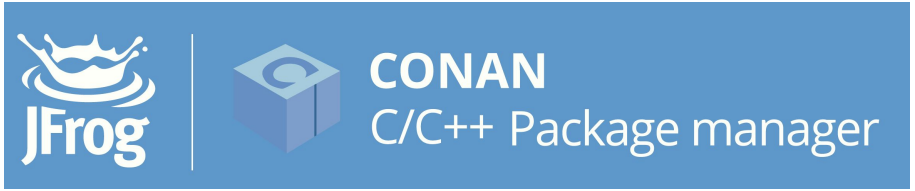


Introduction to Conan C++ Package Manager: Advanced

Diego Rodriguez-Losada, Conan Co-Founder @ JFrog
Javier García Sogo, Sr. SW Engineer @ JFrog

Coaches

Diego Rodriguez-Losada, Conan co-founder

Javier García Sogo, Sr. SW Engineer

Technical Assistants

- Daniel Manzaneque
- Carlos Zoido

Introduction

- If you don’t know Conan
  - At least the “Essentials” training
- Wait until then
  - Enroll the “Essentials” training

Exercise 0 - Setup

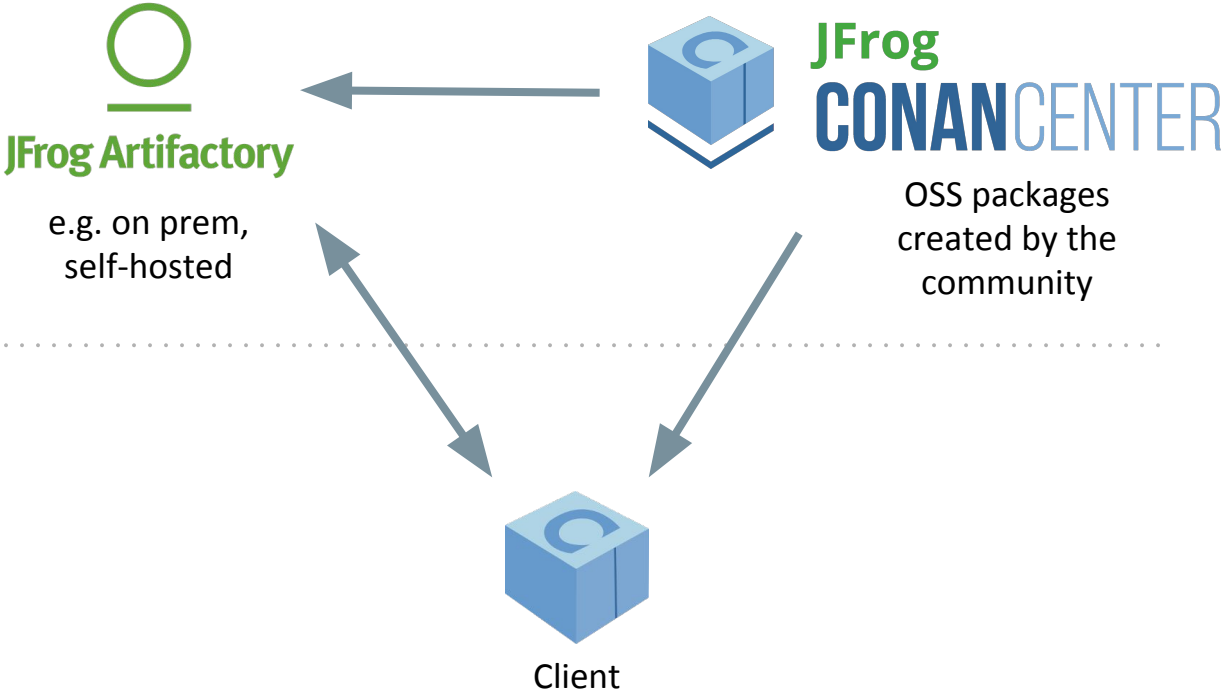
```
# https://jfrog.orbitera.com/c2m/trial/1289

 $ ssh conan@<orbitera-IP>
# Use password from orbitera
$ git clone https://github.com/conan-io/training
```

```
vm-testdriveinstance-1289-88220 con
------------------


------ Outputs ------
Username:
admin

Artifactory URL:
http://35.226.56.161:8082/

Password:
zgQG1h6NjJ

IP:
35.226.56.161

SSH Username:
conan

Jenkins Credential:
uubSq9uN1o

Jenkins URL:
http://35.226.56.161:8080/
---------------------
>
```

