/$bundle_id/users/$uid/cache` (used by `g_get_user_cache_dir`)
- `PATH=/Applications/$bundle_id/bin:/usr/bin:/bin` (used when executing programs)
- `XDG_RUNTIME_DIR=/run/user/$uid` (used by `g_get_user_runtime_dir` and provided automatically by systemd; access is subject to a "whitelist")

Unresolved: Should `LD_LIBRARY_PATH` be set?

This is automatically done by `canterbury-exec` in Apertis 16.06 or later, unless the entry point's bundle ID cannot be determined from its `.desktop` file. For backwards compatibility, Canterbury in Apertis 16.09 still attempts to run entry points whose bundle ID cannot be determined, but this should be prevented in future.

Built-in application bundles should be given the same environment variables, but with `/usr/Applications` replacing `/Applications`.

Unresolved: Is `g_get_home_dir()` bundle-independent?

Unresolved: Is `g_get_temp_dir()` bundle-independent?

In addition, the XDG special directories should be configured as follows for both built-in and store application bundles:

- `g_get_user_special_dir (G_USER_DIRECTORY_DESKTOP)`: **Unresolved**: What is the scope of `DESKTOP`, `DOCUMENTS`, `TEMPLATES`?
- `g_get_user_special_dir (G_USER_DIRECTORY_DOCUMENTS)`: **Unresolved**: What is the scope of `DESKTOP`, `DOCUMENTS`, `TEMPLATES`?
- `g_get_user_special_dir (G_USER_DIRECTORY_DOWNLOAD)`: `/var/Applications/$bundle_id/users/$uid/downloads`
- `g_get_user_special_dir (G_USER_DIRECTORY_MUSIC)`: `/home/shared/Music`
- `g_get_user_special_dir (G_USER_DIRECTORY_PICTURES)`: **Unresolved**: Is `PICTURES` per-user?
- `g_get_user_special_dir (G_USER_DIRECTORY_PUBLIC_SHARE)`: `/home/shared`
- `g_get_user_special_dir (G_USER_DIRECTORY_TEMPLATES)`: **Unresolved**: What is the scope of `DESKTOP`, `DOCUMENTS`, `TEMPLATES`?
- `g_get_user_special_dir (G_USER_DIRECTORY_VIDEOS)`: `/home/shared/Videos`

Again, this is automatically done by `canterbury-exec` in Apertis 16.06 or later.

Permissions and ownership

All files under `/usr/Applications` and `/Applications` should be owned by root, with the standard system permissions (`u=rwX,og=rX` — that is, root may write, and all users may read all files, execute programs that are marked executable and enter directories).

`/var/Applications`, `/var/Applications/$bundle_id` and `/var/Applications/$bundle_id/users/` are also owned by root, with the standard system permissions.

If they exist, `/var/Applications/$bundle_id/{config,data,cache}/` are owned by root, with permissions `a=rwx`. If they are not required and allowed by a permissions flag, they must not exist.

Unresolved: Can we prevent symlink attacks in shared directories?

`/var/Applications/$bundle_id/users/$uid/` and all of its subdirectories are owned by `$uid`, with permissions `u=rwx,og-rwx` for privacy (in other words, only accessible by the owner or by root).

Physical layout

The application-visible directories in `/var/Applications` and `/Applications` are only mount points. Applications' real storage is situated on the general storage volume, in the following layout:

```
<general storage volume>
├─app-bundles/
| ├─com.example.MyApp/                  (store app-bundle)
| | ├─current → version-1.2.2-1         (symbolic link)
| | ├─rollback → version-1.0.8-2        (symbolic link)
| | ├─version-1.0.8-2/
| | | ├─static/                         (subvolume)
| | | | ├─bin/
| | | | └─share/ (etc.)
| | | └─variable/                       (subvolume)
| | |   └─users/
| | |     └─1001/
| | |       ├─cache/
| | |       ├─config/
| | |       └─data/ (etc.)
| | └─version-1.2.2-1/
| |   ├─static/                         (subvolume)
| |   └─variable/                       (subvolume)
| └─org.apertis.Frampton/               (store app-bundle)
|   ├─current → version-2.5.1-1         (symbolic link)
|   └─version-2.5.1-1/
|     └─variable/                       (subvolume)
⋯ <other directories subvolumes unrelated to application bundles>
```

The `static` and `variable` directories are `btrfs` subvolumes so that they can be copied using snapshots, while the other directories shown may be either subvol-

umes or ordinary directories. The `current` and `rollback` symbolic links indicate the currently active version, and the version to which a rollback would move, respectively.

