A Common Package Specification:

Getting Build Tools to Talk to Each Other: Lessons Learned From Making Thousands of Binaries Consumable by Any Build System

DIEGO RODRIGUEZ-LOSADA GONZALEZ

Cppcon
The C++ Conference

2023

October 01 - 06

JFrog | CONAN 2.0

C/C++ Package Manager

Outline

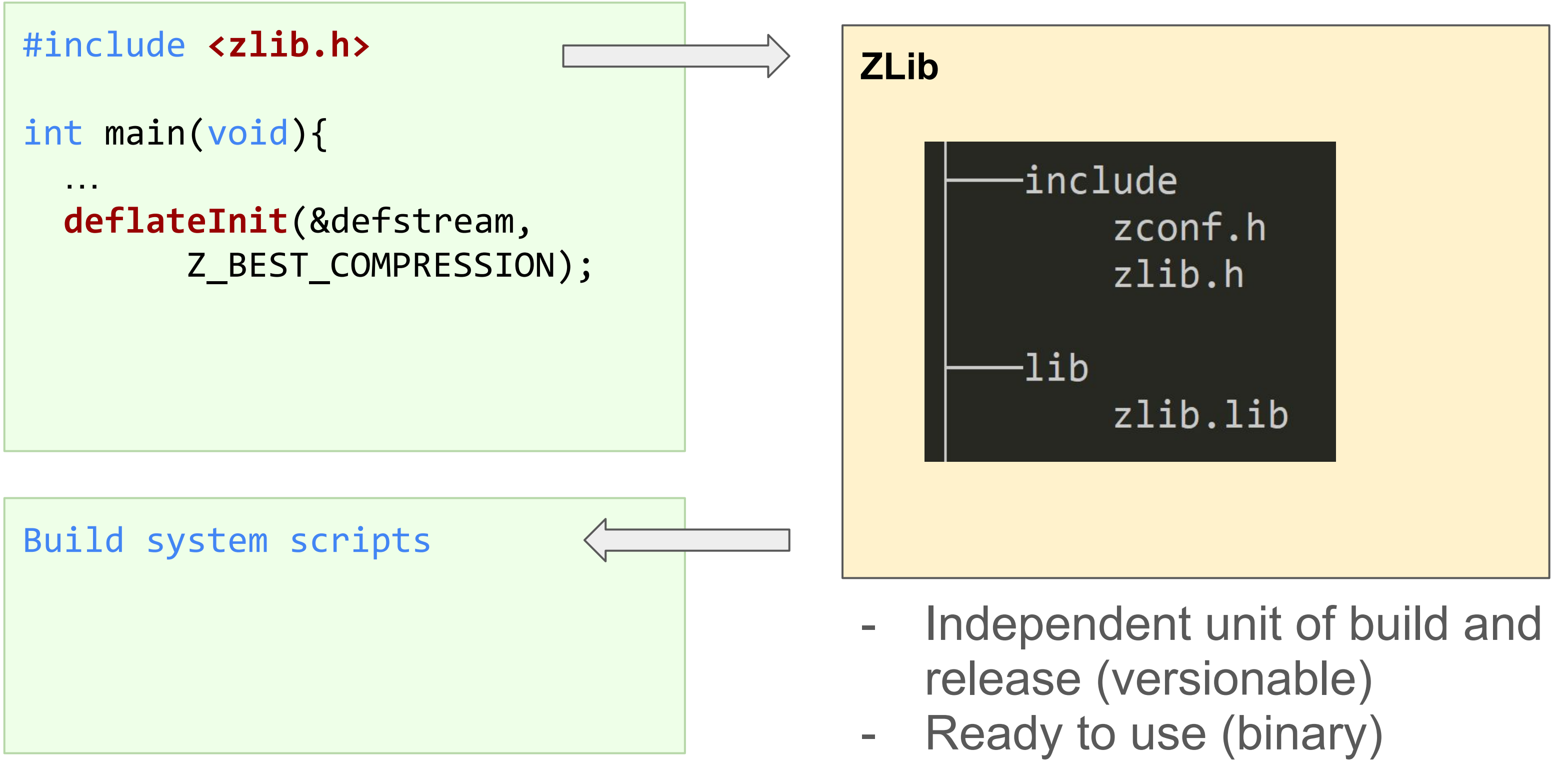
Problem definition and scope

```
#include <zlib.h>

int main(void){
  …
  deflateInit(&defstream,
        Z_BEST_COMPRESSION);
```

```
Build system scripts
```

1. Define the version to use
2. Define the configuration: Windows, x86_64, VS-2022, Release, static library
3. Install (system package manager or language package manager), build from source by the user, with that configuration
4. **Pass information to the build system so it can locate and use it successfully**

What is a package



```c
#include <zlib.h>

int main(void){
  …
  deflateInit(&defstream,
        Z_BEST_COMPRESSION);
```

```
Build system scripts
```

ZLib

```
├──include
│       zconf.h
│       zlib.h
│
└──lib
        zlib.lib
```

- Independent unit of build and release (versionable)
- Ready to use (binary)

Consuming a package

```c
#include <zlib.h>

int main(void){
   …
   deflateInit(&defstream,
         Z_BEST_COMPRESSION);
```

ZLib

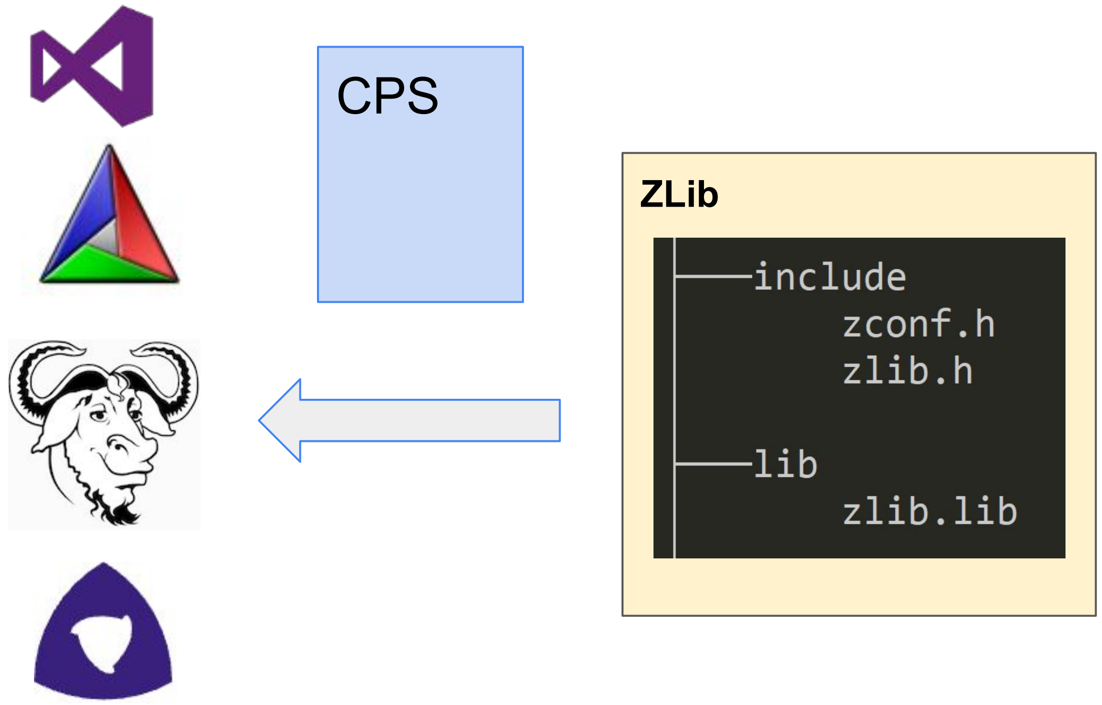
```
├──include
│       zconf.h
│       zlib.h
│
├──lib
│       zlib.lib
```

```
Build system scripts
```

?

```
-I<path>/include
-L<path>/lib
-lzlib
```

Common Package Specification (CPS): Scope



```
├───include
│       zconf.h
│       zlib.h
│
└───lib
        zlib.lib
```

…

Related work

- CPS: https://cps-org.github.io/cps by Matthew Woehlke et al
  - https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1313r0.html
- ISO C++: https://github.com/isocpp/pkg-fmt
- Libman https://api.csswg.org/bikeshed/?force=1&url=https://raw.githubusercontent.com/vector-of-bool/libman/develop/data/spec.bs by Colby Pike (@vectorofbool)

Existing solutions

```
prefix=@CMAKE_INSTALL_PREFIX@
exec_prefix=@CMAKE_INSTALL_PREFIX@
libdir=@INSTALL_LIB_DIR@
sharedlibdir=@INSTALL_LIB_DIR@
includedir=@INSTALL_INC_DIR@

Name: zlib
Description: zlib compression library
Version: @VERSION@

Requires:
Libs: -L${libdir} -L${sharedlibdir} -lz
Cflags: -I${includedir}
```

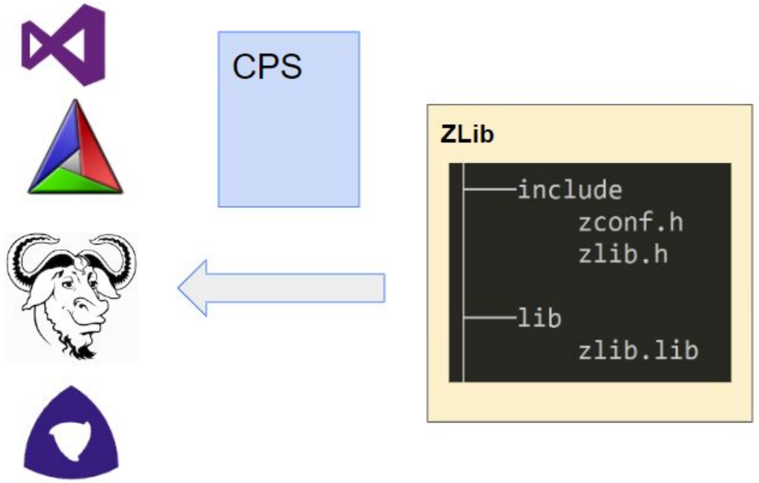
```
set(_ZLIB_x86 "(x86)")
set(_ZLIB_SEARCH_NORMAL PATHS
"[HKEY_LOCAL_MACHINE\\SOFTWARE\\GnuWin32\\Zlib;InstallPath]"
list(APPEND _ZLIB_SEARCHES _ZLIB_SEARCH_NORMAL)

if(ZLIB_USE_STATIC_LIBS)
  set(ZLIB_NAMES zlibstatic zlibstat zlib z)
  set(ZLIB_NAMES_DEBUG zlibstaticd zlibstatd zlibd zd)
else()
  set(ZLIB_NAMES z zlib zdll zlib1 zlibstatic zlibwapi ..)
  set(ZLIB_NAMES_DEBUG zd zlibd zdlld zlibd1 zlib1d ..)
endif()

if(ZLIB_FOUND)
    set(ZLIB_INCLUDE_DIRS ${ZLIB_INCLUDE_DIR})

    if(NOT TARGET ZLIB::ZLIB)
      add_library(ZLIB::ZLIB UNKNOWN IMPORTED)
      set_target_properties(ZLIB::ZLIB PROPERTIES
                                      INTERFACE_INCLUDE_DIRECTORIES
                                     "${ZLIB_INCLUDE_DIRS}")
```

Searching for Convergence in C++ Package Management - Bret Brown & Daniel Ruoso - CppNow 2022

Case For a Standardized Package Description Format for External C++ Libraries by Luis Caro Campos - CppCon22

Why this scope?

