Click Packages Documentation

Release 0.4.45.1+16.10.20160916-0ubuntu1

Colin Watson

2016-09-16

Contents

1 Compatibility **3**

2 Build **5**
2.1 Dependencies 5

3 Testing **7**
3.1 Test coverage 7
3.2 Integration Tests 7

4 Documentation **9**
4.1 “Click” package file format, version 0.4 9
4.2 Design constraints 12
4.3 Hooks 12
4.4 Databases 15
4.5 To do 16

5 Indices and tables **17**

Click is the code name used to describe a packaging format for Ubuntu mobile applications. This format specifies how individual apps are delivered to mobile devices, how they are packed into distributable format, and how they are installed on a mobile device by a system provided package manager. At a minimum they assume that a system framework exists providing all the necessary infrastructure and dependencies needed in order to install and run such apps.

The click packaging format is completely independent from facilities to do full-system installations or upgrades.

CHAPTER 1

Compatibility

Currently, this package should remain compatible with Python 2.7, 3.2, 3.3, and 3.4; Ubuntu 12.04 LTS, Ubuntu 13.10, and Ubuntu 14.04 LTS.

CHAPTER 2

Build

If you run from a fresh bzr checkout, please ensure you have the required build dependencies first by running:

```
$ dpkg-checkbuilddeps
```

and installing anything that is missing here.

Then run:

```
$ ./autogen.sh
$ ./configure --prefix=/usr \
    --sysconfdir=/etc \
    --with-systemdsystemunitdir=/lib/systemd/system \
    --with-systemduserunitdir=/usr/lib/systemd/user
$ make
```

to build the project.

Dependencies

For Ubuntu 14.04, make sure you have the *python2.7* and *python3.4* packages installed. Unless you upgraded from a previous version of Ubuntu and haven't removed it yet, you won't have Python 3.3 and Python 3.2 available. Build them from source if necessary, install them say into `/usr/local`, and make sure they are on your `$PATH`.

You'll need *tox* (Ubuntu package *python-tox*) installed in order to run the full test suite. You should be able to just say:

```
$ tox
```

to run the full suite. Use tox's `-e` option to run the tests against a subset of Python versions. You shouldn't have to install anything manually into the virtual environments that tox creates, but you might have to if you don't have all the dependencies installed in your system Pythons.

You'll need the *mock* and *python-debian* libraries. For Ubuntu 13.10, apt-get install the following packages:

```
* python-mock
* python-debian
* python3-debian
```

CHAPTER 3

Testing

After all of the above is installed, you can run `tox` to run the test suite against all supported Python versions. The `./run-tests` scripts just does an additional check to make sure you've got the preload shared library built.

To run a specific testcase, use the standard python unittest syntax like:

```
$ python3 -m unittest click.tests.test_install
```

or:

```
$ python2 -m unittest click.tests.test_build.TestClickBuilder.test_build
```

Test coverage

If you have python-coverage installed, you can get a Python test coverage report by typing:

$ python-coverage combine $ python-coverage report

This works also for python3-coverage.

To get Vala/C coverage information, install the gcovr and lcov packages and run:

$ ./configure –enable-gcov $ make coverage-html

which will generate a "coveragereport/index.html" file for you.

The combined coverage information can be obtained via:

$ make coverage.xml

Integration Tests

There is also a set of integration tests that have additional test dependencies that are listed in debian/test/control.

Beware that some require to be run as root and they are designed to be run in a safe environment (like a schroot or a autopkgtest container) and may alter the system state (e.g adding test users). By default the tests will run against the installed click binary, but you can also use:

```
$ LD_LIBRARY_PATH=$(pwd)/lib/click/.libs PYTHONPATH=$(pwd)
    GI_TYPELIB_PATH=$(pwd)/lib/click CLICK_BINARY=$(pwd)/bin/click
    TEST_INTEGRATION=1 python3 -m unittest discover click.tests.integration
```

to run against the build tree.

CHAPTER 4

Documentation

To build the HTML version of the documentation, you'll need Sphinx (Ubuntu package *python-sphinx*). Then do:

```
$ (cd doc && make html)
```

Contents:

"Click" package file format, version 0.4

This specification covers a packaging format intended for use by self-contained third-party applications. It is intentionally designed to make it easy to create such packages and for the archive of packages to be able to scale to very large numbers, as well as to ensure that packages do not execute any unverified code as root during installation and that installed packages are sandboxable.

