



Preferences and persistence

1	Contents	
2	Introduction	4
3	Terminology and concepts	4
4	System Settings	4
5	User settings	4
6	App settings	4
7	Preferences	5
8	User services	5
9	Persistent data	5
10	Main storage	5
11	GSettings	6
12	AppArmor	6
13	Requirements	6
14	Access permissions	6
15	Writability	7
16	Rollback	7
17	System and app bundle upgrades	7
18	Factory reset	7
19	Abstraction level	8
20	Minimising I/O bandwidth	8
21	Atomic updates	8
22	Transactional updates	8
23	Performance tradeoffs	8
24	Data size tradeoffs	9
25	Concurrency control	9
26	Vendor overrides	9
27	Vendor lockdown	9
28	User interface	9
29	Control over user interface	10
30	Rearrangeable preferences	10
31	Searchable preferences	10
32	Storage of user secrets and passwords	10
33	Preferences hard key	10
34	Existing preferences systems	10
35	GNOME Linux desktop	10
36	Preferences	10
37	Persistent data	11
38	Secrets and passwords	12
39	Android	12
40	Preferences	12
41	Persistent data	13
42	Secrets and passwords	14

43	ios	14
44	Preferences	14
45	Persistent data	15
46	Secrets and passwords	16
47	GENIVI	16
48	Preferences and persistent data	16
49	Secrets and passwords	17
50	Approach	17
51	Preferences approach	18
52	Overall architecture	18
53	Requirements	20
54	Proxied dconf backend	20
55	Requirements	21
56	Development backend	22
57	Requirements	22
58	Key-file backend	23
59	Requirements	24
60	Security policy	24
61	Application access to system settings	24
62	User interface	25
63	System preferences application	25
64	Per-application preferences windows	26
65	Generating a preferences window from a GSettings schema file	27
66	Support for custom preferences windows	28
67	Searchability of preferences	29
68	Reorganising preferences	29
69	Preferences list widget	29
70	Vendor lockdown	30
71	Discussion of automatically generated versus manually coded	
72	preferences UIs	30
73	Preferences hard key	31
74	Existing preferences schemas	32
75	Persistent data approach	33
76	Overall architecture	33
77	Well-known state directories	33
78	Recommended serialisation APIs	34
79	GKeyFile	34
80	GVDB	35
81	SQLite	35
82	GNOME-DB	36
83	When to save persistent data	36
84	Recently used and favourite items	36

86 Introduction

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

91 The Applications Design, and Global Search Design documents are relevant
92 reading. The [Applications Design](#)¹ and the [Global Search Design](#)² reference
93 the need for storage of persistent data for apps. See [Overall architecture](#) for a
94 design covering this.

95 The [Robustness Design](#)³ document gives more detail on the requirements for
96 robustness of main storage in the face of power loss.

97 Terminology and concepts

98 System Settings

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

107 User settings

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

110 App settings

111 An *app setting* is one which varies by user and also by an [app bundle](#)⁴. App
112 settings apply only to a specific app bundle, and would not make sense outside

¹<https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application/applications/>

²<https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application/global-search/>

³https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application_security/robustness/

⁴<https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application/applications/#bundle>

113 the context of that app. For example, whether to enable shuffling tracks in the
114 media player; whether to open hyperlinks in a new tab by default in the web
115 browser; or the details for accessing a user's e-mail account.

116 Preferences

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

119 User services

120 A *user service* is as defined in the Multiuser Design document —a service that
121 runs on behalf of a particular user. Throughout this document, this is addition-
122 ally assumed to mean a *platform* user service, which is not tied to a particular
123 app-bundle. Services inside of app bundles have the same access to settings as
124 the app's UI.

125 Persistent data

126 Persistent data is app state which persists across multiple user sessions. For ex-
127 ample, documents which the user has written, or the state of the user's pending
128 downloads.

129 One distinguishing factor between preferences and persistent data is that ven-
130 dors may override the default values for preferences (see **Vendor overrides**), but
131 not for persistent data. For example, a vendor would not want to override in-
132 formation about in-progress downloads; but they might want to override the
133 default background image filename for a user.

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

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

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

145 Main storage

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

149 GSettings

150 [GSettings](#)⁵ is an interface provided by GLib for accessing settings. As an in-
151 terface, it can be backed by different storage backends —the most common is
152 dconf, but a key file backend is available for storage in simple key files.

153 GSettings uses a concept of ‘schemas’, which define available settings, their data
154 types, and their default values. Each setting is strictly typed and must have a
155 default value. A schema has an ID, and is ‘instantiated’ at one or more schema
156 paths. Typically, a schema will be instantiated at a single path, but may be
157 instantiated at multiple paths to support storing the same settings for multiple
158 objects. For example, a schema for an e-mail account could require a server
159 name, username and protocol, and be instantiated at [multiple paths](#)⁶, one path
160 for each configured e-mail account.

161 AppArmor

162 [AppArmor](#)⁷ is an access control framework used by Apertis to enforce fine-
163 grained permissions across the entire system, restricting which files each process
164 can open.

165 Requirements

166 Access permissions

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

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

- 174 • **Apps** may read their own app settings, user settings for the current user,
175 and all system settings.
- 176 • **User services** may read the user’s application settings, user settings for
177 the current user, and all system settings.
- 178 • **System services** may read their own app settings, and all system set-
179 tings.

180 **Writes** are generally only allowed at the same level:

- 181 • **Apps** may write their own app settings.

⁵<https://developer.gnome.org/gio/stable/GSettings.html#GSettings.description>

⁶<https://developer.gnome.org/gio/stable/GSettings.html#gsettings-relocatable>

⁷<http://apparmor.net/>

- 182 • **User services** may write user settings for the current user.
- 183 • **System services** may write system settings for all users, user settings for
184 any user, and app settings for any app for any user.

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

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

189 Writability

190 As well as the value of a preference, components must be able to find out whether
191 the preference is writable. A preference may be read-only if the component
192 doesn't have write permission for it ([Access permissions](#)) or if it is locked down
193 by the vendor [vendor lockdown](#)).

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

196 Rollback

197 As discussed in the [Applications Design document](#)⁸, applications may be rolled
198 back to a previously installed version, but it is the application's responsibility
199 to handle any preference changes that occurred in a future version.

200 System and app bundle upgrades

201 As per the [Applications Design](#)⁹ and the [System Update and Rollback design](#)¹⁰,
202 applications must also support upgrading preferences and persistent data from
203 previous application versions to the current version.

204 Factory reset

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

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

⁸<https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application/applications/#roll-back>

⁹<https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application/applications/>

¹⁰<https://apertis-website-0b3586.pages.apertis.org/concepts/platform/system-updates-and-rollback/>

212 **Abstraction level**

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

219 Conversely, any more than one layer of abstraction is an unnecessary complica-
220 tion.

221 **Minimising I/O bandwidth**

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

226 **Atomic updates**

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

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

235 See the Robustness Design document, §3.1.1 for more details on general robust-
236 ness requirements.

237 **Transactional updates**

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

241 **Performance tradeoffs**

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

246 Data size tradeoffs

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

251 Concurrency control

252 As system preferences may affect security policy, reading them should be race
253 free, particularly from [time-of-check-to-time-of-use](#)¹¹ race conditions. For exam-
254 ple, if a preference is changed by process C while process R is reading it, process
255 R must either see the new value of the preference, or see the old value of the
256 preference *and* subsequently be notified that it has changed.

