



Preferences and persistence

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

43	iOS	12
44	Preferences	12
45	Persistent data	13
46	Secrets and passwords	14
47	GENIVI	14
48	Preferences and persistent data	14
49	Secrets and passwords	15
50	Approach	16
51	Preferences approach	17
52	Overall architecture	17
53	Requirements	18
54	Proxied dconf backend	19
55	Requirements	20
56	Development backend	20
57	Requirements	20
58	Key-file backend	21
59	Requirements	22
60	Security policy	22
61	User interface	23
62	System preferences application	24
63	Per-application preferences windows	24
64	Generating a preferences window from a GSettings schema file	25
65	Support for custom preferences windows	26
66	Searchability of preferences	27
67	Reorganising preferences	27
68	Preferences list widget	28
69	Vendor lockdown	28
70	Discussion of automatically generated versus manually coded	
71	preferences UIs	29
72	Preferences hard key	30
73	Existing preferences schemas	30
74	Persistent data approach	31
75	Overall architecture	31
76	Well-known state directories	32
77	Recommended serialisation APIs	32
78	GKeyFile	33
79	GVDB	33
80	SQLite	34
81	GNOME-DB	34
82	When to save persistent data	35
83	Recently used and favourite items	35
84	Summary of recommendations	35

85 Introduction

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

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

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

96 The principles described in this document also apply to [The next-gen Apertis
97 application framework](#)⁴ for which Apertis will use Flatpak.

98 Terminology and concepts

99 System Settings

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

108 User settings

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

111 App settings

112 An *app setting* is one which varies by user and also by app. Throughout this
113 document, the term 'app' is used to mean an app-bundle, including the UI and
114 any associated agent programs, analogous to an Android .apk, with a single
115 security domain shared between all executables in the bundle. The precise

¹<https://martyn.pages.apertis.org/apertis-website/concepts/applications/>

²<https://martyn.pages.apertis.org/apertis-website/concepts/global-search/>

³<https://martyn.pages.apertis.org/apertis-website/concepts/robustness/>

⁴<https://martyn.pages.apertis.org/apertis-website/concepts/application-framework/>

116 terminology is currently under discussion, and this document will be updated
117 to reflect the result of that.

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

122 **Preferences**

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

125 **User services**

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

132 **Persistent data**

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

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

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

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

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

152 Main storage

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

156 GSettings

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

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

168 AppArmor

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

172 Requirements

173 Access permissions

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

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

- 181 • **Apps** may read their own app settings, user settings for the current user,
182 and all system settings.
- 183 • **User services** may read the user’s application settings, user settings for
184 the current user, and all system 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/>

185 • **System services** may read their own app settings, and all system set-
186 tings.

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

188 • **Apps** may write their own app settings.

189 • **User services** may write user settings for the current user.

190 • **System services** may write system settings for all users, user settings
191 for any user, and app settings for any app for any user.

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

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

196 **Writability**

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

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

203 **Rollback**

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

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

213 **System and app bundle upgrades**

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

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

219 **Factory reset**

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

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

227 **Abstraction level**

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

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

236 **Minimising I/O bandwidth**

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

241 **Atomic updates**

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

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

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

252 **Transactional updates**

253 The system must allow updates to preferences to be wrapped in transactions,
254 such that either all of the preferences within a transaction are updated, or none

255 of them are. Transactions must be revertable before being applied permanently.

256 **Performance tradeoffs**

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

261 **Data size tradeoffs**

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

266 **Concurrency control**

267 As system preferences may affect security policy, reading them should be race
268 free, particularly from [time-of-check-to-time-of-use](#)⁸ race conditions. For exam-
269 ple, if a preference is changed by process C while process R is reading it, process
270 R must either see the new value of the preference, or see the old value of the
271 preference *and* subsequently be notified that it has changed.

272 Similarly for persistent data.

273 **Vendor overrides**

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

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

282 This does not apply to persistent data.

283 **Vendor lockdown**

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

⁸http://en.wikipedia.org/wiki/Time_of_check_to_time_of_use

288 This does not apply to persistent data.

289 **User interface**

290 There must be some user interface (UI) for setting preferences. This may be
291 provided by a system preferences application, as a separate window in each
292 application, or as individual widgets embedded throughout an application's in-
293 terface; or a combination of these options.

294 This does not apply to persistent data.

295 **Control over user interface**

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

299 This does not apply to persistent data.

300 **Rearrangeable preferences**

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

304 **Searchable preferences**

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

307 **Storage of user secrets and passwords**

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

311 **Preferences hard key**

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

317 Existing preferences systems

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

323 GNOME Linux desktop

324 Preferences

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

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

337 [dconf](#)⁹ supports multiple layered databases, each stored separately. For each
338 settings key, a value set for it in one layer overrides any values set in the layers
339 below. The bottom (read-only) layer is always the set of default values which
340 are provided by the schema file. This layered approach allows the system admin-
341 istrator to change settings system-wide in a system database, but also allows
342 users to override those settings in their per-user database. It allows a user to
343 reset all their settings by deleting their per-user database — at which point,
344 the values from the next layer down (typically either a system database or the
345 defaults from schema files) will be used for all settings keys.

346 [Lockdown](#)¹⁰ is supported in dconf in the opposite direction: keys may be locked
347 down at a particular level, and may not be set at levels above that one (but
348 may be set at levels below it, as defaults).

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

⁹<https://developer.gnome.org/dconf/unstable/dconf-overview.html>

¹⁰<https://developer.gnome.org/dconf/unstable/dconf-overview.html#id-1.2.7>

354 dconf does *not* support access controls, and does not support storing different
355 schemas in different databases at the same layer. Hence a user either has write
356 access to the whole of a system database, or write access to none of it. As the
357 dconf daemon runs per user, any app accessing the daemon may write to any
358 settings key, either its own app settings, another app's settings, or the user's
359 settings.

360 **Persistent data**

361 Persistent data is stored in application-defined formats, in application-defined
362 locations, although many follow the [XDG Base Directory Specification](#)¹¹, which
363 puts cache data in XDG_CACHE_HOME (typically ~/.cache) and non-cache
364 data in XDG_DATA_HOME (typically ~/.local/share). Below these two direc-
365 tories, applications create their own directories or files as they see fit. There is
366 no security separation between applications, but the normal UNIX permissions
367 restrict access to only the current user.