Introduction

- OSS, MIT license
- Multi-platform
- Any build system
- Stable
- Active

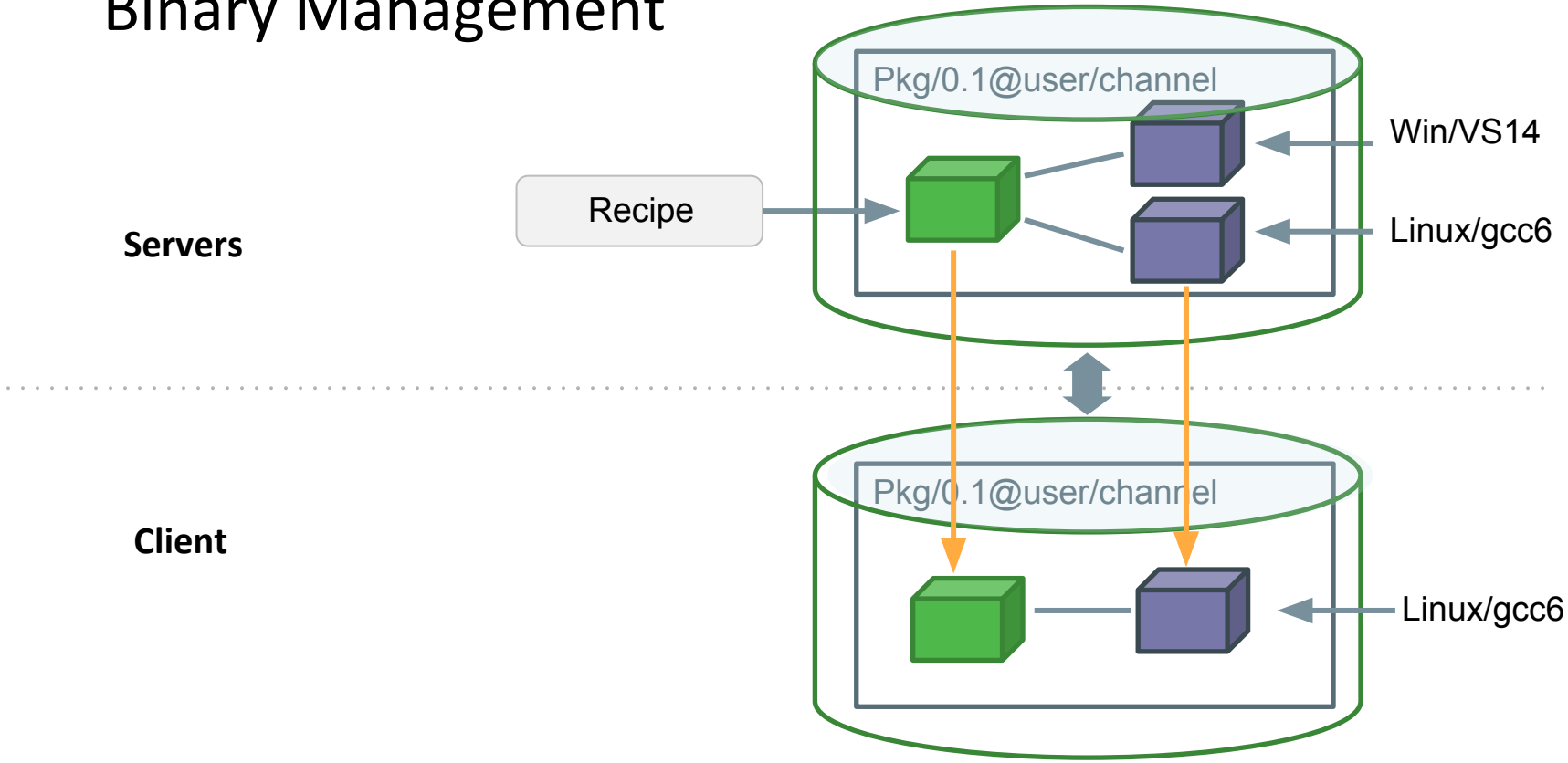
Architecture



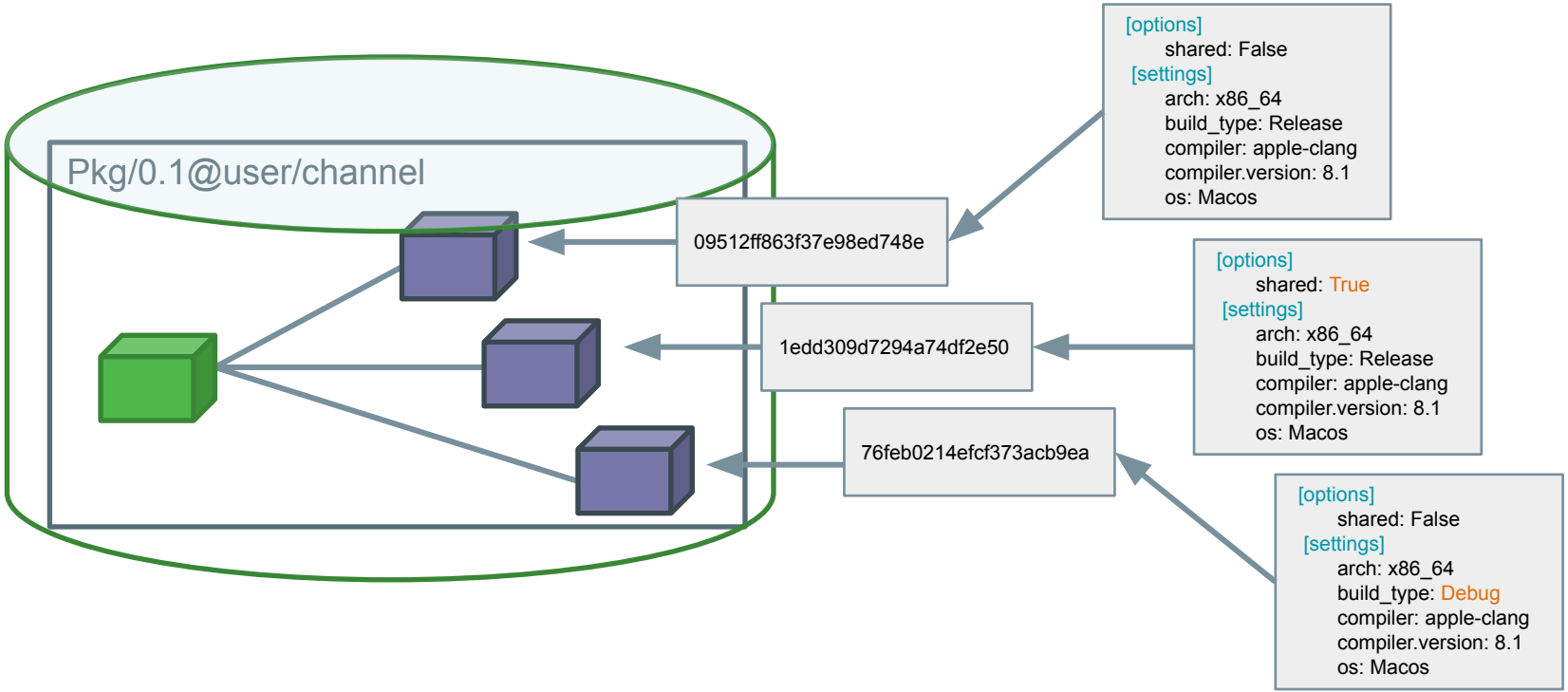
Servers
(artifact storage)

Developer
machine / CI

Binary Management



Binary Management



Outline

- Essentials (Ex 1-15, precondition)
- **Requirements**
  - Build-requires
  - Python-requires
- Versioning
  - Version ranges
  - Revisions
  - Lockfiles
- Package ID
- Hooks and Conan configuration

Here be dragons

- The “catchup.sh” script to be used if lost
- There are bugs, on purpose
- They are solved by the “catchup.sh”

Exercise 16 - Transitive requirements

Goal:

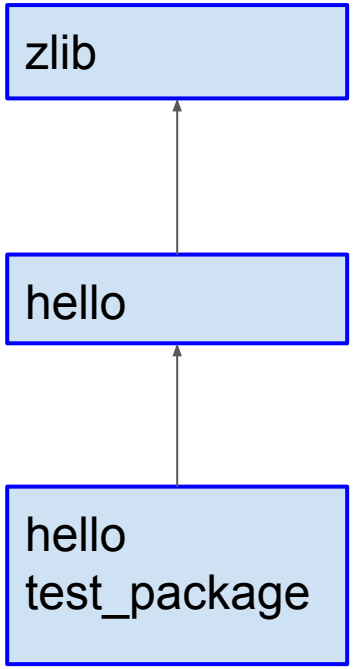
- Use another package libraries in the code of our package

Task:

- Create a “hello” package that depends on the ConanCenter “zlib” one

Success:

- The “test_package” of the “hello” package shows information about compressing strings



Exercise 16 – Transitive requiring zlib [/requires]

$ cd training/requires

src/hello.cpp

```
#include <iostream>
#include "hello.h"
#include <zlib.h>

void hello(){
    std::cout << "Hello world!\n";

    char buffer_in [100] = {"some string"};
    char buffer_out [100] = {0};

    z_stream defstream;

    ...

    printf("size: %lu\n", strlen(buffer_out));
```

conanfile.py

```
class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    settings = "os", "compiler", "arch"
    generators = "cmake"
    exports_sources = "src/*"
    requires = "zlib/1.2.11"
```

Exercise 16 – Transitive requiring zlib [/requires]

3:00

```
$ conan create . user/testing # dragons!
# What if we try to create the package for RPI?
$ conan create . user/testing -pr=rpi_armv7 # Error
$ conan create . user/testing -pr=rpi_armv7 --build=missing
```

Exercise 16 – Transitive requiring zlib [/requires]

- Dragon: ZLib ⇔ zlib ⇒ Use lower case always
- What to do when “Binary missing”?:
  - Build missing dependencies as zlib on the fly ✗
    - conan create . --build=missing
  - The binary of zlib should have been there (created by CI)
    - Create zlib with armv7 profile correctly ✓

Exercise 17 - Conflicts

Goal:

- Learn about requirements version conflicts and how to solve them downstream with overrides

Task:

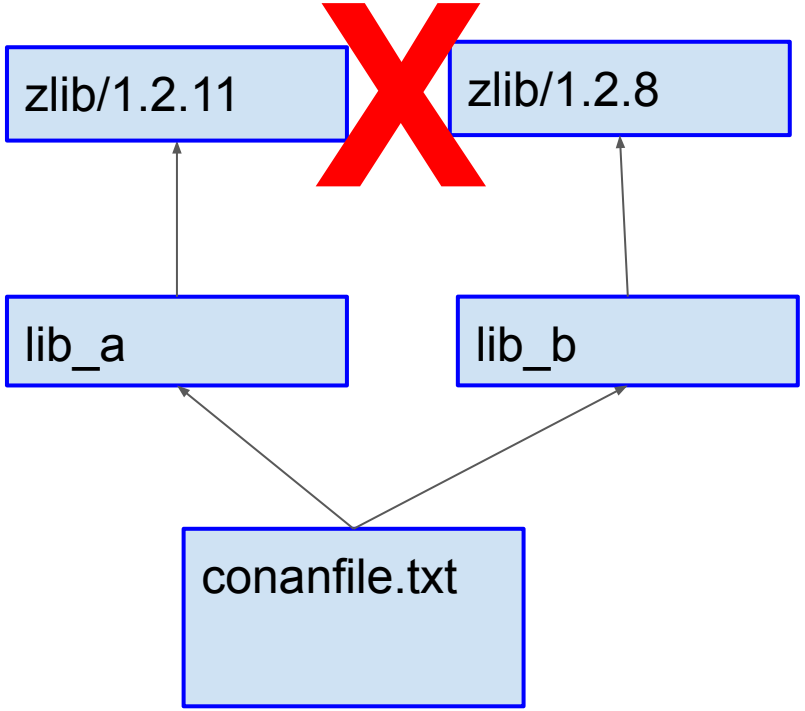
- Solve a version conflict, overriding with the desired dependency version