257 Similarly for persistent data.

258 Vendor overrides

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

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

267 This does not apply to persistent data.

268 Vendor lockdown

269 It may also be desirable to support *vendor lockdowns*, where a vendor shipping
270 Apertis can lock a preference so that end users or non-privileged applications
271 may not change it. For example, they may wish to lock the URI which is checked
272 for system updates.

273 This does not apply to persistent data.

274 User interface

275 There must be some user interface (UI) for setting preferences. This may be
276 provided by a system preferences application, as a separate window in each appli-
277 cation, or as individual widgets embedded throughout an application's interface;
278 or a combination of these options.

279 This does not apply to persistent data.

¹¹http://en.wikipedia.org/wiki/Time_of_check_to_time_of_use

280 **Control over user interface**

281 It must be possible for the vendor to have complete control over the way pref-
282 erences are presented if all applications' preferences are presented in a system
283 preferences application.

284 This does not apply to persistent data.

285 **Rearrangeable preferences**

286 It must be possible for a vendor to rearrange the preferences from applications
287 if they are presented in a system preferences application, so that (for example)
288 all 'privacy' preferences are presented in a page together.

289 **Searchable preferences**

290 It must be possible for a system preferences application provided by the vendor
291 to allow the user to search all preferences from all applications.

292 **Storage of user secrets and passwords**

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

296 **Preferences hard key**

297 There must be support for a preferences hard key (a physical button in the vehi-
298 cle) which when pressed causes the currently active application's settings to be
299 displayed. If no applications are active, it could display the system preferences.
300 Some vehicles may not have such a hard key, in which case the functionality
301 should be ignored.

302 **Existing preferences systems**

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

308 **GNOME Linux desktop**

309 **Preferences**

310 On a modern GNOME desktop, from which Apertis uses a lot of components,
311 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`¹² supports multiple layered databases, 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`¹³ 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 app accessing the daemon may write to any settings key, either its own app settings, another app's settings, or the user's settings.

Persistent data

Persistent data is stored in application-defined formats, in application-defined locations, although many follow the [XDG Base Directory Specification](#)¹⁴, which puts cache data in `XDG_CACHE_HOME` (typically `~/.cache`) and non-cache

¹²<https://developer.gnome.org/dconf/unstable/dconf-overview.html>

¹³<https://developer.gnome.org/dconf/unstable/dconf-overview.html#id-1.2.7>

¹⁴<http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html>

349 data in XDG_DATA_HOME (typically ~/.local/share). Below these two direc-
350 tories, applications create their own directories or files as they see fit. There is
351 no security separation between applications, but the normal UNIX permissions
352 restrict access to only the current user.

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

356 Secrets and passwords

357 On a GNOME or KDE desktop, all user secrets, passwords and credentials are
358 stored using the [Secret Service](#)¹⁵ API. In GNOME, this API is implemented by
359 GNOME Keyring; in KDE, by KWallet.

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

367 An application must open a session with the secret service in order to access
368 secrets. The session may be used to encrypt secrets while they are in tran-
369 sit between the service and application, and allows for encryption algorithm
370 negotiation for this purpose.

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

375 Android

376 Preferences

377 Apps can use the [SharedPreferences class](#)¹⁶ to read and write preferences from
378 named preferences files, with apps typically using a single preferences file with
379 a default name. These files are stored per-app, and are private to that app by
380 default, but may be shared with other apps, either read-only or read-write.

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

384 Preferences are saved to disk immediately.

¹⁵<https://specifications.freedesktop.org/secret-service/latest/index.html>

¹⁶<http://developer.android.com/guide/topics/data/data-storage.html#pref>

385 Android uses a [custom XML format](#)¹⁷ to allow apps to define preference UIs
386 (known as ‘activities’ in Android terminology). This format can define simple lists
387 of preferences, through to complex UIs with grouped preferences, subscreens,
388 lists of subscreens, and custom preference widgets. Implementing features such
389 as making one preference conditional on another is possible, but requires com-
390 plex XML.

391 A [PreferenceFragment](#)¹⁸ can be used to automatically build a screen in an ap-
392 plication to display preferences, loading them from the XML file. It will load
393 the current values of the preferences from the SharedPreferences store, and will
394 write new values back to the store as the preferences are modified in the UI.

395 In order for the system to display the preferences for a particular application,
396 it must execute one or more of the PreferencesFragment classes from that ap-
397 plication.

398 Persistent data

399 Android offers several options for [persistent data](#)¹⁹:

- 400 • **Internal storage:** Files in a per-(user, app) directory, which may option-
401 ally be made world-readable or writable to allow access to other apps or
402 users (though this is strongly discouraged).
- 403 • **External storage:** Files in a world-readable storage area which is
404 accessible to the user, such as an SD card. Accessible to all other
405 apps and users which hold the READ_EXTERNAL_STORAGE or
406 WRITE_EXTERNAL_STORAGE permissions.
- 407 • **SQLite database:** Arbitrary app-defined tables in a per-(user, app)
408 SQLite database. This cannot be shared with other apps or users.
- 409 • **Network connection:** Using the normal networking APIs, Android sug-
410 gests that data can be stored on servers controlled by the app developers.
411 It provides no special API for this.

412 For saving an application’s state, Android offers a persistence API on the [Ac-
413 tivity class](#)²⁰. This automatically saves the state of all UI elements (such as
414 the text in an entry widget, and the position of a list), but cannot automati-
415 cally save application-specific internal state (member variables). For this, the
416 application must override two toolkit methods (onSaveInstanceState() and on-
417 RestoreInstanceState()) and implement its own serialisation and deserialisation
418 of state to a set of key-value pairs which are then stored by Android.

¹⁷<http://developer.android.com/guide/topics/ui/settings.html#DefiningPrefs>

¹⁸<http://developer.android.com/guide/topics/ui/settings.html#Fragment>

¹⁹<http://developer.android.com/guide/topics/data/data-storage.html>

²⁰<http://developer.android.com/training/basics/activity-lifecycle/recreating.html>

419 Secrets and passwords

420 Android recommends storing secrets and passwords in two ways. For authentication credentials for online services, it provides an AccountManager API²¹ 421 which abstracts authentication for known online services (which are supported 422 by pluggable backends, potentially provided by application bundles) and stores 423 the credentials in an OS-wide store. The service handles authenticating and 424 re-authenticating when the login session ends. 425

426 For secrets which are not for online accounts, or otherwise do not fit the Account- 427 Manager pattern, Android recommends²² using the normal preferences API (428 Preferences), as while preferences are not encrypted in storage, they are only 429 accessible to the application which owns them, so cannot be stolen by other 430 applications. However, if the sandboxing system is compromised (potentially 431 by an attacker with physical access to the device), the stored secrets will be 432 accessible in plaintext.

433 iOS

434 Preferences

435 iOS stores preferences as key-value pairs²³, which are separated into domains 436 by user, application and machine. The same preference may be set in multiple 437 domains²⁴, and they are searched in a defined priority order to determine which 438 value to use. This means that an application may, for example, choose to share 439 a given preference between all users of that application on a given machine.

440 Application IDs use the standard reverse domain name syntax to ensure unique- 441 ness.

442 Preference values may be any type supported by Core Foundation property 443 lists²⁵, including strings, integers and arrays. Default values must be coded into 444 the application.

445 Preference keys may be generated at runtime by the application, and do not have 446 to be defined in a schema in advance. However, it is typical to use pre-defined 447 property lists.

