deb building 101

A first dive into the world of debs

Adapting a dh step

- The three most important build targets are
  - *build* (build the source)
  - *clean* (clean up build environment)
  - *binary* (create a deb from the build)
- You can run `dh TARGET --no-act` to see what debhelper sub commands are executed

Why do we use debs a lot?

- Integrates well with the rest of the OS
- Ensures integrity of package retrieval via „Secure Apt“
- Coherent tracking of package state (Debmonitor)

What is a deb?

- A deb package is at the core just an „ar“ archive which can be extracted
  - ar x emacs.deb
- Internally it‘s comprised of two *control.tar.X* and *data.tar.X* archives (where X can be .gz, .bz2 or .xz)

Complexity

- Debian packages are very powerful and cater to many needs
  - Libreoffice 6.4 is a 500 MiB source package building 201 packages in C++, Java and Python
  - Debian provides plumbing to ship code written in C/C++, Fortran, Perl, Python, Go, Rust, Haskell, various ML, various Lisps, Pascal, D and probably a few more
- But a small set of features actually covers many use cases well
  - Typical 80/20 situation
  - Some complexity only applies to Debian and typically not to site-local deployments (e.g. we don‘t need to care about whether our code builds on ARM)

Typical use cases

- Rebuild a package for a (typically older) distro release
- Modify a package (apply a patch)
- Package something from scratch (which is not yet in Debian or WMF-local)
- Maybe you even want to bring a package to Debian going forward
  - We do that e.g. with gdnsd (maintained by Faidon)

Package repositories

- Our repository: `apt.wikimedia.org`
  - Primarily used in production and Cloud VPS/Toolforge, but also externally available to others
  - CNAME directing to `install1002.wikimedia.org` (and synched to `install2002.wikimedia.org`)
  - Only SREs can upload! Being able to upload packages to the repo is pretty much equivalent to fleet-wide root
- Anything we install from Debian unmodified comes from our internal Debian mirror (`sodium.wikimedia.org`)
  - Exception: `security.debian.org` (fetched from the Debian CDN)
  - `sodium` also serves as a mirror to external users (part of the US mirrors)

Structure of our repository

- Structured in repository components
  - Allows separation of potentially mutually exclusive package sets
  - Allows isolation of risky changes („Our new service needs libfoo in the latest version, can you please build it?“)
  - Components are managed via Puppet: `modules/aptrepo/files/distributions-wikimedia`
- Our repository contains two types of debs:
  - Packages built by us
  - Other repositories we sync locally (and unmodified)
    - More reliability in case external repos are down
    - All package installations happen locally
    - Package import configures Secure Apt keys to secure import
    - Managed via Puppet: `modules/aptrepo/files/updates`

Repository structure

- The repository contains packages per Debian release
  - Code typically needs to be rebuilt for each release
  - Sometimes that‘s not necessary, but when in doubt, rebuild!
- By default every server includes the **main** component
  - And every baremetal server adds **thirdparty/hwraid**
- Everything else must be included via Puppet
  - `apt::package_from_component` define helps with this

Terminology: Source package

- A **source package** is an umbrella term for a package
  - Typically is relates to what you download as a tarball or git tag
- In addition to installing packages you can typically also download the source package via `apt-get source foo`
  - Also works on any of our servers
- All the Debian-specific config is in `debian/` sub dir

Terminology: Binary packages

- A **binary package** is a what you install
  - Slightly confusing: Despite the name it can also be just data or code written in Shell, it‘s unrelated to binary machine code
  - A source package can build just one binary package (then they must be identical) or many
  - Typical use case: Split functionality
    - `foo-server`, `foo-client`
  - Typical use case: Libraries
    - `libfoo0`, `libfoo-dev`

Version

- Typically all binary package have the same version as the source package („Here be dragons“ if not)
  - Defined in the top-most entry of `debian/changelog` (more to that later)
- Typical version: mutt 1.13.2-1
  - The 1.13.2 release of the mutt developers
  - -1 denotes the first upload of that package to Debian
- Update to a released Debian version: mutt 1.7.2-1+deb9u1
  - The 1.7.2 release of mutt is present in the with it‘s first upload
  - After the ninth Debian release was made, there was a need for an update in that stable release
  - (In this case a security update)

Core files: debian/changelog

- `debian/changelog`
  - Contains the change history to a package and who last changed the package
  - Defines the version of the package
  - The distribution a package is intended to build for
  - The „urgency“ field is irrelevant for us, only applicable to Debian
- Use `dch -i` to create a new entry
  - `dpkg-dev-el` ships an Emacs mode

Core files: debian/control

- `debian/control` defines what binary packages are build (each block starting with *Package:*)
- The *Source:* entry needs to match the source name referenced in `debian/changelog`
- Several fields are only relevant to Debian, you can ignore them:
  - *Section:* and *Priority:* structure the Debian archive and whether a package ends up in a default installation
  - *Standards-Version:* declares compliance with Debian policy docs

Core files: debian/control, pt 2

- *Build-Depends:* defines which packages need to be present at build time
  - A server may need OpenSSL
  - A Python package may need a another Python package
  - Build dependencies are installed as debs, fetching external sources during build is frowned upon
- Build deps are per source package