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

371 **Secrets and passwords**

372 On a GNOME or KDE desktop, all user secrets, passwords and credentials are
373 stored using the [Secret Service](#)¹² API. In GNOME, this API is implemented by
374 GNOME Keyring; in KDE, by KWallet.

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

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

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

¹¹<http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html>

¹²<https://specifications.freedesktop.org/secret-service/latest/index.html>

390 **Android**

391 **Preferences**

392 Apps can use the [SharedPreferences class](#)¹³ to read and write preferences from
393 named preferences files, with apps typically using a single preferences file with
394 a default name. These files are stored per-app, and are private to that app by
395 default, but may be shared with other apps, either read-only or read-write.

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

399 Preferences are saved to disk immediately.

400 Android uses a [custom XML format](#)¹⁴ to allow apps to define preference UIs
401 (known as ‘activities’ in Android terminology). This format can define sim-
402 ple lists of preferences, through to complex UIs with grouped preferences, sub-
403 screens, lists of subscreens, and custom preference widgets. Implementing fea-
404 tures such as making one preference conditional on another is possible, but
405 requires complex XML.

406 A [PreferenceFragment](#)¹⁵ can be used to automatically build a screen in an ap-
407 plication to display preferences, loading them from the XML file. It will load
408 the current values of the preferences from the SharedPreferences store, and will
409 write new values back to the store as the preferences are modified in the UI.

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

413 **Persistent data**

414 Android offers several options for [persistent data](#)¹⁶:

- 415 • **Internal storage:** Files in a per-(user, app) directory, which may option-
416 ally be made world-readable or writable to allow access to other apps or
417 users (though this is strongly discouraged).
- 418 • **External storage:** Files in a world-readable storage area which is
419 accessible to the user, such as an SD card. Accessible to all other
420 apps and users which hold the READ_EXTERNAL_STORAGE or
421 WRITE_EXTERNAL_STORAGE permissions.
- 422 • **SQLite database:** Arbitrary app-defined tables in a per-(user, app)
423 SQLite database. This cannot be shared with other apps or users.

¹³<http://developer.android.com/guide/topics/data/data-storage.html#pref>

¹⁴<http://developer.android.com/guide/topics/ui/settings.html#DefiningPrefs>

¹⁵<http://developer.android.com/guide/topics/ui/settings.html#Fragment>

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

424 • **Network connection:** Using the normal networking APIs, Android sug-
425 gests that data can be stored on servers controlled by the app developers.
426 It provides no special API for this.

427 For saving an application’s state, Android offers a persistence API on the [Ac-](#)
428 [tivity class](#)¹⁷. This automatically saves the state of all UI elements (such as
429 the text in an entry widget, and the position of a list), but cannot automati-
430 cally save application-specific internal state (member variables). For this, the
431 application must override two toolkit methods (`onSaveInstanceState()` and `on-`
432 `RestoreInstanceState()`) and implement its own serialisation and deserialisation
433 of state to a set of key–value pairs which are then stored by Android.

434 Secrets and passwords

435 Android recommends storing secrets and passwords in two ways. For authen-
436 tication credentials for online services, it provides an `AccountManager` [API](#)¹⁸
437 which abstracts authentication for known online services (which are supported
438 by pluggable backends, potentially provided by application bundles) and stores
439 the credentials in an OS-wide store. The service handles authenticating and
440 re-authenticating when the login session ends.

441 For secrets which are not for online accounts, or otherwise do not fit the `Account-`
442 `Manager` pattern, Android [recommends](#)¹⁹ using the normal preferences API (
443 `Preferences`), as while preferences are not encrypted in storage, they are only
444 accessible to the application which owns them, so cannot be stolen by other
445 applications. However, if the sandboxing system is compromised (potentially
446 by an attacker with physical access to the device), the stored secrets will be
447 accessible in plaintext.

448 iOS

449 Preferences

450 iOS stores preferences as [key–value pairs](#)²⁰, which are separated into domains
451 by user, application and machine. The same preference may be set in [multiple](#)
452 [domains](#)²¹, and they are searched in a defined priority order to determine which
453 value to use. This means that an application may, for example, choose to share
454 a given preference between all users of that application on a given machine.

455 Application IDs use the standard reverse domain name syntax to ensure unique-
456 ness.

¹⁷<http://developer.android.com/training/basics/activity-lifecycle/recreating.html>

¹⁸<http://developer.android.com/reference/android/accounts/AccountManager.html>

¹⁹<http://stackoverflow.com/questions/785973/what-is-the-most-appropriate-way-to-store-user-settings-in-android-application/786588#786588>

²⁰https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFPreferences/CFPreferences.html#//apple_ref/doc/uid/10000129-SW1

²¹<https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFPreferences/Concepts/PreferenceDomains.html>

457 Preference values may be any type supported by Core Foundation [property](#)
458 [lists](#)²², including strings, integers and arrays. Default values must be coded into
459 the application.

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

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

465 On certain Apple operating systems, [preferences may be ‘managed’ by the ad-](#)
466 [ministrator](#)²³, setting an override value which overrides any value set by the
467 user for a given preference key.

468 Application preferences can either be presented as part of the application, using
469 normal UI widgets, and accessing the [NSUserDefaults class](#)²⁴ for the preference
470 values. Or they can be presented as part of the [system-wide settings applica-](#)
471 [tion](#)²⁵, which builds the UI for each application’s preferences dynamically from
472 that application’s property list file for preferences. An application may provide
473 multiple property list files to build a hierarchy of preferences pages. The system-
474 wide settings application accesses NSUserDefaults on behalf of the application
475 to update the stored preferences.

476 Persistent data

477 iOS offers several options for persistent data:

- 478 • **Filesystem:** Arbitrary files may be written to the filesystem in various
479 [app-specific locations](#)²⁶.
- 480 • **Core Data API:** This is an [object-graph management API](#)²⁷, which
481 allows versioned control of instances of objects created from a schema.
482 Instead of being used by an application to persist data, this API is designed
483 to form the core of the application’s data model. It supports editing and
484 discarding edits, undo, redo, versioning of the object schema, and large
485 data sets.

