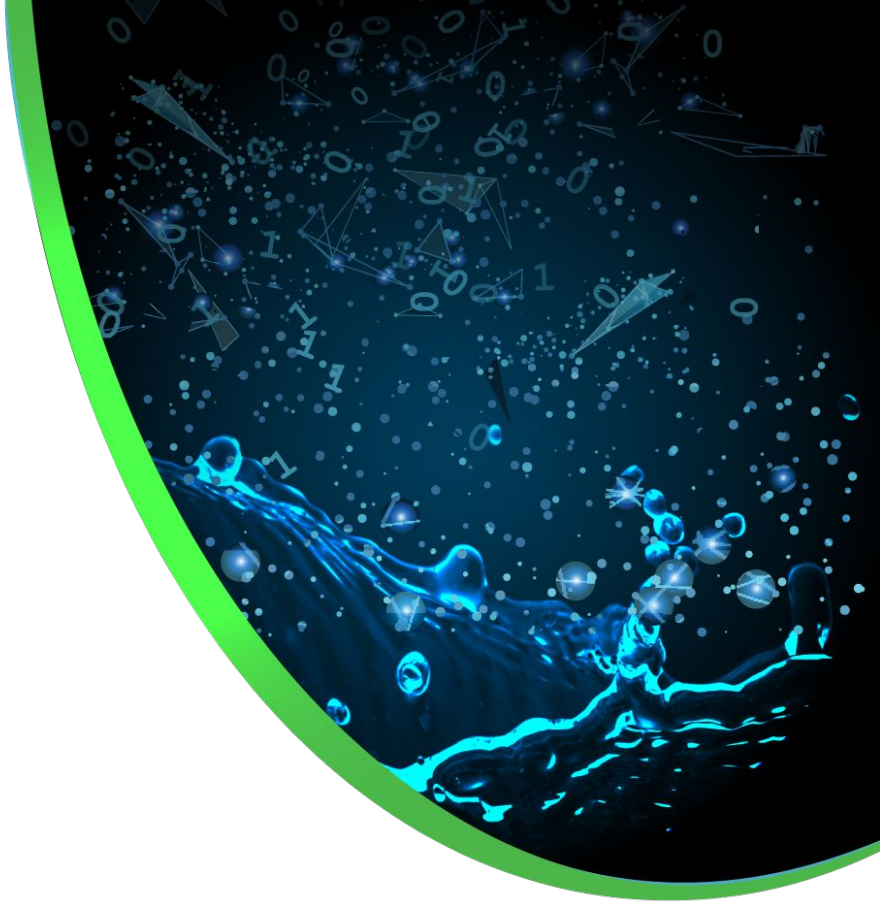
Launch Orbitera now

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

- PREREQUISITES
  - Wi-Fi enabled Mac or PC
  - SSH client
  - Internet Browser



Core C++ 2019

JFrog | CONAN C/C++ Package manager

Introduction to C/C++ Package Management with Conan & Artifactory

Diego Rodriguez-Losada, Conan Founder
Luis Martinez de Bartolome, Conan Founder

Outline

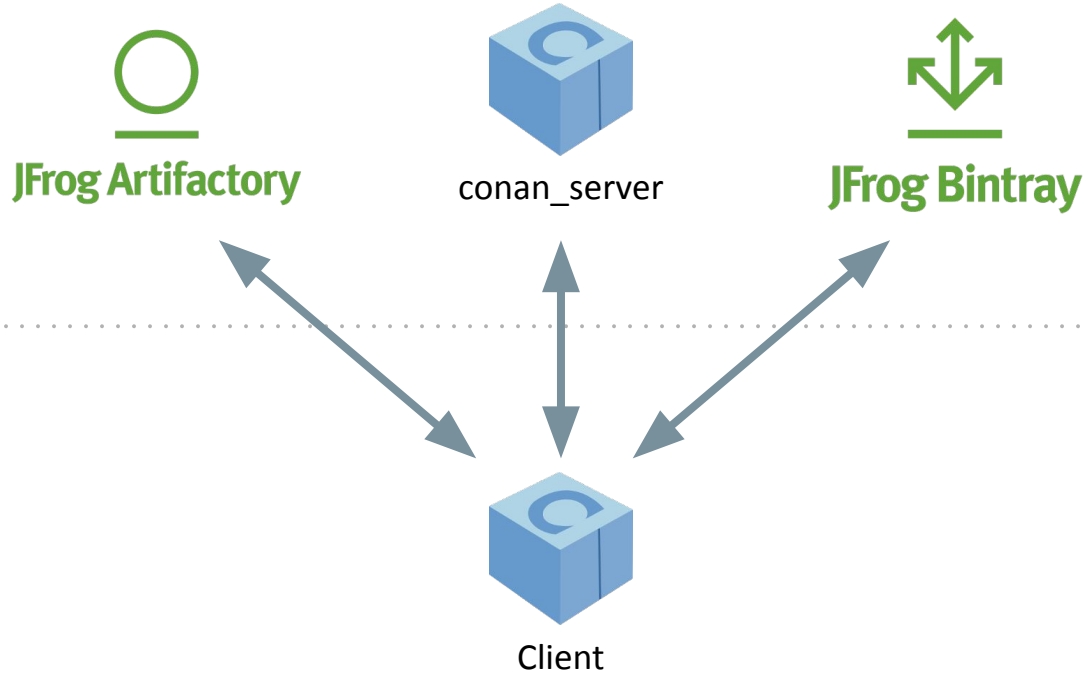
- **Introduction**
- Consume Conan Packages
- Create Conan Packages
- Uploading Packages to Artifactory
- Build Configuration & cross-build
- Special requirements
- Extensions and configuration
- Versioning approaches
- Jenkins Artifactory Conan CI