Built-in application bundles do not have a `static` subvolume, because their static files are part of `/usr` on the read-only operating system volume.

All other filenames in this hierarchy are reserved for the application manager, which may create temporary directories and symbolic links during its operation. It must create these in such a way that it can recover from abrupt power loss at any point, for example by making careful use of POSIX atomic filesystem operations to implement "transactions".

During normal operation, the subvolumes would be mounted as follows:

```
com.example.MyApp/current/static     → /Applications/com.example.MyApp
com.example.MyApp/current/variable   → /var/Applications/com.example.MyApp
org.apertis.Frampton/current/variable → /var/Applications/org.apertis.Frampton
```

so that the expected paths such as `/var/Applications/com.example.MyApp/users/1001/config/` are made available.

Only one subvolume per application is mounted – under normal circumstances, this will be the one with the highest version. After a system rollback it might be an older version if the most recent is unlaunchable.

Installation and upgrading

Suppose we are installing `com.example.MyApp` version 2, or upgrading it from version 1 to version 2. An optimal implementation would look something like this:

- If it was already installed:
  - Instruct any running processes belonging to that bundle to exit
  - Wait for the processes to save their state and exit; if a timeout is reached, kill the processes
  - Unmount the `com.example.MyApp/version-1/static` subvolume from `/Applications/com.example.MyApp`
  - Unmount the `com.example.MyApp/version-1/variable` subvolume from `/var/Applications/com.example.MyApp`
  - Create a snapshot of `com.example.MyApp/version-1/static` named `com.example.MyApp/version-2/static`
  - Create a new snapshot of `com.example.MyApp/version-1/variable`, named `com.example.MyApp/version-2/variable`
  - Recursively delete the `cache` and `users/*/cache` directories from `com.example.MyApp/version-1/variable`

- If it was not already installed, instead:
  - Create a new, empty subvolume `com.example.MyApp/version-2/variable` to be mounted at `/var/Applications/com.example.MyApp`
  - Create a new, empty subvolume `com.example.MyApp/version-2/static` to be mounted at `/Applications/com.example.MyApp`
- For each existing static file in `com.example.MyApp/version-2/static` that was carried over from `com.example.MyApp/version-1/static`:
  - If there is no corresponding file in version 2, delete it
  - If its contents do not match the corresponding file in version 2, delete it
  - If its metadata do not match the one in version 2, update the metadata
- For each static file in version 2:
  - If there is no corresponding file in `com.example.MyApp/version-2/static`, the file is either new or changed. Unpack the new version.
- *(Optional, if support for this feature is required)* Copy any files required from `share/factory/{etc,var}` to `{etc,var}`, overwriting files retained from previous versions if and only if the retained version matches what is in version 1's `share/factory/{etc,var}` but does not match version 2's `share/factory/{etc,var}`

A simpler procedure would be to create the `com.example.MyApp/version-2/static` subvolume as empty, and then unpack all of the static files from the new version. However, that procedure would not provide de-duplication between consecutive versions if a file has not changed. As of Apertis 16.09, only this simpler procedure has been implemented.

Ribchester (and perhaps Canterbury) must be modified to create the per-user directories `/var/Applications/$bundle_id/users/$uid`. This was implemented in Apertis 16.06.

Store application system integration links

Application installation for store applications may set up symbolic links in `/var/lib/apertis_extensions` for the categories of system integration files described in System integration links for built-in applications, but the files and their contents must be restricted[36] unless the bundle has special permissions flags. In particular, all entry points (agents and applications) in a bundle must be in the relevant ISV[37]'s namespace.

For example, an application bundle containing a user interface and an agent could be linked like this:

- `/var/lib/apertis_extensions/applications/com.example.MyApp.UI.desktop` → `/Applications/com.example.MyApp/share/applications/com.example.MyApp.UI.desktop`

[36] https://sjoerd.pages.apertis.org/apertis-website/concepts/app_store_approval/
[37] https://sjoerd.pages.apertis.org/apertis-website/glossary/#isv

- `/var/lib/apertis_extensions/applications/com.example.MyApp.Agent.desktop` → `/Applications/com.example.MyApp/share/applications/com.example.MyApp.Agent.desktop`

The designers of Apertis can introduce new system integration points in future versions if required.