²²https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFPropertyLists/CFPropertyLists.html#//apple_ref/doc/uid/10000130i

²³https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFPreferences/Concepts/BestPractices.html#//apple_ref/doc/uid/TP30001219-118191

²⁴https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSUserDefaults_Class/index.html#//apple_ref/occ/cl/NSUserDefaults

²⁵https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/UserDefaults/Preferences/Preferences.html#//apple_ref/doc/uid/10000059i-CH6-SW6

²⁶https://developer.apple.com/library/ios/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/AccessingFilesandDirectories/AccessingFilesandDirectories.html#//apple_ref/doc/uid/TP40010672-CH3-SW11

²⁷https://developer.apple.com/library/prerelease/ios/documentation/DataManagement/Devpedia-CoreData/coreDataOverview.html#//apple_ref/doc/uid/TP40010398-CH28

- 486 • **Property List API:** A property list is a hierarchical, structured piece
487 of data, consisting of primitive data types, arrays and dictionaries which
488 may be nested [arbitrarily](#)²⁸. Property lists can therefore be used to store
489 arbitrary application data. There is an API to serialise them to the file
490 system.
- 491 • **SQLite:** The standard SQLite API may be used, backed by a file, to store
492 relational data in a database.

493 For persisting an entire application’s state, iOS provides a [solution](#)²⁹ simi-
494 lar to [Android][Persistent data]. The developer must annotate each UI view
495 class which needs to be saved and restored, and the UI toolkit will automati-
496 cally persist the state of the widgets in that view when the application is sus-
497 pended. As with Android, the developer must implement two methods for
498 serialising and deserialising application-specific state from member variables:
499 `encodeRestorableStateWithCoder` and `decodeRestorableStateWithCoder`.

500 Secrets and passwords

501 iOS uses the same [keychain API](#)³⁰ as OS X. This provides a system service for
502 storing secrets, passwords and certificates. They are encrypted in storage, using
503 an encryption key which is derived from the iOS application’s ID and the user’s
504 password.

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

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

513 GENIVI

514 Preferences and persistent data

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

²⁸<https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/PropertyLists/AboutPropertyLists/AboutPropertyLists.html>

²⁹<https://developer.apple.com/library/ios/featuredarticles/ViewControllerPGforiPhoneOS/PreservingandRestoringState.html>

³⁰https://developer.apple.com/library/ios/documentation/Security/Conceptual/keychainServConcepts/01introduction/introduction.html#//appl_ref/doc/uid/TP30000897-CH203-TP1

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

523 It is split into four components:

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

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

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

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

545 GENIVI has investigated providing an SQLite API for relational data storage,
546 and has provided [recommendations for it](#)³¹, but has not shipped a version with
547 SQLite support (as of version 0.3.0 of this document).

548 To persist an application’s state, the developer must manually implement seri-
549 alisation and deserialisation of all UI and internal state of the application using
550 the Persistence client library.

551 **Secrets and passwords**

552 Similarly, GENIVI has no specialised API for storing secrets and passwords
553 — applications must use the persistence management system. The system does

³¹http://docs.projects.genivi.org/persistence-client-library/1.0/Persistence_ClientLibrary_UserGuide.pdf

554 allow for encrypted storage of persistent data using a plugin — but that encrypts
555 all stored data, including preferences and application state.

556 Approach

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

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

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

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

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

581 As the system explicitly does not support full-disk encryption (for performance
582 reasons), user secrets and passwords should be stored via the freedesktop.org
583 **Secrets D-Bus API**³², rather than the preferences or persistence APIs. The
584 Secrets D-Bus API explicitly handles encryption of the secret store, whereas a
585 general design for a preferences system should have no need for encryption, and
586 hence adding it to the API would be an unnecessary complication for 90% of the
587 use cases. Accordingly, confidential data will not be considered in the approach
588 below.

589 For further discussion and designs on the topic of secrets and passwords, see the
590 **Security design document**³³.

³²<http://standards.freedesktop.org/secret-service/>

³³<https://martyn.pages.apertis.org/apertis-website/concepts/security/>

591 Preferences approach

592 Overall architecture

593 Access to app, user and system settings should be through the GSettings API,
594 most likely backed by dconf. (Refer to [GNOME Linux desktop](#) for an overview
595 of the way GSettings and dconf fit together.) As system settings are defined as
596 those settings which are accessed by multiple components, settings which are
597 solely for the use of a single system service may be stored in other ways, and
598 are beyond the scope of this document.

599 Each component should have its own GSettings schema:

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

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

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

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

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

631 migrate its stored value to a new key, and use the new key in newer versions of
632 the app bundle.

633 Requirements

634 Through the use of the GSettings API, the following requirements are automat-
635 ically fulfilled:

- 636 • **Writability** — using `g_settings_is_writable()`
- 637 • **System and app bundle upgrades** — old keys are either kept, or superseded
638 by new keys with migrated values if their type or semantics change
- 639 • **Factory reset** — for individual keys, using `g_settings_reset()`; support for
640 resetting entire schemas needs to be supported by the designs below
- 641 • **Abstraction level** — GSettings serves as the abstraction layer, with the
642 individual backends below adding no further abstractions
- 643 • **Transactional updates** — GSettings provides `g_settings_delay()`,
644 `g_settings_apply()` and `g_settings_revert()` to implement in-memory
645 transactions which are serialised in the backend on calling `apply`
- 646 • **Concurrency control** — `g_settings_get()` automatically returns the de-
647 fault value if no user-set value exists; there is no atomic API for setting
648 settings
- 649 • **User interface** — `g_settings_bind()` can be used to bind a GSettings key
650 to a particular UI widget, allowing interface UIs to be built easily (not-
651 ing the argument in **User interface** that preferences UIs should not be
652 automatically generated)

653 Other requirements are fulfilled separately:

- 654 • **Control over user interface** — by generating preferences windows from
655 GSettings schemas in the system preferences application (**Searchable pref-**
656 **erences**)
- 657 • **Rearrangeable preferences** — by hard-coding more behaviour in the system
658 preferences application (**User interface**)
- 659 • **Searchable preferences** — searching over summaries and descriptions in
660 GSettings schemas (**Security policy**)
- 661 • **Storage of user secrets and passwords** — using the freedesktop.org Secrets
662 D-Bus API as in the Security design (section 5)

663 **-preferences hard key** — implemented according to the Hard Keys design **pref-**
664 **erences hard key1**)

665 Proxied dconf backend

666 In its current state (May 2015, detailed in [GNOME Linux desktop](#)), dconf does
667 not support the necessary fine-grained access controls for multiple components
668 accessing the preferences. However, a design is being implemented upstream to
669 proxy access to dconf through a separate service which imposes access controls
670 based on AppArmor (mostly implemented as of January 2016).