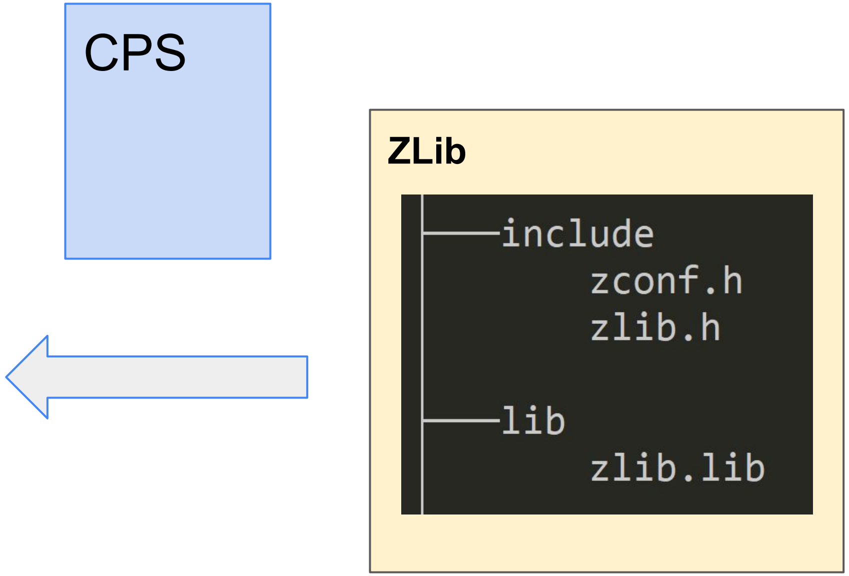
- Easiest **consensus**
- It will deliver the **most value**, the sooner, for users and the community
- Largest contributor to **interoperability**
- **Wide** scope: system packages, closed-source pre-compiled binaries from vendors, to open source built with any build system



```
├──include
│        zconf.h
│        zlib.h
│
└──lib
         zlib.lib
```

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1177r1.pdf by Rene Rivera (ISO C++ SG15)

Assumptions

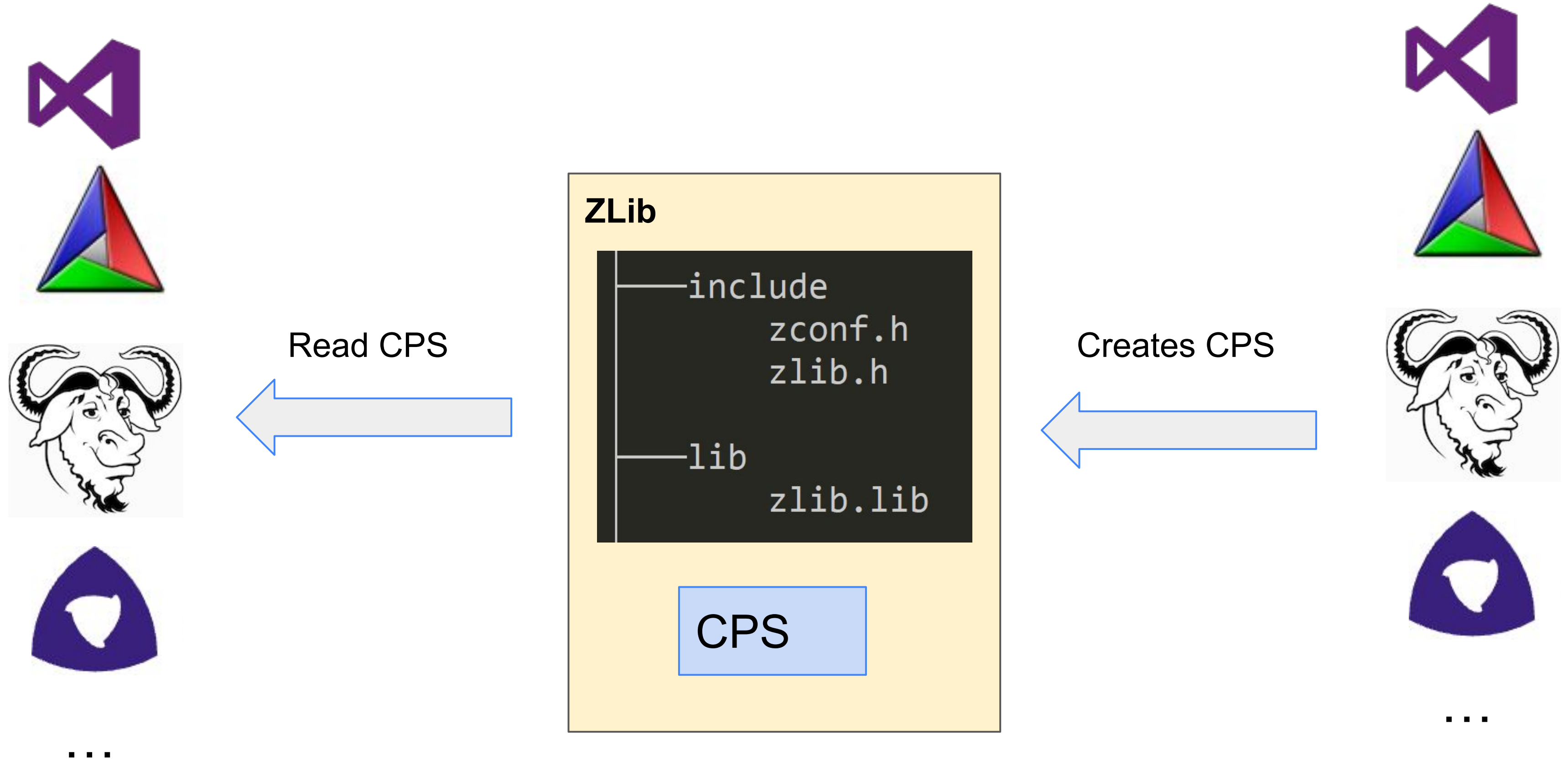


```
├──include
│       zconf.h
│       zlib.h
│
├──lib
│       zlib.lib
```

...

- Compiled **binary**
  - Any build system
  - Closed source
- **Single**-configuration
  - Valid for the current build
- **No version** information
- **No ABI** information

Interoperability

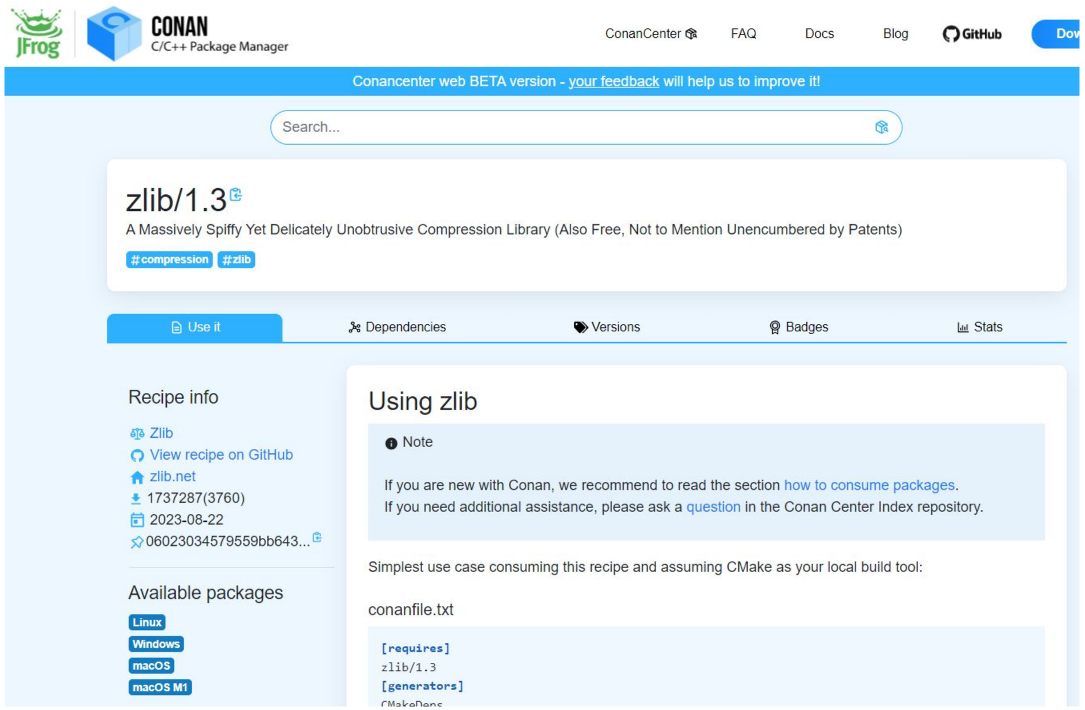


```
├──include
│       zconf.h
│       zlib.h
│
├──lib
│       zlib.lib
```

Outline

Implementation experience

- ConanCenter:
  - 1500 recipes x 3 versions x 100 binaries = **500K packages**
  - **3,1 million** packages download/month = **16Tb/month**



ConanCenter FAQ Docs Blog GitHub

Conancenter web BETA version - your feedback will help us to improve it!

zlib/1.3

A Massively Spiffy Yet Delicately Unobtrusive Compression Library (Also Free, Not to Mention Unencumbered by Patents)

#compression #zlib

Use it | Dependencies | Versions | Badges | Stats

Recipe info

- Zlib
- View recipe on GitHub
- zlib.net
- 1737287(3760)
- 2023-08-22
- 06023034579559bb643...

Available packages

- Linux
- Windows
- macOS
- macOS M1

Using zlib

Note

If you are new with Conan, we recommend to read the section how to consume packages.
If you need additional assistance, please ask a question in the Conan Center Index repository.

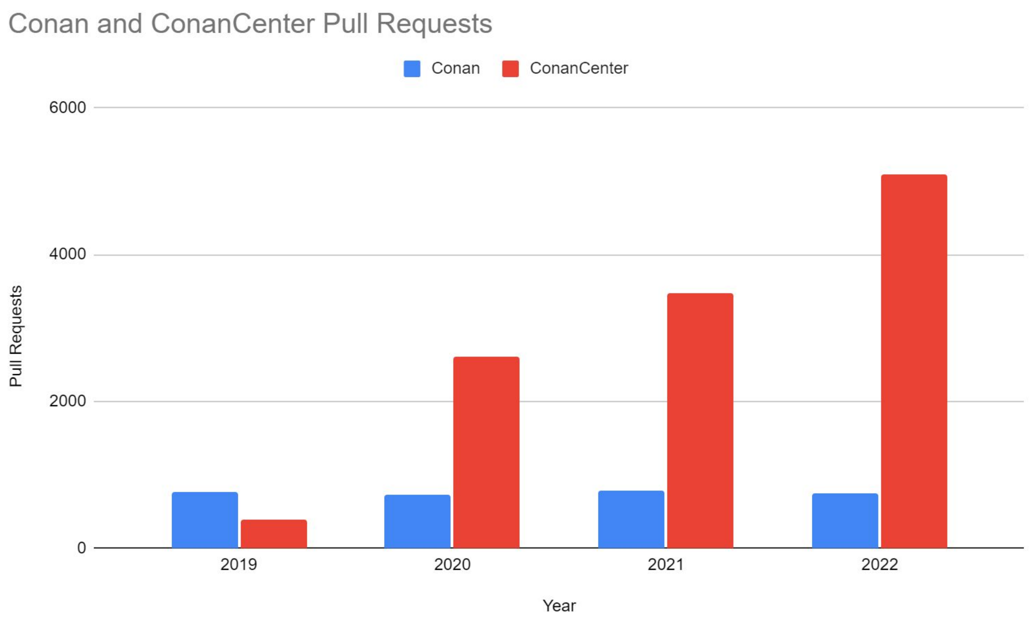
Simplest use case consuming this recipe and assuming CMake as your local build tool:

conanfile.txt

```
[requires]
zlib/1.3
[generators]
CMakeDeps
```

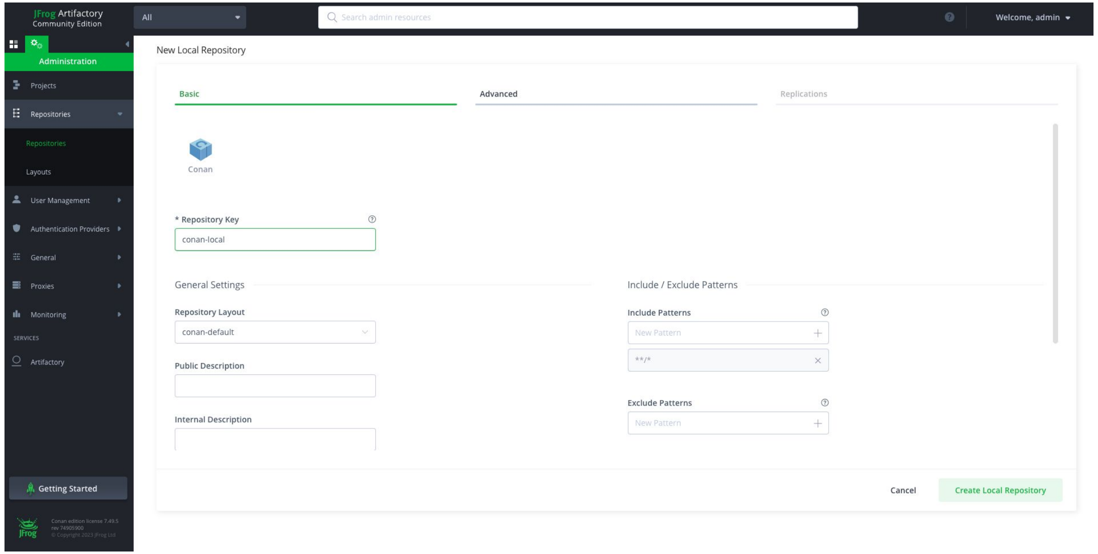
Implementation experience

- + 5000 PRs / year



Implementation experience

- Private packages
  - **75%** of users don't use ConanCenter
  - Client **750K downloads/month** (PyPI 1% critical project)
  - Many thousands of organizations using Artifactory-Conan in production



New Local Repository

* Repository Key: conan-local

General Settings

Repository Layout: conan-default

Public Description:

Internal Description:

Include / Exclude Patterns

Include Patterns: New Pattern

**/*

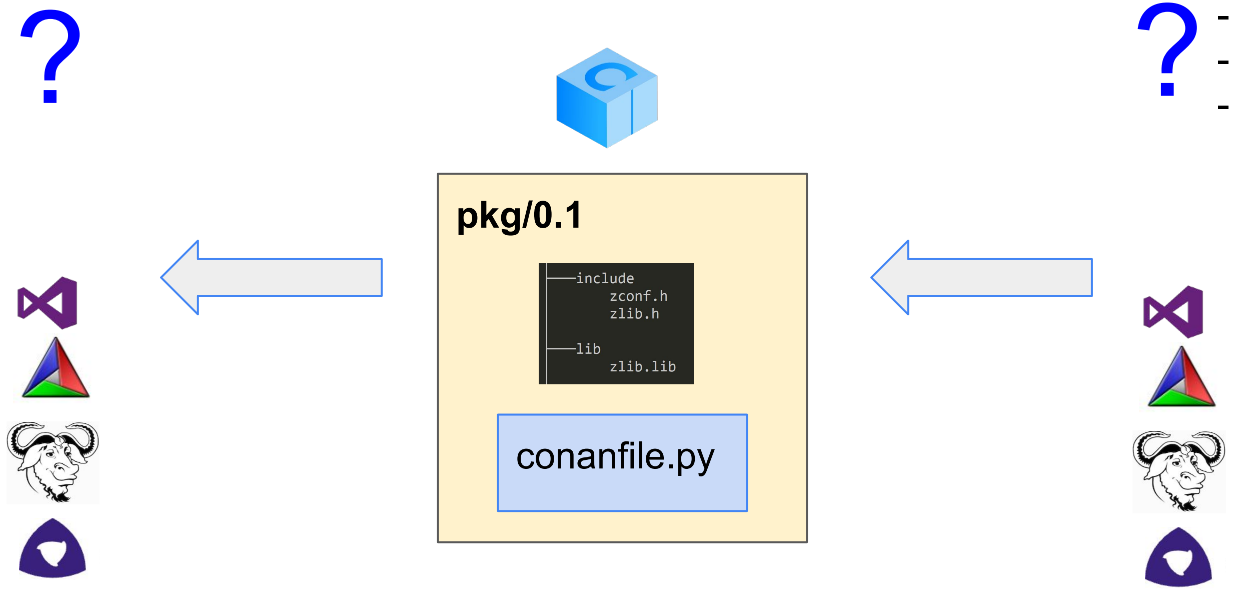
Exclude Patterns: New Pattern

What brings this implementation experience?