This implementation proposal uses the existing dpkg as its core, although that is entirely concealed from both users and application developers. The author believes that using something based on dpkg will allow us to reuse substantial amounts of package-management-related code elsewhere, not least the many years of careful design and bug-fixing of dpkg itself; although there are clearly several things we need to adjust.

General format

The top-level binary format for Click packages is an ar archive containing control and data tar archives, as for .deb packages: see deb(5) for full details.

The deb(5) format permits the insertion of underscore-prefixed ar members, so a "_click-binary" member should be inserted immediately after "debian-binary"; its contents should be the current version number of this specification followed by a newline. This makes it possible to assign a MIME type to Click packages without having to rely solely on their extension.

Despite the similar format, the file extension for these packages is .click, to discourage attempts to install using dpkg directly (although it is still possible to use dpkg to inspect these files). Click packages should not be thought of as .deb packages, although they share tooling. Do not rely on the file extension remaining .click; it may change in the future.

Control area

control

Every Click package must include the following control fields:

- Click-Version: the current version number of this specification

The package manager must refuse to process packages where any of these fields are missing or unparseable. It must refuse to process packages where Click-Version compares newer than the corresponding version it implements (according to rules equivalent to "dpkg –compare-versions"). It may refuse to process packages whose Click-Version field has an older major number than the version it implements (although future developers are encouraged to maintain the maximum possible degree of compatibility with packages in the wild).

Several other fields are copied from the manifest, to ease interoperation with Debian package manipulation tools. The manifest is the primary location for these fields, and Click-aware tools must not rely on their presence in the control file.

All dependency relations are forbidden. Packages implicitly depend on the entire contents of the Click system framework they declare.

manifest

There must be a "manifest" file in the control area (typically corresponding to "manifest.json" in source trees), which must be a dictionary represented as UTF-8-encoded JSON. It must include the following keys:

- name: unique name for the application
- version: version number of the application
- framework: the system framework(s) for which the package was built
- installed-size: the size of the unpacked package in KiB; this should not be set directly in the source tree, but will be generated automatically by "click build" using "du -k -s –apparent-size"

The package manager must refuse to process packages where any of these fields are missing or unparseable. It must refuse to process packages where the value of "framework" does not declare a framework implemented by the system on which the package is being installed.

The value of "name" identifies the application, following Debian source package name rules; every package in the app store has a unique "name" identifier, and the app store will reject clashes. It is the developer's responsibility to choose a unique identifier. The recommended approach is to follow the Java package name convention, i.e. "com.mydomain.myapp", starting with the reverse of an Internet domain name owned by the person or organisation developing the application; note that it is not necessary for the application to contain any Java code in order to use this convention.

The value of "version" provides a unique version for the application, following Debian version numbering rules. See deb-version(5) for full details.

The syntax of "framework" is formally that of a Debian dependency relationship field. Currently, only a simple name is permitted, e.g. "framework": "ubuntu-sdk-13.10", or a list of simple names all of which must be satisfied, e.g. "framework": "ubuntu-sdk-14.04-qml, ubuntu-sdk-14.04-webapps"; version relationships and alternative dependencies are not currently allowed.

The manifest may contain arbitrary additional optional keys; new optional keys may be defined without changing the version number of this specification. The following are currently recognised:

- title: short (one-line) synopsis of the application
- description: extended description of the application; may be multi-paragraph

- maintainer: name and email address of maintainer of the application
- architecture: one of the following:
  - "all", indicating a package containing no compiled code
  - a dpkg architecture name (e.g. "armhf") as a string, indicating a package that will only run on that architecture
  - a list of dpkg architecture names, indicating a package that will run on any of those architectures
- hooks: see *Hooks*
- icon: icon to display in interfaces listing click packages; if the name refers to an existing file when resolved relative to the base directory of the package, the given file will be used; if not, the algorithm described in the Icon Theme Specification will be used to locate the icon

Keys beginning with the two characters "x-" are reserved for local extensions: this file format will never define such keys to have any particular meaning.

Keys beginning with an underscore ("_") are reserved for use as dynamic properties of installed packages. They must not appear in packages' manifest files, and attempts to set them there will be ignored. The following dynamic keys are currently defined:

- _directory: the directory where a package is unpacked
- _removable: 1 if a package is unpacked in a location from which it can be removed, otherwise 0 (this may be changed to a proper boolean in future; client code should be careful to permit either)

Maintainer scripts

Maintainer scripts are forbidden, with one exception: see below. (If they are permitted in future, they will at most be required to consist only of verified debhelper-generated fragments that can be statically analysed.) Packages in Click system frameworks are encouraged to provide file triggers where appropriate (e.g. "interest /usr/share/facility"); these will be processed as normal for dpkg file triggers.