448 Preferences are synchronised with the on-disk store manually, so the application 449 chooses when they are written to disk.

²¹<http://developer.android.com/reference/android/accounts/AccountManager.html>

²²<http://stackoverflow.com/questions/785973/what-is-the-most-appropriate-way-to-store-user-settings-in-android-application/786588#786588>

²³https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFPreferences/CFPreferences.html#//apple_ref/doc/uid/10000129-SW1

²⁴<https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFPreferences/Concepts/PreferenceDomains.html>

²⁵https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFPropertyLists/CFPropertyLists.html#//apple_ref/doc/uid/10000130i

450 On certain Apple operating systems, preferences may be ‘managed’ by the ad-
451 ministrator²⁶, setting an override value which overrides any value set by the
452 user for a given preference key.

453 Application preferences can either be presented as part of the application, using
454 normal UI widgets, and accessing the `NSUserDefaults` class²⁷ for the preference
455 values. Or they can be presented as part of the system-wide settings applica-
456 tion²⁸, which builds the UI for each application’s preferences dynamically from
457 that application’s property list file for preferences. An application may provide
458 multiple property list files to build a hierarchy of preferences pages. The system-
459 wide settings application accesses `NSUserDefaults` on behalf of the application
460 to update the stored preferences.

461 Persistent data

462 iOS offers several options for persistent data:

- 463 • **Filesystem:** Arbitrary files may be written to the filesystem in various
464 app-specific locations²⁹.
- 465 • **Core Data API:** This is an object-graph management API³⁰, which
466 allows versioned control of instances of objects created from a schema.
467 Instead of being used by an application to persist data, this API is designed
468 to form the core of the application’s data model. It supports editing and
469 discarding edits, undo, redo, versioning of the object schema, and large
470 data sets.
- 471 • **Property List API:** A property list is a hierarchical, structured piece
472 of data, consisting of primitive data types, arrays and dictionaries which
473 may be nested arbitrarily³¹. Property lists can therefore be used to store
474 arbitrary application data. There is an API to serialise them to the file
475 system.
- 476 • **SQLite:** The standard SQLite API may be used, backed by a file, to store
477 relational data in a database.

²⁶https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFPreferences/Concepts/BestPractices.html#//apple_ref/doc/uid/TP30001219-118191

²⁷https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSUserDefaults_Class/index.html#//apple_ref/occ/cl/NSUserDefaults

²⁸https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/UserDefault/Preferences/Preferences.html#//apple_ref/doc/uid/10000059i-CH6-SW6

²⁹https://developer.apple.com/library/ios/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/AccessingFilesandDirectories/AccessingFilesandDirectories.html#//apple_ref/doc/uid/TP40010672-CH3-SW11

³⁰https://developer.apple.com/library/prerelease/ios/documentation/DataManagement/Devpedia-CoreData/coreDataOverview.html#//apple_ref/doc/uid/TP40010398-CH28

³¹<https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/PropertyLists/AboutPropertyLists/AboutPropertyLists.html>

478 For persisting an entire application's state, iOS provides a [solution](#)³² similar
479 to [Android][Persistent data]. The developer must annotate each UI view
480 class which needs to be saved and restored, and the UI toolkit will automati-
481 cally persist the state of the widgets in that view when the application is sus-
482 pended. As with Android, the developer must implement two methods for
483 serialising and deserialising application-specific state from member variables:
484 `encodeRestorableStateWithCoder` and `decodeRestorableStateWithCoder`.

485 Secrets and passwords

486 iOS uses the same [keychain API](#)³³ as OS X. This provides a system service for
487 storing secrets, passwords and certificates. They are encrypted in storage, using
488 an encryption key which is derived from the iOS application's ID and the user's
489 password.

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

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

498 GENIVI

499 Preferences and persistent data

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

503 The GENIVI [Persistence Management system](#)³⁴ handles all data read and writ-
504 ten during the lifetime of an IVI system. It aims to provide a standard API
505 for all GENIVI platforms to use, which reliably stores data in the face of power
506 disturbances, and the limited write-cycle lifetime of some non-volatile storage
507 devices (flash memory).

508 It is split into four components:

- 509 • Client library: API for writing key-value or arbitrary data to a file, which

³²<https://developer.apple.com/library/ios/featuredarticles/ViewControllerPGforiPhoneOS/PreservingandRestoringState.html>

³³https://developer.apple.com/library/ios/documentation/Security/Conceptual/keychainServConcepts/01introduction/introduction.html#/appl_ref/doc/uid/TP30000897-CH203-TP1

³⁴http://docs.projects.genivi.org/persistence-client-library/1.0/Persistence_ArchitectureManual.pdf

510 may be used by only the current application, or shared between all appli-
511 cations.

512 • Administration service: system for installing default values and configura-
513 tion for the data storage for each application; backing up and restoring
514 stored data; and implementing factory reset of data.

515 • Common object: used by the other components to access key-value
516 databases through a caching layer.

517 • Health monitor: system under development to implement data recovery
518 in the case of corruption or loss, using existing backups.

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

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

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

530 GENIVI has investigated providing an SQLite API for relational data storage,
531 and has provided [recommendations for it](#)³⁵, but has not shipped a version with
532 SQLite support (as of version 0.3.0 of this document).

533 To persist an application's state, the developer must manually implement seri-
534 alisation and deserialisation of all UI and internal state of the application using
535 the Persistence client library.

536 Secrets and passwords

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

541 Approach

542 Preferences and persistent data have largely separate requirements: preferences
543 are small amounts of data; need to be accessed by multiple components; will
544 typically be read much more frequently than they are written; and need to

³⁵http://docs.projects.genivi.org/persistence-client-library/1.0/Persistence_ClientLibrary_UserGuide.pdf

545 support features like **Vendor overrides** and **vendor lockdown**. Persistent data may
546 vary from small to large amounts of data; will be read *and* written frequently;
547 in app-specific formats; and do not need to be accessed by other components.

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

554 For these reasons, we suggest preferences and persistent data are handled en-
555 tirely separately. The following sections (6 and 7) will cover them separately,
556 giving our recommended approach and justifications which refer back to the
557 requirements (section 3).

558 User secrets and passwords (**Storage of user secrets and passwords**) have differ-
559 ent requirements again:

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

566 As the system explicitly does not support full-disk encryption (for performance
567 reasons), user secrets and passwords should be stored via the freedesktop.org
568 **Secrets D-Bus API**³⁶, rather than the preferences or persistence APIs. The
569 Secrets D-Bus API explicitly handles encryption of the secret store, whereas a
570 general design for a preferences system should have no need for encryption, and
571 hence adding it to the API would be an unnecessary complication for 90% of the
572 use cases. Accordingly, confidential data will not be considered in the approach
573 below.

574 For further discussion and designs on the topic of secrets and passwords, see the
575 **Security design document**³⁷.

576 Preferences approach

577 Overall architecture

578 Access to app, user and system settings should be through the GSettings API,
579 through the dconf backend for system / user schemas and the key-file backend

³⁶<http://standards.freedesktop.org/secret-service/>

³⁷https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application_security/security/

for app schemas, the latter being chosen due to the integrated support for sandboxed settings in Flatpak. (Refer to [GNOME Linux desktop](#) 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:

- **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 ([System and app bundle upgrades](#)). 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.