Introduction

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

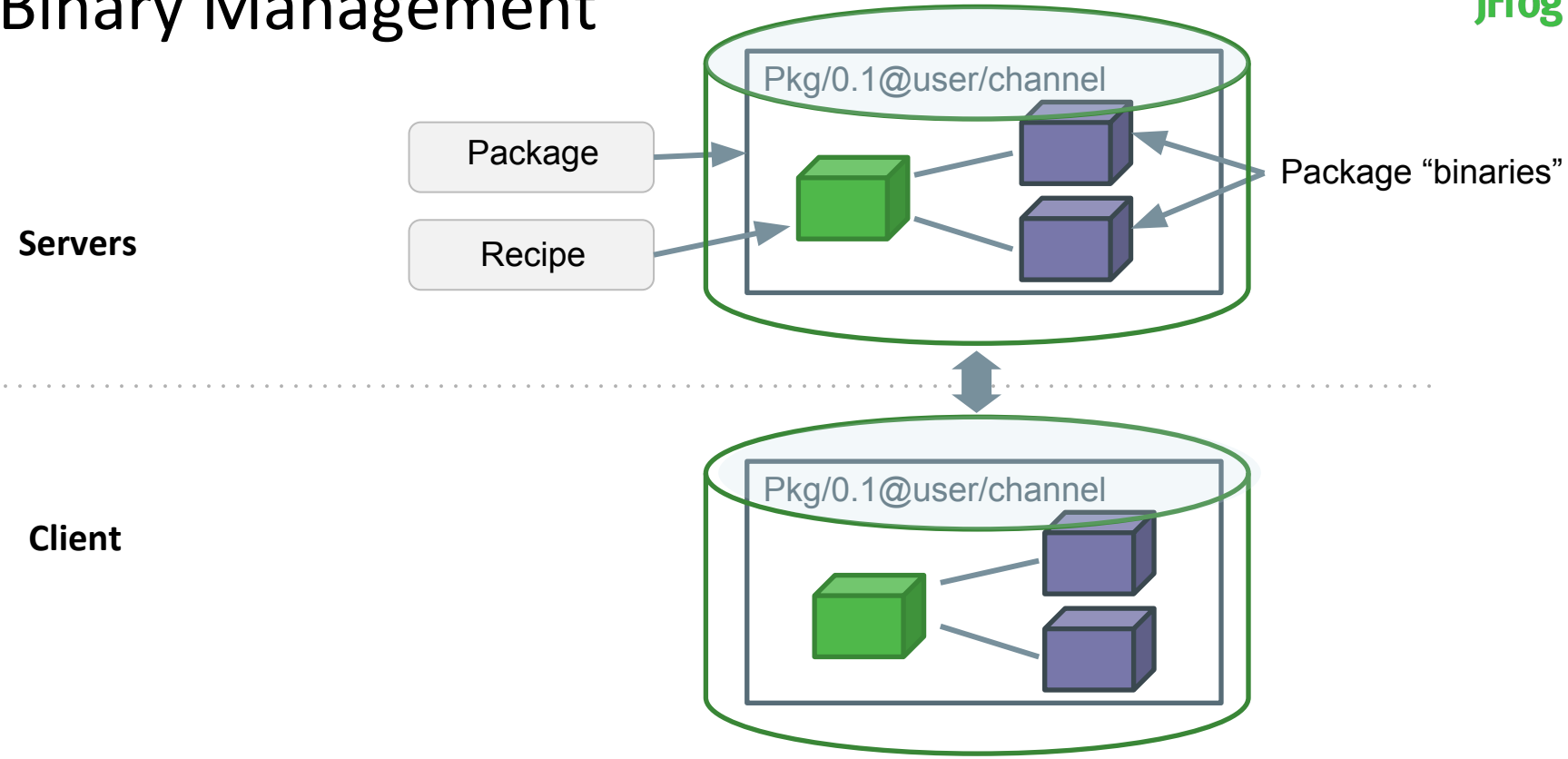
Architecture