The platform components that need to support loading "extension" components from store application bundles will be modified or configured to look in `/var/lib/apertis_extensions`. For example, Canterbury could be run with `XDG_DATA_DIRS=/var/lib/apertis_extensions:/usr/share` so that it will pick up activatable services from `/var/lib/apertis_extensions/dbus-1/services`.

System integration links for public extensions

`/var/lib/apertis_extensions` should *not* be included in the `XDG_DATA_DIRS` for store applications, so that store applications do not automatically attempt to read these restricted directories and receive AppArmor denials. However, a few types of system extension should be loaded by all programs, not just privileged platform components. For example, GUI themes would typically provide icons in `$datadir/icons` and other related files in `$datadir/themes`, which are intended to be loaded by arbitrary applications (so that those applications coordinate with the theme).

We recommend that the system bind-mounts or copies these files into the corresponding subdirectory of `/var/lib/apertis_extensions/public`. In conjunction with the environment variables described above, this means that libraries and applications that follow the XDG Base Directory specification[38], for example Gtk's theme support, will load them automatically.

Please note that symbolic links are *not* suitable for public extensions, because AppArmor access-control is based on the result of dereferencing the symbolic link: if a store application `com.example.ShoppingList` renders widgets using the `org.example.metallic` theme, it would not be allowed to read through a symbolic link that points into `/Applications/org.example.metallic/share/themes/org.example.metallic/`, but it can be allowed to read the same directory indirectly by bind-mounting that directory onto `/var/lib/apertis_extensions/public/themes/org.example.metallic/`.

Uninstallation

- Uninstalling a store application bundle consists of removing `/Applications/$bundle_id`, `/var/Applications/$bundle_id` and the corresponding subvolumes.
- Uninstalling a built-in application bundle is not possible, but it can be reset (equivalent to uninstallation and reinstallation) by deleting and recreating `/var/Applications/$bundle_id` and its corresponding subvolumes.

[38] http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html

- Deleting a user should delete every directory matching `/var/Applications/*/users/$uid`, in addition to the user's home directory.
- A "data reset" consists of:
  - deleting and re-creating `/var/Applications/$bundle_id` for every application bundle
  - *(optional, if a data reset is intended to uninstall store app bundles)* clearing `/Applications`
  - *(optional, if this feature is required)* populating `{etc,var}` from `share/factory/{etc,var}` as if for initial installation

AppArmor profiles

Every application bundle should have rules similar to these in its AppArmor profile:

- `#include <abstractions/chaiwala-base>` (normal "safe" functionality)
- `/{usr/,}Applications/$bundle_id/{bin,lib,libexec}/** mr` (map libraries and the executable described by the profile; read arch-dependent static files)
- `/{usr/,}Applications/$bundle_id/{bin,libexec}/** pix` (run other executables from the same bundle under their own profile, or inherit current profile if they do not have their own)
- `/{usr/,}Applications/$bundle_id/share/** r` (read arch-independent static files)
- `owner /var/Applications/$bundle_id/users/** rwk` (read, write and lock per-app, per-user files for the user running the app)

Note that a write is only allowed if it is allowed by both AppArmor and file permissions, so user A is normally prevented from accessing user B's files by file permissions. The last rule is given the `owner` keyword only for completeness.

Application bundles that require them may additionally have rules similar to these:

- `/var/Applications/$bundle_id/{config,data,cache}/** rwk` (read, write, lock per-bundle, cross-user variable files)
- `/home/shared/{Music,Videos} rwk` (read, write, lock cross-bundle, cross-user media files)
- `/home/shared/{,**} rwk` (read, write, lock all cross-bundle, cross-user files)
- `owner /home/*/$something rwk` (read, write, lock selected cross-bundle, per-user files for the user running the app)

`<abstractions/chaiwala-base>` should be modified to include

- `/var/lib/apertis_extensions/public/** r`

to support public extensions.

Unresolved design questions

Are downloads rolled back?

Newport stores downloaded files in a directory per (bundle ID, user) pair. When an app is rolled back, are those files treated like a cache (deleted), or treated like user data (also rolled back), or left as they are?

Does data reset uninstall apps?

Does a data reset leave the installed store apps installed, or does it uninstall them all? (In other words, does it leave store apps' static files intact, or does it delete them?)