620 Requirements

621 Through the use of the GSettings API, the following requirements are automat-
622 ically fulfilled:

- 623 • **Writability** —using `g_settings_is_writable()`
- 624 • **System and app bundle upgrades** —old keys are either kept, or superseded
625 by new keys with migrated values if their type or semantics change
- 626 • **Factory reset** —for individual keys, using `g_settings_reset()`; support for
627 resetting entire schemas needs to be supported by the designs below
- 628 • **Abstraction level** —GSettings serves as the abstraction layer, with the
629 individual backends below adding no further abstractions
- 630 • **Transactional updates** —GSettings provides `g_settings_delay()`,
631 `g_settings_apply()` and `g_settings_revert()` to implement in-memory
632 transactions which are serialised in the backend on calling apply
- 633 • **Concurrency control** —`g_settings_get()` automatically returns the default
634 value if no user-set value exists; there is no atomic API for setting settings
- 635 • **User interface** —`g_settings_bind()` can be used to bind a GSettings key
636 to a particular UI widget, allowing interface UIs to be built easily (not-
637 ing the argument in **User interface** that preferences UIs should not be
638 automatically generated)

639 Other requirements are fulfilled separately:

- 640 • **Control over user interface** —by generating preferences windows from GSet-
641 tings schemas in the system preferences application (**Searchable prefer-**
642 **ences**)
- 643 • **Rearrangeable preferences** —by hard-coding more behaviour in the system
644 preferences application (**User interface**)
- 645 • **Searchable preferences** —searching over summaries and descriptions in
646 GSettings schemas (**Security policy**)
- 647 • **Storage of user secrets and passwords** —using the freedesktop.org Secrets
648 D-Bus API as in the Security design (section 5)
- 649 • **preferences hard key** —implemented according to the Hard Keys design
650 **preferences hard key1)**

651 Proxied dconf backend

652 In its current state (May 2015, detailed in **GNOME Linux desktop**), dconf does
653 not support the necessary fine-grained access controls for multiple components
654 accessing the user and system schemas. However, a design is being implemented
655 upstream to proxy access to dconf through a separate service which imposes
656 access controls based on AppArmor (mostly implemented as of January 2016).

657 On the assumption that this work can be completed and integrated into Apertis
658 on an appropriate timescale (see [Summary of recommendations](#)), this leads to
659 a design where the dconf daemon runs as a system service, storing all settings
660 in one database file per default layer:

- 661 • **User database:** `~/.config/dconf/user`
- 662 • **System database:** `/etc/dconf/db/local`

663 This would be implemented as the dconf profile:

```
664 user-db:user  
665 system-db:local
```

666 All accesses to dconf would go through GSettings, and then through the proxy
667 service which applies AppArmor rules to restrict access to specific settings, im-
668 plementing the chosen security policy ([Access permissions](#)). The rules may, for
669 example, match against settings path and the AppArmor label of the calling
670 process.

671 The proxy service would therefore implement a system preferences service.

672 [Vendor lockdown](#) is supported already by [dconf](#)³⁸ through the use of lockdown
673 files, which specify particular keys or settings sub-trees which may not be mod-
674 ified.

675 Resetting all system settings would be a matter of deleting the appropriate
676 databases —the keys in that database will revert to the default values provided
677 by the schema files. As this is a simple operation, it does not have to be imple-
678 mented centrally by a preferences service. Resetting the value of an individual
679 key is supported by the `g_settings_reset()` API, which is already implemented
680 as part of GSettings.

681 The existing Apertis system puts

```
682 include <abstractions/gsettings>
```

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

686 Requirements

687 This design fulfills the following requirements:

- 688 • [Access permissions](#) —through use of the proxy service and AppArmor rules
- 689 • [Factory reset](#) —by deleting the user’s database or the user’s per-app
690 database
- 691 • [Minimising io bandwidth](#) —dconf’s database design is optimised for this

³⁸<https://developer.gnome.org/dconf/unstable/dconf-overview.html>

- 692 • **Atomic updates** —dconf performs atomic overwrites of the database
- 693 • **Performance tradeoffs** —dconf is heavily optimised for reads rather than
- 694 writes
- 695 • **Data size tradeoffs** —dconf uses GVDB for storage, so can handle small to
- 696 large amounts of data
- 697 • **Vendor overrides** —dconf supports vendor overrides inherently
- 698 • **vendor lockdown** —dconf supports vendor lockdown inherently

699 Development backend

700 In the interim, we recommend that the standard dconf backend be used to store
 701 all system, user and app settings. This will *not* allow for access controls to be
 702 applied to the settings (**Access permissions**), but will allow for development of
 703 OS components against the final GSettings interface.

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

707 This development backend would support vendor lockdown as normal.

708 Requirements

709 This design fails the following requirements:

- 710 • **Access permissions** —**unsupported** by the current version of dconf

711 It supports the following requirements:

- 712 • **Factory reset** —**partially supported** by deleting the user's database; re-
 713 setting a (user, app) pair is not supported as all settings are stored in the
 714 same dconf database file
- 715 • **Minimising io bandwidth** —dconf's database design is optimised for this
- 716 • **Atomic updates** —dconf performs atomic overwrites of the database
- 717 • **Performance tradeoffs** —dconf is heavily optimised for reads rather than
- 718 writes
- 719 • **Data size tradeoffs** —dconf uses GVDB for storage, so can handle small to
- 720 large amounts of data
- 721 • **Vendor overrides** —dconf supports vendor overrides inherently
- 722 • **vendor lockdown** —dconf supports vendor lockdown inherently

723 Key-file backend

724 For Flatpaks, the GSettings key-file backend is used. (This could also be used
725 as an alternative to the development backend before the proxied dconf backend
726 is ready.) This allows the use of Flatpak's native filesystem sandboxing for
727 security, but it creates a unique set of trade-offs that make it less suitable for
728 the other use cases:

- 729 • lower read performance due to not being optimised for reads (or in general);
730 this may be less of an issue for applications, which may not have as many
731 settings in a single file
- 732 • requiring code changes in user services to switch from the key-file backend
733 to the proxied dconf backend once it's ready; applications will stay using
734 the key-file backend, so no migration is necessary
- 735 • requiring settings values to be migrated from the key-file store to dconf
736 at the time of switch over; as before, the backend will stay being used by
737 applications, so no migration is necessary
- 738 • not supporting vendor lockdown or vendor overrides; these are primarily
739 intended for system settings, not application settings

740 Due to the need for code changes to switch away from this backend to a more
741 suitable long-term solution such as the proxied dconf backend, we do not rec-
742 ommend this approach for system components.

743 In detail, the approach would be to use a separate key file for each schema
744 instance. Flatpak applications handle this automatically, but for user and sys-
745 tem schemas, this would require using `g_settings_key_file_backend_new()` and
746 `g_settings_new_with_backend_and_path()` to manually construct the GSettings in-
747 stance for each schema.

748 Access control for each schema instance would be enforced using AppArmor rules
749 and Flatpak sandboxing which restrict access to each key file as appropriate. For
750 example, apps would be given read-write access to the key file for their own app
751 settings, but any key files or dconf databases used by user and system services
752 would be unreadable or read-only.

753 For apps, they will be unable to directly access any non-key-file settings, thus
754 vendor lockdown for those components is inherent in the design (see [Application
755 access to system settings](#)). However, vendor lockdown for application settings
756 is not supported.

757 For the system and user services themselves, vendor lockdown would be sup-
758 ported by vendors patching the AppArmor files to limit write access to specific
759 schema instances. It would not support per-key lockdown at the granularity
760 supported by dconf.

