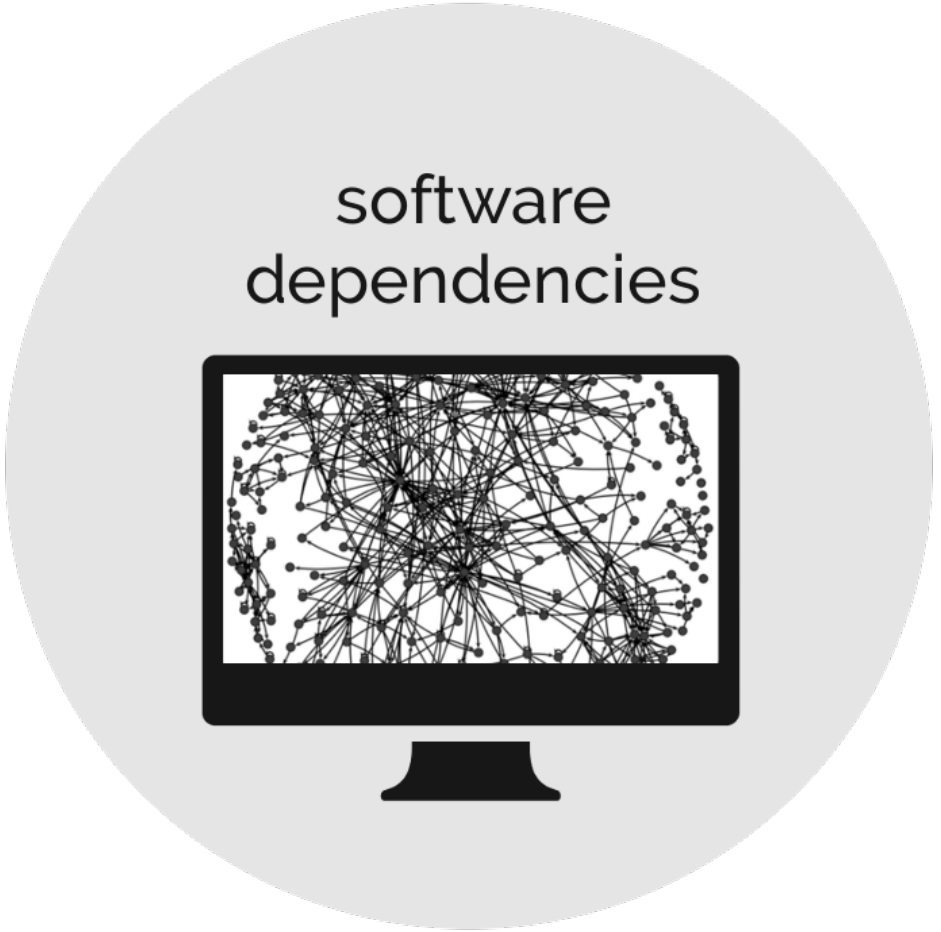
Francesco Lamonica

Handling (C++) Deps

Handling cpp deps

- Statement of the problem: Today every software depends on something else…
- Each language has defined a way to deal with this ‘problem’
  - Ruby gems
  - Linux distro package managers
  - Php composer
  - Python’s pip
  - Etc.

- What about C++?
- There is no standard! ▮▮
- Notable mentions: conan / vcpkg

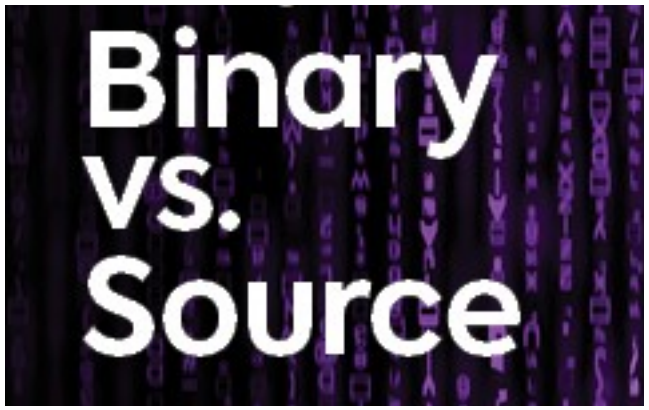


NetResults
Building the digital society

Type of dependencies

- At the very base level we can divide deps in two main categories
  - Source dependencies (we can download the dep and compile along our project)
  - Binary dependencies (maybe the source is not an option)

- Each option has its own advantages and quirks



?

NetResults
Building the digital society

Source dependencies

- Pros:
  - Allows to test code automatically when a commit is done to a dependency (be proactive for updating deps)
  - Avoid ABI clashing, missing symbols, different interface classes shared between projects (each component can be recompiled against the same version of the dependency)
- Cons:
  - Unnecessary recompilations
  - Not a straightforward way to distribute source deps
  - Not always available

NetResults
Building the digital society

Binary Deps

- Pros:
  - Usually production projects always keep the dependency stable (i.e. they use a specific version) so it is pointless to recompile each time the same thing
  - Allow easy distribution of all deps.
  - Sometimes they are the only option available (see IPP libraries)

- Cons:
  - Can be huge! (extreme case VDK static for iOS is about 800M)
  - Sometimes can be difficult to track binary versions (i.e. static libs or dynamic with no versioning)
  - Hard to track and deal with cross-project dependencies (if some common binary dep changes, most probably you need to change it in all projects)

NR solution

- How we dealt with this problem?
  - Src deps: SVN externals (binary not optimized)
  - Binary deps:
    - Included in repo along with src
    - Downloaded manually

- And after switching to git?
  - Svn externals (vdk deps are still on svn)
  - Manual download of new git repos
  - Is this any better?