Are inactive themes visible to all?

Suppose the system-wide theme is "blue", and the user has installed but not activated "red" and "green" themes from the app store. Is it OK for an unprivileged app-bundle to be able to see that the "red" and "green" themes exist?

- The same applies to any other Public system extensions.
- For simplicity, we recommend the answer "yes, this is acceptable" unless there is a reason to do otherwise.

Are built-in bundles visible to all?

We know that unprivileged app-bundles are not allowed to enumerate the store application bundles that are installed. Is it OK for an unprivileged app-bundle to be allowed to enumerate the built-in application bundles?

- For simplicity, we recommend the answer "yes, this is acceptable" unless there is a reason to do otherwise.

Standard icon sizes?

Are there specific icon sizes that we want to require every app to supply? As of November 2015, the "Mildenhall" reference HMI uses 36x36 icons. Launchers should be prepared to scale icons as a fallback, but scaled icons at small pixel sizes tend to look blurry and low-quality, so icons of exactly the size required for the HMI should be preferred.

How do bundles discover the per-user, bundle-independent location?

The precise location to be used for per-user, bundle-independent data, and the API to get it, has not been decided.

Is `g_get_home_dir()` bundle-independent?

It is undecided whether the `HOME` environment variable and `g_get_home_dir()` should point to `/home/$user`, or to a per-user, per-bundle location. If those point

to a per-user, per-bundle location, then a separate API will need to be provided by libcanterbury with which a program can access per-user, bundle-independent data.

Is `g_get_temp_dir()` bundle-independent?

It is undecided whether the `TMPDIR` environment variable and `g_get_temp_dir()` should point to `/tmp` as they normally do, or to a per-user, per-bundle location.

Is `PICTURES` per-user?

Should `G_USER_DIRECTORY_PICTURES` be shared between users and between bundles like `G_USER_DIRECTORY_MUSIC` and `G_USER_DIRECTORY_VIDEOS`, or should it be per-user like `$HOME`, or should it be per-user per-bundle like `g_get_user_cache_dir()`?

As of Apertis 16.06, it has been implemented as shared, like `G_USER_DIRECTORY_MUSIC`.

What is the scope of `DESKTOP`, `DOCUMENTS`, `TEMPLATES`?

What should the scope of `G_USER_DIRECTORY_DESKTOP`, `G_USER_DIRECTORY_DOCUMENTS`, `G_USER_DIRECTORY_TEMPLATES` be? Or should we declare these to be unsupported on Apertis, and set them to the same place as `$HOME` as documented by their specification?

As of Apertis 16.06, these were marked as unsupported and set to be the same as `$HOME`.

Unresolved implementation questions

Can we prevent symlink attacks in shared directories?

Can we use AppArmor to prevent the creation of symbolic links in directories that are shared between users or between bundles, so that applications do not need to take precautions to avoid writing through a symbolic link, which could allow one trust domain to make another trust domain overwrite a chosen file if the writing application is insufficiently careful? We probably cannot use `+t` permissions (the "sticky bit", which activates restricted deletion and symlink protection), because that would prevent one user from deleting a file created by another user, which is undesired here.

Should `LD_LIBRARY_PATH` be set?

The Autotools build system (`autoconf`, `automake` and `libtool`) will automatically configure executables to load libraries built from the same source tree in their installed locations, using the `DT_RPATH` ELF header, so it is unnecessary to set `LD_LIBRARY_PATH`.

However, we might wish to set `LD_LIBRARY_PATH=/Applications/${bundle_id}/lib` (or the obvious `/usr/Applications` equivalent) so that app-bundles built with a non-Automake build system will "just work".

Similarly, we might wish to set `GI_TYPELIB_PATH=/Applications/${bundle_id}/lib/girepository-1.0` for app-bundles that use GObject-Introspection.

Alternative designs

Merge static and variable files for store applications

One option that was considered was to separate the read-only parts of built-in application bundles (in `/usr/Applications`) from the read/write parts (in `/Applications`), but not separate the read-only parts of store application bundles (in `/Applications`) from the read/write parts (also in `/Applications`).

This reduces the number of subvolumes (one subvolume per store bundle instead of two), but requires additional complexity in the store bundle installer: it would have to distinguish between the static data directories (`bin`, `share`, etc.) and the variable data directories (`cache`, `users`, etc.) by name.

