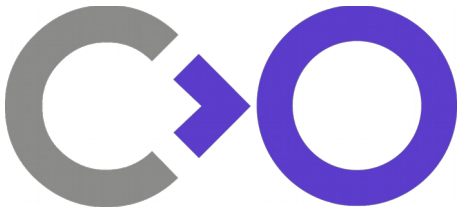

Apertis Preferences and Persistence Design

Author:	Philip Withnall
Contributors:	Simon McVittie, Gustavo Noronha, Guillaume Desmottes
Version:	0.3.0
Status:	Draft
Date:	2015-11-25
Last Reviewer:	Ekaterina Gerasimova

This design was produced exclusively using free and open source software.

Please consider the environment before printing this document.

DOCUMENT CHANGE LOG

Version	Date	Changes
0.3.0	2015-11-25	• Expand background sections (4). • Give alternative designs for the system preferences application (5.1.5) • Clarify recommendation to store secrets and passwords via a different API
0.2.3	2015-11-16	• Delete obsolete document properties • Improve language
0.2.2	2015-10-20	• Clarify wording of rollback sections.
0.2.1	2015-10-14	• Add cross-references to the System Updates and Rollback design for the rollback requirement (3.3). • Clarify system and app bundle upgrades requirement (3.4).
0.2.0	2015-06-01	• Add references to the Robustness Design document. • Add Rollback (3.3), Transactional updates (3.9) and Storage of user secrets and passwords (3.16) requirements. • Expand Factory reset requirement (3.5) to cover clearing a single (user, app) pair. • Clarify suggestions regarding templated preference UIs. • Clarify that apps are responsible for upgrading their own schemas.
0.1.0	2015-05-12	• New document to summarise discussions from various bug reports.

Table of Contents

1 INTRODUCTION

This documents how system services and apps in Apertis may store preferences and persistent data. It considers the security architecture for storage and access to these data; separation of schemas, default values and user-provided values; and guidelines for how to present preferences in the UI.

The Applications Design, and Global Search Design documents are relevant reading. Sections 5.3.1 and 7 of the Applications Design and section 6.3 of the Global Search Design reference the need for storage of persistent data for apps. See section 5.2 for a design covering this.

The Robustness Design document gives more detail on the requirements for robustness of secondary storage in the face of power loss.

2 TERMINOLOGY AND CONCEPTS

2.1 SYSTEM SETTINGS

A *system setting* is one which does not vary by user, and applies to the entire system. For example, networking settings. This document considers system settings which must be readable by multiple components — settings which are solely for the use of a single system service are out of scope, and may be stored in whichever way that service wishes (typically as a configuration file in `/etc`). This is particularly important for sensitive settings, for example the shadow user database in `/etc/shadow`, which must not be readable by anything except the system authentication service (PAM).

2.2 USER SETTINGS

A *user setting* is one which does vary by user, but not by app. User settings apply to the whole of a user's session. For example, the language or theme.

2.3 APP SETTINGS

An *app setting* is one which varies by user and also by app. Throughout this document, the term ‘app’ is used to mean an app-bundle, including the UI and any associated agent programs, analogous to an Android .apk, with a single security domain shared between all executables in the bundle. The precise terminology is currently under discussion, and this document will be updated to reflect the result of that.

App settings apply only to a specific app, and would not make sense outside the context of that app. For example, whether to enable shuffling tracks in the media player; whether to open hyperlinks in a new tab by default in the web browser; or the details for accessing a user's e-mail account.

2.4 PREFERENCES

'*Preferences*' is the general term for system, user and app settings. The terms 'preference' and 'setting' will be used interchangeably throughout this document.

2.5 USER SERVICES

A *user service* is as defined in the Multiuser Design document — a service that runs on behalf of a particular user. Throughout this document, this is additionally assumed to mean a *platform* user service, which is not tied to a particular app-bundle. The alternative is an *agent* user service, which this document considers part of an app-bundle, with the same access to settings as the app-UI.

2.6 PERSISTENT DATA

Persistent data is app state which persists across multiple user sessions. For example, documents which the user has written, or the state of the user's pending downloads.

One distinguishing factor between preferences and persistent data is that vendors may override the default values for preferences (see section 3.13), but not for persistent data. For example, a vendor would not want to override information about in-progress downloads; but they might want to override the default background image filename for a user.

The persistent data for an app may be the same as the data it shares between user sessions, or may differ. The difference between persistent data and data for sharing between apps is discussed in the Multiuser Design document.

Persistent data is stored on secondary storage, whereas shared data is expected to be passed in memory — so while the sets of data are the same, the mechanisms used to handle them are different. Persistent data is always private to an app, and cannot be read by another app or user.

Persistent data might cover all state in an application — such that restoring its persistent data when starting the application is sufficient to make it appear as if it had been suspended, rather than exited. Or persistent data might cover some subset of this. The decision is up to the application authors.

2.7 SECONDARY STORAGE

A flash disk, hard disk, or other persistent data storage medium which can be used by the system. This term has been chosen rather than the more common *persistent storage* to avoid confusion with persistent data.

2.8 GSETTINGS

GSettings[1] is an interface provided by GLib for accessing settings. As an interface, it can be backed by different storage backends — the most common is dconf, but a key file backend is available for storage in simple key files.

GSettings uses a concept of 'schemas', which define available settings, their data types, and their default values. Each setting is strictly typed and must have a default value. A schema has an ID, and is 'instantiated' at one or more schema paths. Typically, a schema will be instantiated at a single path, but may be instantiated at multiple paths to support storing the same settings for multiple objects — for example, a server name, username and protocol to use for multiple different e-mail accounts.

2.9 APPARMOR

AppArmor[2] is an access control framework used by Apertis to enforce fine-grained permissions across the entire system, restricting which files each process can open.

1 https://developer.gnome.org/gio/stable/GSettings.html#GSettings.description
2 http://apparmor.net/

3 REQUIREMENTS

3.1 ACCESS PERMISSIONS

Access controls must be enforceable on preferences. Read and write permissions must be available. It is assumed that if a component has read permission for a preference, it may also be notified of any changes to that preference's value. It is assumed that if a component has write permission for a preference, it may also reset that preference.

A suggested security policy for preferences implements a downwards flow for **reads**:

- **Apps** may read their own app settings, user settings for the current user, and all system settings.
- **User services** may read their own app settings, user settings for the current user, and all system settings.
- **System services** may read their own app settings, and all system settings.

Writes are generally only allowed at the same level:

- **Apps** may write their own app settings.
- **User services** may write user settings for the current user.
- **System services** may write system settings for all users, user settings for any user, and app settings for any app for any user.