NetResults
Building the digital society

Looking for a solution

- Conan (becoming the standard ?)
  - Pretty complex
  - Need a dedicated server
  - Dedicated to binaries

- VCPKG (guess the authors?)
  - Allows both binaries and src (with recipes for compilation)
  - Limited to desktop platforms (?)
  - Dedicated to libraries

NR proposed solution

- How we should deal with this problem?
  - nr_co_deps.py (NetResults CheckOut Dependencies)
  - Python script with standard dependency (i.e. modules installed on every modern platform)
  - Runs with both python 2.7 and 3.7
  - Has a very simple usage: python nr_co_deps.py dependencies.json
  - Git url: https://gitlab.netresults.dev/netresults/utils/scripts

  - What kind of dependecies can it handle?
    - Svn repos
    - Git repos (https / ssh)
    - Gitlab Merge Requests (https / ssh)
    - Download of ZIP / tgz files via http(s)
    - Download of packages from Artifactory
    - Avoid downloading again artifacts already downloaded (SHA-256)

NetResults
Building the digital society

Deps.json structure

- Structure: * means mandatory

```
{
   "deps": [
      {
         "name": an identifier for this dep, *
         "rel_dest_dir": folder where to download dep (relative to working dir), *
         "proto": how to download this dep,  (if omitted defaults to git)
         "url"*: URL to the dep,
         "ignoressl": whether or not ignore ssl error (defaults to no),
         "username": username used for auth,
         "password": password used for auth,
         "branch_or_tag" : the branch / tag / merge request of the deps (NOTE: if omitted master or trunk will be used depending of protocol. MR should always be in the form mr-git_remote-number_of_mr)
         "unzip" : whether the package should be unzipped (makes sense for http / artifacts)
         "depsfile": the name of another deps.json file to be used recursively
      },
...
   ]
}
```


NetResults
Building the digital society

Protocol: artifact

- What is artifactory?
  - Artifact Repository
  - Optimized for deduplication
  - Dozens of REST APIs to handle / search packages
  - Allows storage of different types of packages: pip, deb / rpm, gems, conan, maven, etc.
  - We have now an cloud service: https://nrreleases.jfrog.io (we might switch to on-prem if need be)
  - What can we do?
    - Allow storage of nightly / releases and MR builds
    - Integrates with teamcity for automatic deploy
    - Retrieve specific version of above categories
    - Retrieve ‘latest’ build for each of the above categories

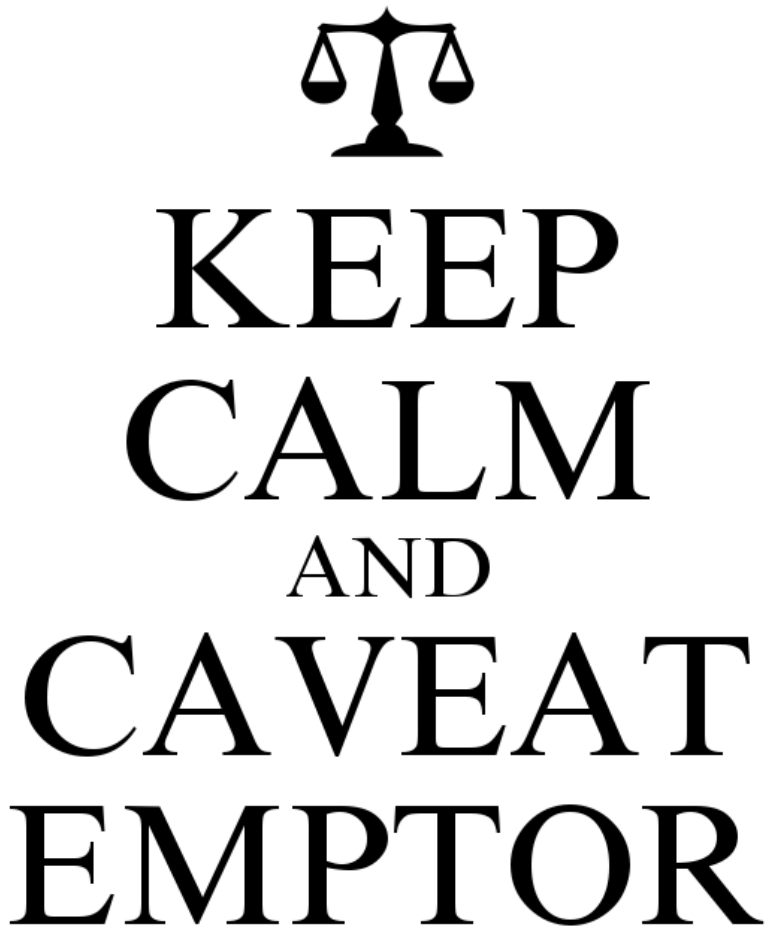
Deps.json structure for artifactory

- Structure: * means mandatory

```
{
   "deps": [
      {
         "name" * : "uniqloggerbin_artifact",
         "rel_dest_dir" * : "src/ext/uniqlogger-bin",
         "proto": "artifact",
         "username": "admin",
         "password": "password",
         "url" * : "https://nrreleases.jfrog.io/nrreleases/cpp-artifacts",
         "module": "bblocks/uniqlogger",
         "branch_or_tag": "nightly",
         "version": "0.7.1",
         "unzip": "true“
      },
...
   ]
}
```


NetResults
Building the digital society

The end (?)



NetResults
Building the digital society

What’s missing (yet)

- No artifact repository for merge requests
- No queries to download ‘latest’ version
- No teamcity rule to build ‘release’ or ‘MR’
- No instructions to manual upload packages
- Surely something else i’m forgetting at the moment

The end (For real!)

NetResults
Building the digital society