Add a third subvolume per app-bundle for cache

Conversely, because cache files are not rolled back, we could consider separating disposable cache files from the other read/write parts; they would not be subject to snapshots, and during a rollback, the cache subvolume would simply be deleted and re-created.

Each user's files under their `$HOME`

This strategy is not recommended, and is only mentioned here to document why we have not taken it.

The recommendations above keep all users' variable files for a given application bundle, and any variable files for that bundle that are shared among all users, together. An alternative design that we could have used would be to keep all of a user's variable files, across all bundles, in one place (for example their home directory, `$HOME`).

Because store application bundles can be rolled back independently, each user would need at least one subvolume per store application bundle plus one subvolume for built-in application bundles, so that the chosen store application bundle's data area could be rolled back without affecting other bundles.

The reason that this design was rejected is that it scales poorly in some cases, including the one that we expect to be most frequent (store app-bundle installation and uninstallation). While it does require fewer subvolume manipulations than the recommended design for some operations, those operations are expected to

be rare. To illustrate this, suppose we have 10 built-in bundles, 20 store bundles and 5 users.

If we install, upgrade or remove the store bundle `com.example.MyApp`, which additionally has some variable files that are shared between users. With the recommended design, we only have to perform O(1) subvolume operations (two with the recommended design, one if we Merge static and variable files for store applications, or three if we Add a third subvolume per app-bundle for cache). In this alternative design, we would have to perform O(number of users) subvolume operations, in this case 7: one for the bundle's static files, one for its variable files shared between users, and one per user.

Similarly, when we upgrade the platform and we wish to take a snapshot of each built-in application's data, the recommended design requires us to take 10 snapshots (more generally O(1), one per built-in bundle), whereas this alternative requires 50-60 snapshots (more generally O(number of users), one per built-in bundle per user, and zero or one per built-in bundle for non-user-specific data).

If we add or delete a user, in the recommended design we would have to perform 31 subvolume operations, or more generally O(number of bundles): one per store or built-in bundle, plus one extra operation for non-bundle-specific data. In this alternative we would need a minimum of 22 subvolume operations, or more generally O(number of store bundles): one per store bundle, one for all built-in bundles together, and one for non-bundle-specific data.

If we perform a data reset without uninstalling store app bundles, the recommended design would require at least 30 subvolume deletions (one per application bundle), whereas this design would require at least 150 subvolume deletions (one per bundle per user).

System integration links for services

It would be technically possible to install user-services (services that run as a particular user, similar to Tracker) in an application bundle, and register them with the wider system via system integration links (System integration links for built-in applications, Store application system integration links) pointing to their systemd user services and D-Bus session services.

We recommend that this is not done, because general systemd user services are powerful and have a global effect. Instead, we recommend that per-app-bundle user-services (agents) are implemented by having the application manager (Canterbury) generate a carefully constrained subset of service file syntax from the entry point metadata.

System services in app-bundles

It would be technically possible to install system services (services that do not run as a specific user) in an application bundle, registering them via system integration links as above.

We recommend that this is not done, because system services are extremely powerful and can have extensive privileges. Instead, system services should be part of the platform[39] layer.

Appendix: application layout in Apertis 15.09

Sudoku is one example of a store application bundle. Its source code is not currently public. `xyz` is used here to represent the common prefix for an Apertis variant. The layout of the store application bundle looks like this:

```
/appstore/
    store.json
    store.sig
    xyz-sudoku_config.tar
        xyz-sudoku_config/
            xyz-sudoku.png
            xyz-sudoku_manifest.json
/xyz-sudoku.tar
    xyz-sudoku/
        bin/
            xyz-sudoku
        share
            glib-2.0
                schemas
                    com.app.xyz-sudoku.gschema.xml
                    com.app.xyz-sudoku.enums.xml
                    gschemas.compiled
            background.png
            icon_sudoku.png
            (more graphics)
```

The manifest indicates that `/xyz-sudoku.tar` is expected to be unpacked into `/Applications`, leading to filenames like `/Applications/xyz-sudoku/bin/xyz-sudoku`.