- What is important and what not. Dos and donts.
- Edge and corner cases to cover
- Both open-source and closed-source experience
- Both lessons learned and open questions

Bring me solutions, not problems
Bring me problems, not solutions
imgflip.com

Every package is consumable by any build system



```
├──include
│      zconf.h
│      zlib.h
│
└──lib
       zlib.lib
```

- Proprietary
- Precompiled
- Legacy

How Conan models the CPS

```
def package_info(self):
   self.cpp_info.set_property("cmake_find_mode", "both")
   self.cpp_info.set_property("cmake_file_name", "ZLIB")
   self.cpp_info.set_property("cmake_target_name", "ZLIB::ZLIB")
   if self.settings.os == "Windows" and not self._is_mingw:
      libname = "zdll" if self.options.shared else "zlib"
   else:
      libname = "z"
   self.cpp_info.libs = [libname]
```

How Conan models the CPS

```
def package_info(self):
    self.cpp_info.set_property("cmake_find_mode", "both")
    self.cpp_info.set_property("cmake_file_name", "ZLIB")
    self.cpp_info.set_property("cmake_target_name", "ZLIB::ZLIB")
    if self.settings.os == "Windows" and not self._is_mingw:
        libname = "zdll" if self.options.shared else "zlib"
    else:
        libname = "z"
    self.cpp_info.libs = [libname]
    import json
    print(json.dumps(self.cpp_info.serialize(), indent=2))
```

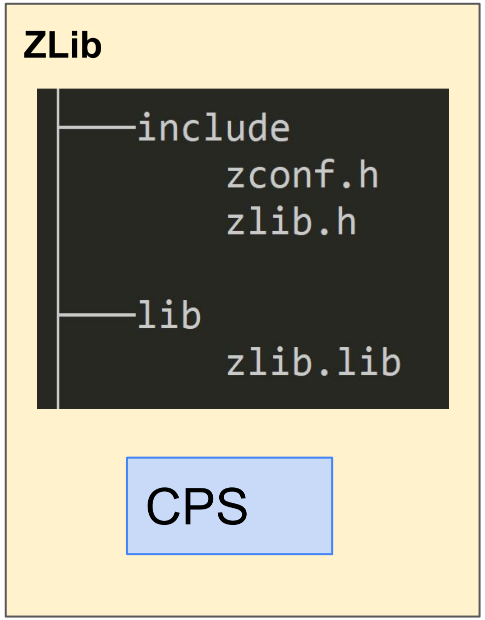
```
{
  ...
  "libs": ["zlib"],
}
```

Can we do better?

```
def package_info(self):
    self.cpp_info.set_property("cmake_find_mode", "both")
    self.cpp_info.set_property("cmake_file_name", "ZLIB")
    self.cpp_info.set_property("cmake_target_name", "ZLIB::ZLIB")
    if self.settings.os == "Windows" and not self._is_mingw:
        libname = "zdll" if self.options.shared else "zlib"
    else:
        libname = "z"
    self.cpp_info.libs = [libname]
```

- Repetition
- For large packages like boost, tedious
- Python

Can we do better



```
├──include
│       zconf.h
│       zlib.h
│
├──lib
│       zlib.lib
```

Outline

ZLib

```
|--include
|     zconf.h
|     zlib.h
|
|--lib
|     zlib.lib
|
|--licenses
      LICENSE
```

zlib.cps

```
{
  "includedirs": ["include"],
  "libdirs": ["lib"],
  "libs": ["zlib"],
  "properties": {
    "cmake_find_mode": "both",
    "cmake_file_name": "ZLIB",
    "cmake_target_name": "ZLIB::ZLIB",
  }
}
```

* Just an instance, for Windows MSVC, static library

Why json?

I don’t care, let’s just use any

ZLib

```
|--include
|     zconf.h
|     zlib.h
|
|--lib
|     zlib.lib
|
|--licenses
      LICENSE
```

zlib.cps

```
{
  "includedirs": ["include"],
  "libdirs": ["lib"],
  "libs": ["zlib"],
  "properties": {
    "cmake_find_mode": "both",
    "cmake_file_name": "ZLIB",
    "cmake_target_name": "ZLIB::ZLIB",
  }
}
```

Include Directories

- Maps to `-I<folder>` or to `-Isystem<folder>?`
- Why it is a list of folders?
- Conventions?

zlib.cps

```
{
  "includedirs": ["include"],
  "libdirs": ["lib"],
  "libs": ["zlib"],
  "properties": {
    "cmake_find_mode": "both",
    "cmake_file_name": "ZLIB",
    "cmake_target_name": "ZLIB::ZLIB",
  }
}
```

Relative paths by default: package “relocatibility”

```
zlib_win
|--include
|     zconf.h
|     zlib.h
|
|--lib
|     zlib.lib
|--zlib.cps
```

```
$ mv zlib zlib_win
```

```
zlib
|--include
|     zconf.h
|     zlib.h
|
|--lib
|     zlib.lib
|--zlib.cps
```

...

Relative paths? “System” packages

- In non-standard (absolute) locations

mypkg.cps

```
{
  "includedirs": ["/usr/nonstandardpath/headers/mylib/include"],
  "libdirs": ["/usr/othernonstandardpath/libs/mylib/lib"]
  "libs": ["mylib2", "mylib1"]
}
```

Litmus test:

- Can it be the output of a “build/install” process?
- Can it be consumed by build systems?

Libraries directories and libraries

Maps to -L<librarydir1> -L<librarydir2>...

Maps to -l<libname1> -l<libname2>..

zlib.cps

```
{
  "includedirs": ["include"],
  "libdirs": ["lib"],
  "libs": ["zlib"],
  "properties": {
    "cmake_find_mode": "both",
    "cmake_file_name": "ZLIB",
    "cmake_target_name": "ZLIB::ZLIB",
  }
}
```

Properties: playing nice with ecosystem

```
find_package(ZLIB REQUIRED)

#  find_package(ZLIB MODULE)

target_link_libraries(... ZLIB::ZLIB)
```

zlib.cps

```
  "includedirs": ["include"],
  "libdirs": ["lib"],
  "libs": ["zlib"],
  "properties": {
    "cmake_find_mode": "both",
    "cmake_file_name": "ZLIB",
    "cmake_target_name": "ZLIB::ZLIB",
   }
}
```

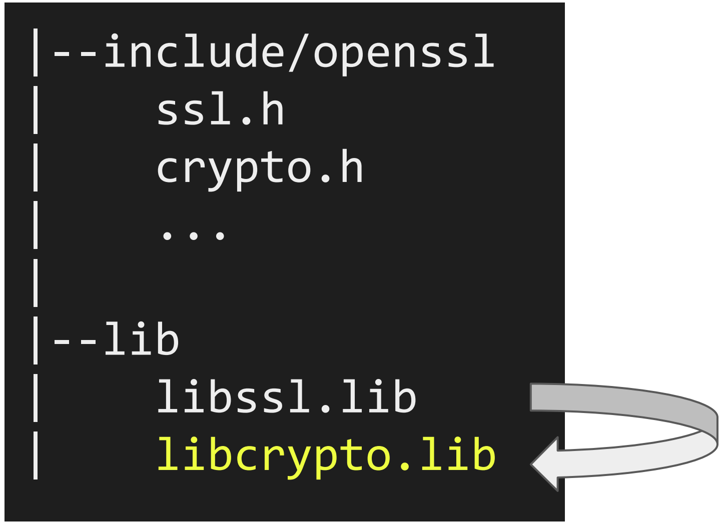
Outline

OpenSSL

```
|--include/openssl
|     ssl.h
|     crypto.h
|     ...
|
|--lib
|     libssl.lib
|     libcrypto.lib
```

What “linking openssl” means?

- By default, most users want to use and link with “ssl” library
  - Link “crypto” transitively
- But, some users will want to use only “crypto”
  - What if we pass “-lssl -lcrypto”?
  - Some linkers can be smart and optimize away
  - Some linkers (embedded cross-toolchains) will not
    - Larger than necessary binaries



```
|--include/openssl
|    ssl.h
|    crypto.h
|    ...
|
|--lib
|    libssl.lib
|    libcrypto.lib
```

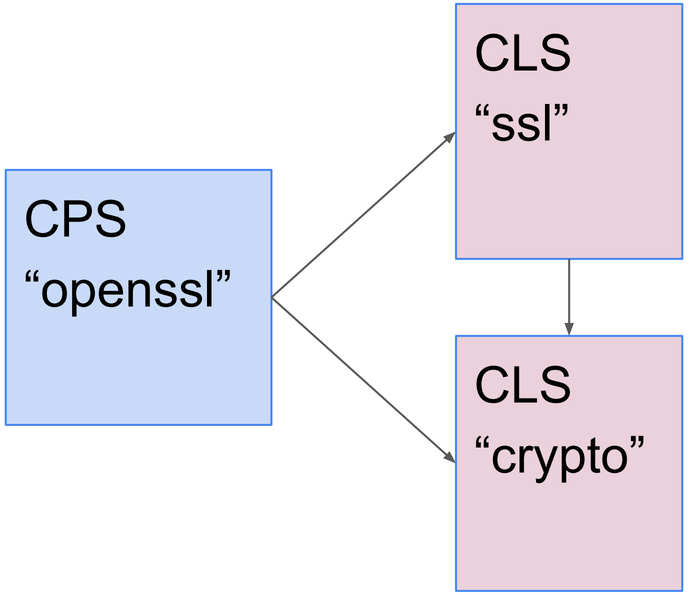
OpenSSL: Components

openssl.cps

```
"root": {
   "properties": {"cmake_file_name": "OpenSSL"}
},
"ssl": {
  "includedirs": ["include"],
  "libs": ["libssl"],
  "requires": ["crypto"],
},
"crypto": {
  "includedirs": ["include"],
  "system_libs": ["crypt32", "ws2_32", "advapi32",...],
  "libs": ["libcrypto"],
  "requires": ["zlib::zlib"],
}
```

Why not 2 separate CPS files, one for each?

- OpenSSL maintainers will not split the project in 2
  - Can we build once, create 2 independent packages?
- Different packages could evolve differently, versioned differently
  - Openssl maintainers don't want that
- It is a "package" specification, not a "library" specification
- But we can certainly separate, **Common Library Specification (Libman)**



OpenSSL: System libraries

openssl.cps

```
"root": {
   "properties": {"cmake_file_name": "OpenSSL"}
},
"ssl": {
  …
  "libs": ["libssl"],
  "requires": ["crypto"],
},
"crypto": {
  …
  "system_libs": ["crypt32", "ws2_32", "advapi32",...],
  "libs": ["libcrypto"],
  "requires": ["zlib::zlib"],
}
```