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

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

678 This would be implemented as the dconf profile:

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

682 All accesses to dconf would go through GSettings, and then through the proxy
683 service which applies AppArmor rules to restrict access to specific settings,
684 implementing the chosen security policy ([Access permissions](#)). The rules may,
685 for example, match against settings path and the AppArmor label of the calling
686 process.

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

688 [Vendor lockdown](#) is supported already by [dconf](#)³⁴ through the use of lockdown
689 files, which specify particular keys or settings sub-trees which may not be mod-
690 ified.

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

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

701 The existing Apertis system puts

³⁴<https://developer.gnome.org/dconf/unstable/dconf-overview.html>

702 `include <abstractions/gsettings>`

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

706 Requirements

707 This design fulfills the following requirements:

- 708 • **Access permissions** — through use of the proxy service and AppArmor
709 rules
- 710 • **Rollback** — by rolling back the user’s per-app database
- 711 • **Factory reset** — by deleting the user’s database or the user’s per-app
712 database
- 713 • **Minimising io bandwidth** — dconf’s database design is optimised for this
- 714 • **Atomic updates** — dconf performs atomic overwrites of the database
- 715 • **Performance tradeoffs** — dconf is heavily optimised for reads rather than
716 writes
- 717 • **Data size tradeoffs** — dconf uses GVDB for storage, so can handle small
718 to large amounts of data
- 719 • **Vendor overrides** — dconf supports vendor overrides inherently
- 720 • **-vendor lockdown** — dconf supports vendor lockdown inherently

721 Development backend

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

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

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

733 Requirements

734 This design fails the following requirements:

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

736 • **Rollback** — **unsupported** by the current version of dconf

737 It supports the following requirements:

738 • **Factory reset** — **partially supported** by deleting the user's database;
739 resetting a (user, app) pair is not supported as all settings are stored in
740 the same dconf database file

741 • **Minimising io bandwidth** — dconf's database design is optimised for this

742 • **Atomic updates** — dconf performs atomic overwrites of the database

743 • **Performance tradeoffs** — dconf is heavily optimised for reads rather than
744 writes

745 • **Data size tradeoffs** — dconf uses GVDB for storage, so can handle small
746 to large amounts of data

747 • **Vendor overrides** — dconf supports vendor overrides inherently

748 **-vendor lockdown** — dconf supports vendor lockdown inherently

749 **Key-file backend**

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

754 • lower read performance due to not being optimised for reads (or in gener-
755 al);

756 • requiring code changes in user services and apps to switch from the key-file
757 backend to the proxied dconf backend once it's ready;

758 • requiring settings values to be migrated from the key-file store to dconf at
759 the time of switch over;

760 • not supporting vendor lockdown or vendor overrides.

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

764 In detail, the approach would be to use a separate key file for each schema in-
765 stance, across all system services, user services and apps. This would require us-
766 ing `g_settings_key_file_backend_new()` and `g_settings_new_with_backend_and_path()`
767 to manually construct the GSettings instance for each schema, using a key file
768 path which corresponds to the schema path.

769 Access control for each schema instance would be enforced using AppArmor
770 rules which restrict access to each key file as appropriate. For example, apps

771 would be given read-only access to the key files for system and user settings,
772 and read–write access to the key file for their own app settings.

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

776 This code for creating the GSettings object could be abstracted away by a helper
777 library, but the API for that library would have to be stable and supported
778 indefinitely, even after changing the backend.

779 Requirements

780 This design fails the following requirements:

- 781 • **Performance tradeoffs** — GKeyFile is **equally non-optimised** for reads
782 and writes
- 783 • **Vendor overrides** — **unsupported** by GKeyFile
- 784 • **-vendor lockdown** — **unsupported** by GKeyFile

785 It supports the following requirements:

- 786 • **Access permissions** — supported by AppArmor rules on the per-schema
787 key files
- 788 • **Rollback** — by snapshotting and rolling back the appropriate key files
- 789 • **Factory reset** — by deleting the appropriate key files
- 790 • **Minimising io bandwidth** — GKeyFile’s I/O bandwidth is proportional to
791 the number of times each key file is loaded and saved
- 792 • **Atomic updates** — GKeyFile performs atomic overwrites of the database
- 793 • **Data size tradeoffs** — GKeyFile’s load and save performance is propor-
794 tional to the amount of data stored in the file, so it is suitable for small
795 amounts of data

796 Security policy

797 All three potential backends enforce security policy through per-app AppArmor
798 rules (if they support implementing security policy at all — the **Development**
799 **backend**, does not).

800 It is beyond the scope of this document to define how each app ships its AppAr-
801 mor rules, and how Apertis can guarantee that third-party apps cannot grant
802 themselves higher privileges using additional rules. The suggestion in section
803 8.3 of the Applications Design document is for the AppArmor rule set for an
804 app to be automatically generated from the app’s manifest file by the app store

805 (which is trusted). The manifest file could contain permissions such as ‘can-
806 change-locale’ or ‘can-add-network’ which would translate to AppArmor rules
807 allowing an app write access to the relevant user and system settings.

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

812 **User interface**

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

- 815 • Individual preferences embedded at different points in the application UI.
- 816 • A preferences window implemented within the application.
- 817 • A system preferences application which controls displaying the preferences
818 for all installed applications, plus system preferences.

819 In all cases, we recommend that preferences are defined using GSettings
820 schemas, as discussed in **Overall architecture**, and that settings are read and
821 written through the **GSettings**³⁵ API. This ensures that access control is
822 enforced, and separates the structure of the preferences (including types and
823 default values) from their presentation.