The exception to maintainer scripts being forbidden is that a Click package may contain a preinst script with the effect of causing direct calls to dpkg to refuse to install it. The package manager must enforce the permitted text of this script.

Data area

Unlike .debs, each package installs in a self-contained directory, and the filesystem tarball must be based at the root of that directory. The package must not assume any particular installation directory: if it needs to know where it is installed, it should look at argv[0] or similar.

Within each package installation directory, the ".click" subdirectory will be used for metadata. This directory must not be present at the top level of package filesystem tarballs; the package manager should silently filter it out if present. (Rationale: scanning the filesystem tarball in advance is likely to impose a performance cost, especially for large packages.)

The package manager should ensure that all unpacked files and directories are group- and world-readable, and (if owner-executable) also group- and world-executable. (Rationale: since packages are unpacked as a dedicated user not used when running applications, and since packages cannot write to their own unpack directories, any files that aren't world-readable are unusable.)

Design constraints

Building packages

- Building packages should not require any more than the Python standard library. In particular, it should not require dpkg, python-debian, or any other such Debian-specific tools.

  Rationale: We want people to be able to build Click packages easily on any platform (or at least any platform that can manage a Python installation, which is not too onerous a requirement).

Installing packages

- For the purpose of rapid prototyping, package installation is also implemented in Python. This may of course use Debian/Ubuntu-specific tools, since it will always be running on an Ubuntu system. In future, it will probably be re-implemented in C for performance.
- Reading the system dpkg database is forbidden. This is partly to ensure strict separation, and partly because the system dpkg database is large and therefore slow to read.
- Nothing should require root, although it may be acceptable to make use of root-only facilities if available (but remembering to pay attention to performance).

Hooks

Rationale

Of course, any sensible packaging format needs a hook mechanism of some kind; just unpacking a filesystem tarball isn't going to cut it. But part of the point of Click packages is to make packages easier to audit by removing their ability to run code at installation time. How do we resolve this? For most application packages, the code that needs to be run is to integrate with some system package; for instance, a package that provides an icon may need to update icon caches. Thus, the best way to achieve both these goals at once is to make sure the code for this is always in the integrated-with package.

dpkg triggers are useful prior art for this approach. In general they get a lot of things right. The code to process a trigger runs in the postinst, which encourages an approach where trigger processing is a subset of full package configuration and shares code with it. Furthermore, the express inability to pass any user data through the trigger activation mechanism itself ensures that triggers must operate in a "catch up" style, ensuring that whatever data store they manage is up to date with the state of the parts of the file system they use as input. This naturally results in a system where the user can install integrating and integrated-with packages in either order and get the same result, a valuable property which developers are nevertheless unlikely to test explicitly in every case and which must therefore be encouraged by design.

There are two principal problems with dpkg triggers (aside from the point that not all integrated-with packages use them, which is irrelevant because they don't support any hypothetical future hook mechanisms either). The first is that the inability to pass user data through trigger activation means that there is no way to indicate where an integrating package is installed, which matters when the hook files it provides cannot be in a single location under /usr/ but might be under /opt/ or even in per-user directories. The second is that processing dpkg triggers requires operating on the system dpkg database, which is large and therefore slow.

Let us consider an example of the sort that might in future be delivered as a Click package, and one which is simple but not too simple. Our example package (com.ubuntu.example) delivers an AppArmor profile and two .desktop files. These are consumed by apparmor and desktop-integration (TBD) respectively, and each lists the corresponding directory looking for files to consume.

We must assume that in the general case it will be at least inconvenient to cause the integrated-with packages to look in multiple directories, especially when the list of possible directories is not fixed, so we need a way to cause files to exist in those directories. On the other hand, we cannot unpack directly into those directories, because that takes us back to using dpkg itself, and is incompatible with system image updates where the root file system is read-only. What we can do with reasonable safety is populate symlink farms.