OpenSSL: Requirements

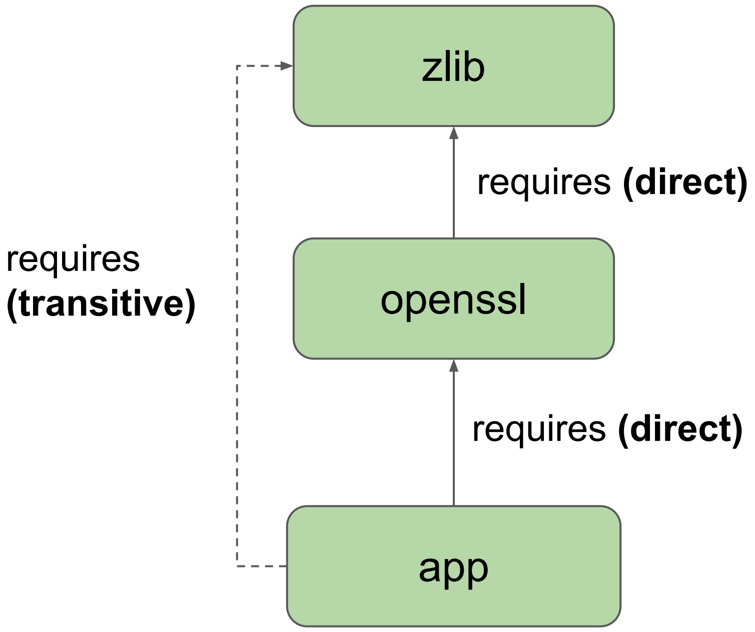
openssl.cps

internal

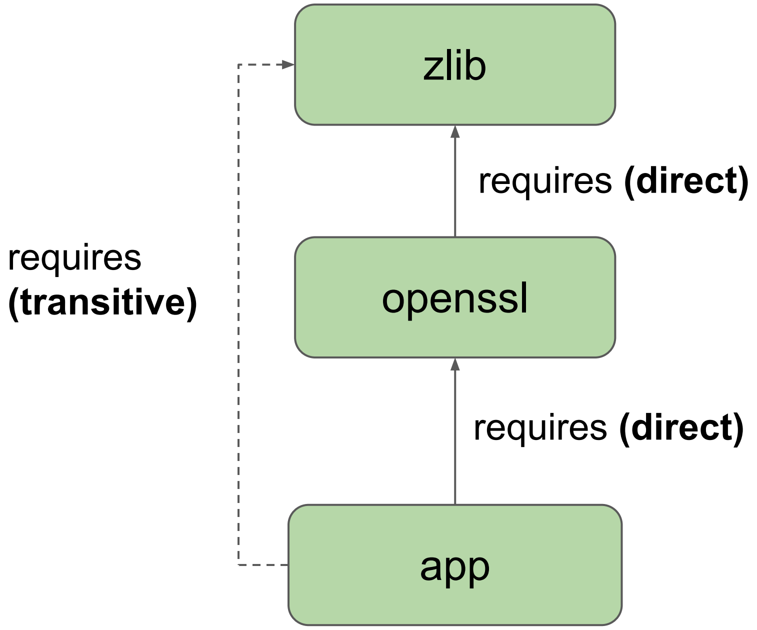
external

```
"root": {
   "properties": {"cmake_file_name": "OpenSSL"}
},
"ssl": {
  …
  "libs": ["libssl"],
  "requires": ["crypto"],
},
"crypto": {
  …
  "system_libs": ["crypt32", "ws2_32", "advapi32",...],
  "libs": ["libcrypto"],
  "requires": ["zlib::zlib"],
}
```

Requirements transitivity



Requirements transitivity



app.cpp flags	ssl.h includes zlib.h	ssl.h not includes zlib.h
app links zlib	`-I…/zlib/include -L…/zlib/lib -lzlib`	`-I…/zlib/include -L…/zlib/lib -lzlib`
app not links zlib	`-I…/zlib/include (-L…/zlib/lib) -lzlib`	`-I…/zlib/include -L…/zlib/lib -lzlib`

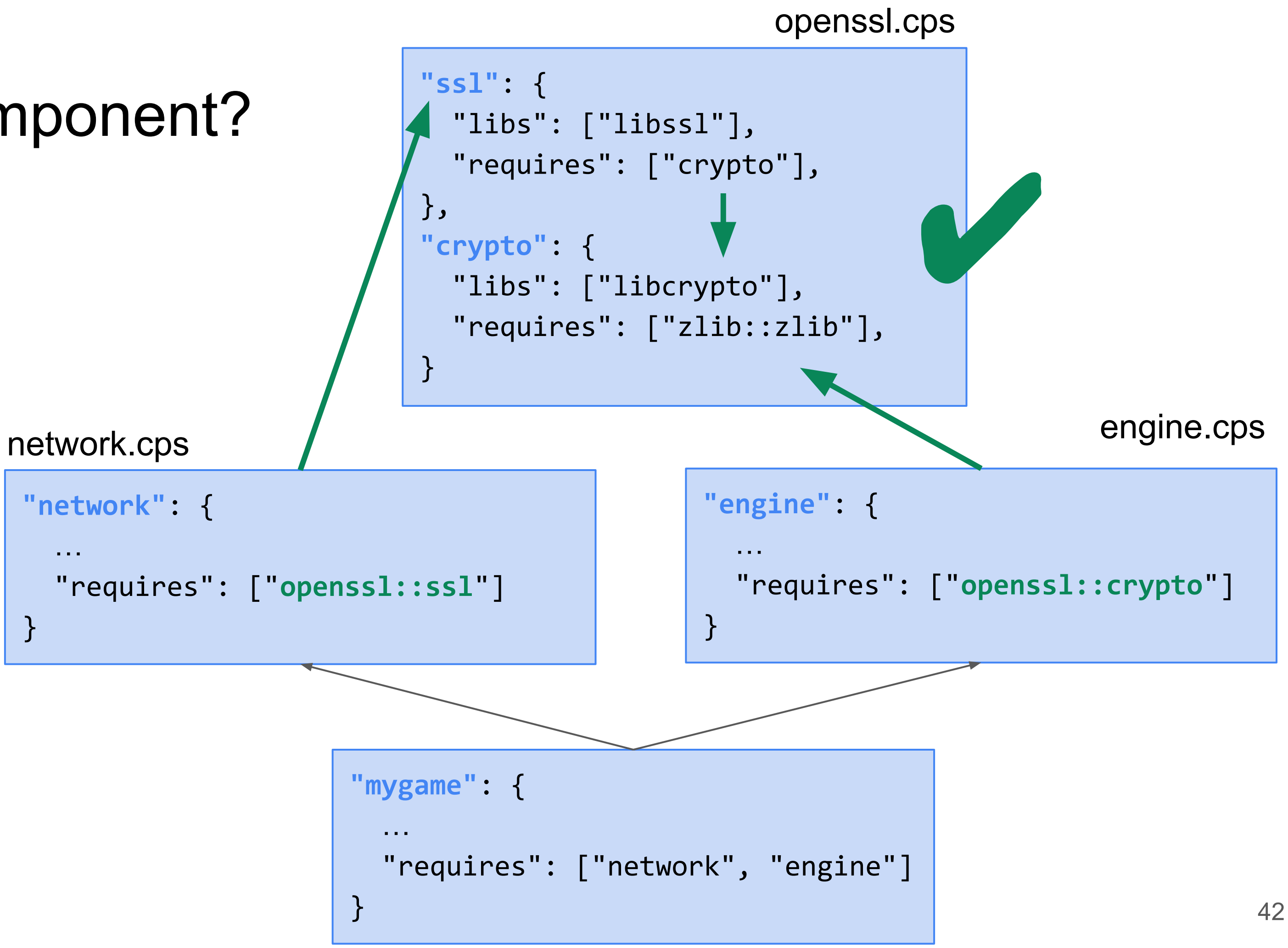
* Not real, just possibilities

OpenSSL: Requirements

openssl.cps

```
"ssl": {
  "libs": ["libssl"],
  "requires": ["crypto"],
},
"crypto": {
  "libs": ["libcrypto"],
  "requires": {
        "zlib::zlib": {
             "headers": True,
             "libs": False
         }
   },
}
```

What is a component?



openssl.cps

```
"ssl": {
  "libs": ["libssl"],
  "requires": ["crypto"],
},
"crypto": {
  "libs": ["libcrypto"],
  "requires": ["zlib::zlib"],
}
```

network.cps

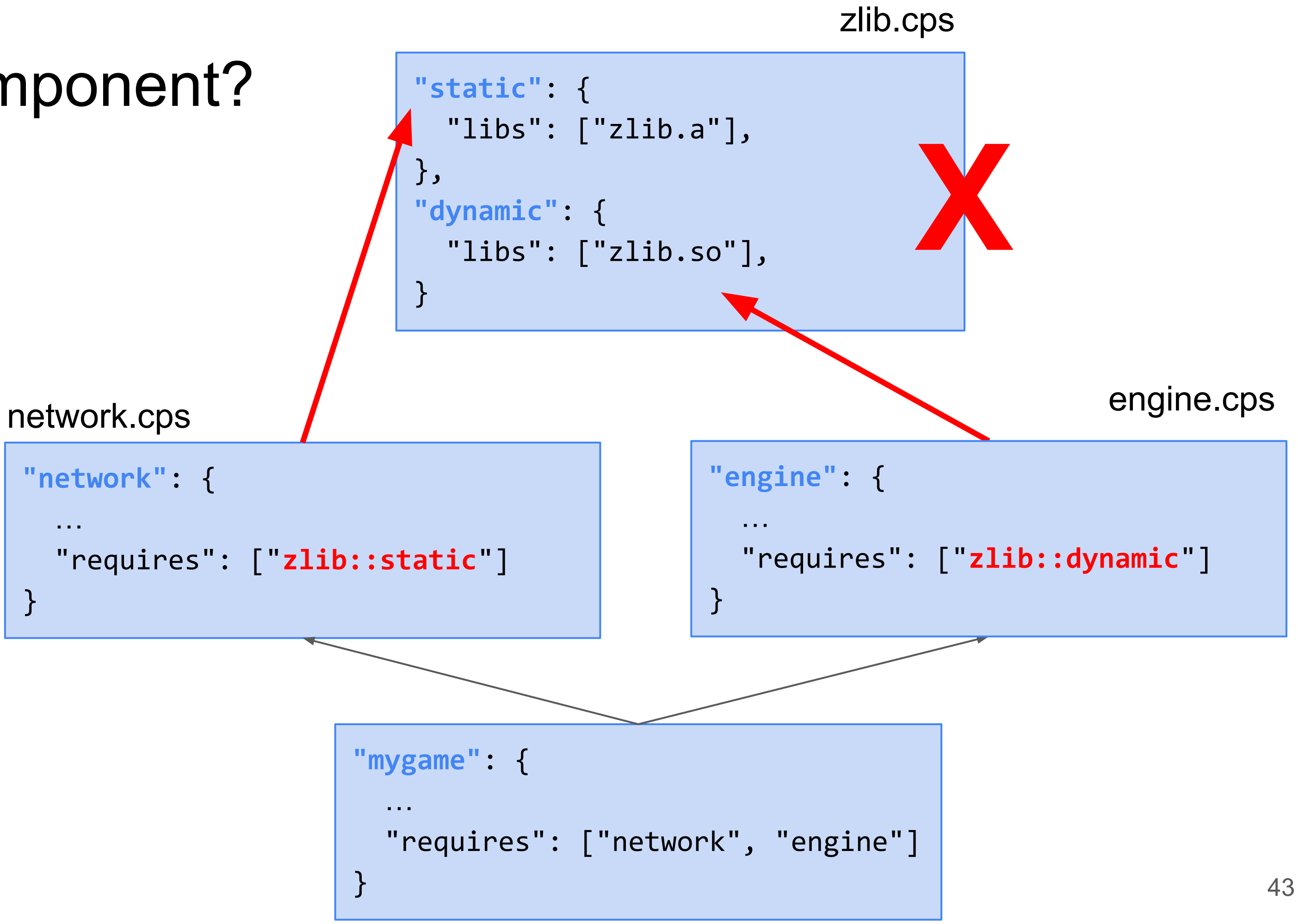
```
"network": {
  …
  "requires": ["openssl::ssl"]
}
```

engine.cps

```
"engine": {
  …
  "requires": ["openssl::crypto"]
}
```

```
"mygame": {
  …
  "requires": ["network", "engine"]
}
```

What is a component?



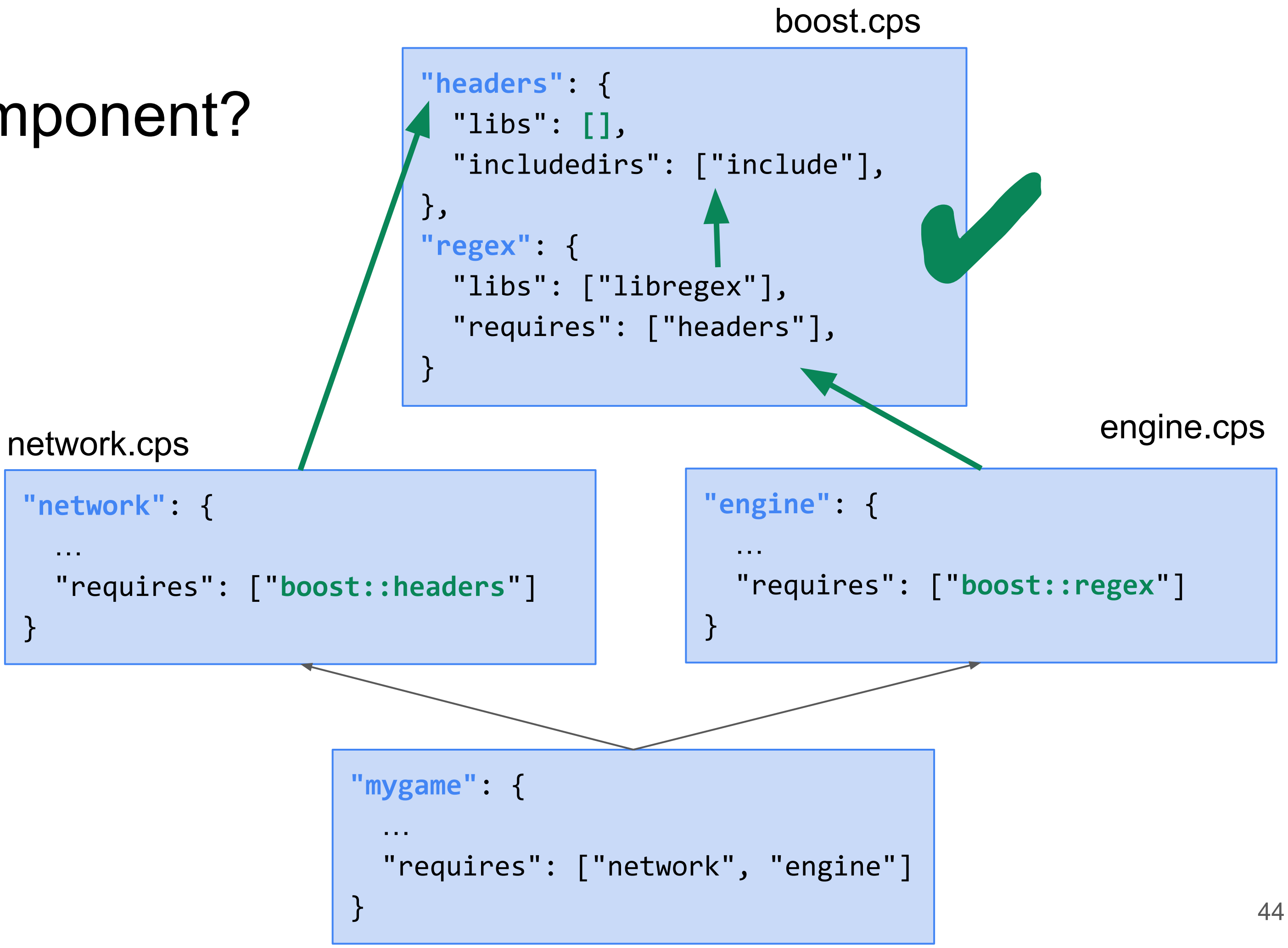
zlib.cps
```
"static": {
  "libs": ["zlib.a"],
},
"dynamic": {
  "libs": ["zlib.so"],
}
```

network.cps
```
"network": {
  …
  "requires": ["zlib::static"]
}
```

engine.cps
```
"engine": {
  …
  "requires": ["zlib::dynamic"]
}
```

```
"mygame": {
  …
  "requires": ["network", "engine"]
}
```