761 This code for creating the GSettings object could be abstracted away by a helper
762 library, but the API for that library would have to be stable and supported

763 indefinitely, even after changing the backend.

764 Requirements

765 This design fails the following requirements:

- 766 • **Performance tradeoffs** —GKeyFile is **equally non-optimised** for reads
767 and writes
- 768 • **Vendor overrides** —**unsupported** by GKeyFile
- 769 • **vendor lockdown** —**unsupported** by GKeyFile

770 It supports the following requirements:

- 771 • **Access permissions** —supported by AppArmor rules and Flatpak permis-
772 sions on the per-schema key files
- 773 • **Factory reset** —by deleting the appropriate key files
- 774 • **Minimising io bandwidth** —GKeyFile's I/O bandwidth is proportional to
775 the number of times each key file is loaded and saved
- 776 • **Atomic updates** —GKeyFile performs atomic overwrites of the database
- 777 • **Data size tradeoffs** —GKeyFile's load and save performance is proportional
778 to the amount of data stored in the file, so it is suitable for small amounts
779 of data

780 Security policy

781 The key-file backend enforces security policy for apps through Flatpak's filesys-
782 tem sandboxing, and the proxied dconf backend enforces security policy for user
783 and system schemas through AppArmor rules. (The **Development backend** does
784 not support implementing security policy at all.)

785 It is beyond the scope of this document to define how AppArmor rules and
786 Flatpak manifests are configured.

787 Application access to system settings

788 Flatpak applications should not be able to see the full host dconf database.
789 Therefore, access to that is not granted by default, rather allowing access to
790 select settings by the use of the [XDG Settings portal](https://flatpak.github.io/xdg-desktop-portal/#gdbus-org.freedesktop.portal.Settings)³⁹. In order for this to
791 function, a custom [portal backend service](https://flatpak.github.io/xdg-desktop-portal/#gdbus-org.freedesktop.impl.portal.Settings)⁴⁰ must be created that exposes the
792 settings to applications. The use of the portals to access system and user settings
793 does not place any restrictions on the actual storage of the settings, which may
794 still be stored in dconf.

³⁹<https://flatpak.github.io/xdg-desktop-portal/#gdbus-org.freedesktop.portal.Settings>

⁴⁰<https://flatpak.github.io/xdg-desktop-portal/#gdbus-org.freedesktop.impl.portal.Settings>

795 User interface

796 Different options for building preferences user interfaces need to be supported
797 by the system (**Control over user interface**):

- 798 • Individual preferences embedded at different points in the application UI.
- 799 • A preferences window implemented within the application.
- 800 • A system preferences application which controls displaying the preferences
801 for all installed applications, plus system preferences.

802 In all cases, we recommend that preferences are defined using GSettings schemas,
803 as discussed in **Overall architecture**, and that settings are read and written
804 through the **GSettings**⁴¹ API. This ensures that access control is enforced, and
805 separates the structure of the preferences (including types and default values)
806 from their presentation.

807 The choice of how preferences are presented ultimately lies with the vendor. In
808 certain cases, an application may choose to display a preference embedded into
809 its UI (for example, as a satellite/hybrid/standard view selector overlaid on a
810 map view), if it makes sense for that preference to be displayed in-context as
811 opposed to in a preferences window. This user experience is something which
812 should be checked as part of app validation.

813 The majority of preferences should be displayed in a separate preferences win-
814 dow. In order to allow this window to be embedded into a system preferences
815 application if the vendor desires it, the preferences window must be automati-
816 cally generated. This is because:

- 817 • arbitrary code from arbitrary applications must not be run in the context
818 of the system preferences application; and
- 819 • the system preferences application cannot be shipped with manually-coded
820 preferences windows for all applications which could ever be installed.

821 However, automatically generated UIs generally give a bad user experience, due
822 to the limited flexibility a designer has on them, so are suitable only for basic
823 preferences (such as toggle switches; see **Discussion of automatically generated
824 versus manually coded preferences UIs**). There may be cases where an appli-
825 cation has a particular preference which Apertis provides no widgets suitable
826 for editing it. In these infrequent cases, it must be possible for the system
827 preferences application to execute a stand-alone preferences window from the
828 application to set that particular preference.

829 System preferences application

830 If an application has preferences, it must give the path to the GSettings schema
831 file which defines them in its application manifest.

⁴¹<https://developer.gnome.org/gio/stable/GSettings.html#GSettings.description>

832 The system preferences application should display a list of applications as its
833 initial screen, including entries for system preferences which it implements itself.
834 The applications listed should be the ones whose manifests specify GSettings
835 schema files, and the application name and icon should also be retrieved from
836 the application manifest and displayed.

837 If the user selects an application, a preferences window should be displayed
838 which shows all the preferences in the application's GSettings schema file. See
839 [Generating a preferences window from a GSettings schema file](#) for details of how
840 this is done. Note that if the schema file defines multiple levels of schema, they
841 should be presented as a hierarchy of pages, with preferences only being shown
842 on leaf pages.

843 As a system application, the system preferences application would have permis-
844 sion to read and write any application settings via GSettings, so forms part of
845 the trusted computing base (TCB) for preferences.

846 The vendor may choose the security policy for which users may edit system
847 preferences (such as the language or background) —they could either allow all
848 users to edit these, or only allow administrative users (such as the vehicle owner)
849 to edit them. If so, we recommend showing the entries for these preferences
850 anyway, but making the widgets insensitive and presenting an authentication
851 dialogue for the administrator to authenticate with before allowing the settings
852 to be edited, see the [Multi-User Transactional Switching document](#)⁴².

853 **Per-application preferences windows**

854 If the vendor wishes to implement a user experience where each application
855 shows its own preferences window, this should be implemented using the system
856 preferences application in a different mode. A settings button or menu entry in
857 the application should launch the system preferences application.

858 It should support being launched with the name of a GSettings schema to show,
859 and it would render a preferences window from that schema (see [Generating
860 a preferences window from a GSettings schema file](#)). If the schema file defines
861 multiple levels of schema, they should be presented as a hierarchy of pages, with
862 preferences only being shown on leaf pages. It is up to the vendor whether the
863 user can navigate 'up' from the top level of the schema to a list of all applications.

864 As the system preferences application is part of the TCB for preferences, it must
865 not allow an application to launch it with the name of a GSettings schema file
866 which does not belong to that application. For example, that would allow one
867 application to trick the user into editing their preferences for another applica-
868 tion.