Note that apps must not be able to read or write each others' settings. Similarly for user services and system services.

Persistent data is always private to a (user, app) pair, though it can be accessed by user services and system services.

3.2 WRITABILITY

As well as the value of a preference, components must be able to find out whether the preference is writable. A preference may be read-only if the component doesn't have write permission for it (section 3.1) or if it is locked down by the vendor (section 3.14).

This does not apply to persistent data, which is always read–write by the (user, app) pair which owns it.

3.3 ROLLBACK

As per section 4.1.5 of the Applications Design document, and section 6 of the System Update and Rollback Design document, applications must support rollback to a previously installed version, including restoring the user’s settings for that application by reverting the stored preferences to those from the earlier version. The storage backends for the preferences and persistence APIs must support restoring stored preferences from an earlier version — they should not support context-sensitive conversion of newer preferences to older ones.

Applications do not have to support running with preferences or persistent data from a newer version than the application code.

3.4 SYSTEM AND APP BUNDLE UPGRADES

As per the Applications Design and the System Update and Rollback design, applications must also support upgrading preferences and persistent data from previous application versions to the current version.

They do not need to support downgrading preferences or persistent data by converting it from a newer version to an older one.

3.5 FACTORY RESET

The system must provide some means for the user to reset the state of all apps to a factory default for a particular user, or for all users. This is necessary for supporting removing user accounts, refreshing the car for transfer to a new owner, or clearing the state of a temporary guest account (see the Multiuser Design document). Similarly, it must support clearing the state of a single (user, app) pair.

The factory reset must support resetting preferences, persistent data, or both.

3.6 ABSTRACTION LEVEL

The preferences and persistent data APIs may want to abstract the underlying storage backend, for example to support uniform access to preferences stored in multiple locations. If so, details of the underlying storage backend must not be present in the abstraction (a 'leaky abstraction') — for example, SQL fragments must not be used in the interface, as they tie the implementation to an SQL-based backend and a specific schema.

Conversely, any more than one layer of abstraction is an unnecessary complication.

3.7 MINIMISING I/O BANDWIDTH

As with all components which use secondary storage, the preferences and persistent data stores should minimise the I/O load they impose on secondary storage. This is a particular concern at system startup, where typically a lot of data must be loaded from secondary storage, and hence I/O read efficiency is important.

3.8 ATOMIC UPDATES

The system must make atomic writes to secondary storage, so that preferences or persistent data are not corrupted or lost if power is lost part-way through saving changes.

An atomic write is one where the stored state is either the old state, or the new state, but never an intermediate between the two, and never missing entirely. In other words, if power is lost while updating a preference, upon rebooting either the old value of the preference must be loadable, or the new value must be loadable.

See the Robustness Design document, §3.1.1 for more details on general robustness requirements.

3.9 TRANSACTIONAL UPDATES

The system must allow updates to preferences to be wrapped in transactions, such that either all of the preferences within a transaction are updated, or none of them are. Transactions must be revertable being being applied permanently.

3.10 PERFORMANCE TRADEOFFS

Preferences are typically written infrequently and read frequently; access patterns for persistent data depend on the app. The implementation should play to those access patterns, for example by using locking which favours readers over writers.

3.11 DATA SIZE TRADEOFFS

It is not expected that preference values will be large — a few tens of kilobytes at most. Conversely, persistent data may range in size from a few bytes to many megabytes. The implementation should use a storage format suitable to the expected data size.

3.12 RACE CONDITIONS

As system preferences may affect security policy, reading them should be race free, particularly from time-of-check-to-time-of-use races[3]. For example, if a preference is changed by process C while process R is reading it, process R must either see the new value of the preference, or see the old value of the preference *and* subsequently be notified that it has changed.

Similarly for persistent data.

3.13 VENDOR OVERRIDES

It may be desirable to support *vendor overrides*, where a vendor shipping Apertis can change the default values of the (app, user or system) preferences before shipping to the end user. For example, they may change the default background image shown to the user.

If these are supported, resetting a preference to its default value (for example, if doing a factory reset, section 3.5) must restore it to the vendor-supplied default, rather than the Apertis default. There is no need to be able to access the Apertis default at any time.

This does not apply to persistent data.

3.14 VENDOR LOCKDOWN

It may also be desirable to support *vendor lockdowns*, where a vendor shipping Apertis can lock a preference so that end users may not change it. For example, they may wish to lock

3 http://en.wikipedia.org/wiki/Time_of_check_to_time_of_use

the URI which is checked for system updates.

This does not apply to persistent data.

3.15 USER INTERFACE

There must be some user interface (UI) for setting preferences. This may be provided by a special preferences app, or as a separate window in each app, or a combination of the two.

This does not apply to persistent data.

3.16 STORAGE OF USER SECRETS AND PASSWORDS

There must be a secure way to store user secrets and passwords, which preserves confidentiality of these data. This may be separate from the main preferences or persistent data stores.

4 EXISTING PREFERENCES SYSTEMS

This chapter describes the conceptual model, user experience and design elements used in various non-Apertis operating systems' support for preferences and persistent data, because it might be useful input for decision-making. Where available, it also provides some details of the implementations of features that seem particularly interesting or relevant.

4.1 GNOME LINUX DESKTOP

4.1.1 PREFERENCES

On a modern GNOME desktop, from which Apertis uses a lot of components, settings are stored in multiple places.

- **System settings**: Stored in `/etc` by each system service, typically in a text file with a service-specific format. A lot of them have a system-wide default value, and may be overridden per user (for example, each user can set their own timezone and locale, with a system-wide default).
- **User settings**: Defined by shared GSettings schemas (such as `org.gnome.system.locale`), or schemas specific to individual user services (such as `org.freedesktop.Tracker`). The values are stored in dconf (see below).
- **App settings**: Defined by app-specific GSettings schemas. The values are stored in dconf (see below).

dconf supports multiple layered databases[4], each stored separately. For each settings key, a value set for it in one layer overrides any values set in the layers below. The bottom (read-only) layer is always the set of default values which are provided by the schema file. This layered approach allows the system administrator to change settings system-wide in a system database, but also allows users to override those settings in their per-user database. It allows a user to reset all their settings by deleting their per-user database — at which point, the values from the next layer down (typically either a system database or the defaults from schema files) will be used for all settings keys.

Lockdown[5] is supported in dconf in the opposite direction: keys may be locked down at a particular level, and may not be set at levels above that one (but may be set at levels below it, as defaults).