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

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

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

838 However, automatically generated UIs generally give a bad user experience, due
839 to the limited flexibility a designer has on them, so are suitable only for basic
840 preferences (such as toggle switches; see **Discussion of automatically generated**

³⁵<https://developer.gnome.org/gio/stable/GSettings.html#GSettings.description>

841 versus manually coded preferences UIs). There may be cases where an appli-
842 cation has a particular preference which Apertis provides no widgets suitable
843 for editing it. In these infrequent cases, it must be possible for the system
844 preferences application to execute a stand-alone preferences window from the
845 application to set that particular preference.

846 System preferences application

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

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

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

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

863 The vendor may choose the security policy for which users may edit system
864 preferences (such as the language or background) — they could either allow all
865 users to edit these, or only allow administrative users (such as the vehicle owner)
866 to edit them. If so, we recommend showing the entries for these preferences
867 anyway, but making the widgets insensitive and presenting an authentication
868 dialogue for the administrator to authenticate with before allowing the settings
869 to be edited, see the [Multi-User Transactional Switching document](#)³⁶.

870 Per-application preferences windows

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

875 It should support being launched with the name of a GSettings schema to show,
876 and it would render a preferences window from that schema (see [Generating a
877 preferences window from a GSettings schema file](#)). If the schema file defines

³⁶<https://martyn.pages.apertis.org/apertis-website/concepts/multiuser-transactional-switching/>

878 multiple levels of schema, they should be presented as a hierarchy of pages,
879 with preferences only being shown on leaf pages. It is up to the vendor whether
880 the user can navigate ‘up’ from the top level of the schema to a list of all
881 applications.

882 As the system preferences application is part of the TCB for preferences, it
883 must not allow an application to launch it with the name of a GSettings schema
884 file which does not belong to that application. For example, that would al-
885 low one application to trick the user into editing their preferences for another
886 application.

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

888 A GSettings [schema file](#)³⁷ can be turned into a UI using the following rules:

- 889 • A `<schema>` element is turned into a preference page. If it has an ex-
890 tends attribute, the widgets from the schema it extends are added to the
891 preferences page first.
- 892 • The first non-relocatable `<schema>` element in a `<schemalist>` will be
893 taken as providing the preferences page for the application. Subsequent
894 `<schema>` elements will be ignored unless pulled in as preferences sub-
895 pages using a `<child>` element.
- 896 • A `<child>` element is turned into an entry to show a preferences sub-page
897 for the corresponding sub-schema. The label for this entry should come
898 from a new (non-standard) label attribute on the `<child>` element.
- 899 • Relocatable `<schema>` elements (those without a path attribute) are ig-
900 nored unless pulled in as a preferences sub-page using a `<child>` element.
- 901 • A `<key>` element is turned into a widget with its label set from the
902 `<summary>` element and its description set from the `<description>` ele-
903 ment. The type of widget is set by the type attribute, which specifies a
904 [GVariant type](#)³⁸:
 - 905 – b (boolean): Switch or checkbox widget.
 - 906 – y, n, q, i, u, x, t (integers): Integer spin button. Its range is set to
907 the smaller of the bounds of the integer type or the values of the
908 `<range>` element (if present).
 - 909 – h (handle): Not supported.
 - 910 – d (double): Floating point spin button. Its range is set to the smaller
911 of the bounds of the double type or the values of the `<range>` element
912 (if present).

³⁷<https://gitlab.gnome.org/GNOME/glib/-/blob/main/gio/gschema.dtd>

³⁸<https://developer.gnome.org/glib/stable/glib-GVariantType.html#id-1.6.18.6.9>

- 913 – s (string): Text entry widget. If a <choices> element is present, a
- 914 drop-down box should be used instead, displaying the options from
- 915 the <choice> elements.
- 916 – o (object path): Not supported.
- 917 – g (type string): Not supported.
- 918 – ? (basic type): Not supported.
- 919 – v (variant): Not supported.
- 920 – a (array): Not supported in any form.
- 921 – m (maybe): Not supported in any form.
- 922 – (), r (tuple): Not supported in any form.
- 923 – {} (dictionary): Not supported in any form.
- 924 – * (any): Not supported in any form.
- 925 • If a <key> element contains an enum attribute and no type attribute,
- 926 a drop-down box should be used, displaying the options from the nick
- 927 attributes of the <value> elements in the corresponding <enum> element.
- 928 • If a <key> element contains a flags attribute and no type attribute, a
- 929 checkbox list should be used, displaying a checkbox for each each of the
- 930 nick attributes of the <value> elements in the corresponding <flags>
- 931 element.
- 932 • If a key’s name attribute matches a mapping to a wizard application
- 933 (see [Support for custom preferences windows](#)) in the application’s man-
- 934 ifest, that key should be displayed as a menu entry which, when selected,
- 935 launches the wizard application as a new window.

936 **Support for custom preferences windows**

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

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

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

948 For example, a particular manifest could map the key `/org/foo/MyApp/complex-`
949 `setting` to the command `my-app --show-complex-setting`. Or a manifest could
950 map the schema `/org/foo/MyApp/EmailAddress` to the command `my-app`
951 `--configure-email-account`.

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

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

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

964 **Searchability of preferences**

965 To allow the system preferences application to search over all applications' pref-
966 erences ([Searchable preferences](#)), it must load all the GSettings schemas from
967 applications whose manifests specify a schema. Searching must be performed
968 over the user-visible parts of the schema (the `<summary>` and `<description>`
969 elements), and results should be returned as a link to the relevant application
970 preferences window. System preferences should be included in the search results
971 too.

972 **Reorganising preferences**

973 Implementing arbitrary reorganisation of preferences ([Rearrangeable prefer-](#)
974 [ences](#)) is difficult, as that requires an OEM to know the semantics of all prefer-
975 ences for all possibly installable applications.

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

981 An alternative implementation which is more flexible, but which devolves more
982 control to application developers, is to tag each preference in the GSettings
983 schemas with well-defined tags which summarise the preference's semantics.
984 For example, an application's preference for whether to submit usage data to
985 the application data could be tagged as 'privacy'; or a preference determining
986 the colour scheme to use in an application could be tagged as 'appearance'.
987 The OEM could then implement a custom preferences window which queries

988 all installed GSettings schemas for a specific tag and displays the resulting
989 preferences. We do not recommend this option, as even with app store validation
990 of the chosen tags, this would allow application developers too much control over
991 the appearance of a system preferences window.