Specification

- Only system packages (i.e. .debs) may declare hooks. Click packages must be declarative in that they may not include code executed outside AppArmor confinement, which precludes declaring hooks.
- "System-level hooks" are those which operate on the full set of installed package/version combinations. They may run as any (system) user. (Example: AppArmor profile handling.)
- "User-level hooks" are those which operate on the set of packages registered by a given user. They run as that user, and thus would generally be expected to keep their state in the user's home directory or some similar user-owned file system location. (Example: desktop file handling.)
- System-level and user-level hooks share a namespace.
- A Click package may contain one or more applications (the common case will be only one). Each application has a name.
- An "application ID" is a string unique to each application instance: it is made up of the Click package name, the application name (must consist only of characters for a Debian source package name, Debian version and [A-Z]), and the Click package version joined by underscores, e.g. `com.ubuntu.clock_alarm_0.1`.
- A "short application ID" is a string unique to each application, but not necessarily to each instance of it: it is made up of the Click package name and the application name (must consist only of characters for a Debian source package name, Debian version and [A-Z]) joined by an underscore, e.g. `com.ubuntu.clock_alarm`. It is only valid in user-level hooks, or in system-level hooks with `Single-Version: yes`.
- An integrated-with system package may add `*.hook` files to `/usr/share/click/hooks/`. These are standard Debian-style control files with the following keys:

  User-Level: yes (optional) If the `User-Level` key is present with the value `yes`, the hook is a user-level hook.

  Pattern: <file-pattern> (required) The value of `Pattern` is a string containing one or more substitution placeholders, as follows:

  `${id}` The application ID.

  `${short-id}` The short application ID (user-level or single-version hooks only).

  `${user}` The user name (user-level hooks only).

  `${home}` The user's home directory (user-level hooks only).

  `$$` The character '`$`'.

  At least one `${id}` or `${short-id}` substitution is required. For user-level hooks, at least one of `${user}` and `${home}` must be present.

  On install, the package manager creates the target path as a symlink to a path provided by the Click package; on upgrade, it changes the target path to be a symlink to the path in the new version of the Click package; on removal, it unlinks the target path.

  The terms "install", "upgrade", and "removal" are taken to refer to the status of the hook rather than of the package. That is, when upgrading between two versions of a package, if the old version uses a given hook

but the new version does not, then that is a removal; if the old version does not use a given hook but the new version does, then that is an install; if both versions use a given hook, then that is an upgrade.

For system-level hooks, one target path exists for each unpacked version, unless "`Single-Version: yes`" is used (see below). For user-level hooks, a target path exists only for the current version registered by each user for each package.

Upgrades of user-level hooks may leave the symlink pointed at the same target (since the target will itself be via a `current` symlink in the user registration directory). `Exec` commands in hooks should take care to check the modification timestamp of the target.

Exec: <program> (optional) If the `Exec` key is present, its value is executed as if passed to the shell after the above symlink is modified. A non-zero exit status is an error; hook implementors must be careful to make commands in `Exec` fields robust. Note that this command intentionally takes no arguments, and will be run on install, upgrade, and removal; it must be written such that it causes the system to catch up with the current state of all installed hooks. `Exec` commands must be idempotent.

Trigger: yes (optional) It will often be valuable to execute a dpkg trigger after installing a Click package to avoid code duplication between system and Click package handling, although we must do so asynchronously and any errors must not block the installation of Click packages. If "`Trigger: yes`" is set in a `*.hook` file, then "`click install`" will activate an asynchronous D-Bus service at the end of installation, passing the names of all the changed paths resulting from Pattern key expansions; this will activate any file triggers matching those paths, and process all the packages that enter the triggers-pending state as a result.

User: <username> (required, system-level hooks only) System-level hooks are run as the user whose name is specified as the value of `User`. There is intentionally no default for this key, to encourage hook authors to run their hooks with the least appropriate privilege.

Single-Version: yes (optional, system-level hooks only) By default, system-level hooks support multiple versions of packages, so target paths may exist at multiple versions. "`Single-Version: yes`" causes only the current version of each package to have a target path.

Hook-Name: <name> (optional) The value of `Hook-Name` is the name that Click packages may use to attach to this hook. By default, this is the base name of the `*.hook` file, with the `.hook` extension removed.

Multiple hooks may use the same hook-name, in which case all those hooks will be run when installing, upgrading, or removing a Click package that attaches to that name.

- A Click package may attach to zero or more hooks, by including a "hooks" entry in its manifest. If present, this must be a dictionary mapping application names to hook sets; each hook set is itself a dictionary mapping hook names to paths. The hook names are used to look up `*.hook` files with matching hook-names (see `Hook-Name` above). The paths are relative to the directory where the Click package is unpacked, and are used as symlink targets by the package manager when creating symlinks according to the `Pattern` field in `*.hook` files.
- There is a dh_click program which installs the `*.hook` files in system packages and adds maintainer script fragments to cause click to catch up with any newly-provided hooks. It may be invoked using `dh $@ --with click`.