Success:

- “conan install .” installs without raising a conflict error

Exercise 17 - Conflicts [/requires_conflict]

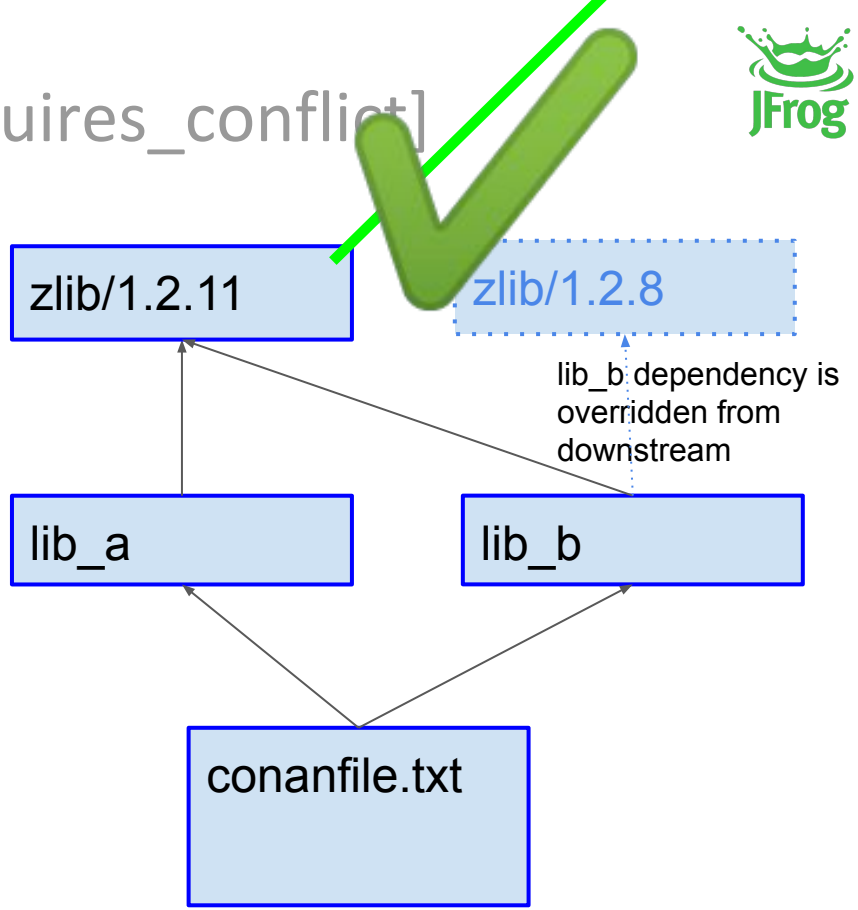
```
$ cd requires_conflict
$ conan create lib_a user/testing
$ conan create lib_b user/testing
$ conan install . # Error
```



Exercise 17 - Conflicts [/requires_conflict]

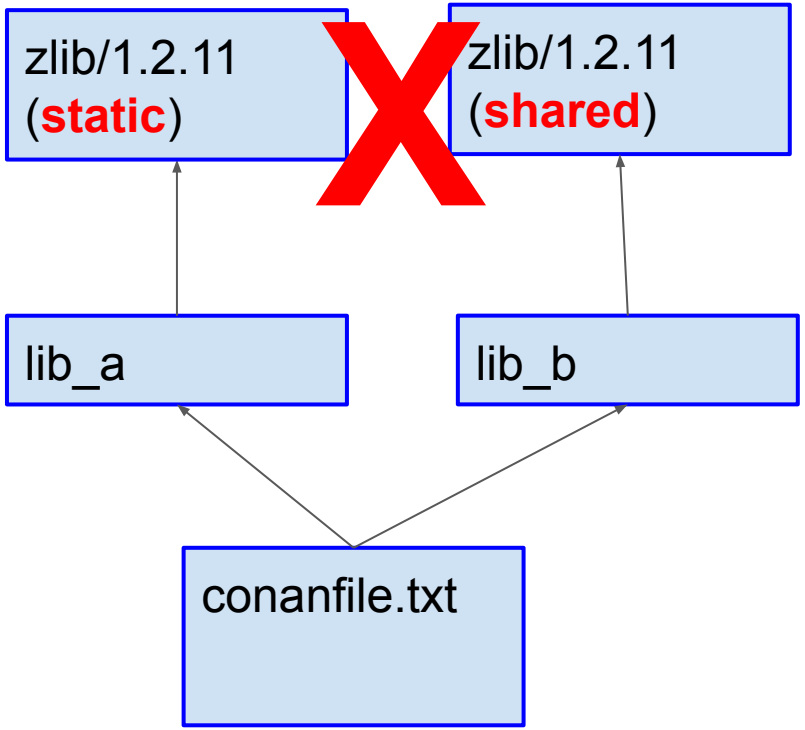
3:00

```
$ cd requires_conflict
$ conan create lib_a user/testing
$ conan create lib_b user/testing
$ conan install . # Error
# Edit consumer conanfile.txt
# add zlib/1.2.11 to [requires]
$ conan install .
```



Exercise 17 – Conflicts

- Versions generate conflicts
- Configuration can generate conflicts too:
  - Different options
  - Solved in the same way



Exercise 18 - Conditional requirements

Goal:

- Learn to conditionally depend on one library based on the value of one option

Task:

- Complete the “requirements()” method to make it conditional to the option

Success:

- “conan create . user/testing -o hello:zip=False” works without depending on the “zlib” package

Exercise 18 - Conditional requirements

conanfile.py

```
class HelloConan(ConanFile):
    options = {"zip": [True, False]}
    default_options = {"zip": True}

    requires = "zlib/1.2.11"

    def build(self):
        cmake = CMake(self)
        if self.options.zip:
            cmake.definitions["WITH_ZIP"] = "1"
        else:
            cmake.definitions["WITH_ZIP"] = "0"
        cmake.configure(source_folder="src")
```

Exercise 18 - Conditional [/requires_conditional]

3:00

```
$ cd requires_conditional
$ conan create . user/testing -o hello:zip=False
# check in output that zlib is in the deps

# Edit conanfile.py
$ conan create . user/testing -o hello:zip=False
# zlib should NOT be in the deps
```

conanfile.py

```
class HelloConan(ConanFile):
    options = {"zip": [True, False]}
    default_options = {"zip": True}

    def requirements(self):
        self.requires("zlib/1.2.11")

    def build(self):
        ...
```

Exercise 19 - Unit tests with gtest [/gtest]

Goal:

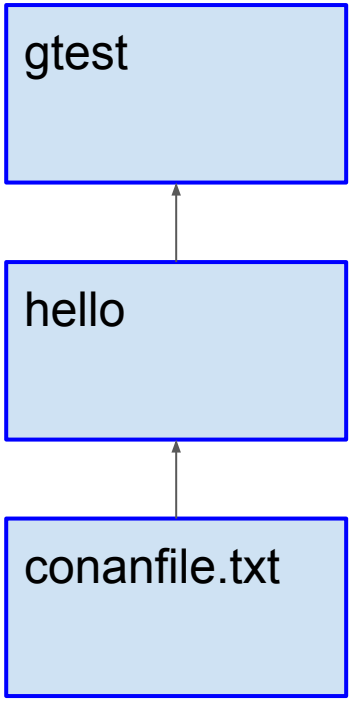
- Build and run unit tests with gtest

Task:

- Create the “hello” package, that runs unit tests

Success:

- See the unit tests running while creating the “hello” package



Exercise 19 – Unit tests with gtest [/gtest]