Architecturally, dconf allows direct read-only access to all databases — each app reads settings values directly from the database. Writes to the databases are arbitrated through a per-user dconf daemon which then forces each app to refresh its read-only view of the settings. This allows for fast concurrent reads of settings, at the cost of making writes expensive.

dconf does *not* support access controls, and does not support storing different schemas in different databases at the same layer. Hence a user either has write access to the whole of a system database, or write access to none of it. As the dconf daemon runs per user, any

4 https://developer.gnome.org/dconf/unstable/dconf-overview.html
5 https://developer.gnome.org/dconf/unstable/dconf-overview.html#id-1.2.7

app accessing the daemon may write to any settings key, either its own app settings, another app's settings, or the user's settings.

4.1.2 PERSISTENT DATA

Persistent data is stored in application-defined formats, in application-defined locations, although many follow the XDG Base Directory Specification[6], which puts cache data in `XDG_CACHE_HOME` (typically `~/.cache`) and non-cache data in `XDG_DATA_HOME` (typically `~/.local/share`). Below these two directories, applications create their own directories or files as they see fit. There is no security separation between applications, but the normal UNIX permissions restrict access to only the current user.

There are no APIs available in GNOME for automatically persisting an entire application's state — if an application wishes to do this, it must implement its own serialisation and deserialisation functions and save to a file, as above.

4.1.3 SECRETS AND PASSWORDS

On a GNOME or KDE desktop, all user secrets, passwords and credentials are stored using the Secret Service API[7]. In GNOME, this API is implemented by GNOME Keyring; in KDE, by KWallet.

The API[8] allows storage of byte array 'secrets' (such as passwords), along with non-secret attributes used to look them up, in an encrypted storage file which must be unlocked by the user before it can be accessed by applications. Unlocking it may be automatic if the user does not set a password on the file (or if the password is identical to the user's login password). Secrets are stored in 'collections', which may group them for different purposes, and which are encrypted separately.

An application must open a session with the secret service in order to access secrets. The session may be used to encrypt secrets while they are in transit between the service and application, and allows for algorithm negotiation for this purpose.

For certain actions, the secret service may need to interact directly with the user in order to establish a trusted path to the user, and avoid (for example) requiring the user to enter their password into a potentially untrusted application for that application to forward it to the service.

4.2 ANDROID

4.2.1 PREFERENCES

Apps can use the `SharedPreferences` class[9] to read and write preferences from named preferences files, with apps typically using a single preferences file with a default name. These files are stored per-app, and are private to that app by default, but may be shared with other apps, either read-only or read–write.

6 http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html
7 http://standards.freedesktop.org/secret-service/
8 http://standards.freedesktop.org/secret-service/pt02.html
9 http://developer.android.com/guide/topics/data/data-storage.html#pref

Preferences are strongly typed, and default values are provided by the app at runtime. There is no concept of layering or of schemas — all definition of the preferences files is handled at runtime.

Preferences are saved to disk immediately.

Android uses a custom XML format[10] to allow apps to define preference UIs (known as 'activities' in Android terminology). This format can define simple lists of preferences, through to complex UIs with grouped preferences, subscreens, lists of subscreens, and custom preference widgets. Implementing features such as making one preference conditional on another is possible, but requires complex XML.

4.2.2 PERSISTENT DATA

Android offers several options for persistent data[11]:

- **Internal storage**: Files in a per-(user, app) directory, which may optionally be made world-readable or writable to allow access to other apps or users (though this is strongly discouraged).
- **External storage**: Files in a world-readable storage area which is accessible to the user, such as an SD card. Accessible to all other apps and users which hold the `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE` permissions.
- **SQLite database**: Arbitrary app-defined tables in a per-(user, app) SQLite database. This may not be shared with other apps or users.
- **Network connection**: Using the normal networking APIs, Android suggests that data can be stored on servers controlled by the app developers. It provides no special API for this.

For saving an application's state, Android offers a persistence API on the Activity class[12]. This automatically saves the state of all UI elements (such as the text in an entry widget, and the position of a list), but cannot automatically save application-specific internal state (member variables). For this, the application must override two toolkit methods (`onSaveInstanceState()` and `onRestoreInstanceState()`) and implement its own serialisation and deserialisation of state to a set of key–value pairs which are then stored by Android.

4.2.3 SECRETS AND PASSWORDS

Android recommends storing secrets and passwords in two ways. For authentication credentials for online services, it provides an `AccountManager` API[13] which abstracts authentication for known online services (which are supported by pluggable backends, potentially provided by application bundles) and stores the credentials in an OS-wide store. The service handles authenticating and re-authenticating when the login session ends.

For secrets which are not for online accounts, or otherwise do not fit the `AccountManager`

10 http://developer.android.com/guide/topics/ui/settings.html#DefiningPrefs
11 http://developer.android.com/guide/topics/data/data-storage.html
12 http://developer.android.com/training/basics/activity-lifecycle/recreating.html
13 http://developer.android.com/reference/android/accounts/AccountManager.html

pattern, Android recommends[14] using the normal preferences API (section 4.2.1), as while preferences are not encrypted in storage, they are only accessible to the application which owns them, so cannot be stolen by other applications. However, if the sandboxing system is compromised (potentially by an attacker with physical access to the device), the stored secrets will be accessible in plaintext.

4.3 iOS

4.3.1 PREFERENCES

iOS stores preferences as key–value pairs[15], which are separated into domains by user, application and machine. The same preference may be set in multiple domains, and they are searched in a defined priority order to determine which value to use[16]. This means that an application may, for example, choose to share a given preference between all users of that application on a given machine.

Application IDs use the standard reverse domain name syntax to ensure uniqueness.

Preference values may be any type supported by Core Foundation property lists[17], including strings, integers and arrays. Default values must be coded into the application.

Preference keys may be generated at runtime by the application, and do not have to be defined in a schema in advance.

Preferences are synchronised with the on-disk store manually, so the application chooses when they are written to disk.

On certain Apple operating systems, preferences may be 'managed' by the administrator[18], setting an override value which overrides any value set by the user for a given preference key.