What is a component?



boost.cps

```
"headers": {
  "libs": [],
  "includedirs": ["include"],
},
"regex": {
  "libs": ["libregex"],
  "requires": ["headers"],
}
```

network.cps

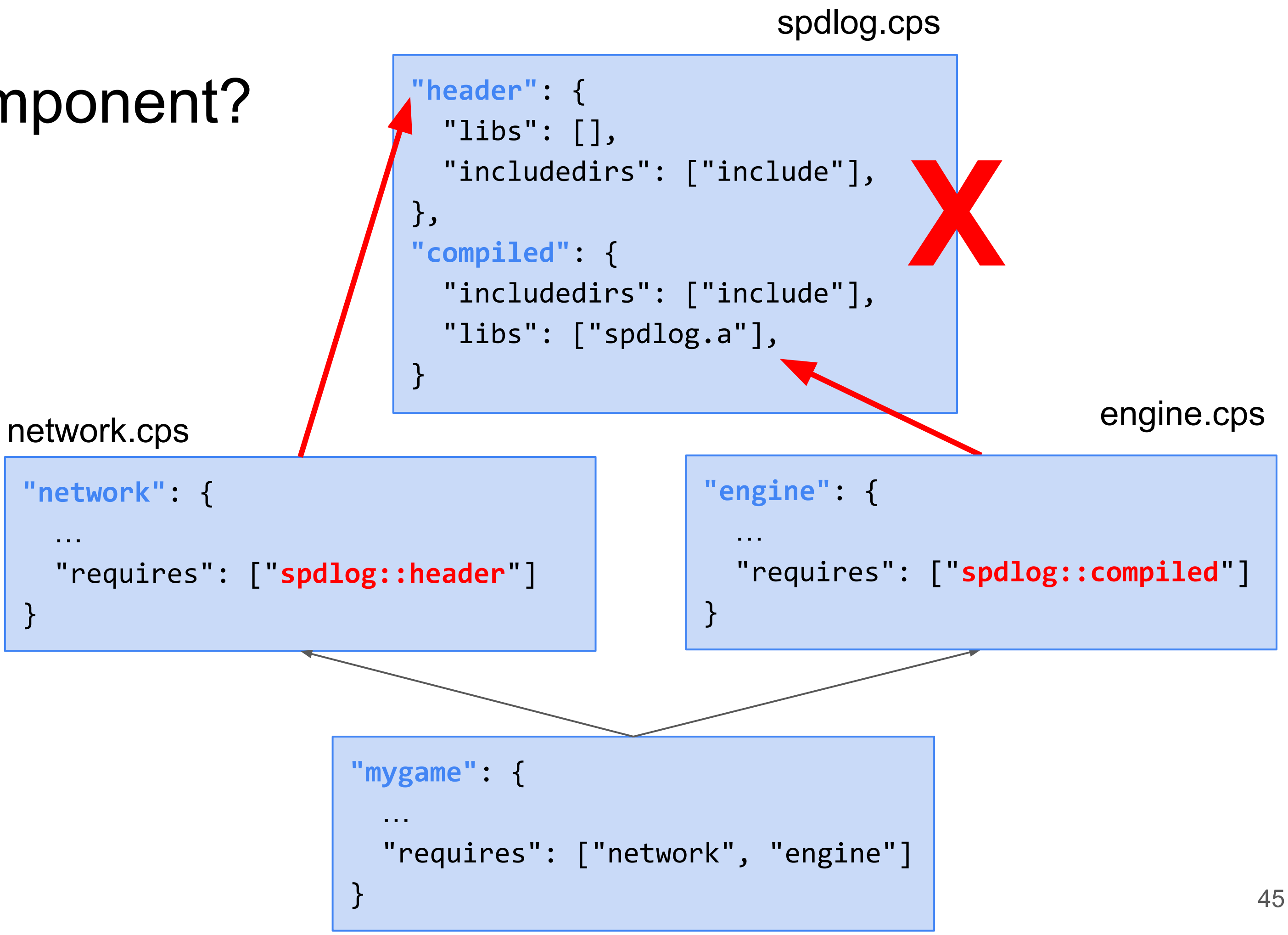
```
"network": {
  …
  "requires": ["boost::headers"]
}
```

engine.cps

```
"engine": {
  …
  "requires": ["boost::regex"]
}
```

```
"mygame": {
  …
  "requires": ["network", "engine"]
}
```

What is a component?



spdlog.cps

```
"header": {
  "libs": [],
  "includedirs": ["include"],
},
"compiled": {
  "includedirs": ["include"],
  "libs": ["spdlog.a"],
}
```

network.cps

```
"network": {
  …
  "requires": ["spdlog::header"]
}
```

engine.cps

```
"engine": {
  …
  "requires": ["spdlog::compiled"]
}
```

```
"mygame": {
  …
  "requires": ["network", "engine"]
}
```

What is a component?

openssl.cps

```
"ssl": {
  "libs": ["libssl"],
  "requires": ["crypto"],
},
"crypto": {
  "libs": ["libcrypto"],
  "requires": ["zlib::zlib"],
}
```

✓

Components:
- Should be different “parts” of a package, that can be optionally consumed
- Shouldn’t be mutually exclusive
- Will typically have inter-dependencies

spdlog.cps

```
"header": {
  "libs": [],
  "includedirs": ["include"],
},
"compiled": {
  "includedirs": ["include"],
  "libs": ["spdlog.a"],
}
```

✗

Components:
- Shouldn’t be different binary “variants” of the same code/functionality
- Aren’t an optimization over building or distribution

What is not a component?

spdlog.cps

```
"header": {
  "libs": [],
  "includedirs": ["include"],
},
"compiled": {
  "includedirs": ["include"],
  "libs": ["spdlog.a"],
}
```

✗

spdlog.cps

```
"spdlog": {
  "libs": [],
  "includedirs": ["include"],
},
```

```
|--include
|     spdlog.h
|     spdlog_impl.h
|--lib (empty)
|--spdlog.cps
```

spdlog.cps

```
"spdlog": {
  "includedirs": ["include"],
  "libs": ["spdlog.a"],
}
```

✓

```
|--include
|     spdlog.h
|--lib
|     spdlog.a
|--spdlog.cps
```

Outline

Full CPS

```
"includedirs": ["include"],
"srcdirs": null,
"libdirs": ["lib"],
"resdirs": null,
"bindirs": [ "bin"],
"builddirs": null,
"frameworkdirs": null,
"system_libs": null,
"frameworks": null,
"libs": ["zlib"],
"defines": null,
"cflags": null,
"cxxflags": null,
"sharedlinkflags": null,
"exelinkflags": null,
"objects": null,
"sysroot": null,
"requires": null,
"properties": {
  "cmake_find_mode": "both",
  "cmake_file_name": "ZLIB",
  "cmake_target_name": "ZLIB::ZLIB",
  "pkg_config_name": "zlib"
 }
```

Defines

```
|--include
|     zconf.h
|     zlib.h
|
|--lib
|     zlib.lib
|
|--licenses
      LICENSE
```

```
{
  "includedirs": ["include"],
  "libdirs": ["lib"],
  "libs": ["zlib"],
  "defines": ["ZLIB_STATIC"]
}
```

zlib.h

```
#ifdef ZLIB_STATIC
void deflateInit(...
#else
__declspec(dllexport) void deflateInit(...
#endif
```

Source packages

- No build system! (aka submodules)
- Build system agnostic
- Usage up to the consumer
- Abuse as source management to be avoided
- ODRs violations not checkeable by tooling

```
"srcdirs": ["mysrc"],  # or
"sources": ["file.h", "file.cpp", …],
```

Objects

- No build system! (aka submodules)
- Build system agnostic
- Usage up to the consumer
- ODRs violations not checkeable by tooling

```
"objdirs": ["mysrc"],  # or
"objects": ["file.obj", …],
```

Outline

Library definitions

- Paths and extensions?
- Types:
    - Header
    - Static
    - Shared
    - Dynamic

zlib.cps

```
{
  "includedirs": ["include"],
  "libdirs": ["lib"],
  "bindirs": ["bin"],
  "libs": {
      "zlib": {
          "type": "shared",
          "importlib": "zlib.lib",
          "sharedlib": "zlib.dll",
      },
   }
}
```

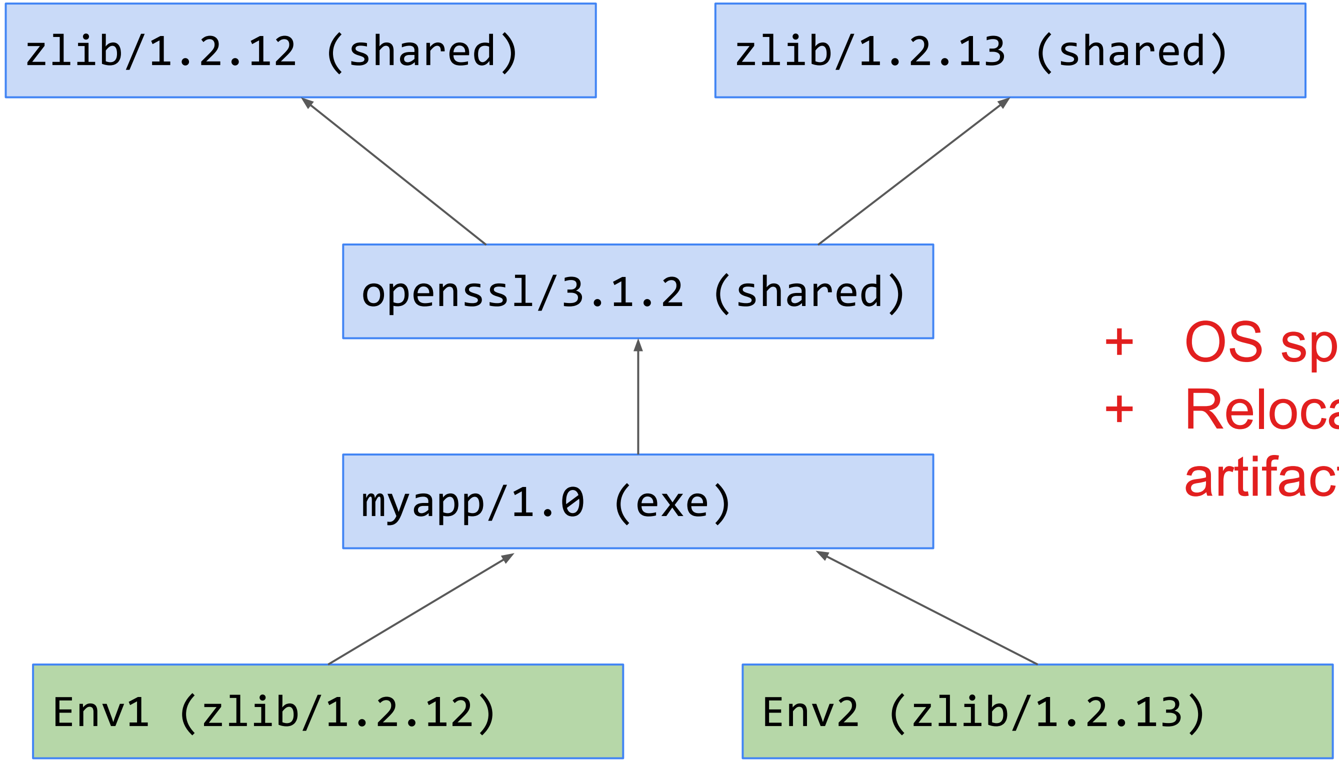
Runtime

3 mechanisms:

- RPaths
- Copy .dll to exe location
- Env-vars (PATH, LD_LIBRARY_PATH, …)

```
{
  "includedirs": ["include"],
  "libdirs": ["lib"],
  "bindirs": ["bin"],
  "libs": {
      "myplugin": {
          "type": "plugin",
          "sharedlib": "plugin.dll",
      },
   }
}
```

Runtime: why RPATHs are not great for dev-dependencies



- + OS specific
- + Relocation of artifacts

Runtime: Information is necessary

3 mechanisms:

- RPaths
- Copy .dll to exe location
- Env-vars (PATH, LD_LIBRARY_PATH, …)

```
{
  "includedirs": ["include"],
  "libdirs": ["lib"],
  "bindirs": ["bin"],
  "libs": {
      "myplugin": {
          "type": "plugin",
          "sharedlib": "plugin.dll",
      },
   }
}
```

Outline

Conditional CPS?

```
|--include
|     zconf.h
|     zlib.h
|
|--lib
|     zlib.lib
|     zlib_d.lib
```

```
{
  "includedirs": ["include"],
  "libdirs": ["lib"],
  "libs": {
      "Release": ["zlib"],
      "Debug": ["zlib_d"]
  }
}
```

Conditional CPS? Better avoid it

```
|--include
|     zconf.h
|     zlib.h
|
|--lib
|     zlib.lib
|     zlib_d.lib
```

```
{
  "includedirs": ["include"],
  "libdirs": ["lib"],
  "libs": ["zlib"]
}
```

```
{
  "includedirs": ["include"],
  "libdirs": ["lib"],
  "libs": ["zlib_d"]
}
```

Conditional CPS: Better avoid it, also binary

```
|--include
|     zlib.h
|--lib
|     zlib.lib
```

```
{
  "includedirs": ["include"],
  "libdirs": ["lib"],
  "libs": ["zlib"]
}
```

```
|--include
|     zlib.h
|--lib
|     zlib_d.lib
```

```
{
  "includedirs": ["include"],
  "libdirs": ["lib"],
  "libs": ["zlib_d"]
}
```

Conditional CPS?

```
|--include
|     zconf.h
|     zlib.h
|
|--lib
|     zlib.lib
|     zlib_safe.lib
|
|--licenses
      LICENSE
```

```
{
  "includedirs": ["include"],
  "libdirs": ["lib"],
  "libs": {
      "fast_unsafe": ["zlib"],
      "slow_safe": ["zlib_safe"]
  }
}
```

Conditional CPS?

```
|--include
|      zconf.h
|      zlib.h
|
|--lib
|    zlib.a(OSX fat)
|
|
|--licenses
       LICENSE
```

```
{
  "includedirs": ["include"],
  "libdirs": ["lib"],
  "libs": ["zlib"],
  "defines": {
      "x86_64": ["-DMY_Z_ARCH_X86"],
      "armv8": ["-DMY_Z_ARCH_ARM"]
  }
}
```

Conditional CPS for a header-only?

```
|--include
|     my_algos.h
|
```

```
{
  "includedirs": ["include"],
  "defines": {
      "fast": ["-DMY_FAST_VERSION"],
      "slow": ["-DMY_SLOW_VERSION"]
  }
}
```

Conditional CPS

- The amount of conditionals in Conan package_info() is very large:
    - Toolchains, NDKs
    - The space is very continuous and large (Android NDK api level)
    - Not easy to generate a ton of **android_ndk_apiXX_archYY.cps**
- Avoid as much as possible
    - Regular libraries

Focus on 1 package = 1 binary configuration = 1 cps

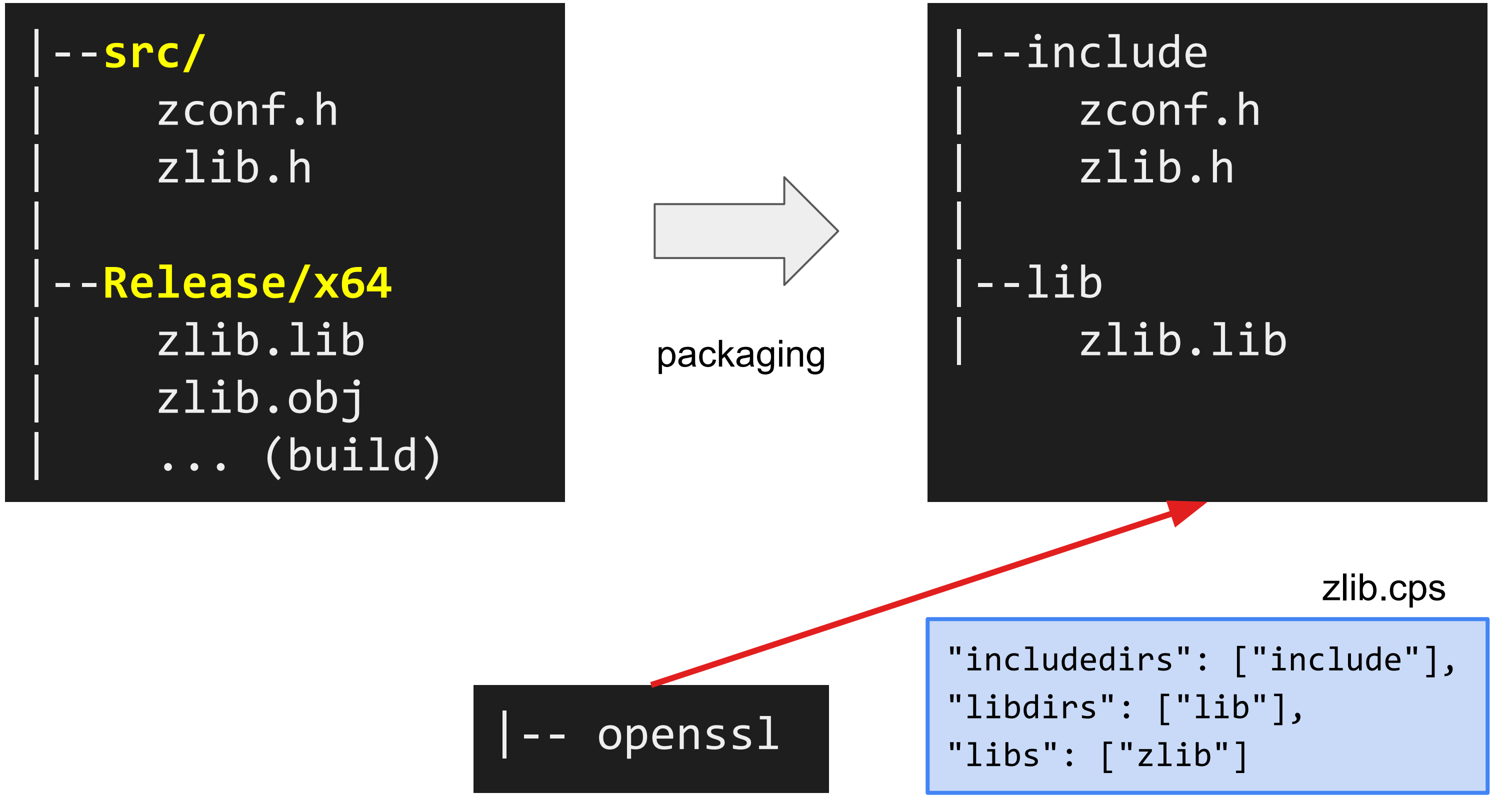
“System” packages

```python
def system_requirements(self):
    yum = package_manager.Yum(self)
    yum.install(["mesa-libGL-devel"])
    apt = package_manager.Apt(self)
    apt.install_substitutes(["libgl-dev"],
                            ["libgl1-mesa-dev"])
    ...

def package_info(self):
    if self.settings.os == "Macos":
      self.cpp_info.frameworks.append("OpenGL")
    elif self.settings.os == "Windows":
      self.cpp_info.system_libs = ["opengl32"]
    ...
```

```
{
  "system_libs": ["opengl32"],
}
```

Outline

“Editable” packages



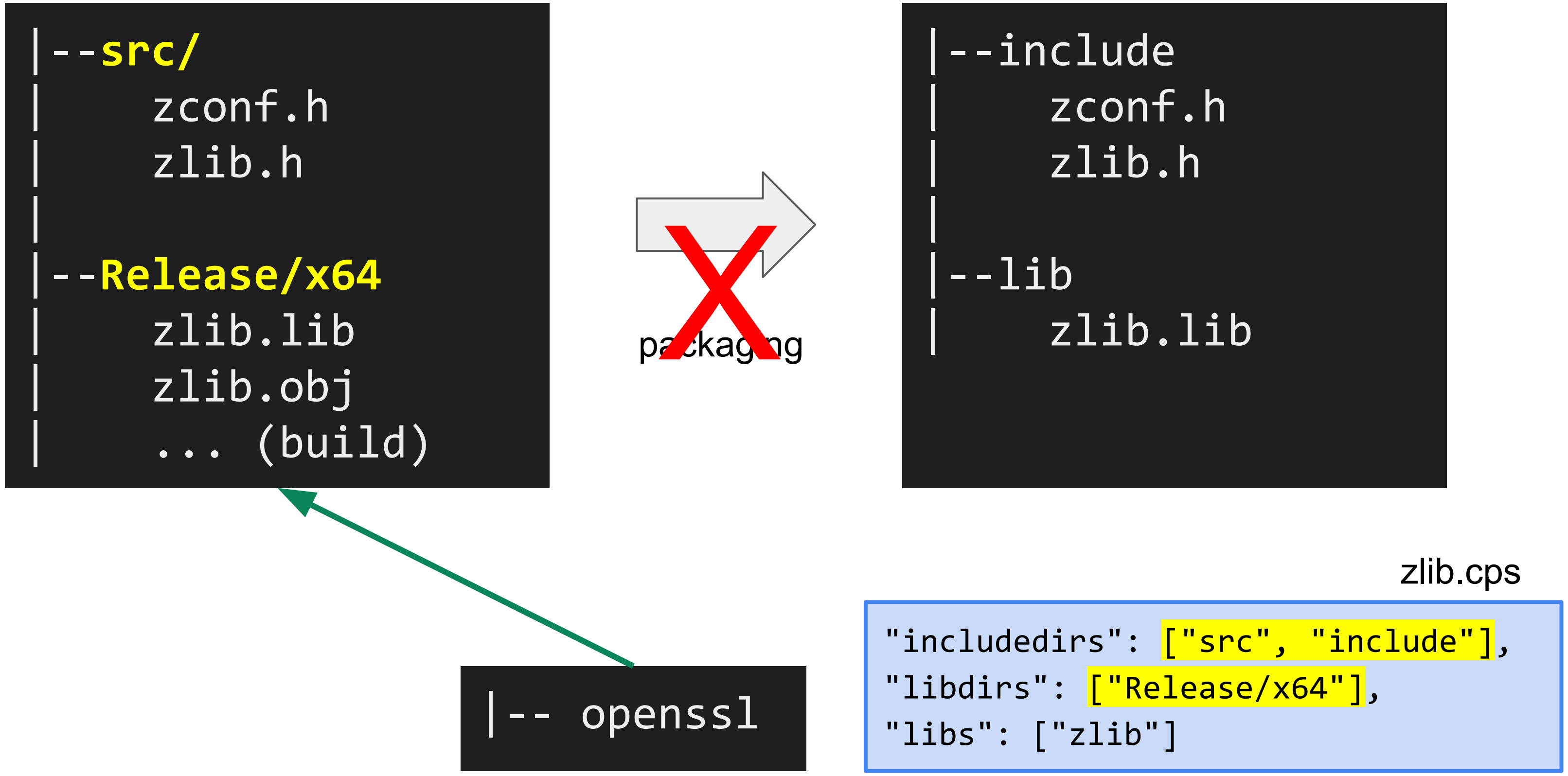
```
|--src/
|     zconf.h
|     zlib.h
|
|--Release/x64
|     zlib.lib
|     zlib.obj
|     ... (build)
```

```
|--include
|     zconf.h
|     zlib.h
|
|--lib
|     zlib.lib
```

```
|-- openssl
```

```
"includedirs": ["include"],
"libdirs": ["lib"],
"libs": ["zlib"]
```

“Editable” packages



```
|--src/
|     zconf.h
|     zlib.h
|
|--Release/x64
|     zlib.lib
|     zlib.obj
|     ... (build)
```

```
|--include
|     zconf.h
|     zlib.h
|
|--lib
|     zlib.lib
```

```
|-- openssl
```

zlib.cps

```
"includedirs": ["src", "include"],
"libdirs": ["Release/x64"],
"libs": ["zlib"]
```

Outline

About CMake modules: Protobuf

CMakeLists.txt

```
find_package(protobuf CONFIG REQUIRED)

add_executable(${PROJECT_NAME} test_package.cpp
                               addressbook.proto)
target_link_libraries(${PROJECT_NAME} PRIVATE
                      protobuf::libprotobuf)



protobuf_generate_cpp(PROTO_SRCS PROTO_HDRS TARGET
                      ${PROJECT_NAME})
protobuf_generate(LANGUAGE cpp TARGET ${PROJECT_NAME}
                  PROTOS addressbook.proto)
```

About CMake modules: Protobuf

CMakeLists.txt

```
find_package(protobuf CONFIG REQUIRED)

add_executable(${PROJECT_NAME} test_package.cpp
                               addressbook.proto)
target_link_libraries(${PROJECT_NAME} PRIVATE
                      protobuf::libprotobuf)



protobuf_generate_cpp(PROTO_SRCS PROTO_HDRS TARGET
                      ${PROJECT_NAME})
protobuf_generate(LANGUAGE cpp TARGET ${PROJECT_NAME}
                  PROTOS addressbook.proto)
```

```
|--include
|     protobuf.h
|--lib
|     protobuf.lib
|--bin
|     protoc.exe
|--cmake
|     protobuf-config.cmake
```