$ cd gtest/hello

test.cpp

```
#include <gtest/gtest.h>
#include "hello.h"

TEST(SalutationTest, Static) {
  EXPECT_EQ(string("Hello World!"), message());
}
```

Exercise 19 – Unit tests with gtest [/gtest]

conanfile.py

```
class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"

    requires = "gtest/1.8.1"

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()
        self.run("bin/runUnitTests")
```

Exercise 19 – Unit tests with gtest [/gtest]

3:00

```
# create the “hello” package, check that it runs tests
$ cd gtest/hello
$ conan create . user/testing # dragon!
# Now move to a consumer of “hello”, and install
$ cd ../consumer
$ conan install .
# check dependencies (gtest installed!)
```

Exercise 20 – Gtest as build-require [/gtest]

Goal:

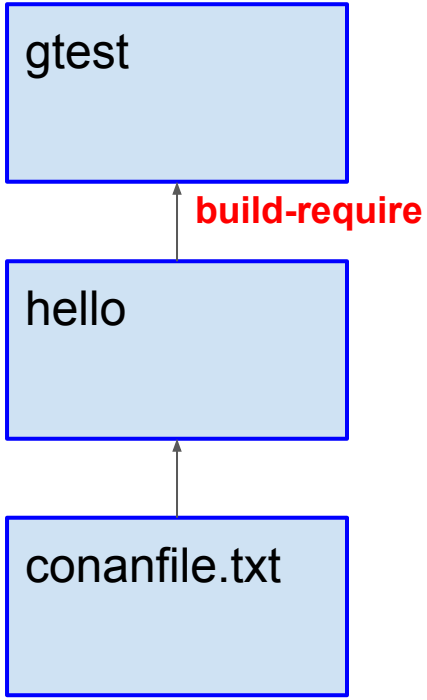
- Learn about “build_requires”

Task:

- Change the “hello” “requires=gtest” to a “build_requires”

Success:

- See the dependency graph of consumers of “hello” do NOT depend on gtest



Exercise 20 – Gtest as build-require [/gtest]

3:00

```
$ cd gtest/hello
# replace “requires” ⇒ “build_requires”
# create the “hello” package, check that it runs tests
$ conan create . user/testing
# Now move to a consumer of “hello”, and install
$ cd ../consumer
$ conan install .
# check dependencies (gtest not installed!)
```

Exercise 21 – CMake as build-require

Goal:

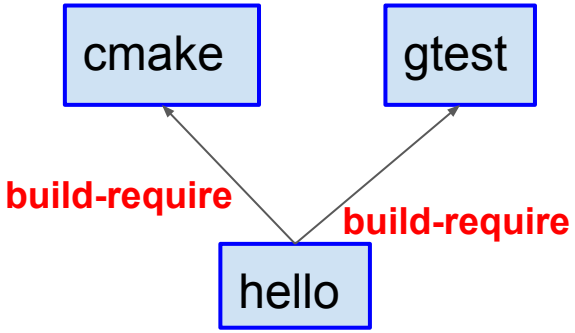
- Use “cmake” as build-require

Task:

- Build the “hello” package with a more modern cmake version injecting “cmake” package as “build_requires” in the profile

Success:

- Read in the output of “conan create” of “hello” that it is being built with another cmake version



Exercise 21 – CMake as build-require [/gtest]

conanfile.py

```
class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"
    generators = "cmake"
    exports_sources = "*"
    build_requires = "gtest/1.8.1", "cmake/3.16.3"
```

Exercise 21 – CMake from build_require [/gtest]

```
$ cd gtest/hello
$ cmake --version
# check line in CMakeLists.txt:
  message(STATUS "CMAKE VERSION ${CMAKE_VERSION}")
$ conan create . user/testing
# Read output cmake version (should be 3.7)
$ vim myprofile  # create the profile as defined at right
$ conan create . user/testing -pr=myprofile
# Read output cmake version (should be 3.16)
$ cmake --version # Check system cmake version still 3.7!
```

myprofile

```
include(default)

[build_requires]
cmake/3.16.3
```

A few notes about build_requires

- They shouldn't affect the binary
  - They are not taken into account in the package ID
- Use them **only** for tools:
  - Build tools, like cmake.
    - E.g. OpenSSL in Windows build-requires Nasm and Strawberry Perl
  - Testing frameworks
- Use them in profiles for common things (cmake)
- Use them in recipes for specific, and package specific things (testing framework)

Ex. 22 – Run apps from packages [/running_apps]

Goal:

- Learn how to use and run apps contained in Conan packages

Task:

- Run cmake 3.16.3 from the Conan package in the user terminal

Success:

- cmake --version shows 3.16.3 and we can easily get back to use the system cmake 3.7

Ex. 22 – Run apps from packages [/running_apps]

- **3 ways:**
  - Add a **method to conanfile.py** (also in conanfile.txt) to copy dependencies artifacts to current folder
    - conanfile.py `imports()` and `deploy()` methods: fine control what to import from cache
  - Use the **deploy generator** to copy dependencies to current folder

```
$ conan install cmake/3.16.3@ -g deploy
```

  - Use **virtualenv generators** to use dependencies from the cache

```
$ conan install cmake/3.16.3@ -g virtualrunenv
```

Ex. 22 – Run apps from packages [/running_apps]

3:00

```
$ cd running_apps
# use the deploy generator, inspect the folders that are installed
$ conan install cmake/3.16.3@ -g deploy
$ cmake/bin/cmake --version
$ rm -rf cmake

# use the virtualrunenv generator, inspect the created files
$ conan install cmake/3.16.3@ -g virtualrunenv
$ cmake --version  # system one (3.7)
$ source activate_run.sh
$ cmake --version  # conan package (3.16)
$ source deactivate_run.sh
$ cmake --version   # system one (3.7)
```

Exercise 23 – Python requires [/python_requires]

Goal:

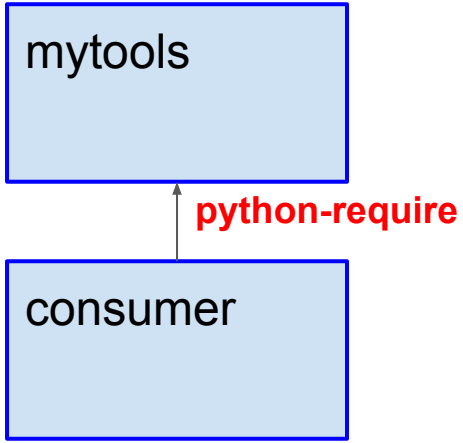
- Learn to reuse python code in recipes

Task:

- Change the “mytools” package to display the name of the package that uses the “mymsg()” function

Success:

- See the message “My tool cool message” while creating the “consumer” package



Exercise 23 - Python requires [/python_requires]

```
$ cd python_requires/mytools
$ conan export . user/testing
```

conanfile.py

```python
from conans import ConanFile

def mymsg(conanfile):
    print("MyTool working cool message Pkg:%s!!!" % conanfile.name)

class ToolConan(ConanFile):
    name = "mytools"
    version = "0.1"
```

Exercise 23 - Python requires [/python_requires]

```
$ cd python_requires/consumer
```

conanfile.py

```python
from conans import ConanFile

class ConsumerConan(ConanFile):
    python_requires = "mytools/0.1@user/testing"

    def build(self):
        mytools = self.python_requires["mytools"].module
        mytools.mymsg(self)
```

Exercise 23 - Python requires [/python_requires]

```
$ cd python_requires/mytools
$ conan export . user/testing
# Use the python requires in another package
$ cd ../consumer
$ conan create . user/testing # Dragon!
```

3:00

Exercise 23 - Python requires

NOTES

- python-requires DO NOT have binary packages, only python code
- python-requires can have dependencies to other python-requires (keep minimum)
- A recipe can have multiple python requires
- They might contain other files (source file, build scripts)
- They affect the package-ID (changing python-require version might require build new binaries for packages using them)

Python requires (inheritance)

```
from conans import ConanFile


class BaseConanFile(ConanFile):
    ...
    def build(self):
        ...
    def package(self):
        ...
    def package_info(self):
```