4.3.2 PERSISTENT DATA

iOS offers several options for persistent data:

- **Filesystem**: Arbitrary files may be written to the filesystem in various app-specific locations[19].
- **Core Data API**: This is an object-graph management API[20], which allows versioned

14 http://stackoverflow.com/questions/785973/what-is-the-most-appropriate-way-to-store-user-settings-in-android-application/786588#786588

15 https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFPreferences/CFPreferences.html#//apple_ref/doc/uid/10000129-SW1

16 https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFPreferences/Concepts/PreferenceDomains.html

17 https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFPropertyLists/CFPropertyLists.html#//apple_ref/doc/uid/10000130i

18 https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFPreferences/Concepts/BestPractices.html#//apple_ref/doc/uid/TP30001219-118191

19 https://developer.apple.com/library/ios/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/AccessingFilesandDirectories/AccessingFilesandDirectories.html#//apple_ref/doc/uid/TP40010672-CH3-SW11

20 https://developer.apple.com/library/prerelease/ios/documentation/DataManagement/Devpedia-CoreData/coreDataOverview.html#//apple_ref/doc/uid/TP40010398-CH28

control of instances of objects created from a schema. Instead of being used by an application to persist data, this API is designed to form the core of the application's data model. It supports editing and discarding edits, undo, redo, versioning of the object schema, and large data sets.

- **Property List API**: A property list is a hierarchical, structured piece of data, consisting of primitive data types, arrays and dictionaries which may be nested arbitrarily[21]. Property lists can therefore be used to store arbitrary application data. There is an API to serialise them to the file system.
- **SQLite**: The standard SQLite API may be used, backed by a file, to store relational data in a database.

For persisting an entire application's state, iOS provides a solution similar to Android (section 4.2.2)[22]. The developer must annotate each UI view class which needs to be saved and restored, and the UI toolkit will automatically persist the state of the widgets in that view when the application is suspended. As with Android, the developer must implement two methods for serialising and deserialising application-specific state from member variables: `encodeRestorableStateWithCoder` and `decodeRestorableStateWithCoder`.

4.3.3 SECRETS AND PASSWORDS

iOS uses the same keychain API[23] as OS X. This provides a system service for storing secrets, passwords and certificates. They are encrypted in storage, using an encryption key which is derived from the iOS application's ID and the user's password.

The keychain is encrypted in backups, and stored without its encryption key, so an attacker cannot extract secrets from backups.

An iOS application can access the secrets it has stored in the keychain, but cannot access secrets from other applications. There is no way to (for example) share login details for a given website between all applications which access that website — they must all query the user for the details and store them separately. This differs from OS X, where all applications can access any stored secrets, subject to the user approving the access (trusting the application).

4.4 GENIVI

4.4.1 PREFERENCES AND PERSISTENT DATA

GENIVI does not differentiate between preferences and persistent data, and provides one low-level API for saving and loading persistent data. It does not support automatically persisting an entire application's state.

21 https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/PropertyLists/AboutPropertyLists/AboutPropertyLists.html

22 https://developer.apple.com/library/ios/featuredarticles/ViewControllerPGforiPhoneOS/PreservingandRestoringState.html

23 https://developer.apple.com/library/ios/documentation/Security/Conceptual/keychainServConcepts/01introduction/introduction.html#//apple_ref/doc/uid/TP30000897-CH203-TP1

The GENIVI Persistence Management system[24] handles all data read and written during the lifetime of an IVI system. It aims to provide a standard API for all GENIVI platforms to use, which reliably stores data in the face of power disturbances, and the limited write-cycle lifetime of some non-volatile storage devices (flash memory).

It is split into four components:

- Client library: API for writing key–value or arbitrary data to a file, which may be used by only the current application, or shared between all applications.
- Administration service: system for installing default values and configuration for the data storage for each application; backing up and restoring stored data; and implementing factory reset of data.
- Common object: used by the other components to access key–value databases through a caching layer.
- Health monitor: system under development to implement data recovery in the case of corruption or loss, using existing backups.

The GENIVI Persistence Management system only supports storage of data as byte arrays — applications must serialise and deserialise their data formats themselves. Similarly, it does not implement versioning of stored data.

The data storage code is implemented as a set of plugins for the client library, implementing different methods for storing data. There are various types of plugins implementing layers of functionality such as hardware information querying, encryption, early loading of data, and the default storage backend.

Key–value data is limited to 16KB per key. Keys are stored per-application, namespaced by an application-chosen arbitrary identifier. As persistent data is stored in a separate file per application, Unix users and groups may be used to enforce access control on the persisted data.

GENIVI has investigated providing an SQLite API for relational data storage, and has provided recommendations for it[25], but has not shipped a version with SQLite support (as of version 0.3.0 of this document).

To persist an application's state, the developer must manually implement serialisation and deserialisation of all UI and internal state of the application using the Persistence client library.

4.4.2 SECRETS AND PASSWORDS

Similarly, GENIVI has no specialised API for storing secrets and passwords — applications must use the persistence management system. The system does allow for encrypted storage of persistent data using a plugin — but that encrypts all stored data, including preferences and application state.

24 http://docs.projects.genivi.org/persistence-client-library/1.0/Persistence_ArchitectureManual.pdf
25 http://docs.projects.genivi.org/persistence-client-library/1.0/Persistence_ClientLibrary_UserGuide.pdf, §11.7

5 APPROACH

Preferences and persistent data have largely separate requirements: preferences are small amounts of data; need to be accessed by multiple components; will typically be read much more frequently than they are written; and need to support features like vendor overrides (section 3.13) and vendor lockdown (section 3.14). Persistent data may vary from small to large amounts of data; will be read *and* written frequently; in app-specific formats; and do not need to be accessed by other components.

The expected amount of data to be stored, and the relative frequency of reads and writes of that data, is an important factor in the choice of storage format to use. Preferences should be stored in a format which is optimised for reads; persistent data should be stored in a format which is optimised for frequent reads and writes, since apps should update it frequently as they may be killed at any time.

For these reasons, we suggest preferences and persistent data are handled entirely separately. The following sections will cover them separately, giving our recommended approach and justifications which refer back to the requirements (section 3).

User secrets and passwords (section 3.16) have different requirements again:

- Confidentiality in storage (encryption).
- Sharing secrets and passwords for a given resource (such as website) between all applications using that website (i.e. secrets and passwords are not necessarily specific to an application, while preferences typically are).
- No fixed schema: the credentials required to access a given service (such as website) may change over time as that service changes.

As the system explicitly does not support full-disk encryption (for performance reasons), user secrets and passwords should be stored via the freedesktop.org Secrets D-Bus API[26], rather than the preferences or persistence APIs. The Secrets D-Bus API explicitly handles encryption of the secret store, whereas a general design for a preferences system should have no need for encryption, and hence adding it to the API would be an unnecessary complication for 90% of the use cases. Accordingly, confidential data will not be considered in the approach below.

For further discussion and designs on the topic of secrets and passwords, see §13.1 of the Security design document[27].