992 **Preferences list widget**

993 In order to help make all preferences UIs consistent (including those imple-
994 mented by the vendor, [System preferences application](#); and those implemented
995 by application developers as wizard applications, [Per-application preferences
996 windows](#)), Apertis should provide a standard widget which implements the con-
997 version from GSettings schemas to UI as described in [Generating a preferences
998 window from a GSettings schema file](#).

999 This widget should accept a list of GSettings schema paths to display, and may
1000 optionally accept a list of keys within those schemas to display (ignoring the
1001 others), or to ignore (displaying the others); and should display all those keys
1002 as preferences. It should implement reading and writing the keys' values using
1003 the GSettings API, and must assume that the application has permission to do
1004 so (see [Security policy](#)). It must check for writability of preferences and make
1005 them insensitive if they are read-only (see [vendor lockdown1](#)). It cannot give the
1006 application more permissions than it already has.

1007 If application developers use this widget, the vendor can ensure that preferences
1008 UIs are consistent between applications and the system preferences application
1009 through the theming of the widget.

1010 **Vendor lockdown**

1011 If the vendor locks down a key in a GSettings schema for an application (or
1012 system preference) [vendor lockdown](#) — supported by [Proxied dconf backend](#)
1013 and [Development backend](#), but not [Key-file backend](#)), that is enforced by the
1014 underlying settings service (most likely dconf), and cannot be overridden or
1015 worked around by applications.

1016 However, it is up to applications to reflect whether a preference is read-only
1017 (due to being locked down) in their UIs. This is typically achieved by hid-
1018 ing a preference or making its widget insensitive. Applications can use the
1019 [g_settings_is_writable](#)³⁹ method to determine whether a preference is read-
1020 only. Any preferences widgets provided by Apertis ([Preferences list widget](#))
1021 must implement this already.

1022 If an application developer uses a custom widget to display a preference, and
1023 forgets to check whether that preference is read-only, their application might
1024 enter an inconsistent state (which is their fault), but the system will not let

³⁹<https://developer.gnome.org/gio/unstable/GSettings.html#g-settings-is-writable>

1025 that preference be written. Convenience APIs like `g_settings_bind_writable`⁴⁰
1026 can reduce the risk of this happening.

1027 **Discussion of automatically generated versus manually coded prefer-** 1028 **ences UIs**

1029 In an ideal world, our recommendation would be that: while automatically
1030 generating preference UIs can rapidly produce rough drafts, in our experience
1031 it can never result in a high-quality finished UI with:

- 1032 • logically grouped options;
- 1033 • correctly aligned controls;
- 1034 • a concept of which preferences are most important, which ones are ‘ad-
1035 vanced’, and which ones should be hidden;
- 1036 • conditional defaults (for example, when you set up IMAP e-mail, the
1037 default port should be 143, except if you have selected old-style SSL in
1038 which case it should be 993); and
- 1039 • the ability to hide or disable preferences that do not apply because of
1040 the value of another preference (for example, if you switch off Bluetooth
1041 completely, then the widget to change the name that is broadcast over
1042 Bluetooth should be hidden or disabled).

1043 If the uniform appearance of preferences UIs is a concern, we believe this should
1044 be addressed through: convention; the default appearance of widgets in the UI
1045 toolkit; and the use of a set of human interface guidelines such as the [GNOME](#)
1046 [HIG](#)⁴¹. Specifically, we recommend that preferences are:

- 1047 • integrated into the main application UI if there are only a small number
1048 of them;
- 1049 • `instant-apply`⁴² unless doing so would be dangerous, in which case they
1050 should be `explicit-apply` for all preferences in the dialogue (for example,
1051 changing monitor resolutions is dangerous, and hence is `explicit-apply`);
1052 and
- 1053 • grouped logically in the UI.

1054 If, after the preferences UIs of several applications have been implemented, some
1055 common widget patterns have been identified, we suggest that they could be
1056 abstracted out into new widgets in the UI toolkit. The goal of this would be to
1057 increase consistency between preferences UIs, without implementing essentially
1058 a separate UI toolkit for them, which would be the result of any template- or
1059 auto-generation-based approach.

⁴⁰<https://developer.gnome.org/gio/stable/GSettings.html#g-settings-bind-writable>

⁴¹<https://developer.gnome.org/hig/stable/dialogs.html.en>

⁴²<https://developer.gnome.org/hig/stable/dialogs.html.en#instant-and-explicit-apply>

1060 An alternative way of thinking about this is that preferences are subject to a
1061 model–view split (the model is GSettings schema files; the view is the prefer-
1062 ences UI), and it is typically inadvisable to generate a view from a model when
1063 following that pattern.

1064 However, we realise that the goal of having a unified system preferences ap-
1065 plication with a consistent appearance (which is enforced) conflicts with these
1066 recommendations, and hence these recommendations are not part of our overall
1067 suggested approach.

1068 **Preferences hard key**

1069 A preferences hard key must be supported as detailed in the Hard Keys de-
1070 sign. In a configuration where a system preferences application is used, it must
1071 launch that application, already open on the preferences window for the active
1072 application. If no application is active, or if the currently active application has
1073 no GSettings schemas listed in its manifest file, the main page of the system
1074 preferences application should be shown.

1075 In a configuration where applications implement their own preferences windows,
1076 the active application must be sent a ‘hard key pressed’ signal for the preferences
1077 hard key, which the application can handle how it wishes (i.e. by showing its
1078 preferences window). If there is no active application, the system preferences
1079 application (which in this configuration only contains system preferences) should
1080 be shown.

1081 The policy for exactly what happens in each situation and configuration is under
1082 the control of the hard keys service, which is provided by the vendor. It should
1083 have access to the manifest for the active application so it can find information
1084 about GSettings schemas.

1085 **Existing preferences schemas**

1086 As GSettings is used widely within the open source software components used
1087 by Apertis, particularly GNOME, there are many standard GSettings schemas
1088 for common user settings. We recommend that Apertis re-use these schemas as
1089 much as possible, as support for them has already been implemented in various
1090 components. If that is not possible, they could be studied to ensure we learn
1091 from their design successes or failures.