⁴²https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application_security/multiuser-transactional-switching/

869 **Generating a preferences window from a GSettings schema file**

870 A GSettings [schema file](#)⁴³ can be turned into a UI using the following rules:

- 871 • A `<schema>` element is turned into a preference page. If it has an `extends` attribute, the widgets from the schema it extends are added to the
872 preferences page first.
873
- 874 • The first non-relocatable `<schema>` element in a `<schemalist>` will be
875 taken as providing the preferences page for the application. Subsequent
876 `<schema>` elements will be ignored unless pulled in as preferences sub-
877 pages using a `<child>` element.
- 878 • A `<child>` element is turned into an entry to show a preferences sub-page
879 for the corresponding sub-schema. The label for this entry should come
880 from a new (non-standard) label attribute on the `<child>` element.
- 881 • Relocatable `<schema>` elements (those without a path attribute) are ig-
882 nored unless pulled in as a preferences sub-page using a `<child>` element.
- 883 • A `<key>` element is turned into a widget with its label set from the `<sum-`
884 `mary>` element and its description set from the `<description>` element.
885 The type of widget is set by the type attribute, which specifies a [GVariant](#)
886 [type](#)⁴⁴:
 - 887 – b (boolean): Switch or checkbox widget.
 - 888 – y, n, q, i, u, x, t (integers): Integer spin button. Its range is set to
889 the smaller of the bounds of the integer type or the values of the
890 `<range>` element (if present).
 - 891 – h (handle): Not supported.
 - 892 – d (double): Floating point spin button. Its range is set to the smaller
893 of the bounds of the double type or the values of the `<range>` element
894 (if present).
 - 895 – s (string): Text entry widget. If a `<choices>` element is present, a
896 drop-down box should be used instead, displaying the options from
897 the `<choice>` elements.
 - 898 – o (object path): Not supported.
 - 899 – g (type string): Not supported.
 - 900 – ? (basic type): Not supported.
 - 901 – v (variant): Not supported.
 - 902 – a (array): Not supported in any form.
 - 903 – m (maybe): Not supported in any form.

⁴³<https://gitlab.gnome.org/GNOME/glib/-/blob/main/gio/gschema.dtd>

⁴⁴<https://developer.gnome.org/glib/stable/glib-GVariantType.html#id-1.6.18.6.9>

- 904 – (), r (tuple): Not supported in any form.
- 905 – {} (dictionary): Not supported in any form.
- 906 – * (any): Not supported in any form.
- 907 • If a <key> element contains an enum attribute and no type attribute,
908 a drop-down box should be used, displaying the options from the nick
909 attributes of the <value> elements in the corresponding <enum> element.
- 910 • If a <key> element contains a flags attribute and no type attribute, a
911 checkbox list should be used, displaying a checkbox for each each of the
912 nick attributes of the <value> elements in the corresponding <flags>
913 element.
- 914 • If a key's name attribute matches a mapping to a wizard application
915 (see [Support for custom preferences windows](#)) in the application's man-
916 ifest, that key should be displayed as a menu entry which, when selected,
917 launches the wizard application as a new window.

918 **Support for custom preferences windows**

919 If an application has a particularly esoteric preference or set of preferences which
920 are not supported by the generated preferences UI (see [Generating a preferences](#)
921 [window from a GSettings schema file](#)), it may provide a 'wizard' application
922 as part of its application bundle which allows setting those preferences (and
923 only those preferences). For example, this could be used to show a 'wizard' for
924 configuring an e-mail account; or a map widget for selecting a location.

925 A wizard application presents a single window of preferences, and its widgets
926 cannot be integrated into a preferences window generated by the system prefer-
927 ences application —it must be launched using a menu entry from there.

928 The wizard application must be listed in the application's manifest as part of a
929 dictionary which maps GSettings schemas or keys to commands to run.

930 For example, a particular manifest could map the key /org/foo/MyApp/complex-
931 setting to the command my-app --show-complex-setting. Or a manifest could
932 map the schema /org/foo/MyApp/EmailAccount to the command my-app --
933 configure-email-account.

934 Application bundles which contain keys for this in their manifest should be
935 subjected to extra app store validation checks, to establish that the wizard
936 application's UI is consistent with other preferences UIs, and that it does not
937 implement preferences which should be handled by a generated UI.

938 The wizard application must set the relevant preferences itself before exiting,
939 and runs with the same privileges as the rest of the application bundle (so will
940 only have access to that application's preferences, as per [Security policy](#)).

941 It may be necessary for the window manager to treat windows from wizard
942 applications specially, so that they appear more like a window which is part of
943 the system preferences application than a window from a separate application.
944 This can be solved by adding appropriate metadata to the wizard application
945 windows so the window manager treats them differently.

946 **Searchability of preferences**

947 To allow the system preferences application to search over all applications' pref-
948 erences (**Searchable preferences**), it must load all the GSettings schemas from
949 applications whose manifests specify a schema. Searching must be performed
950 over the user-visible parts of the schema (the <summary> and <description>
951 elements), and results should be returned as a link to the relevant application
952 preferences window. System preferences should be included in the search results
953 too.

954 **Reorganising preferences**

955 Implementing arbitrary reorganisation of preferences (**Rearrangeable prefer-**
956 **ences**) is difficult, as that requires an OEM to know the semantics of all prefer-
957 ences for all possibly installable applications.

958 We recommend that if an OEM wants to present a new group of a certain set
959 of preferences, they must choose specific preferences from known applications,
960 and implement a custom window in the system preferences application which
961 displays those preferences. Each preference should only be shown if the relevant
962 application is installed.

963 An alternative implementation which is more flexible, but which devolves more
964 control to application developers, is to tag each preference in the GSettings
965 schemas with well-defined tags which summarise the preference's semantics. For
966 example, an application's preference for whether to submit usage data to the
967 application data could be tagged as 'privacy'; or a preference determining the
968 colour scheme to use in an application could be tagged as 'appearance'. The
969 OEM could then implement a custom preferences window which queries all
970 installed GSettings schemas for a specific tag and displays the resulting prefer-
971 ences. We do not recommend this option, as even with app store validation of
972 the chosen tags, this would allow application developers too much control over
973 the appearance of a system preferences window.

974 **Preferences list widget**

975 In order to help make all preferences UIs consistent (including those imple-
976 mented by the vendor, **System preferences application**; and those implemented
977 by application developers as wizard applications, **Per-application preferences**
978 **windows**), Apertis should provide a standard widget which implements the con-
979 version from GSettings schemas to UI as described in **Generating a preferences**
980 **window from a GSettings schema file**.

981 This widget should accept a list of GSettings schema paths to display, and may
982 optionally accept a list of keys within those schemas to display (ignoring the
983 others), or to ignore (displaying the others); and should display all those keys
984 as preferences. It should implement reading and writing the keys' values using
985 the GSettings API, and must assume that the application has permission to do
986 so (see [Security policy](#)). It must check for writability of preferences and make
987 them insensitive if they are read-only (see [vendor lockdown1](#)). It cannot give the
988 application more permissions than it already has.

989 If application developers use this widget, the vendor can ensure that preferences
990 UIs are consistent between applications and the system preferences application
991 through the theming of the widget.

992 **Vendor lockdown**

993 If the vendor locks down a key in a GSettings schema for an application (or
994 system preference) [vendor lockdown](#)—supported by [Proxied dconf backend](#) and
995 [Development backend](#), but not [Key-file backend](#)), that is enforced by the under-
996 lying settings service (most likely dconf), and cannot be overridden or worked
997 around by applications.

998 However, it is up to applications to reflect whether a preference is read-only
999 (due to being locked down) in their UIs. This is typically achieved by hid-
1000 ing a preference or making its widget insensitive. Applications can use the
1001 [g_settings_is_writable](#)⁴⁵ method to determine whether a preference is read-
1002 only. Any preferences widgets provided by Apertis ([Preferences list widget](#))
1003 must implement this already.

1004 If an application developer uses a custom widget to display a preference, and
1005 forgets to check whether that preference is read-only, their application might
1006 enter an inconsistent state (which is their fault), but the system will not let
1007 that preference be written. Convenience APIs like [g_settings_bind_writable](#)⁴⁶
1008 can reduce the risk of this happening.