Python requires (inheritance II)

```
from conans import ConanFile


class Pkg(ConanFile):
    python_requires = "mytools/0.1@user/testing"
    python_requires_extend = "mytools.BaseConanFile"
    # inherits the source(), build()...
```

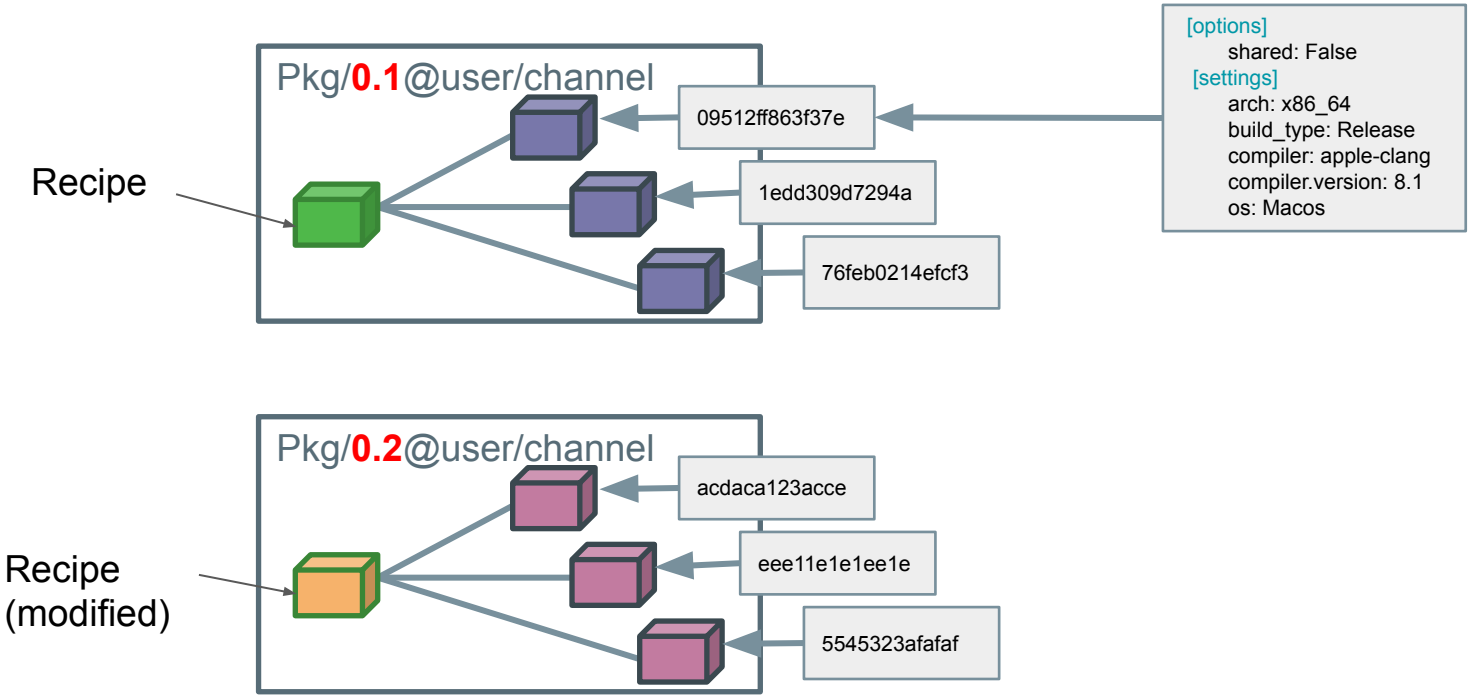
Outline

- Essentials (Ex 1-15, precondition)
- Requirements
  - Build-requires
  - Python-requires
- **Versioning**
  - Version ranges
  - Revisions
  - Lockfiles
- Package ID
- Hooks and Conan configuration

Approaches to versioning

- Bump version (semver):
  - 1.2.3->1.2.4
  - 2.8.12->3.0.0
  - What if you are packaging Boost 1.64.0, and need to do a change to the recipe?
    - 1.64.1? Mismatch to the Boost version you are packaging
  - Versions might use version ranges requirements
- Revisions:
  - pkg/version@user/channel#revision
  - revision is internal, automatic (hash)

Recipe & Binary Management



Exercise 24 – Version ranges [/version_ranges]

Goal:

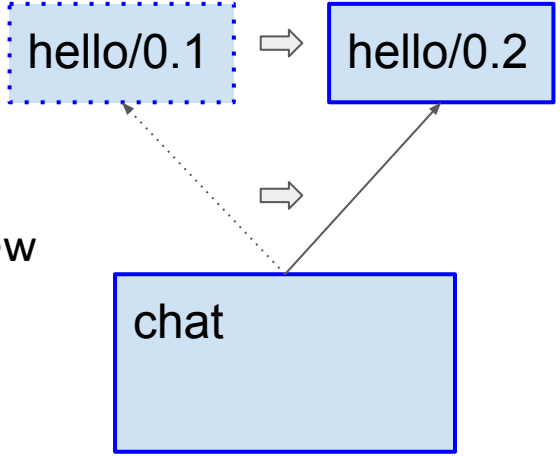
- Learn version ranges to depend on versions

Task:

- Do a change in C++ code and create a “hello/0.2” new version, and use it

Success:

- See the new message from the hello() function without modifying the “chat” conanfile.py



Exercise 24 – Version ranges [/version_ranges]

$ cd version_ranges

chat/conanfile.py

```
class ChatConan(ConanFile):
    name = "chat"
    version = "0.1"
    ..
    requires = "hello/[>0.0 <1.0]@user/testing"
```

chat/src/chat.cpp

```
void chat(){
    hello();
    hello();
    hello();
}
```

Exercise 24 – Version ranges [/version_ranges]

3:00

```
$ cd version_ranges
$ conan create hello    hello/0.1@user/testing  # note the syntax!
$ conan create chat  user/testing # read output

# do a change in hello.cpp msg and create 0.2
$ conan create hello     hello/0.2@user/testing

# the chat package will use it because it is inside its valid range
$ conan create chat user/testing
# you should see the new message from 0.2
```

Version ranges syntax

```
$ conan install "hello/[>0.0 <1.0]@user/testing"
$ conan install "hello/[*]@user/testing"
$ conan install "hello/[~1.1]@user/testing"
```

Exercise 25 – Revisions [/revisions]

Goal:

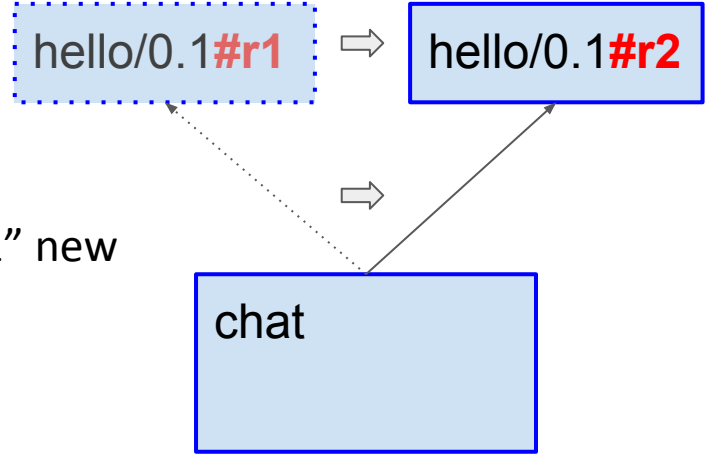
- Learn package revisions

Task:

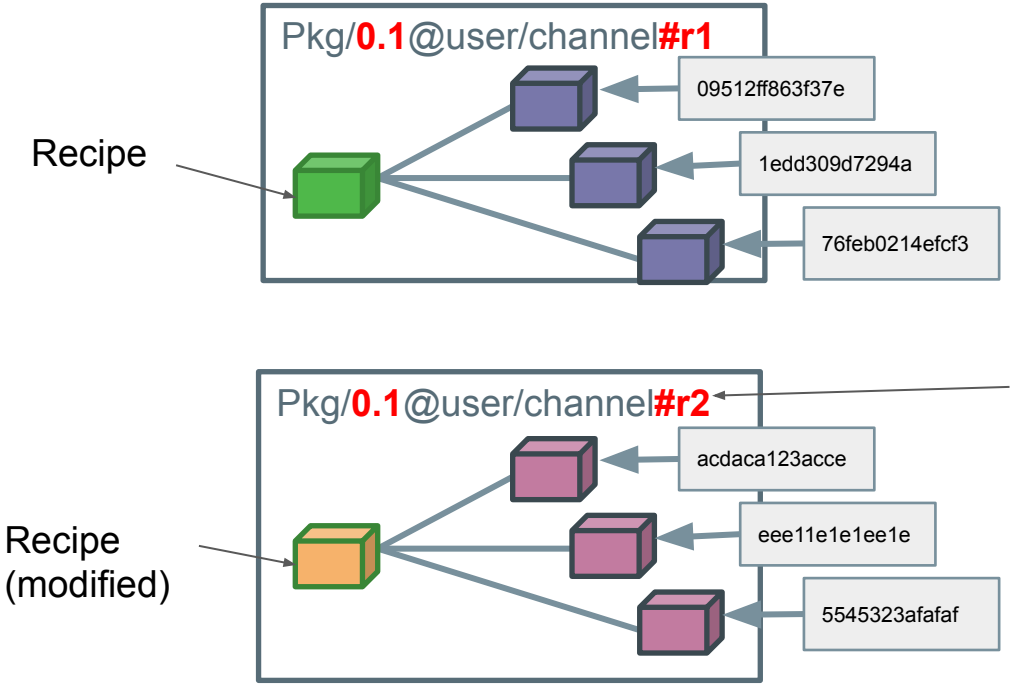
- Do a change in C++ code and create a “hello/0.1” new revision

Success:

- See the new message from the hello() function without modifying the “chat” conanfile.py



Recipe & Binary Management



pkg/version@user/channel#rrev
#rrev = hash(recipe contents)

Exercise 25.a - Package revisions [/revisions]

3:00

```
# Configuration
$ conan remote add artifactory http://localhost:8081/artifactory/api/conan/conan-local

# Enable revisions & remove previous packages
$ conan config get  # show the conan.conf file
$ conan config set general.revisions_enabled=True
$ conan config get  # show the conan.conf file

# Remove previous “hello” packages
$ conan remove hello* -f
```

Exercise 25.b - Package revisions [/revisions]

3:00

```
$ cd revisions

# Create and upload package “hello”
$ conan create hello user/testing
$ conan upload hello* --all -r=artifactory --confirm

# Do some change in code hello.cpp
$ conan create hello user/testing
$ conan upload hello* --all -r=artifactory --confirm

# check in Artifactory (in web UI, URL in Orbitera)
$ conan search hello/0.1@user/testing --revisions -r=artifactory
```

Package revisions (consuming)

```
$ conan remove hello* -f

$ conan install hello/0.1@user/testing
# By default latest revision
$ conan remove hello* -f

# Can be explicit (for debugging)
$ conan install hello/0.1@user/testing#<revision>
```

Exercise 26– Lockfiles [/lockfiles]

Goal:

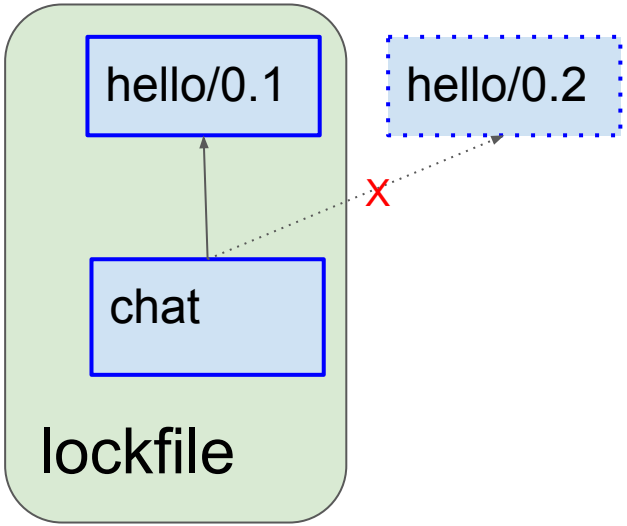
- Learn to generate and use a lockfile

Task:

- Create the “chat” package with a dependency to “hello/0.1” after creating “hello/0.2”

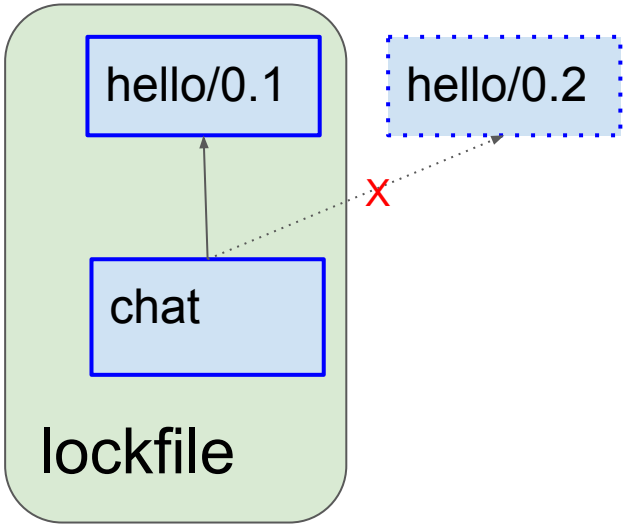
Success:

- See the old message from the “hello/0.1” version while creating the “chat” package



Exercise 26– Lockfiles [/lockfiles]

- A snapshot of a dependency graph at a given time.
- Can be used to reconstruct the exact same graph of dependencies



Exercise 26 - Lockfiles

3:00

```
$ cd lockfiles
$ conan remove hello* -f
$ conan create hello hello/0.1@user/testing
$ conan graph lock chat # will generate a conan.lock file
# change hello/src/hello.cpp message
$ conan create hello hello/0.2@user/testing

$ conan create chat user/testing # NOT locked (hello/0.2)
$ conan create chat user/testing --lockfile # locked (hello/0.1)
```

Outline

- Essentials (Ex 1-15, precondition)
- Requirements
  - Build-requires
  - Python-requires
- Versioning
  - Version ranges
  - Revisions
  - Lockfiles
- **Package ID**
- Hooks and Conan configuration

Exercise 27– Package ID [/package_id]

Goal:

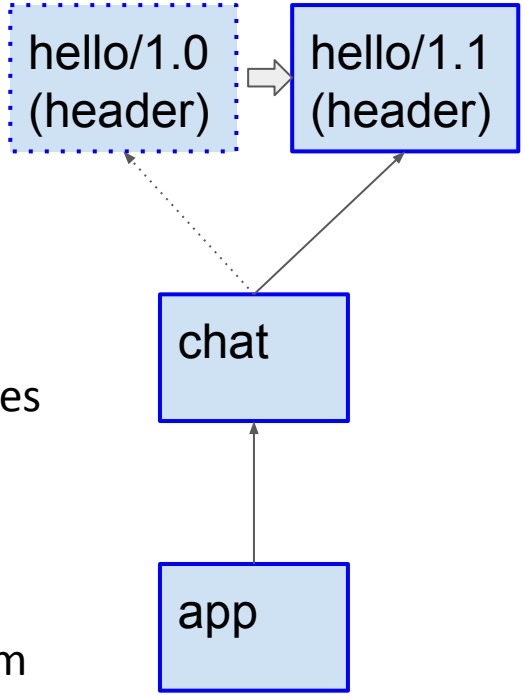
- Learn about package IDs and use different “package_id” modes

Task:

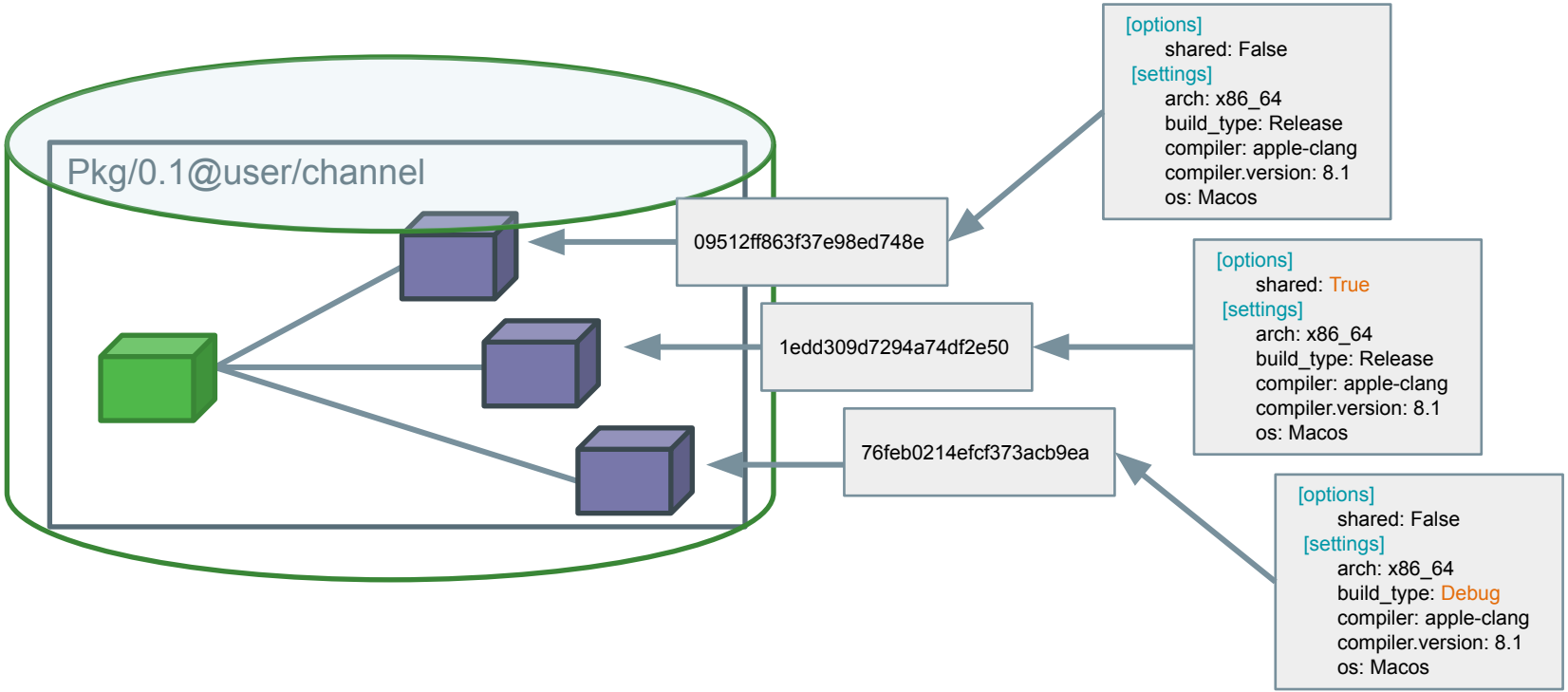
- Change the “package_id” default mode, so changes in a header only lib (without bumping the major version) causes a re-build of consumers

Success:

- See that consumer of package uses the right message from the modified header-only lib



Package ID

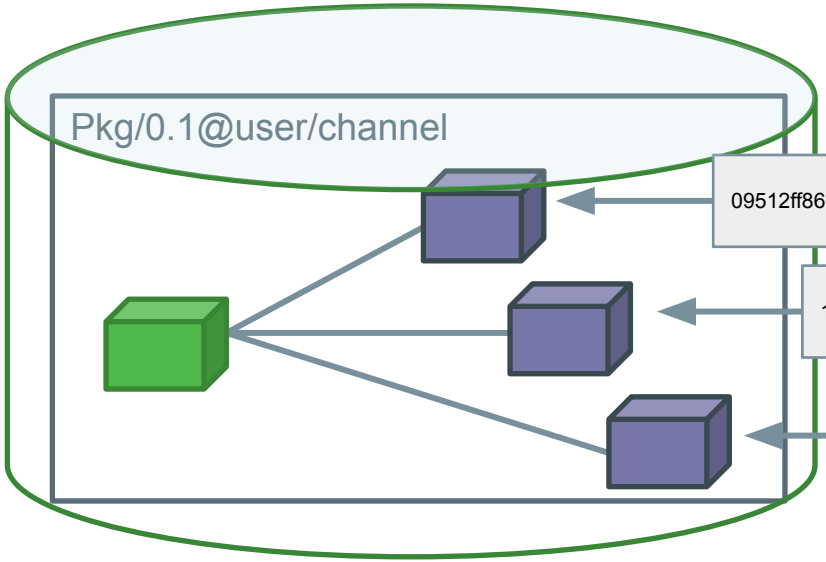


package_id()

conanfile.py

```
class Pkg(ConanFile):

    def package_id(self):
        if self.settings.compiler == "gcc" and
                self.settings.compiler.version == "4.9":
            for version in ("4.8", "4.7"):
                compatible_pkg = self.info.clone()
                compatible_pkg.settings.compiler.version = version
                self.compatible_packages.append(compatible_pkg)
```

Package ID



09512ff863f37e98ed748e

1edd309d7294a74df2e50

76feb0214efcf373

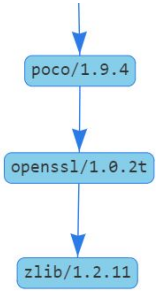
[options]
shared: False
[settings]
arch: x86_64
build_type: Release
compiler: apple-clang
compiler.version: 8.1
os: Macos
[requires]
poco/1.Y.Z

[options]
shared: False
[settings]
arch: x86_64
build_type: Debug
compiler: apple-clang
compiler.version: 8.1
os: Macos

package_id() (in recipes)

conanfile.py

```
class Pkg(ConanFile):

    requires = "poco/1.9.4"
    # using the default package_id()
    # only the semver of the direct deps
```

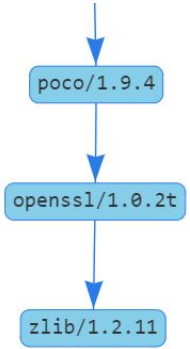


[options]
    shared: False
[settings]
    arch: x86_64
    build_type: Release
    compiler: apple-clang
    compiler.version: 8.1
    os: Macos
[requires]
    poco/1.X.Y

package_id() (in recipes)

conanfile.py

```
class Pkg(ConanFile):

    requires = "poco/1.9.4"

    def package_id(self):
        # apply full_version_mode for all
        self.info.requires.full_version_mode()
```



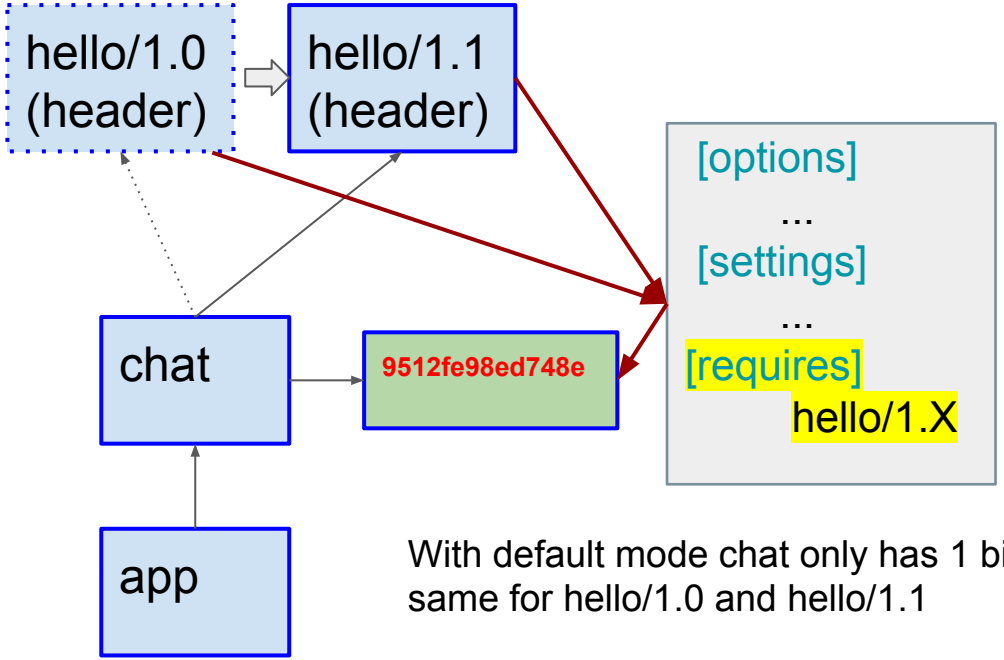
[options]
shared: False
[settings]
arch: x86_64
build_type: Release
compiler: apple-clang
compiler.version: 8.1
os: Macos
[requires]
zlib/1.2.11
openssl/1.0.2t
poco/1.9.4