- 1092 • org.gnome.system.locale
- 1093 • org.gnome.system.proxy
- 1094 • org.gnome.desktop.default-applications
- 1095 • org.gnome.desktop.media-handling
- 1096 • org.gnome.desktop.interface

- 1097 • org.gnome.desktop.lockdown
- 1098 • org.gnome.desktop.background
- 1099 • org.gnome.desktop.notifications
- 1100 • org.gnome.crypto
- 1101 • org.gnome.desktop.privacy
- 1102 • org.gnome.system.dns_sd
- 1103 • org.gnome.desktop.sound
- 1104 • org.gnome.desktop.datetime
- 1105 • org.gnome.system.location
- 1106 • org.gnome.desktop.thumbnailers
- 1107 • org.gnome.desktop.thumbnail-cache
- 1108 • org.gnome.desktop.file-sharing

1109 Various Apertis dependencies (for example, Mutter, Tracker, libfolks, IBus, Geo-
 1110 clue, Telepathy) use their own GSettings schemas already — as these are not
 1111 shared, they are not listed.

1112 *Alternative model:* If the locale is a system setting, rather than a user setting,
 1113 systemd’s [localed](#)⁴³ should be used. This would require the locale to be changed
 1114 via the locale D-Bus API, rather than GSettings, which would affect the im-
 1115 plementation of the system preferences app.

1116 Persistent data approach

1117 Overall architecture

1118 As discussed in sections 5.3.1 and 7 of the Applications Design, and the Multi-
 1119 user Design, there is a difference between state which an app needs to persist
 1120 (for example, if it is being terminated to switch users), and state which an app
 1121 explicitly needs to share (for example, if a transactional user switch is taking
 1122 place to execute an action as a different user). The Multiuser Design encourages
 1123 app authors to think explicitly about these two sets of state, and the differences
 1124 between them. It is the app which chooses the state to persist, rather than the
 1125 operating system — storage space is too limited to persist the entire address
 1126 space of an app, effectively suspending it.

1127 The state each app chooses to persist will differ, and cannot be predicted by
 1128 Apertis. There could be a lot of state, or very little. It could be representable as
 1129 a simple key–value dictionary, or might have a complex hierarchical structure.

⁴³<http://www.freedesktop.org/wiki/Software/systemd/localed/>

1130 Well-known state directories

1131 As mentioned in the Applications Design document (sections 5.3.1 and 7),
1132 we recommend that Apertis provide a per-(user, app) directory for storage
1133 of persisted data, and a public API the app can call to find out that di-
1134 rectory. The API should differentiate between cache and non-cache state,
1135 with cache state going in `$XDG_CACHE_HOME/net.example.MyApp/` and
1136 non-cache state going in `$XDG_DATA_HOME/net.example.MyApp/`. Alter-
1137 natively, as suggested in the Applications Design, the latter could be `/Appli-
1138 cations/net.example.MyApp/Storage/username/state/`. This has the advantage
1139 of allowing all data for a particular app to be removed by deleting `/Appli-
1140 cations/net.example.MyApp`, at the cost of not following the XDG standard used
1141 by most existing software. This fulfils the factory reset requirement (**Factory
1142 reset**).

1143 The former is effectively equivalent to a per-(user, app) `XDG_CACHE_HOME`
1144 directory, and the latter to a `XDG_DATA_HOME`, as defined by the [XDG Base
1145 Directory Specification](#)⁴⁴.

1146 AppArmor rules should exist to allow apps to write to these directories (and
1147 not to other apps' state directories). This is the extent of the security needed,
1148 as state storage is simply an interaction between an app and the filesystem.

1149 This approach automatically allows for rollback of persistent data (**Rollback**)
1150 using the normal snapshotting mechanism described in the Applications Design
1151 document.

1152 As with preferences, app bundles must be in charge of upgrading their own per-
1153 sistent data when the system is upgraded (or the app is upgraded) (**System and
1154 app bundle upgrades**). Recommendations are given in the subsections below.

1155 Recommended serialisation APIs

1156 As each app's state storage requirements are different, we suggest that Apertis
1157 provide several recommended serialisation APIs, and allow apps to choose the
1158 most appropriate one — or something completely different if that fulfils their
1159 requirements better.

1160 Alongside, Apertis should provide guidelines to app developers to allow them
1161 to choose an appropriate serialisation API, and avoid common problems in se-
1162 rialisation:

- 1163 • minimise writes to main storage (**Minimising io bandwidth**);
- 1164 • ensure all updates to stored state are atomic (requirement **Atomic up-
1165 dates**); and
- 1166 • ensure transactions are used for groups of updates where appropriate (**1167 Transactional updates**).

⁴⁴<http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html>

1168 Atomic in the sense that either the old or new states are stored in
1169 entirety, rather than some intermediate state, if power is lost part-
1170 way through an update.

1171 Depending on the requirements it is believed that apps will have, some or all of
1172 the following APIs could be recommended for serialising state to main storage.
1173 For comparison, Android only provides a generic file storage API, and an SQLite
1174 API, with no implemented [key-value store APIs](#)⁴⁵. Apps must implement those
1175 themselves.

1176 **GKeyFile**

1177 <https://developer.gnome.org/glib/stable/glib-Key-value-file-parser.html>

1178 Suitable for small amounts of key-value state with simple types. Suitable for
1179 small amounts of data.

1180 All updates to a GKeyFile are atomic, as it uses the atomic-overwrite technique:
1181 the new file contents are written to a temporary file, which is then atomically
1182 renamed over the top of the old file. Transactional updates can be implemented
1183 by saving the key file to apply the transaction, and discarding the in-memory
1184 GKeyFile object to revert it.

1185 The amount of I/O with a GKeyFile is small, as the amount of data which
1186 should be stored in a GKeyFile is small, and the file is only written out when
1187 explicitly requested by the app.

1188 System upgrades have to be handled manually by app bundles — if the persis-
1189 tence data format has to change, the app must migrate data from the old format
1190 to the new format the first time it is run after an upgrade. In this case, it is
1191 recommended that all GKeyFiles used for persistent data contain a ‘Version’
1192 key specifying the data format version in use.

1193 **GVDB**

1194 <https://git.gnome.org/browse/gvdb>

1195 Memory-mapped hash table with [GVariant](#)⁴⁶-style types, suitable for small to
1196 large amounts of data which are read much more frequently than they are writ-
1197 ten. This is what dconf uses for storage.