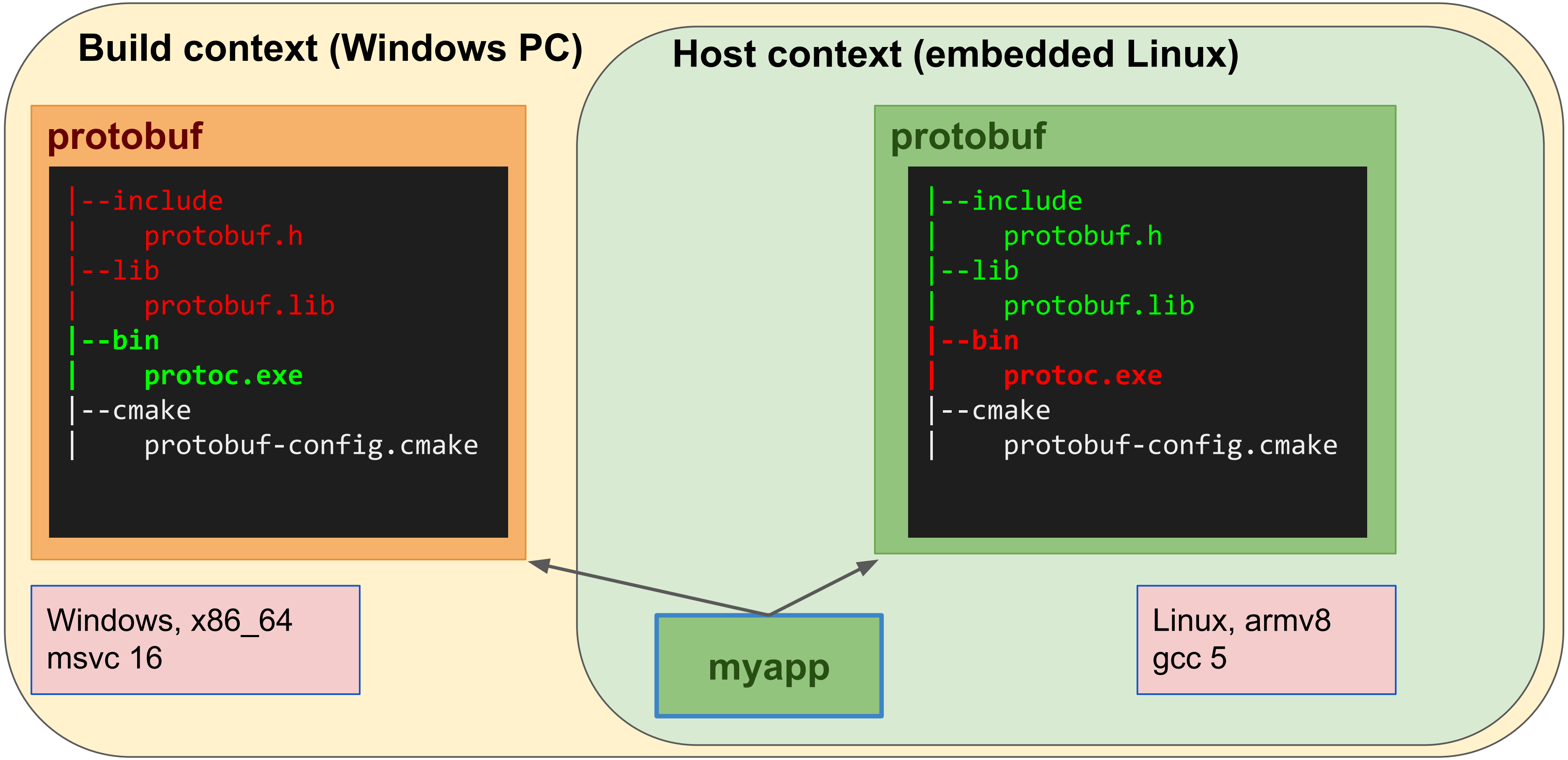
```
function(protobuf_generate_cpp)
   find_program(protoc …)
   execute_process(protoc …)
function(protobuf_generate)
...
```

About CMake modules: Protobuf

```
"includedirs": ["include"],
"libdirs": ["lib"],
"libs": ["protobuf"],
"properties": {
   "cmake_modules":
       ["cmake/protoutils.cmake"],
}
```

```
|--include
|    protobuf.h
|--lib
|    protobuf.lib
|--bin
|    protoc.exe
|--cmake
|    protoutils.cmake
```

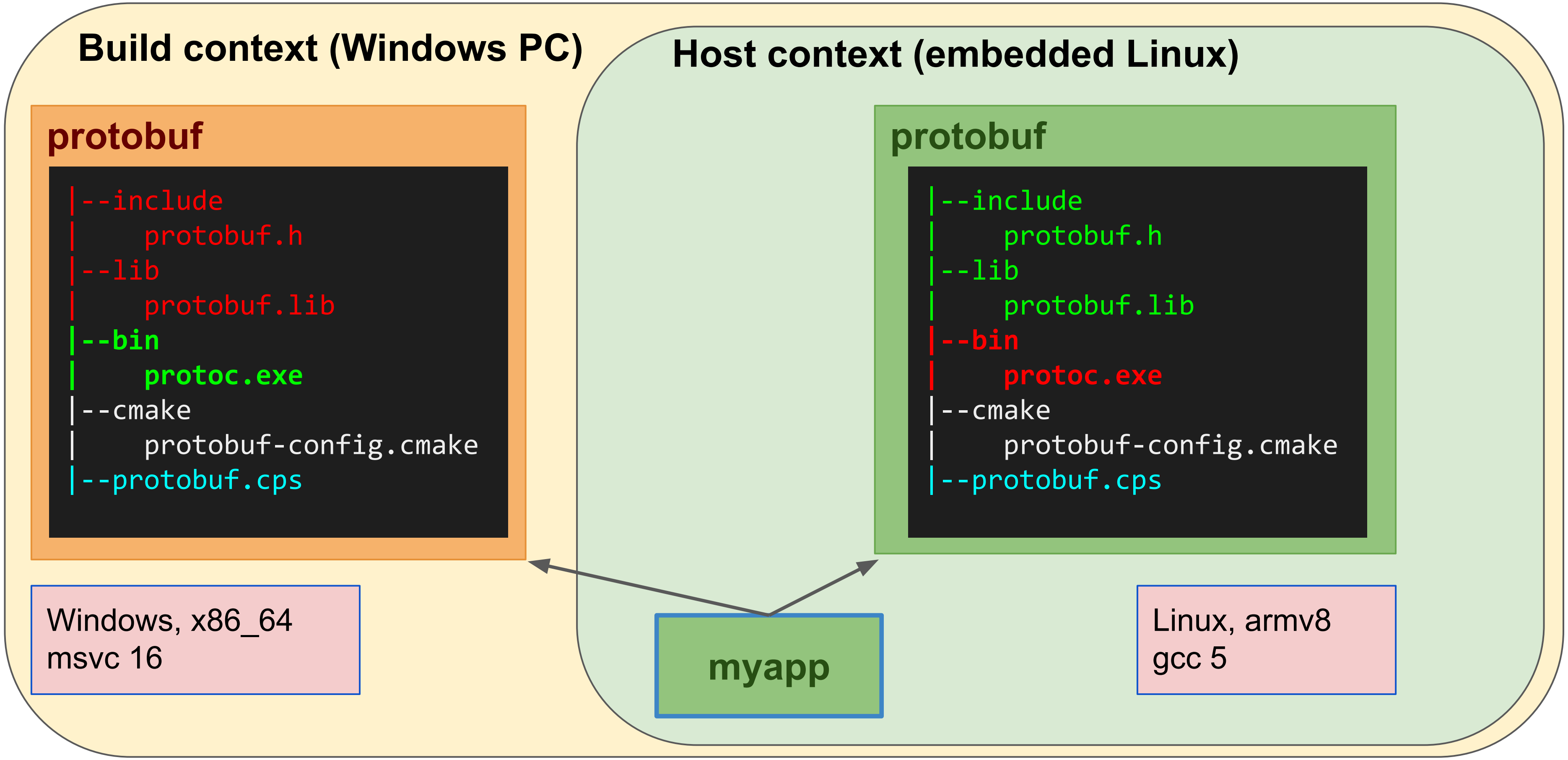
Cross-building with build & host contexts



```
|--include
|    protobuf.h
|--lib
|    protobuf.lib
|--bin
|    protoc.exe
|--cmake
|    protobuf-config.cmake
```

```
|--include
|    protobuf.h
|--lib
|    protobuf.lib
|--bin
|    protoc.exe
|--cmake
|    protobuf-config.cmake
```

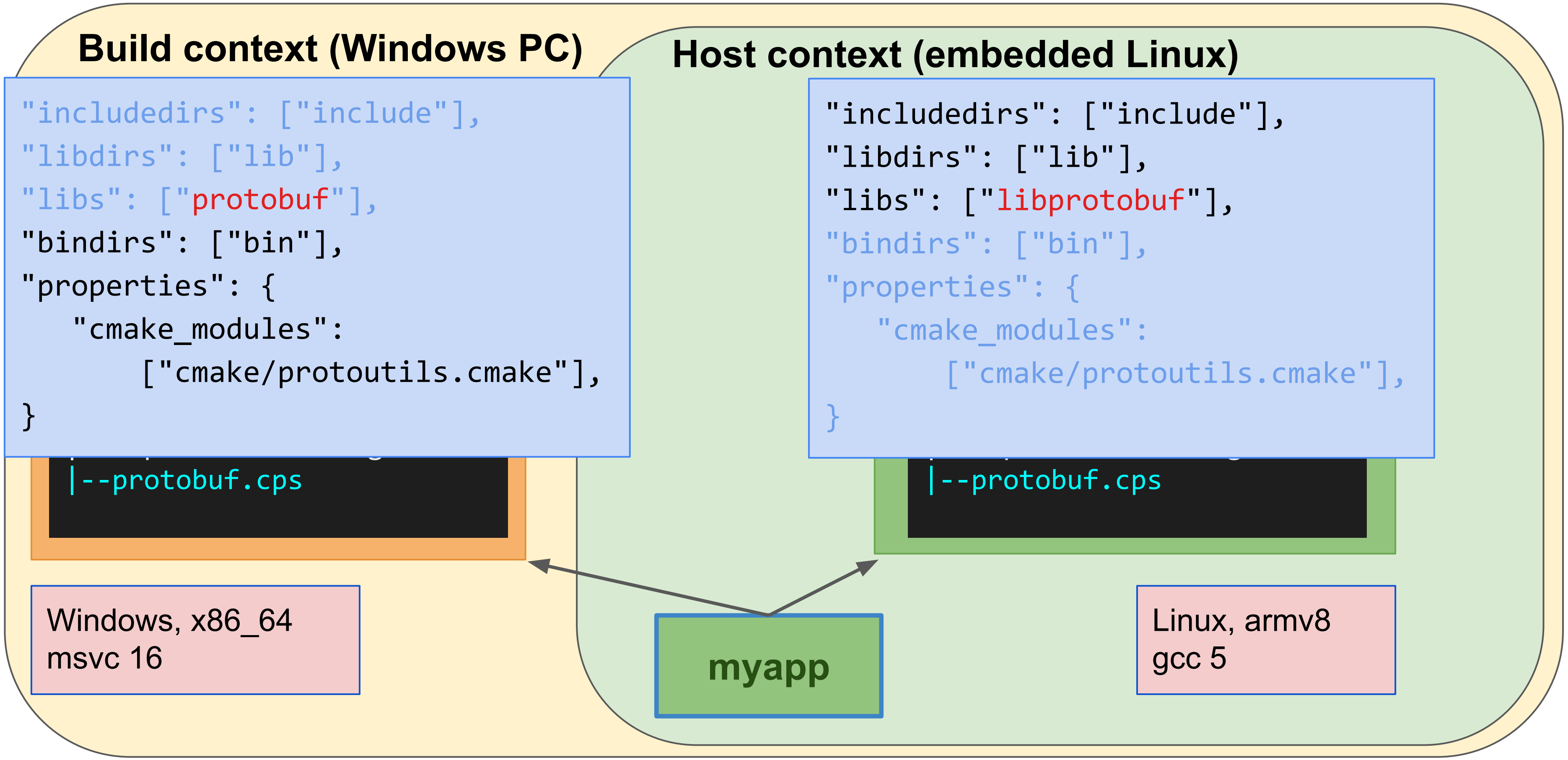
Cross-building with build & host contexts



```
|--include
|    protobuf.h
|--lib
|    protobuf.lib
|--bin
|    protoc.exe
|--cmake
|    protobuf-config.cmake
|--protobuf.cps
```

```
|--include
|    protobuf.h
|--lib
|    protobuf.lib
|--bin
|    protoc.exe
|--cmake
|    protobuf-config.cmake
|--protobuf.cps
```

Cross-building with build & host contexts



```
"includedirs": ["include"],
"libdirs": ["lib"],
"libs": ["protobuf"],
"bindirs": ["bin"],
"properties": {
   "cmake_modules":
       ["cmake/protoutils.cmake"],
}
```

```
 |--protobuf.cps
```