Examples

```
/usr/share/click/hooks/apparmor.hook:
  Pattern: /var/lib/apparmor/clicks/${id}.json
  Exec: /usr/bin/aa-clickhook
  User: root

/usr/share/click/hooks/click-desktop.hook:
```

```
  User-Level: yes
  Pattern: /opt/click.ubuntu.com/.click/desktop-files/${user}_${id}.desktop
  Exec: click desktophook
  Hook-Name: desktop

com.ubuntu.example/manifest.json:
  "hooks": {
    "example-app": {
      "apparmor": "apparmor/example-app.json",
      "desktop": "example-app.desktop"
    }
  }
```

TODO: copy rather than symlink, for additional robustness?

Databases

(This is a lightly-edited copy of a brain-dump sent by Colin Watson to the ubuntu-phone mailing list, preserved here since it may be useful.)

Click has multiple databases where packages may be unpacked: by default we have the "core" database for core apps (`/usr/share/click/preinstalled/`), the "custom" database for carrier/OEM customisations (`/custom/click/`), and the "default" database for user-installed applications (`/opt/click.ubuntu.com/`), although these are configurable in `/etc/click/databases/`. Each database may have multiple unpacked versions of any given package.

Each database may also have user registrations, which live in `.click/users/` relative to the database root. Each user has a subdirectory of that, which contains symlinks to the versions of each package they have registered. This means that on a tablet, say, I can install an app without it also showing up on my children's accounts; they'd need to install it separately, although the disk space for the unpacked copy of the app would be shared.

There was an idea early on that we'd deal with preinstalled apps by going round and registering them all for all active users on first boot. This would have lots of problems for the packaging system, though. Most notably, doing it that way makes it hard for a user to remove an app and make it stick, because it would tend to reappear on system updates. You can probably fudge your way around this somehow, but it gets very fiddly and easy to get wrong.

What we do instead is: we have an `@all` pseudo-user which you can register packages for, typically in the core database (`click register --root=/usr/share/click/preinstalled --all-users`). If a user wants to remove a package, we do this by creating a deliberately broken symlink pointing to `@hidden` in their user registration area in `/opt/click.ubuntu.com/.click/users/$USERNAME/`. When click is asked to list the set of packages for a given user, it walks its way down the list of databases from top (default) to bottom (core). For each database, it checks registrations for that user, followed by registrations for `@all`. It takes the first registration for any given package name that it finds. If that registration is `@hidden`, then it ignores the package, otherwise it must be a link to the unpacked copy of the appropriate version of the package.

There are still some things that can't be done just with static files in the image and instead have to be done at boot time and on session startup: we have to make sure the right AppArmor profiles are loaded, do things to the user's home directory like creating .desktop files, and that kind of thing. We run `click hook run-system` at boot time and `click hook run-user` on session startup, and these deal with running hooks for whatever packages are visible in context, according to the rules above.

The effect of all this is that we can hide a core app for a carrier by doing this as root when preparing their custom overlay image:

```
click unregister --root=/custom/click --all-users PACKAGE-NAME
```

This will create a symlink `/custom/click/.click/users/@all/PACKAGE-NAME` pointing to `@hidden`. Unless a user explicitly installs the app in question, the effect of this will be that it's as if the app just isn't there. It shouldn't incur any more than a negligible cost at startup (basically just a readlink call); at the moment I think we might still create an AppArmor profile for it, which isn't free, but that can be fixed easily enough.

To do

- hook that gets notified about all installations
- dbus interface etc. as backend for UI
  - method may not be feasible because caller may want to go away
  - but where do we send a completion/failure signal back to?
- some way to manage shared data files
- association with developer ID, to allow sharing of data
- debug symbols
- define exit statuses for "click install"
- command to generate manifest template, like `dh_make`
- check whether a package contains compiled code for an architecture not listed in the "architecture" manifest field

Delta updates

It would be helpful to have some kind of delta update format.

Tools such as `rsync` and `zsync` are probably the wrong answer. There's no particular reason to keep the .click file around as an rsync target, particularly since the unpacked application directory is kept pristine, and many devices won't have the kind of disk space where you want to keep 4.2GB files around just for the sake of it.

We could do something ad-hoc with `xdelta` or `bsdiff` or whatever.

debdelta seems like a good possibility. We're already using the .deb format, and debdelta is capable of doing patch upgrades without having the old .deb around (though it will need minor adjustments to cope with the different installation location of Click packages). Under the hood, it uses xdelta/bsdiff/etc. and can be extended with other backends if need be. If we used this then we could take advantage of a good deal of existing code.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search