5.1 PREFERENCES ARCHITECTURE

Access to app, user and system settings should be through the GSettings API, most likely backed by dconf. (Refer to section 4.1 for an overview of the way GSettings and dconf fit together.) As system settings are defined as those settings which are accessed by multiple components, settings which are solely for the use of a single system service may be stored in other ways, and are beyond the scope of this document.

Each component should have its own GSettings schema:

26 http://standards.freedesktop.org/secret-service/
27 This document refers to version 1.1.4 of the Security design.

- **App schemas**: In the form `net.example.MyApplication.SchemaName`. Each app may have zero or more schemas, but all must be prefixed by the app ID (in this case, `net.example.MyApplication`; see the Applications Design document for details on the application ID scheme) to provide a level of namespacing.
- **User schemas**: These may have any form, and will typically re-use existing cross-desktop schemas, such as `org.gnome.system.locale`, as these are supported by many existing software components used by Apertis.
- **System schemas**: These may have any form, similarly.

Schema files for apps should be packaged with their app. For user services, they could be packaged with the most relevant service, or in a general purpose `gsettings-desktop-schemas` package (adapted from Debian) and an accompanying `apertis-schemas` package for Apertis-specific schemas.

All reads and writes of all settings should go through the normal GSettings interface — leaving access controls and policy to be implemented in the backend. App code therefore does not need to treat reads and writes differently, or treat app, user and system settings differently.

The use of GSettings also means that a single schema may be instantiated at multiple schema paths. Typically, a schema will only be instantiated at the path matching its ID; but a *relocatable* schema may be instantiated at other paths. This can be used to store settings for multiple accounts, for example.

It is expected that each app will handle any upgrades to its preference schemas, for example from one major version of the app to the next (section 3.4). Apertis will not provide any special APIs for this. As this is highly dependent on the structure of the preference keys an app is storing, Apertis can provide no recommendations here. Note, however, that GSettings is designed with upgradability in mind: new preference keys take their value from the schema-provided defaults until the user sets them; the values for old preferences which are no longer in the schema are ignored. It is recommended that the type or semantics of a given GSettings key is not changed between versions of an app bundle — if it needs to be changed, stop using the old key, migrate its stored value to a new key, and use the new key in newer versions of the app bundle.

Requirements

Through the use of the GSettings API, the following requirements are automatically fulfilled:

- 3.2: Writability — using `g_settings_is_writable()`
- 3.4: System updates — Old keys are either kept, or superseded by new keys with migrated values if their type or semantics change
- 3.5: Factory reset — for individual keys, using `g_settings_reset()`; support for resetting entire schemas needs to be supported by the designs below
- 3.6: Abstraction level — GSettings serves as the abstraction layer, with the individual backends below adding no further abstractions
- 3.9: Transactional updates — GSettings provides `g_settings_delay()`,

`g_settings_apply()` and `g_settings_revert()` to implement in-memory transactions which are serialised in the backend on calling apply

- 3.12: Race conditions — `g_settings_get()` automatically returns the default value if no user-set value exists; there is no atomic API for setting settings
- 3.15: User interface — `g_settings_bind()` can be used to bind a GSettings key to a particular UI widget, allowing interface UIs to be built easily (noting the argument in section 5.1.5 that preferences UIs should not be automatically generated)

5.1.1 PROXIED DCONF BACKEND

In its current state (May 2015, detailed in section 4.1), dconf does not support the necessary fine-grained access controls for multiple components accessing the preferences. However, a design is being implemented upstream to proxy access to dconf through a separate service which imposes access controls based on AppArmor (intended to be ready by the end of May 2015).

On the assumption that this work can be completed and integrated into Apertis on an appropriate timescale, this leads to a design where the dconf daemon runs as a system service, storing all settings in one database file per default layer:

- **App database**: `/Applications/`*`net.example.MyApplication/username`*`/config/dconf/app`
- **User database**: `~/.config/dconf/user`
- **System database**: `/etc/dconf/db/local`

This would be implemented as the dconf profile:

```
user-db:user
file-db:/Applications/net.example.MyApplication/username/config/dconf/app
system-db:local
```

All accesses to dconf would go through GSettings, and then through the proxy service which applies AppArmor rules to restrict access to specific settings, implementing the chosen security policy (section 3.1). The rules may, for example, match against settings path and the AppArmor label of the calling process.

The proxy service would therefore implement a system preferences service.

Vendor lockdown (section 3.14) is supported already by dconf[28] through the use of lockdown files, which specify particular keys or settings sub-trees which may not be modified.

Rollback (section 3.3) is supported by having one database file per (user, app) pair, which can be snapshotted and rolled back using the normal app snapshot mechanism described in the Applications Design. dconf will detect the rollback of the database and reload it.

Resetting all system settings would be a matter of deleting the appropriate databases — the keys in that database will revert to the default values provided by the schema files. As

28 https://developer.gnome.org/dconf/unstable/dconf-overview.html#id-1.2.7

this is a simple operation, it does not have to be implemented centrally by a preferences service. Resetting the value of an individual key is supported by the `g_settings_reset()` API, which is already implemented as part of GSettings.

The existing Apertis system puts

```
#include <abstractions/gsettings>
```

in several of the AppArmor profiles, which gives unrestricted access to the user dconf database. This must change with the new system, only allowing the dconf daemon access to the database.

Requirements

This design fulfills the following requirements:

- 3.1: Access permissions — through use of the proxy service and AppArmor rules
- 3.3: Rollback — by rolling back the user's per-app database
- 3.5: Factory reset — by deleting the user's database or the user's per-app database
- 3.7: Minimising I/O bandwidth — dconf's database design is optimised for this
- 3.8: Atomic updates — dconf performs atomic overwrites of the database
- 3.10: Performance tradeoffs — dconf is heavily optimised for reads rather than writes
- 3.11: Data size tradeoffs — dconf uses GVDB for storage, so can handle small to large amounts of data
- 3.13: Vendor overrides — dconf supports vendor overrides inherently
- 3.14: Vendor lockdown — dconf supports vendor lockdown inherently

5.1.2 DEVELOPMENT BACKEND

In the interim, we recommend that the standard dconf backend be used to store all system, user and app settings. This will *not* allow for access controls to be applied to the settings (requirement 3.1), but will allow for app development against the final GSettings interface.

Once the proxied dconf backend is ready, it can be packaged and the system configuration changed — no changes should be necessary in user services or apps to make use of the changed backend.

