Packaging training

1 - Intro

Introduction

This training focuses on deb and rpm packaging.
More to come… (brew, docker)

Packages?

deb, rpm, Perl cpan, Ruby gem, Node npm, python, R, JS bower, brew …

Why a package?

Software life cycle (install/upgrade/remove/…)
Auto update via repo (official or owned managed)
Dependency management

Repositories: official vs owned

<u>Official</u>

Team managed/community help

Easier bug report for end user (reportbug, ...)

Dependency updates (fixes, security,...)

Builds via distribution system (many archs/systems)

But restrictive rules (copyright, files hierarchy,...)

Repositories: official vs owned

<u>Owned</u>

Less rules to follow/match

No stable release freeze, update package at any time

Can include anything in package

But need to build for many archs/versions

Good practices

A package build must be fully reproducible. For open source project, this means that documentation can be regenerated too. If you embed pdf files for example, that’s fine, but original file (doc, latex, …) should be in source archive too.

Good practices

Do not embed in your package libraries from other projects (use them if packaged, or package them).
If a library is needed in a specific directory for example, then add a symlink to the library location.

Follow distribution policies (filesystem hierarchy, compilation flags, ...)

Provide documentation (man pages,...)

Good practices

A good package needs a good software!

Use build systems (autotools, ant, cmake, ...)
Do not use hard coded path (use env or config files)
Use env variables for your builds (CXXFLAGS, LD_LIBRARY_PATH, ...: CXXFLAGS = CXXFLAGS:-O4, not CXXFLAGS = -O4)

It will ease your package creation/maintenance.

Good practices

At installation, avoid automatic setup/install such as database creation.
Provide documentation, user will execute post-install scripts (or your program manages this at runtime)