1009 **Discussion of automatically generated versus manually coded prefer-** 1010 **ences UIs**

1011 In an ideal world, our recommendation would be that: while automatically
1012 generating preference UIs can rapidly produce rough drafts, in our experience
1013 it can never result in a high-quality finished UI with:

- 1014 • logically grouped options;
- 1015 • correctly aligned controls;
- 1016 • a concept of which preferences are most important, which ones are ‘ad-
1017 vanced’, and which ones should be hidden;

⁴⁵<https://developer.gnome.org/gio/unstable/GSettings.html#g-settings-is-writable>

⁴⁶<https://developer.gnome.org/gio/stable/GSettings.html#g-settings-bind-writable>

- 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, we believe 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](#)⁴⁷. Specifically, we recommend that preferences are:

- integrated into the main application UI if there are only a small number of them;
- [instant-apply](#)⁴⁸ unless doing so would be dangerous, 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 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.

An alternative way of thinking about this is that preferences are subject to a model-view split (the model is GSettings schema files; the view is the preferences UI), and it is typically inadvisable to generate a view from a model when following that pattern.

However, we realise that the goal of having a unified system preferences application with a consistent appearance (which is enforced) conflicts with these recommendations, and hence these recommendations are not part of our overall suggested approach.

Preferences hard key

A preferences hard key must be supported as detailed in the Hard Keys design. In a configuration where a system preferences application is used, it must launch that application, already open on the preferences window for the active application. If no application is active, or if the currently active application has

⁴⁷<https://developer.gnome.org/hig/stable/dialogs.html.en>

⁴⁸<https://developer.gnome.org/hig/stable/dialogs.html.en#instant-and-explicit-apply>

1055 no GSettings schemas listed in its manifest file, the main page of the system
1056 preferences application should be shown.

1057 In a configuration where applications implement their own preferences windows,
1058 the active application must be sent a 'hard key pressed'signal for the preferences
1059 hard key, which the application can handle how it wishes (i.e. by showing its
1060 preferences window). If there is no active application, the system preferences
1061 application (which in this configuration only contains system preferences) should
1062 be shown.

1063 The policy for exactly what happens in each situation and configuration is under
1064 the control of the hard keys service, which is provided by the vendor. It should
1065 have access to the manifest for the active application so it can find information
1066 about GSettings schemas.

1067 **Existing preferences schemas**

1068 As GSettings is used widely within the open source software components used
1069 by Apertis, particularly GNOME, there are many standard GSettings schemas
1070 for common user settings. We recommend that Apertis re-use these schemas as
1071 much as possible, as support for them has already been implemented in various
1072 components. If that is not possible, they could be studied to ensure we learn
1073 from their design successes or failures.

- 1074 • org.gnome.system.locale
- 1075 • org.gnome.system.proxy
- 1076 • org.gnome.desktop.default-applications
- 1077 • org.gnome.desktop.media-handling
- 1078 • org.gnome.desktop.interface
- 1079 • org.gnome.desktop.lockdown
- 1080 • org.gnome.desktop.background
- 1081 • org.gnome.desktop.notifications
- 1082 • org.gnome.crypto
- 1083 • org.gnome.desktop.privacy
- 1084 • org.gnome.system.dns_sd
- 1085 • org.gnome.desktop.sound
- 1086 • org.gnome.desktop.datetime
- 1087 • org.gnome.system.location
- 1088 • org.gnome.desktop.thumbnailers
- 1089 • org.gnome.desktop.thumbnail-cache

1090 • org.gnome.desktop.file-sharing

1091 Various Apertis dependencies (for example, Mutter, Tracker, libfolks, IBus, Geo-
1092 clue, Telepathy) use their own GSettings schemas already —as these are not
1093 shared, they are not listed.

1094 *Alternative model:* If the locale is a system setting, rather than a user setting,
1095 systemd’s [locale](#)⁴⁹ should be used. This would require the locale to be changed
1096 via the locale D-Bus API, rather than GSettings, which would affect the im-
1097 plementation of the system preferences app.

1098 Persistent data approach

1099 Overall architecture

1100 As discussed in sections 5.3.1 and 7 of the Applications Design, and the Mul-
1101 tiuser Design, there is a difference between state which an app needs to persist
1102 (for example, if it is being terminated to switch users), and state which an app
1103 explicitly needs to share (for example, if a transactional user switch is taking
1104 place to execute an action as a different user). The Multiuser Design encourages
1105 app authors to think explicitly about these two sets of state, and the differences
1106 between them. It is the app which chooses the state to persist, rather than
1107 the operating system —storage space is too limited to persist the entire address
1108 space of an app, effectively suspending it.

1109 The state each app chooses to persist will differ, and cannot be predicted by
1110 Apertis. There could be a lot of state, or very little. It could be representable as
1111 a simple key-value dictionary, or might have a complex hierarchical structure.

1112 Well-known state directories

1113 As mentioned in the Applications Design document (sections 5.3.1 and 7), we
1114 recommend that Apertis provides a per-(user, app) directory for storage of per-
1115 sisted data, and a public API the app can call to find out that directory. The
1116 API should follow Flatpak conventions, differentiating between cache and non-
1117 cache state, with cache state going in `$HOME/.var/net.example.MyApp/cache`
1118 and non-cache state going in `$HOME/.var/net.example.MyApp/data`. This ful-
1119 fills the factory reset requirement (**Factory reset**).

1120 The former is effectively equivalent to a per-(user, app) `XDGLCACHEHOME`
1121 directory, and the latter to a `XDGLDATAHOME`, as defined by the [XDGL Base](#)
1122 [Directory Specification](#)⁵⁰.

1123 Flatpak permissions, by default, grant an app write access exclusively to its
1124 cache and data directories, and not to other apps’state directories, with AppAr-
1125 mor profiles potentially providing further defense-in-depth. This is the extent

⁴⁹<http://www.freedesktop.org/wiki/Software/systemd/locale/>

⁵⁰<http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html>

1126 of the security needed, as state storage is simply an interaction between an app
1127 and the filesystem.

1128 As with preferences, app bundles must be in charge of upgrading their own per-
1129 sistent data when the system is upgraded (or the app is upgraded) (**System and**
1130 **app bundle upgrades**). Recommendations are given in the subsections below.

1131 Recommended serialisation APIs

1132 As each app's state storage requirements are different, we suggest that Apertis
1133 provide several recommended serialisation APIs, and allow apps to choose the
1134 most appropriate one —or something completely different if that fulfils their
1135 requirements better.

1136 Alongside, Apertis should provide guidelines to app developers to allow them
1137 to choose an appropriate serialisation API, and avoid common problems in se-
1138 rialisation:

- 1139 • minimise writes to main storage (**Minimising io bandwidth**);
- 1140 • ensure all updates to stored state are atomic (requirement **Atomic up-**
1141 **dates**); and
- 1142 • ensure transactions are used for groups of updates where appropriate (**Transactional updates**).

1144 Atomic in the sense that either the old or new states are stored in
1145 entirety, rather than some intermediate state, if power is lost part-
1146 way through an update.

1147 Depending on the requirements it is believed that apps will have, some or all of
1148 the following APIs could be recommended for serialising state to main storage.
1149 For comparison, Android only provides a generic file storage API, and an SQLite
1150 API, with no implemented **key-value store APIs**⁵¹. Apps must implement those
1151 themselves.