1198 All updates to a GVDB file are atomic, as it uses the same atomic-overwrite
1199 technique as [GKeyFile](#). Transactions are supported similarly — by writing out
1200 the updated database or discarding it.

1201 The amount of I/O for reads from a GVDB file is small, as it memory-maps
1202 the database, so only pages in the data it actually reads (plus some metadata).

⁴⁵<http://developer.android.com/guide/topics/data/data-storage.html>

⁴⁶<https://developer.gnome.org/glib/stable/glib-GVariant.html>

1203 Writes require the entire file to be updated, but are only done when explicitly
1204 requested by the app.

1205 GVDB supports per-file versioning (though this is not currently exposed in
1206 the public API). This can be used for handling system upgrades ([System and
1207 app bundle upgrades](#)) — the database must be explicitly migrated from an old
1208 version to a new version when an upgraded app is first started.

1209 SQLite

1210 <http://sqlite.org/>

1211 <https://wiki.gnome.org/Projects/Gom>

1212 Full SQL database implementation, supporting simple SQL types and more
1213 complex relational types if implemented manually by the app. Suitable for
1214 medium to large amounts of data which are read and written frequently. It
1215 supports SQL transactions.

1216 SQLite is not a panacea. It is designed for the specific use pattern of SQL
1217 databases with indexes and relational tables, with frequent reads and writes,
1218 and infrequent deletions of data. Apps will only get the best performance from
1219 SQLite by defining their own table structure, indices and relations; imposing a
1220 common key–value-style API on top of SQLite would give lower performance.

1221 SQLite has limited support for SQL schema upgrades with its [ALTER TABLE](#)⁴⁷
1222 statement, which supports renaming tables and adding new columns to tables.
1223 Apps must implement their own data migration from old to new versions of
1224 their database schema; documenting this is beyond the scope of this design.

1225 Apps should only use SQLite if they have considered issues like their vacuuming
1226 policy — how frequently to vacuum the database after deleting data from it.
1227 See:

- 1228 • https://blogs.gnome.org/jnelson/2015/01/06/sqlite-vacuum-and-auto_vacuum/
- 1229 • https://wiki.mozilla.org/Performance/Avoid_SQLite_In_Your_Next_Firefox_Feature

1230 If using GObject to represent entries in an SQLite database, the [GOM](#)⁴⁸ wrap-
1231 per around SQLite may be useful to simplify code.

1232 GNOME-DB

1233 <http://www.gnome-db.org/>

1234 This is **not** recommended. It is an abstraction layer over multiple SQL database
1235 implementations, allowing apps to access remote SQL databases. In almost all
1236 cases, directly using [Sqlite](#) is a more appropriate choice.

⁴⁷https://www.sqlite.org/lang_altertable.html

⁴⁸<https://wiki.gnome.org/Projects/Gom>

1237 **When to save persistent data**

1238 As specified in the Applications Design (section 5.3.1), state is saved to main
1239 storage at times chosen by both the operating system and the app. The oper-
1240 ating system knows when the logged in user is about to change, or when the
1241 system is about to be shut down; the app knows when it has changed some of
1242 its persistent state in memory, and hence needs to write it out to main storage.

1243 An action could be implemented in each app which is triggered by the Acti-
1244 vateAction method of the org.freedesktop.Application [D-Bus interface](#)⁴⁹ if, for
1245 example, that interface is implemented by apps. When triggered, this action
1246 would cause the app to store its persistent state.

1247 **Recently used and favourite items**

1248 Section 6.3 of the Global Search Design specifies that an API for apps to store
1249 their favourite and recently used items in will be provided. As this is data shared
1250 from an app to the operating system, and is typically append-only rather than
1251 strongly read-write, we recommend that it be designed separately from the
1252 persistent data API covered in this document, following the recommendations
1253 given in the Global Search Design document.

1254 **Summary of recommendations**

1255 As discussed in the above sections, we recommend:

- 1256 • Splitting preferences, persistent data storage and confidential data storage
1257 ([Approach](#)).
- 1258 • Providing one API for preferences: GSettings ([Overall architecture](#)).
- 1259 • Apps provide a GSettings schema file for their preferences, named after
1260 the app ([Overall architecture](#)).
- 1261 • Existing GSettings schemas are re-used where possible for user and system
1262 settings ([Existing preferences schemas](#)).
- 1263 • Using the normal GSettings approach for handling app upgrades ([Overall
1264 architecture](#)).
- 1265 • Developing against the normal dconf backend for GSettings (section [De-
1266 velopment backend](#)).
- 1267 • Switching to the proxied dconf backend once it's ready, to support access
1268 control ([Proxied dconf backend](#)).
- 1269 • A key-file backend is an alternative we do *not* recommend ([Key-file back-
1270 end](#)).

⁴⁹<http://standards.freedesktop.org/desktop-entry-spec/desktop-entry-spec-latest.html#dbus>

- 1271 • Permissions to modify user or system settings are controlled by the app's
1272 manifest ([Security policy](#)).
- 1273 • Permissions are converted to backend-specific AppArmor rules by the app
1274 store ([Security policy](#)).
- 1275 • User interfaces for preferences are provided by the vendor, automatically
1276 generated from GSettings schemas; or provided by applications ([User
1277 interface](#)).
- 1278 • Apertis provides a standard widget to present GSettings schemas as a
1279 preferences UI ([Preferences list widget](#)).
- 1280 • Preferences hard key support is added according to the Hard Keys design
1281 [preferences hard key](#)).
- 1282 • Providing API to get a persistent data storage location ([Well known state
1283 directories](#)).
- 1284 • Persistent data is private to each (user, app) pair ([Well known state
1285 directories](#)).
- 1286 • Recommending various different data storage APIs to suit different apps'
1287 use cases ([Recommended serialisation APIs](#)).
- 1288 • Apps explicitly define which data will persist, and are responsible for sav-
1289 ing it and migrating it from older to newer versions ([Overall architecture](#)).
- 1290 • Apps can be instructed to save their persistent state by the operating
1291 system via a D-Bus interface ([When to save persistent data](#)).
- 1292 • User secrets and passwords are stored using the freedesktop.org Secrets
1293 D-Bus API, not the Apertis preferences or persistence APIs ([Approach](#)).