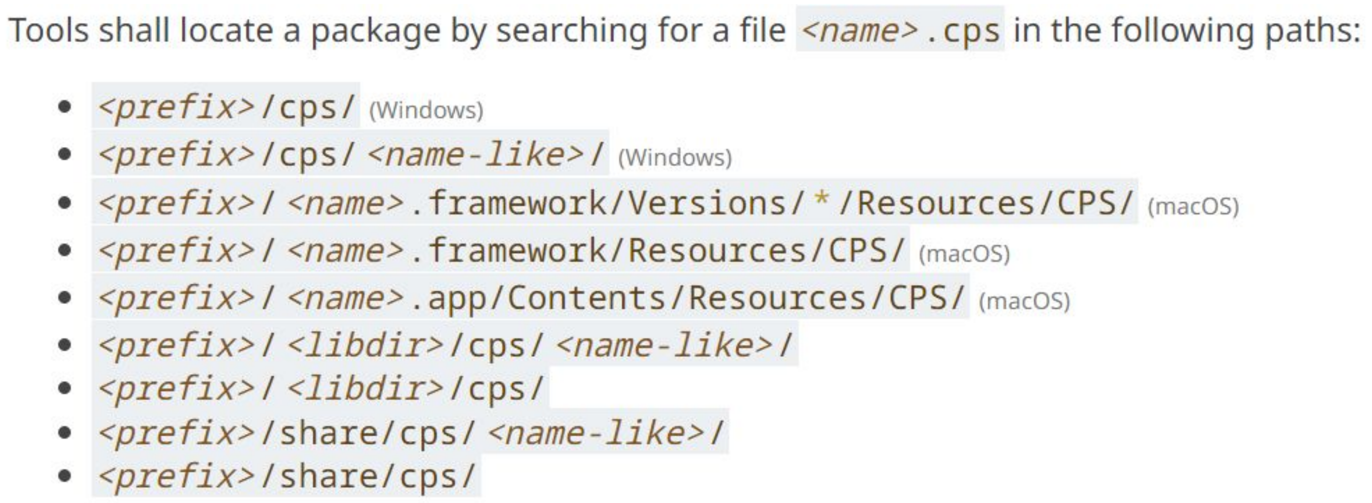
```
"includedirs": ["include"],
"libdirs": ["lib"],
"libs": ["libprotobuf"],
"bindirs": ["bin"],
"properties": {
   "cmake_modules":
       ["cmake/protoutils.cmake"],
}
```

```
 |--protobuf.cps
```

Outline

CPS files mapping

https://cps-org.github.io/cps/searching.html



Tools shall locate a package by searching for a file `<name>.cps` in the following paths:

- `<prefix>/cps/` (Windows)
- `<prefix>/cps/<name-like>/` (Windows)
- `<prefix>/<name>.framework/Versions/*/Resources/CPS/` (macOS)
- `<prefix>/<name>.framework/Resources/CPS/` (macOS)
- `<prefix>/<name>.app/Contents/Resources/CPS/` (macOS)
- `<prefix>/<libdir>/cps/<name-like>/`
- `<prefix>/<libdir>/cps/`
- `<prefix>/share/cps/<name-like>/`
- `<prefix>/share/cps/`

CPS file location mapping

Consumer project

CMakeLists.txt

```
find_package(OpenSSL CONFIG REQUIRED)
find_package(ZLIB CONFIG REQUIRED)
```

Built packages

```
|--zlib
|     include
|        zlib.h
|     ...
|     zlib.cps
|--openssl
|     include
|        ssl.h
|     ...
|     openssl.cps
```

zlib.cps

```
"includedirs": ["include"],
"libdirs": ["lib"],
"libs": ["zlib"]
```

openssl.cps

```
"crypto": {
  "libs": ["libcrypto"],
  "requires": ["zlib::zlib"],
}
```

CPS file location mapping

Consumer project

CMakeLists.txt

```
find_package(OpenSSL CONFIG REQUIRED)
find_package(ZLIB CONFIG REQUIRED)
```

mapping.cpsm

```
"zlib": "/path/zlib/zlib.cps",
"openssl": "/path/openssl/openssl.cps"
```

Built packages

```
|--zlib
|    include
|       zlib.h
|    ...
|    zlib.cps
|--openssl
|    include
|       ssl.h
|    ...
|    openssl.cps
```

zlib.cps

```
"includedirs": ["include"],
"libdirs": ["lib"],
"libs": ["zlib"]
```

openssl.cps

```
"crypto": {
  "libs": ["libcrypto"],
  "requires": ["zlib::zlib"],
}
```

CPS file location mapping

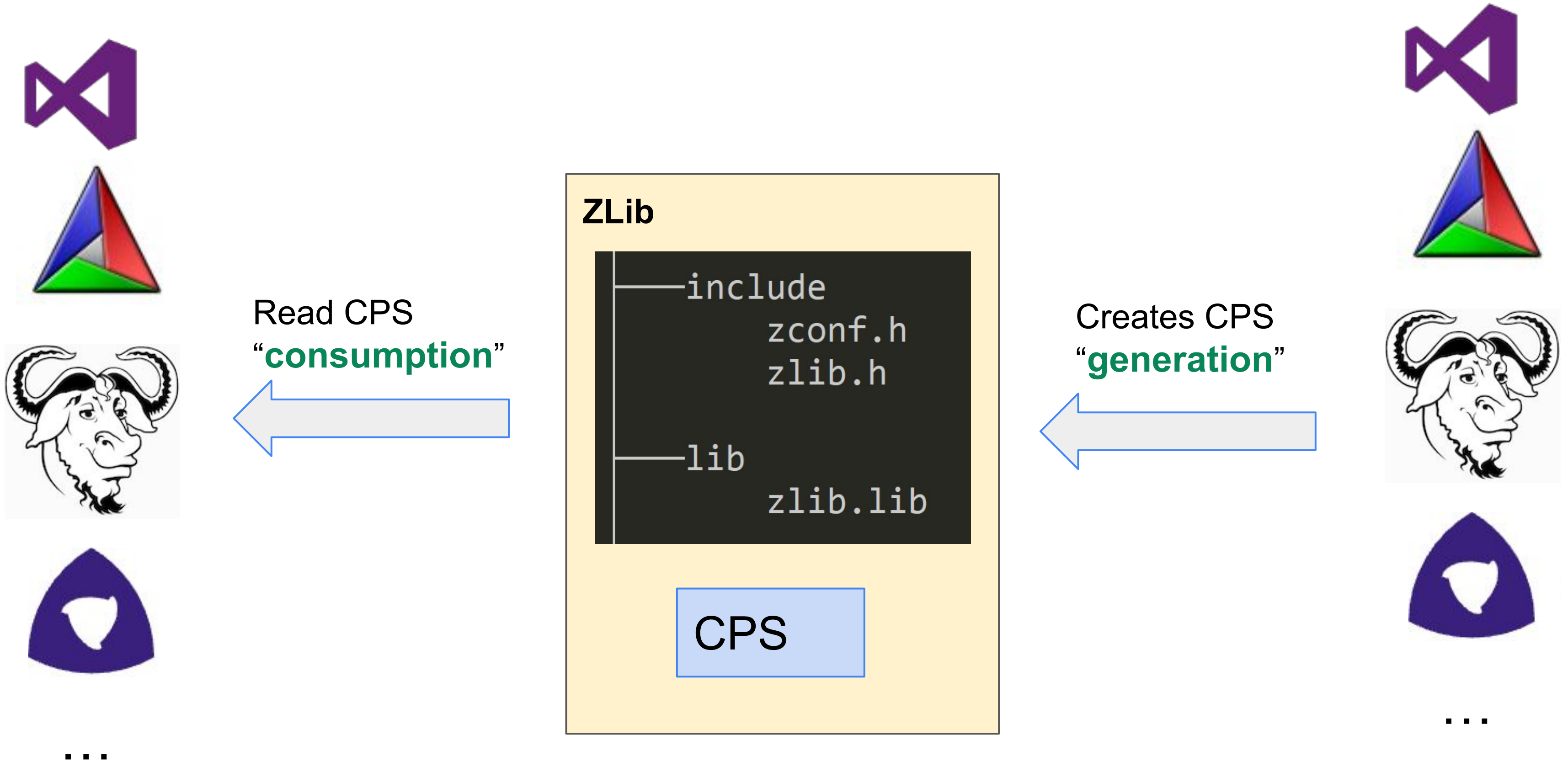
mapping.cpsm

```
"host": {
     "zlib": {
          "ReleaseDLL": "/path/zlib/linux/release/shared/zlib.cps",
          "DebugStatic": "/path/zlib/linux/debug/static/zlib.cps"
          ...
     },
    "openssl": {
          "*": "/system/openssl.cps",
     },
    "protobuf": {  # for libprotobuf for linux
          "ReleaseDLL": "/path/protobuf/linux/release/shared/protobuf.cps",
     },
},
"build": {
     "protobuf": {  # for protoc for windows
          "*": "/path/protobuf/windows/release/static/protobuf.cps",
     },
}
```

Outline

- Definition and scope
- Implementation experience
- CPS basics
    - Directories and libraries: ZLib
    - Components and requirements: Openssl
    - Full CPS
- Advanced use cases
    - Full library definition
    - Runtime
    - Conditionals
    - Editable packages
    - Protobuf modules and cross-building
    - CPS files location
- **Speeding adoption hints**
- Conclusions and next steps

Implementation tips: mapping from/to build systems



```
├──include
│       zconf.h
│       zlib.h
│
├──lib
│       zlib.lib
```

How to help (make, bazel, …) projects to generate CPS?

```
def package_info(self):
   if self.settings.os == "Windows"
          and not self.._is_mingw:
      lib = "zdll" if self.options.shared else "zlib"
   else:
      lib = "z"
   self.cpp_info.libs = [lib]
```

- Declarative
  - Batch/Shell scripts
  - Other scripts
- Introspection
  - Tool
- Custom solutions per-project
  - Templates

Consumption:
Mapping to build systems

zlib.cps

```
"includedirs": ["include"],
"libdirs": ["lib"],
"libs": ["zlib"],
"properties": {
  "cmake_find_mode": "both",
  "cmake_file_name": "ZLIB",
  "cmake_target_name": "ZLIB::ZLIB",
}
```

CMakeLists.txt (user)

```
find_package(ZLIB REQUIRED)
# find_package(ZLIB MODULE)
target_link_libraries(... ZLIB::ZLIB)
```

Mapping to build systems

zlib.cps

```
"includedirs": ["include"],
"libdirs": ["lib"],
"libs": ["zlib"],
"properties": {
  "cmake_find_mode": "both",
  "cmake_file_name": "ZLIB",
  "cmake_target_name": "ZLIB::ZLIB",
}
```

zlib-config.cmake

```
set(zlib_PKG_FOLDER "<full-path>")

set(zlib_INCLUDE_DIRS
              "${zlib_PKG_FOLDER}/include")
set(zlib_LIB_DIRS "${zlib_PKG_FOLDER}/lib")
set(zlib_LIBS zlib)


set_property(TARGET ZLIB::ZLIB
             PROPERTY
              INTERFACE_LINK_LIBRARIES
             ${zlib_LIBRARIES_TARGETS}>
             APPEND)
set_property(TARGET ZLIB::ZLIB
             PROPERTY
             INTERFACE_INCLUDE_DIRECTORIES
             ${zlib_INCLUDE_DIRS}> APPEND)
```

Mapping to existing build systems

zlib.cps

```
"includedirs": ["include"],
"libdirs": ["lib"],
"libs": ["zlib"]
```

zlib-cmake-map.cps

```
"zlib": {
  "cmake_find_mode": "both",
  "cmake_file_name": "ZLIB",
  "cmake_target_name": "ZLIB::ZLIB",
}
```

zlib-config.cmake

```
set(zlib_PKG_FOLDER "<full-path>")

set(zlib_INCLUDE_DIRS
              "${zlib_PKG_FOLDER}/include")
set(zlib_LIB_DIRS "${zlib_PKG_FOLDER}/lib")
set(zlib_LIBS zlib)


set_property(TARGET ZLIB::ZLIB
             PROPERTY
              INTERFACE_LINK_LIBRARIES
             ${zlib_LIBRARIES_TARGETS}>
             APPEND)
set_property(TARGET ZLIB::ZLIB
             PROPERTY
             INTERFACE_INCLUDE_DIRECTORIES
             ${zlib_INCLUDE_DIRS}> APPEND)
```

Mapping to build systems

zlibs.props

```
<ConanzlibRootFolder>path</ConanzlibRootFolder>

<ClCompile>
   <AdditionalIncludeDirectories>$(ConanzlibRootFolder)/include
   </AdditionalIncludeDirectories>
</ClCompile>
<Link>
  <AdditionalLibraryDirectories>$(ConanzlibRootFolder)/lib
  </AdditionalLibraryDirectories>
  <AdditionalDependencies>zlib.lib</AdditionalDependencies>
</Link>
```

zlib.cps

```
"includedirs": ["include"],
"libdirs": ["lib"],
"libs": ["zlib"]
```

Outline

What about modules?

C++20 Modules:
The Packaging and Binary Redistribution Story

Monday, Oct 2, 2023 | 16:45 - 17:45

CONAN
C/C++ Package Manager

JFrog

Luis Caro Campos
SW Tech Lead, JFrog

What about ABI, metadata information?

Read CPS

ZLib

```
build: {
 "includedirs": ["include"],
 "libdirs": ["lib"],
 "libs": ["zlib"]
},
abi: {
   dynamic_runtime: True,
   kernel_version: 4.1,
   glibc: 3.2
},
metadata: {
   compiled: 2023-10-01
   checksum: …
}
```

Creates CPS

...

...

Proposal: Lean MVP

Read CPS

ZLib

```
build: {
 "includedirs": ["include"],
 "libdirs": ["lib"],
 "libs": ["zlib"]
},
abi: {
   dynamic_runtime: True,
   kernel_version: 4.1,
   glibc: 3.2
},
metadata: {
   compiled: 2023-10-01
   checksum: ...
}
```

Creates CPS

...

...

Next steps

- Resume work on this
- Collaborate with CPS, Brett, Bill to mature proposal for ISO C++ (tooling)
- Continue evolving implementation in Conan 2.0
  - Move from Python ``package_info()`` => cps (poc)

Thank you! Questions?

JFrog | CONAN 2.0
C/C++ Package Manager