1152 GKeyFile

1153 <https://developer.gnome.org/glib/stable/glib-Key-value-file-parser.html>

1154 Suitable for small amounts of key-value state with simple types. Suitable for
1155 small amounts of data.

1156 All updates to a GKeyFile are atomic, as it uses the atomic-overwrite technique:
1157 the new file contents are written to a temporary file, which is then atomically
1158 renamed over the top of the old file. Transactional updates can be implemented
1159 by saving the key file to apply the transaction, and discarding the in-memory
1160 GKeyFile object to revert it.

⁵¹<http://developer.android.com/guide/topics/data/data-storage.html>

1161 The amount of I/O with a GKeyFile is small, as the amount of data which
1162 should be stored in a GKeyFile is small, and the file is only written out when
1163 explicitly requested by the app.

1164 System upgrades have to be handled manually by app bundles—if the persistence
1165 data format has to change, the app must migrate data from the old format to
1166 the new format the first time it is run after an upgrade. In this case, it is
1167 recommended that all GKeyFiles used for persistent data contain a ‘Version’ key
1168 specifying the data format version in use.

1169 **GVDB**

1170 <https://git.gnome.org/browse/gvdb>

1171 Memory-mapped hash table with GVariant⁵²-style types, suitable for small to
1172 large amounts of data which are read much more frequently than they are writ-
1173 ten. This is what dconf uses for storage.

1174 All updates to a GVDB file are atomic, as it uses the same atomic-overwrite
1175 technique as GKeyFile. Transactions are supported similarly—by writing out
1176 the updated database or discarding it.

1177 The amount of I/O for reads from a GVDB file is small, as it memory-maps
1178 the database, so only pages in the data it actually reads (plus some metadata).
1179 Writes require the entire file to be updated, but are only done when explicitly
1180 requested by the app.

1181 GVDB supports per-file versioning (though this is not currently exposed in the
1182 public API). This can be used for handling system upgrades (**System and app**
1183 **bundle upgrades**)—the database must be explicitly migrated from an old version
1184 to a new version when an upgraded app is first started.

1185 **SQLite**

1186 <http://sqlite.org/>

1187 <https://wiki.gnome.org/Projects/Gom>

1188 Full SQL database implementation, supporting simple SQL types and more
1189 complex relational types if implemented manually by the app. Suitable for
1190 medium to large amounts of data which are read and written frequently. It
1191 supports SQL transactions.

1192 SQLite is not a panacea. It is designed for the specific use pattern of SQL
1193 databases with indexes and relational tables, with frequent reads and writes,
1194 and infrequent deletions of data. Apps will only get the best performance from
1195 SQLite by defining their own table structure, indices and relations; imposing a
1196 common key-value-style API on top of SQLite would give lower performance.

⁵²<https://developer.gnome.org/glib/stable/glib-GVariant.html>

1197 SQLite has limited support for SQL schema upgrades with its [ALTER TABLE](#)⁵³
 1198 statement, which supports renaming tables and adding new columns to tables.
 1199 Apps must implement their own data migration from old to new versions of
 1200 their database schema; documenting this is beyond the scope of this design.

1201 Apps should only use SQLite if they have considered issues like their vacuuming
 1202 policy —how frequently to vacuum the database after deleting data from it. See:

- 1203 • https://blogs.gnome.org/jnelson/2015/01/06/sqlite-vacuum-and-auto_vacuum/
- 1204 • https://wiki.mozilla.org/Performance/Avoid_SQLite_In_Your_Next_Firefox_Feature

1205 If using GObject to represent entries in an SQLite database, the [GOM](#)⁵⁴ wrap-
 1206 per around SQLite may be useful to simplify code.

1207 GNOME-DB

1208 <http://www.gnome-db.org/>

1209 This is **not** recommended. It is an abstraction layer over multiple SQL database
 1210 implementations, allowing apps to access remote SQL databases. In almost all
 1211 cases, directly using [Sqlite](#) is a more appropriate choice.

1212 When to save persistent data

1213 As specified in the Applications Design (section 5.3.1), state is saved to main
 1214 storage at times chosen by both the operating system and the app. The oper-
 1215 ating system knows when the logged in user is about to change, or when the
 1216 system is about to be shut down; the app knows when it has changed some of
 1217 its persistent state in memory, and hence needs to write it out to main storage.

1218 An action could be implemented in each app which is triggered by the Acti-
 1219 vateAction method of the org.freedesktop.Application [D-Bus interface](#)⁵⁵ if, for
 1220 example, that interface is implemented by apps. When triggered, this action
 1221 would cause the app to store its persistent state.

1222 Recently used and favourite items

1223 Section 6.3 of the Global Search Design specifies that an API for apps to store
 1224 their favourite and recently used items in will be provided. As this is data shared
 1225 from an app to the operating system, and is typically append-only rather than
 1226 strongly read-write, we recommend that it be designed separately from the
 1227 persistent data API covered in this document, following the recommendations
 1228 given in the Global Search Design document.

⁵³https://www.sqlite.org/lang_altertable.html

⁵⁴<https://wiki.gnome.org/Projects/Gom>

⁵⁵[http://standards.freedesktop.org/desktop-entry-spec/desktop-entry-spec-latest.html#](http://standards.freedesktop.org/desktop-entry-spec/desktop-entry-spec-latest.html#dbus)
 dbus

1229 Summary of recommendations

1230 As discussed in the above sections, we recommend:

- 1231 • Splitting preferences, persistent data storage and confidential data storage
1232 ([Approach](#)).
- 1233 • Providing one API for preferences: GSettings ([Overall architecture](#)).
- 1234 • Apps provide a GSettings schema file for their preferences, named after
1235 the app ([Overall architecture](#)).
- 1236 • Existing GSettings schemas are re-used where possible for user and system
1237 settings ([Existing preferences schemas](#)).
- 1238 • Using the normal GSettings approach for handling app upgrades ([Overall
1239 architecture](#)).
- 1240 • Developing against the normal dconf backend for GSettings storage for
1241 user and systems schemas (section [Development backend](#)).
- 1242 • Switching to the proxied dconf backend once it's ready, to support access
1243 control ([Proxied dconf backend](#)).
- 1244 • Use the key-file backend for *only* applications; we do *not* recommend using
1245 it as an alternative to the dconf backends for system and user schemas. ([Key-file
1246 backend](#)).
- 1247 • Permissions to modify user or system settings are controlled by the app's
1248 manifest ([Security policy](#)).
- 1249 • User interfaces for preferences are provided by the vendor, automatically
1250 generated from GSettings schemas; or provided by applications ([User
1251 interface](#)).
- 1252 • Apertis provides a standard widget to present GSettings schemas as a
1253 preferences UI ([Preferences list widget](#)).
- 1254 • Preferences hard key support is added according to the Hard Keys design
1255 ([preferences hard key](#)).
- 1256 • Providing API to get a persistent data storage location ([Well known state
1257 directories](#)).
- 1258 • Persistent data is private to each (user, app) pair ([Well known state
1259 directories](#)).
- 1260 • Recommending various different data storage APIs to suit different apps'
1261 use cases ([Recommended serialisation APIs](#)).
- 1262 • Apps explicitly define which data will persist, and are responsible for sav-
1263 ing it and migrating it from older to newer versions ([Overall architecture](#)).

- 1264 • Apps can be instructed to save their persistent state by the operating
1265 system via a D-Bus interface ([When to save persistent data](#)).
- 1266 • User secrets and passwords are stored using the freedesktop.org Secrets
1267 D-Bus API, not the Apertis preferences or persistence APIs ([Approach](#)).