Frampton[40] is an example of a built-in application bundle shipped in 15.09. Its layout is as follows:

[39] https://sjoerd.pages.apertis.org/apertis-website/glossary/#platform
[40] https://gitlab.apertis.org/appfw/frampton

```
/usr/
    Applications/
        frampton/
            bin/
                frampton
                frampton-agent
                test-frampton-agent
            lib/
                libframptonagentiface.so{,.0,.0.0.0}
            share/
                IconBig_Music.png
                icon_albums_inactive.png
                ...
                artist-album-views/
                    DetailView.json
                    ...
                glib-2.0/
                    schemas/
                        com.app.frampton-agent.gschema.xml
                        ...
                locale/
                    de/
                        ...
/Applications/
    Frampton/
        app-data/
            Internal/
                FramptonAgent.db
    frampton/
        app-data/
            (empty)
```

Issues with the application filesystem layout in these examples:

- There is no "manifest" file with metadata for the built-in application bundle as a whole.
- The "manifest" files for entry points in both store and built-in applications are GSettings schema XML, which is not how GSettings is designed to be used. They are also incorrectly namespaced: the app developer presumably does not own `app.com`. We should use `org.apertis.*` for Apertis components, `{com,net,org}.example.*` for developer examples, and a vendor's name elsewhere.
- There is no separation between users. "user" owns all of /Applications.

- Frampton's app bundle ID is ambiguous: is it Frampton or frampton? We should choose exactly one ID, and make the AppArmor profile forbid using the other.
- Frampton's app bundle ID is not namespaced. The Applications design document[41] specifies use of a reversed domain name[42] such as org.apertis.Frampton.
- Similarly, Sudoku's app bundle ID is not namespaced.
- There is no well-known location for apps' icons: Frampton places its icons in /usr/Applications/frampton/share/, but other apps use /usr/Applications/$bundle_id/share/images, requiring mildenhall-launcher to be allowed to read both locations.
- There is no well-known location into which Newport may download files.

Appendix: comparison with other systems

Desktop Linux (packaged apps)

There are many possibilities, but a common coding standard looks like this:

- Main programs are installed in `$bindir` (which is set to `/usr/bin`)
- Supporting programs are installed in `$libexecdir` (which is set to either `/usr/libexec` or `/usr/lib`), often in a subdirectory per application package
- Public shared libraries are installed in `$libdir` (which is set to either `/usr/lib` or `/usr/lib64` or `/usr/lib/$architecture`)
  - Plugins are installed in a subdirectory of `$libdir`
  - Private shared libraries are installed in a subdirectory of `$libdir`
- `.gresource` resource bundles (and any resource files that cannot use GResource) are installed in `$datadir`, which is set to `/usr/share`
- System-level configuration is installed in a subdirectory of `$sysconfdir`, which is set to `/etc`
- System-level variable data is installed in `$localstatedir/lib/$package` and `$localstatedir/cache/$package`, with `$localstatedir` set to `/var`
- There is normally no technical protection between apps, but per-user variable data is stored according to the XDG Base Directory specification[43] in:
  - `$XDG_CONFIG_HOME/$package`, defaulting to `/home/$username/.config/$package`, where $username is the user's login name and $package is the short name of the application or package
  - `$XDG_DATA_HOME/$package`, defaulting to `/home/$username/.local/share/$package`
  - `$XDG_CACHE_HOME/$package`, defaulting to `/home/$username/.cache/$package`
- The user's home directory, normally `/home/$username`, is shared between apps but private to the user
  - It is usually technically possible for one app to alter another app's subdirectories of `$XDG_CONFIG_HOME` etc.

[41] https://sjoerd.pages.apertis.org/apertis-website/concepts/applications/
[42] https://sjoerd.pages.apertis.org/apertis-website/glossary/#reversed-domain-name
[43] http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html

- There is no standard location that can be read and written by all users, other than temporary directories which are not intended to be shared

Debian Policy §9.1 "File system hierarchy"[44] describes the policy followed on Debian and Ubuntu systems for non-user-specific data. It references the Filesystem Hierarchy Standard, version 2.3[45].

Similar documents:

- The Filesystem Hierarchy Standard, version 3.0[46] has not yet been adopted by Debian Policy.
- The GNU Coding Standards[47] use a similar layout by default.
- systemd's proposals for file hierarchy[48] have been partially adopted by Linux distributions.

Flatpak

Autoconf/Automake software in a Flatpak[49] package is built with `--prefix=/app`, and the static files of the app are mounted at `/app` inside the sandbox. Each Flatpak has its own private view of the filesystem inside its sandbox, so this does not lead to conflict over ownership of `/app` as might be expected.