This development backend would support vendor lockdown as normal. It would support resetting all settings at once, but would not support resetting an individual app's settings (or rolling them back) independently of other apps, as all settings are stored in the same dconf database file.

Requirements

This design fails the following requirements:

- 3.1: Access permissions — **unsupported** by the current version of dconf

- 3.3: Rollback — **unsupported** by the current version of dconf

It supports the following requirements:

- 3.5: Factory reset — **partially supported** by deleting the user's database; resetting a (user, app) pair is not supported as all settings are stored in the same dconf database file
- 3.7: Minimising I/O bandwidth — dconf's database design is optimised for this
- 3.8: Atomic updates — dconf performs atomic overwrites of the database
- 3.10: Performance tradeoffs — dconf is heavily optimised for reads rather than writes
- 3.11: Data size tradeoffs — dconf uses GVDB for storage, so can handle small to large amounts of data
- 3.13: Vendor overrides — dconf supports vendor overrides inherently
- 3.14: Vendor lockdown — dconf supports vendor lockdown inherently

5.1.3 KEY-FILE BACKEND

As an alternative, if it is felt that the development backend is too simplistic to use in the interim before the proxied dconf backend is ready, the GSettings key-file backend could be used. This would allow enforcement of access controls via AppArmor, at the cost of:

- lower read performance due to not being optimised for reads (or in general);
- requiring code changes in user services and apps to switch from the key-file backend to the proxied dconf backend once it's ready;
- requiring settings values to be migrated from the key-file store to dconf at the time of switch over;
- not supporting vendor lockdown or vendor overrides.

Due to the need for code changes to switch away from this backend to a more suitable long-term solution such as the proxied dconf backend, we do not recommend this approach.

In detail, the approach would be to use a separate key file for each schema instance, across all system services, user services and apps. This would require using `g_settings_key_file_backend_new()` and `g_settings_new_with_backend_and_path()` to manually construct the GSettings instance for each schema, using a key file path which corresponds to the schema path.

Access control for each schema instance would be enforced using AppArmor rules which restrict access to each key file as appropriate. For example, apps would be given read-only access to the key files for system and user settings, and read–write access to the key file for their own app settings.

Vendor lockdown would be supported by vendors patching the AppArmor files to limit write access to specific schema instances. It would not support per-key lockdown at the granularity supported by dconf.

This code for creating the GSettings object could be abstracted away by a helper library,

but the API for that library would have to be stable and supported indefinitely, even after changing the backend.

Requirements

This design fails the following requirements:

- 3.10: Performance tradeoffs — GKeyFile is **equally non-optimised** for reads and writes
- 3.13: Vendor overrides — **unsupported** by GKeyFile
- 3.14: Vendor lockdown — **unsupported** by GKeyFile

It supports the following requirements:

- 3.1: Access permissions — supported by AppArmor rules on the per-schema key files
- 3.3: Rollback — by snapshotting and rolling back the appropriate key files
- 3.5: Factory reset — by deleting the appropriate key files
- 3.7: Minimising I/O bandwidth — GKeyFile's I/O bandwidth is proportional to the number of times each key file is loaded and saved
- 3.8: Atomic updates — GKeyFile performs atomic overwrites of the database
- 3.11: Data size tradeoffs — GKeyFile's load and save performance is proportional to the amount of data stored in the file, so it is suitable for small amounts of data

5.1.4 SECURITY POLICY

All three potential backends enforce security policy through per-app AppArmor rules (if they support implementing security policy at all — the development backend, section 5.1.2, does not).

It is beyond the scope of this document to define how each app ships its AppArmor rules, and how Apertis can guarantee that third-party apps cannot grant themselves higher privileges using additional rules. The suggestion in section 8.3 of the Applications Design document is for the AppArmor rule set for an app to be automatically generated from the app's manifest file by the app store (which is trusted). The manifest file could contain permissions such as 'can-change-locale' or 'can-add-network' which would translate to AppArmor rules allowing an app write access to the relevant user and system settings.

Additionally, by generating AppArmor rules from an app's manifest, the precise format of the AppArmor rules is abstracted, allowing the preferences backend to be switched in future (just as app access to preferences is abstracted through GSettings).

5.1.5 USER INTERFACE

We recommend that each app implement its own preferences interface, as a separate window in the app, or in some other manner in the app its UI designers feel is appropriate.

This preferences UI should be launchable via an action which is triggered by the

`ActivateAction` method of the `org.freedesktop.Application` D-Bus interface[29] When triggered, this action launches the app with the preferences UI shown, or brings the preferences UI to the foreground if the app is already running. This gives a method for a system preferences app to list installed apps and launch their preferences centrally, if desired. If an app supports this action, it must set a `supports-show-preferences` flag in its manifest, so that the system preferences app knows which apps to list.

The system preferences app should present system and user preferences which are not specific to any app — for example, language or background settings. It would be a separate app binary, pre-installed, and with security policy allowing it to set user and system settings. Use of the system preferences app may be restricted to administrative users (for example, the car's owner) — the policy about which users may run the system preferences app is beyond the scope of this document, which assumes that if the system preferences app is running, it may modify system and user settings.

All preferences UIs should be designed manually, and *must not* be automatically generated from the GSettings schema files. To do so is tempting, and can rapidly produce rough drafts of preferences UIs, but in our experience can never result in a high-quality finished UI with:

- logically grouped options;
- correctly aligned controls;
- a concept of which preferences are most important, which ones are ‘advanced’, and which ones should be hidden;
- conditional defaults (for example, when you set up IMAP e-mail, the default port should be 143, except if you have selected old-style SSL in which case it should be 993); and
- the ability to hide or disable preferences that do not apply because of the value of another preference (for example, if you switch off Bluetooth completely, then the widget to change the name that is broadcast over Bluetooth should be hidden or disabled).

If the uniform appearance of preferences UIs is a concern, this should be addressed through convention, the default appearance of widgets in the UI toolkit, and the use of a set of human interface guidelines such as the GNOME HIG[30]. Specifically, we recommend that preferences are:

- integrated into the main application UI if there are only a small number of them;
- instant-apply unless doing so would be dangerous[31], in which case they should be explicit-apply for all preferences in the dialogue (for example, changing monitor resolutions is dangerous, and hence is explicit-apply); and
- grouped logically in the UI.

If, after the preferences UIs of several applications have been implemented, some common widget patterns have been identified, we suggest that they could be abstracted out into

29 http://standards.freedesktop.org/desktop-entry-spec/desktop-entry-spec-latest.html#dbus
30 https://developer.gnome.org/hig/stable/dialogs.html.en
31 https://developer.gnome.org/hig/stable/dialogs.html.en#instant-and-explicit-apply

new widgets in the UI toolkit. The goal of this would be to increase consistency between preferences UIs, without implementing essentially a separate UI toolkit for them, which would be the result of any template- or auto-generation-based approach.