default_package_id_mode (conan.conf)

conan.conf

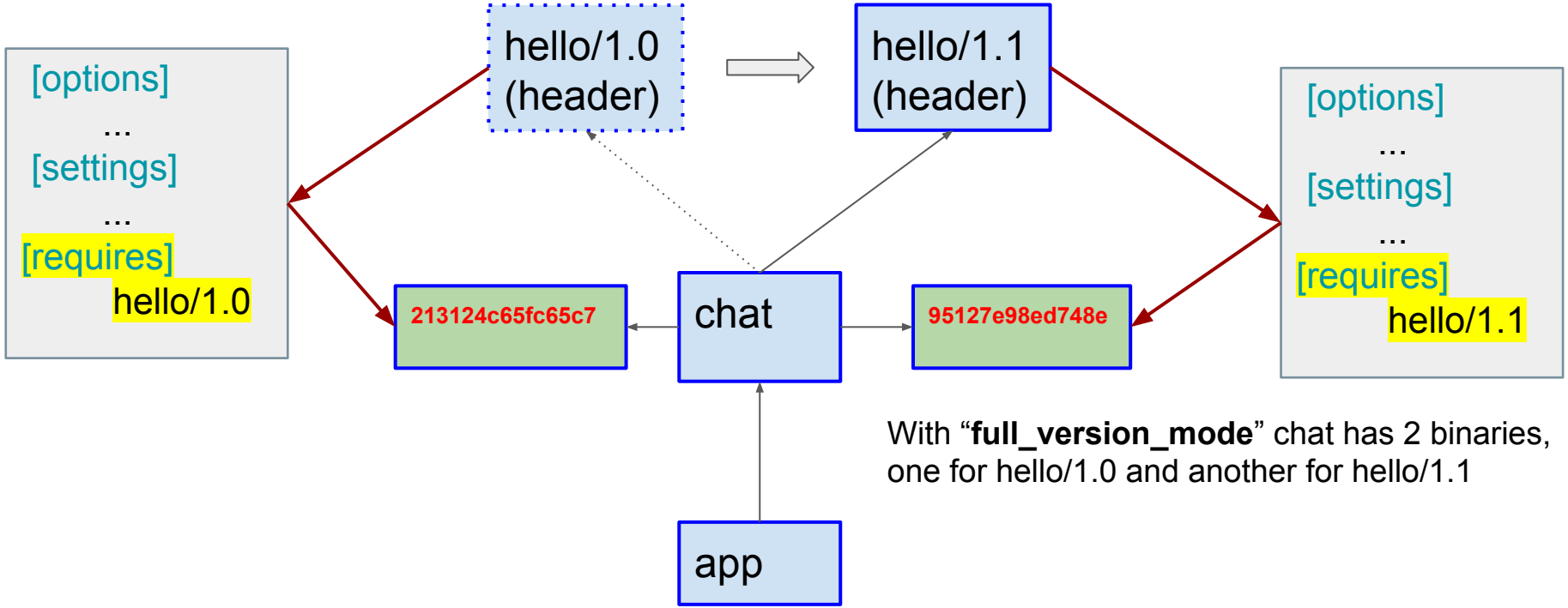
```
[general]
default_package_id_mode=full_version_mode
```

Exercise 27– Package ID (default mode)



With default mode chat only has 1 binary, the same for hello/1.0 and hello/1.1

Exercise 27– Package ID (full_version_mode)



With "**full_version_mode**" chat has 2 binaries, one for hello/1.0 and another for hello/1.1

Exercise 27– Package ID [/package_id]

3:00

```
$ cd package_id
$ conan remove "*" -f
$ conan create hello hello/1.0@user/testing
$ conan create chat user/testing
$ conan create app user/testing # check the package IDs

# do a change in hello/src/hello.h, create new hello/1.1 version
$ conan create hello hello/1.1@user/testing
$ conan create app user/testing   # What?! why not using the new hello/1.1 code?

$ conan config set general.default_package_id_mode=full_version_mode
$ conan create app user/testing  # Error! now we don't have binary for chat
$ conan create app user/testing --build=missing
$ conan search chat/1.0@user/testing # check the different package-IDs
```

Outline

- Essentials (Ex 1-15, precondition)
- Requirements
  - Build-requires
  - Python-requires
- Versioning
  - Version ranges
  - Revisions
  - Lockfiles
- Package ID
- **Hooks and Conan configuration**

Exercise 28– Hooks & config install [/hooks]

Goal:

- Install a “export” hook that avoid UpperCase package names

Task:

- Install the configuration from a folder, that contains a hook

Success:

- Try to create a package with Hello (uppercase) name, and see the hook raising an error

Hooks

- Hooks are users extensions, written in python, that are executed at certain points:
  - pre_build(), post_build(), pre_package(), post_package()...
- Should be orthogonal to recipes: custom checks, auxiliary logic.
- Stored in cache: <userhome>/.conan/hooks
- Activated in: <userhome>/.conan/conan.conf

Hooks

```
$ vim myconfig/hooks/check_name.py
```

```
def pre_export(output, conanfile, conanfile_path,
               reference, **kwargs):
    ref = str(reference)
    if ref.lower() != ref:
        raise Exception("%s should be lowercase" % ref)
```

Hooks: how to activate them

```
# Copy hook in <username>/.conan/hooks
$ cp myconfig/hooks/check_name.py ~/.conan/hooks

# Activate in conan.conf
$ vim ~/.conan/conan.conf
[hooks]
check_name
```

conan config install

- Command that can install/update in cache:
  - Add/update: hooks, profiles
  - Update: settings.yml, remotes.txt
  - Add any other file (pylintrc)
- From:
  - A git repo (master branch)
  - A remote http zip file
  - A local zip file
  - A local folder
-

Exercise 28 - conan config install & hooks

3:00

```
$ conan config install myconfig # can be URL, git

$ cd hooks
$ conan new Hello-Pkg/0.1 -s
$ conan export . user/testing # Error
$ conan new hello-pkg/0.1 -s
$ conan export . user/testing # OK
# Edit the hook in “myconfig” to raise for “-” char
$ conan config install # No arg! It memorizes
$ conan export . user/testing # Error
```

Summary

- Requirements:
  - Do not abuse special requirements:
    - Build-requirements only for tools
- Versioning:
  - Decide your policy: manual < version-ranges < revisions
- Lockfiles:
  - Key to CI at scale
- Package-ID:
  - ABI compatibility is complicated, package_id() powerful
  - Maybe the default “semver” mode is not enough
- Hooks, “conan config install”
- Other trainings: write to “conandays@jfrog.com”

Homework

- Create your own project with:
  - all packages are your recipes
  - build-requires and python-requires
  - conditional requires
- Enable revisions and change to “recipe_revision_mode”
- Do a change in an upstream dependency, and rebuild the project

Resources

- Docs: https://docs.conan.io/
  - Read carefully, explore.
- Issues:
  - CppLang slack (community)
  - Github issues (https://github.com/conan-io/conan) “official” support
- Following trainings:
  - conandays@jfrog.com
- Other Conan questions?
  - info@conan.io

THANK YOU!