- Main programs are installed in `$bindir`, which is `/app/bin`
- Supporting programs are installed in `$libexecdir`, which is `/app/libexec`
- Private shared libraries are installed in `$libdir`, which is `/app/lib`, or in a subdirectory
  - Plugins are installed in a subdirectory of `$libdir`
- Static resources are embedded using GResource, installed in `/app/share` as a `.gresource` resource bundle, or installed in `/app/share` as plain files
- System-level configuration is installed in `/app/etc`
- Per-user variable data is stored in `/home/$username/.var/app/$app_id/{data,config,cache}`, which are bind-mounted into the app's filesystem namespace, with the `XDG_{DATA,CONFIG,CACHE}_HOME` environment variables set to point at those locations
- Shared variable data is stored in `/var/lib/$app_id`, `/var/cache/$app_id`. *(How widely shared is this really?)*

Integration files (systemd units, D-Bus services, etc.) are said to be *exported* by the Flatpak, and they are linked into `$XDG_DATA_HOME/flatpak/exports` or `/var/lib/flatpak/exports` outside the sandbox.

Runtimes (sets of libraries) are mounted at `/usr` inside the sandbox.

[44] https://www.debian.org/doc/debian-policy/ch-opersys.html#s9.1
[45] http://www.pathname.com/fhs/pub/fhs-2.3.html
[46] http://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html
[47] https://www.gnu.org/prep/standards/html_node/Directory-Variables.html#Directory-Variables
[48] http://www.freedesktop.org/software/systemd/man/file-hierarchy.html
[49] http://flatpak.org/

Android

- System app packages (the equivalent of our built-in application bundles[50]) are stored in `/system/app/$package.apk`
- Normal app packages (the equivalent of our store application bundles[51]) are stored in `/data/app/$package.apk`
- Private shared libraries and plugins (and, technically, any other supporting files) are automatically unpacked into `/data/data/$package/lib/` by the OS
- Resource files are loaded from inside the `.apk` file (analogous to GResource) instead of existing as files in the filesystem
- Per-user variable data is stored in `/data/data/$package/` on single-user devices
- Per-user variable data is stored in `/data/user/$user/$package/` on multi-user devices
- There is no location that is private to an app but shared between users. The closest equivalent is `/sdcard/$package`, which is conventionally only used by the app `$package`, but is technically accessible to all apps.
- There is no location that is shared between apps but private to a user.
- `/sdcard` is shared between apps and between users. Large data files such as music and videos are normally stored here.

systemd "revisiting Linux systems" proposal

The authors of systemd propose a structure like this[52]. At the time of writing, no implementations of this idea are known.

- The static files of application bundles are installed in a subvolume named `app:$bundle_id:$runtime:$architecture:$version`, where:
  - `$bundle_id` is a reversed domain name identifying the app bundle itself
  - `$runtime` identifies the set of runtime libraries needed by the application bundle (in our case it might be `org.apertis.r15_09`)
  - `$architecture` represents the CPU architecture
  - `$version` represents the version number
- That subvolume is mounted at `/opt/$bundle_id` in the app sandbox. The corresponding runtime is mounted at `/usr`.
- User-specific variable files are in a subvolume named, for example, `home:user:1000:1000` which is mounted at `/home/user`.
- System-level variable files go in `/etc` and `/var` as usual.
- There is currently no concrete proposal for a trust boundary between apps: all apps are assumed to have full access to `/home`.
- There is no location that is private to an app but shared between users.
- There is no location that is shared between apps and between users, other than removable media.

[50] https://sjoerd.pages.apertis.org/apertis-website/glossary/#built-in-application-bundle
[51] https://sjoerd.pages.apertis.org/apertis-website/glossary/#store-application-bundle
[52] http://0pointer.net/blog/revisiting-how-we-put-together-linux-systems.html

References

- Applications design document[53] (v0.5.4 used)
- Multimedia design document[54] (v0.5.4 used)
- Security design document[55] (v1.1.3 used)
- System Update and Rollback design document[56] (v1.6.2 used)

[53] https://sjoerd.pages.apertis.org/apertis-website/concepts/applications/
[54] https://sjoerd.pages.apertis.org/apertis-website/concepts/multimedia/
[55] https://sjoerd.pages.apertis.org/apertis-website/designs/security/
[56] https://sjoerd.pages.apertis.org/apertis-website/designs/system-updates-and-rollback/