Alternative model 1: embedding preferences for individual applications in the system preferences application

Preferences for individual apps could instead be presented together in the system preferences app. This is not recommended, as it has various disadvantages:

- Increased complexity: there would need to be some way of embedding the UI from the app into the system preferences app, similar to Android Preference Fragments[32] (see below).
- Decreased flexibility for app authors to present preferences as best suits the app's UI design.
- Moves the preferences away from the app to somewhere the user may not immediately look for them.

The advantage it provides is extremely tight integration of application preferences into the system preferences application.

There are three ways to embed a preferences UI from an application (A) into the system preferences app (B):

1. Run code from A in B in order to build and run the preferences UI. As A is untrusted, this allows potentially malicious code to be run with the (presumably raised) privileges of the system preferences application, and with no effective way of keeping the security domains separate, as that's only possible using a process boundary.
2. Define the preferences UI for A statically, for example by generating it from the GSettings schema for A. Implement code in B to generate a UI from this definition. This has the problems of generated preference UIs described above.
3. Use a nested Wayland compositor to embed the UI from A in B while keeping A as a separate process. If B implements a nested Wayland compositor, and all applications which need to display preferences there connect to it using the Wayland protocol, they can send their rendered UI to B, and it can return input events to them to be handled.

The third approach is the best balance of security and feasibility, but would be complex to implement — it requires designing and implementing an extension to the Wayland protocol for handling the sub-compositing. It is for this reason that we do not recommend the overall approach of embedding preferences for individual apps in the system preferences app.

If this model were to be implemented, it would require a more thorough security evaluation first.

---

32 http://developer.android.com/guide/topics/ui/settings.html#Fragment

Alternative model 2: linking to individual applications from the system preferences application

As the first alternative model is complex to implement, a simpler alternative which presents much the same user experience may be for the system preferences application to list applications which have preferences UIs (as determined using their manifests, as described in section 5.1.5). When the user clicks on a listed application, the system preferences application would launch that application's preferences UI, which would run as described in section 5.1.5. When the user is finished editing the preferences for that application, the back button should exit the application and hence return focus to the system preferences application — implementing this could not be enforced by the system preferences application, and would have to be a matter of convention (perhaps checked when an application is submitted to the Apertis store).

This design has the advantage of giving reasonably tight integration of application preferences into the system preferences application, while maintaining separated security domains, and not requiring a complex implementation. It has the disadvantage that it relies on applications to correctly implement the 'back' behaviour once a user has finished editing preferences.

5.1.6 EXISTING PREFERENCES SCHEMAS

As GSettings is used widely within the open source software components used by Apertis, particularly GNOME, there are many standard GSettings schemas for common user settings. We recommend that Apertis re-use these schemas as much as possible, as support for them has already been implemented in various components. If that is not possible, they could be studied to ensure we learn from their design successes or failures.

- `org.gnome.system.locale`
- `org.gnome.system.proxy`
- `org.gnome.desktop.default-applications`
- `org.gnome.desktop.media-handling`
- `org.gnome.desktop.interface`
- `org.gnome.desktop.lockdown`
- `org.gnome.desktop.background`
- `org.gnome.desktop.notifications`
- `org.gnome.crypto`
- `org.gnome.desktop.privacy`
- `org.gnome.system.dns_sd`
- `org.gnome.desktop.sound`
- `org.gnome.desktop.datetime`
- `org.gnome.system.location`
- `org.gnome.desktop.thumbnailers`
- `org.gnome.desktop.thumbnail-cache`

- `org.gnome.desktop.file-sharing`

Various Apertis dependencies (for example, Mutter, Tracker, libfolks, IBus, Geoclue, Telepathy) use their own GSettings schemas already — as these are not shared, they are not listed.

Alternative model: If the locale is a system setting, rather than a user setting, systemd's localed[33] should be used. This would require the locale to be changed via the localed D-Bus API, rather than GSettings, which would affect the implementation of the system preferences app.

5.2 PERSISTENT DATA ARCHITECTURE

As discussed in sections 5.3.1 and 7 of the Applications Design, and the Multiuser Design, there is a difference between state which an app needs to persist (for example, if it is being terminated to switch users), and state which an app explicitly needs to share (for example, if a transactional user switch is taking place to execute an action as a different user). The Multiuser Design encourages app authors to think explicitly about these two sets of state, and the differences between them. It is the app which chooses the state to persist, rather than the operating system — storage space is too limited to persist the entire address space of an app, effectively suspending it.

The state each app chooses to persist will differ, and cannot be predicted by Apertis. There could be a lot of state, or very little. It could be representable as a simple key–value dictionary, or might have a complex hierarchical structure.

5.2.1 WELL-KNOWN STATE DIRECTORIES

As mentioned in the Applications Design document (sections 5.3.1 and 7), we recommend that Apertis provide a per-(user, app) directory for storage of persisted data, and a public API the app can call to find out that directory. The API should differentiate between cache and non-cache state, with cache state going in `$XDG_CACHE_HOME/net.example.MyApp/` and non-cache state going in `$XDG_DATA_HOME/net.example.MyApp/`. Alternatively, as suggested in the Applications Design, the latter could be `/Applications/net.example.MyApp/Storage/username/state/`. This has the advantage of allowing all data for a particular app to be removed by deleting `/Applications/net.example.MyApp`, at the cost of not following the XDG standard used by most existing software. This fulfils the factory reset requirement (3.5).

The former is effectively equivalent to a per-(user, app) `XDG_CACHE_HOME` directory, and the latter to a `XDG_DATA_HOME`, as defined by the XDG Base Directory Specification[34].

AppArmor rules should exist to allow apps to write to these directories (and not to other apps' state directories). This is the extent of the security needed, as state storage is simply an interaction between an app and the filesystem.

This approach automatically allows for rollback of persistent data (requirement 3.3) using the normal snapshotting mechanism described in the Applications Design document.

33 http://www.freedesktop.org/wiki/Software/systemd/localed/
34 http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html

As with preferences, app bundles must be in charge of upgrading their own persistent data when the system is upgraded (or the app is upgraded) (section 3.4). Recommendations are given in the subsections below.

5.2.2 RECOMMENDED SERIALISATION APIS

As each app's state storage requirements are different, we suggest that Apertis provide several recommended serialisation APIs, and allow apps to choose the most appropriate one — or something completely different if that fulfils their requirements better.