Core files: debian/control, pt 3

- *Depends:* defines what a package requires at runtime
  - This setting is per binary package (a server might have different library requirements than a client)
  - Virtually every package will need to depend on something
  - When installing a package, apt will pull in dependencies automatically
  - Most dependencies do not need to be manually specified, but are auto-generated:
    - `${shlibs:Depends}` adds all the dependencies on ELF libs
    - `${python3:Depends}` adds dependencies for Python packages
    - `${misc:Depends}` adds dependencies as required by Debhelper

Core files: debian/control, pt 4

- Other types of package relations:
  - *Recommends:* are packages which are installed by default (not in our production setup, though!), but can be uninstalled as necessary
  - *Suggests:* are packages which may extend the package‘s capabilities, but are not installed by default (e.g. docs or exotic use cases)
  - Special cases out of scope: *Conflicts:* , *Provides:* , *Pre-Depends:* , *Breaks:*
- Every package has a decription (shown by `apt-cache show` or in package managers) (one line summary and long form)
- There‘s also a *Architecture:* field which defines on which platforms the package can be built
  - *any* is for platform-specific code which can be build everywhere
  - *all* is for platform-independent code (Pure perl/Python, shell, documentation, data etc.)
  - Mostly relevant for Debian, as we only use a single architecture (64 bits x86 referred to as *amd64*)

How a package is built

- The package build is defined via debian/rules
  - It‘s „just a Makefile“ with common targets (like *build* or *clean* which are invoked by package build tools
- Common steps are abstracted by debhelper, which provides many `dh_*` commands
  - See `man debhelper` for a complete list
  - Each `dh_` command also has a separate manpage
- Each package declares a „compat level“
  - New functionality is often enabled in new levels
  - Allows gradual rollouts of new central features without impact to „old“ packages
  - As such often migrating to a new Debian-wide feature only means bumping the level
    - E.g. toolchain hardening or automatic support for systemd were enabled this way

dh

- `dh` abstracts the process of calling various debhelper commands
- Many packages build the same way:
  - `./configure ; make; sudo make install`
  - `python setup.py install`
  - `perl Build.pl; ./Build installdeps; ./Build; ./Build test; sudo ./Build install`
- `dh` detects many build systems and automatically invokes the necessary build steps
- Using debhelper without `dh` is only useful for special cases (or a package where the migration is still TBD)
  - 3/4s of the Debian archive use it at this point
  - Some packages use CDBS, similar to dh, but mostly deprecated

A minimal dh file

```
#!/usr/bin/make -f
%:
	dh $@
```

This alone works for quite a few packages!

dh addons

- `dh` has a number of addons
  - Some shipped in the main debhelper package
  - Some installed via addon packages
  - See `dh -l` for a list
  - Addons can be enabled using `--with`
    - e.g. `dh $@ --with phppear`

Overriding defaults

- If a dh automatism isn‘t sufficient, you can override it
  - E.g. to disable a test suite for whatever reason add a *override_dh_auto_test* Makefile target

Package builder

- We have a build host: `boron.eqiad.wmnet`
  - It has a properly puppetised setup of Pbuilder (a package building framework)
- Don‘t build locally on your systems (with exceptions)

Building using pbuilder

- Navigate to the top directory of your source tree (e.g. same directory where the `debian/` dir is)
  - `DIST=[buster|stretch|jessie] pdebuild` covers most use cases
  - `buster-wikimedia` (et al.) will also include the `apt.wikimedia.org` repository for that suite
- pbuilder creates a clean build environment and installs all the build dependencies
- Build result under `/var/cache/pbuilder/result/$SUITE-amd64`
- There‘s a `.build` log file in the package source tree

Importing to apt.wikimedia.org

- When your tests with the built package are fine, the package can be uploaded to `apt.wikimedia.org`
- rsync is setup between build and repo host, e.g. run the following on install1002:

```
rsync rsync://boron.eqiad.wmnet/pbuilder-result/buster-amd64/*foo* .
```

Importing packages

- We use reprepro to manage your repository
- To re-sign the repo, reprepro needs to access PGP keys in the home directory of the root user, so use `sudo -i command` or `sudo -H bash`
- Two major approaches to import:
  - Import via a .changes file (a complete build including source) using *include*: (preferred)
    - `reprepro -C main include stretch-wikimedia emacs_26-3_amd64.changes`
  - Import a single deb using *includedeb*:
    - `reprepro -C main includedeb stretch-wikimedia foo_1.2-1_amd64.deb`

Other reprepro commands

- List all versions of a package
  - `reprepro ls puppet`
- Copy between distributions:
  - `reprepro copy buster-wikimedia stretch-wikimedia prometheus-snmp-exporter`
  - Careful: First destination distro, then source!
  - Usually you need to rebuild, but exceptions exist

Further reading

- These docs are directed at packaging in Debian, but much of the content also applies to out environment:
  - https://www.debian.org/doc/debian-policy/
  - https://www.debian.org/doc/manuals/developers-reference/
  -