Alongside, Apertis should provide guidelines to app developers to allow them to choose an appropriate serialisation API, and avoid common problems in serialisation:

- minimise writes to secondary storage (requirement 3.7);
- ensure all updates to stored state are atomic[35] (requirement 3.8); and
- ensure transactions are used for groups of updates where appropriate (requirement 3.9).

Depending on the requirements it is believed that apps will have, some or all of the following APIs could be recommended for serialising state to secondary storage. For comparison, Android only provides a generic file storage API, and an SQLite API, with no implemented key–value store APIs[36]. Apps must implement those themselves.

GKeyFile

https://developer.gnome.org/glib/stable/glib-Key-value-file-parser.html

Suitable for small amounts of key–value state with simple types. Suitable for small amounts of data.

All updates to a GKeyFile are atomic, as it uses the atomic-overwrite technique: the new file contents are written to a temporary file, which is then atomically renamed over the top of the old file. Transactional updates can be implemented by saving the key file to apply the transaction, and discarding the in-memory GKeyFile object to revert it.

The amount of I/O with a GKeyFile is small, as the amount of data which should be stored in a GKeyFile is small, and the file is only written out when explicitly requested by the app.

System upgrades have to be handled manually by app bundles — if the persistence data format has to change, the app must migrate data from the old format to the new format the first time it is run after an upgrade. In this case, it is recommended that all GKeyFiles used for persistent data contain a 'Version' key specifying the data format version in use.

GVDB

https://git.gnome.org/browse/gvdb

Memory-mapped hash table with GVariant-style[37] types, suitable for small to large amounts of data which are read much more frequently than they are written. This is what

35 Atomic in the sense that either the old or new states are stored in entirety, rather than some intermediate state, if power is lost part-way through an update.

36 http://developer.android.com/guide/topics/data/data-storage.html

37 https://developer.gnome.org/glib/stable/glib-GVariant.html

dconf uses for storage.

All updates to a GVDB file are atomic, as it uses the same atomic-overwrite technique as GKeyFile (above). Transactions are supported similarly — by writing out the updated database or discarding it.

The amount of I/O for reads from a GVDB file is small, as it memory-maps the database, so only pages in the data it actually reads (plus some metadata). Writes require the entire file to be updated, but are only done when explicitly requested by the app.

GVDB supports per-file versioning (though this is not currently exposed in the public API). This can be used for handling system upgrades (section 3.4) — the database must be explicitly migrated from an old version to a new version when an upgraded app is first started.

SQLite

http://sqlite.org/

Full SQL database implementation, supporting simple SQL types and more complex relational types if implemented manually by the app. Suitable for medium to large amounts of data which are read and written frequently. It supports SQL transactions.

SQLite is not a panacea. It is designed for the specific use pattern of SQL databases with indexes and relational tables, with frequent reads and writes, and infrequent deletions of data. Apps will only get the best performance from SQLite by defining their own table structure, indices and relations; imposing a common key–value-style API on top of SQLite would give lower performance.

SQLite has limited support for SQL schema upgrades with its ALTER TABLE statement[38], which supports renaming tables and adding new columns to tables. Apps must implement their own data migration from old to new versions of their database schema; documenting this is beyond the scope of this design.

Apps should only use SQLite if they have considered issues like their vacuuming policy — how frequently to vacuum the database after deleting data from it. See:

- https://blogs.gnome.org/jnelson/2015/01/06/sqlite-vacuum-and-auto_vacuum/
- https://wiki.mozilla.org/Performance/Avoid_SQLite_In_Your_Next_Firefox_Feature

GNOME-DB

http://www.gnome-db.org/

This is **not** recommended. It is an abstraction layer over multiple SQL database implementations, allowing apps to access remote SQL databases. In almost all cases, directly using SQLite (above) is a more appropriate choice.

5.2.3 WHEN TO SAVE PERSISTENT DATA

As specified in the Applications Design (section 5.3.1), state is saved to secondary storage

38 https://www.sqlite.org/lang_altertable.html

at times chosen by both the operating system and the app. The operating system knows when the logged in user is about to change, or when the system is about to be shut down; the app knows when it has changed some of its persistent state in memory, and hence needs to write it out to secondary storage.

An action could be implemented in each app which is triggered by the `ActivateAction` method of the `org.freedesktop.Application` D-Bus interface[39] if, for example, that interface is implemented by apps. When triggered, this action would cause the app to store its persistent state.

5.2.4 RECENTLY USED AND FAVOURITE ITEMS

Section 6.3 of the Global Search Design specifies that an API for apps to store their favourite and recently used items in will be provided. As this is data shared from an app to the operating system, and is typically append-only rather than strongly read–write, Collabora recommends that it be designed separately from the persistent data API covered in this document, following the recommendations given in the Global Search Design document.

39 http://standards.freedesktop.org/desktop-entry-spec/desktop-entry-spec-latest.html#dbus

6 SUMMARY OF RECOMMENDATIONS

As discussed in the above sections, Collabora recommends:

- Splitting preferences, persistent data storage and confidential data storage (section 5).
- Providing one API for preferences: GSettings (section 5.1).
- Apps provide a GSettings schema file for their preferences, named after the app (section 5.1).
- Existing GSettings schemas are re-used where possible for user and system settings (section 5.1.6).
- Using the normal GSettings approach for handling app upgrades (section 5.1).
- Developing against the normal dconf backend for GSettings (section 5.1.2).
- Switching to the proxied dconf backend once it's ready, to support access control (section 5.1.1).
- A key-file backend is an alternative Collabora does *not* recommend (section 5.1.3).
- Permissions to modify user or system settings are controlled by the app's manifest (section 5.1.4).
- Permissions are converted to backend-specific AppArmor rules by the app store (section 5.1.4).
- User interfaces for preferences are designed manually (section 5.1.5).
- Each app implements its preferences UI in its main binary, and provides a GApplication action to show it (section 5.1.5).
- Providing API to get a persistent data storage location (section 5.2.1).
- Persistent data is private to each (user, app) pair (section 5.2.1).
- Recommending various different data storage APIs to suit different apps' use cases (section 5.2.2).
- Apps explicitly define which data will persist, and are responsible for saving it and migrating it from older to newer versions (section 5.2).
- Apps can be instructed to save their persistent state by the operating system via a D-Bus interface (section 5.2.3).
- User secrets and passwords are stored using the freedesktop.org Secrets D-Bus API, not the Apertis preferences or persistence APIs (section 5).