Servers
(artifact storage)



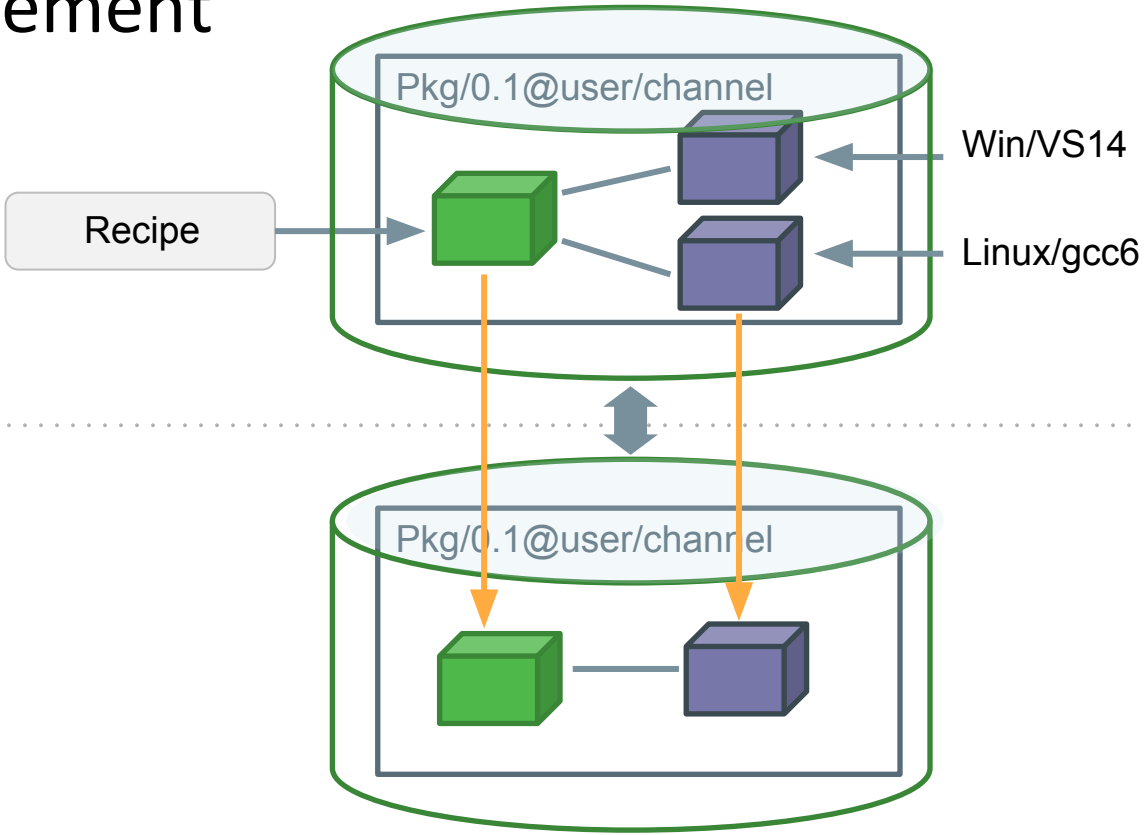
Developer
machine / CI

Binary Management



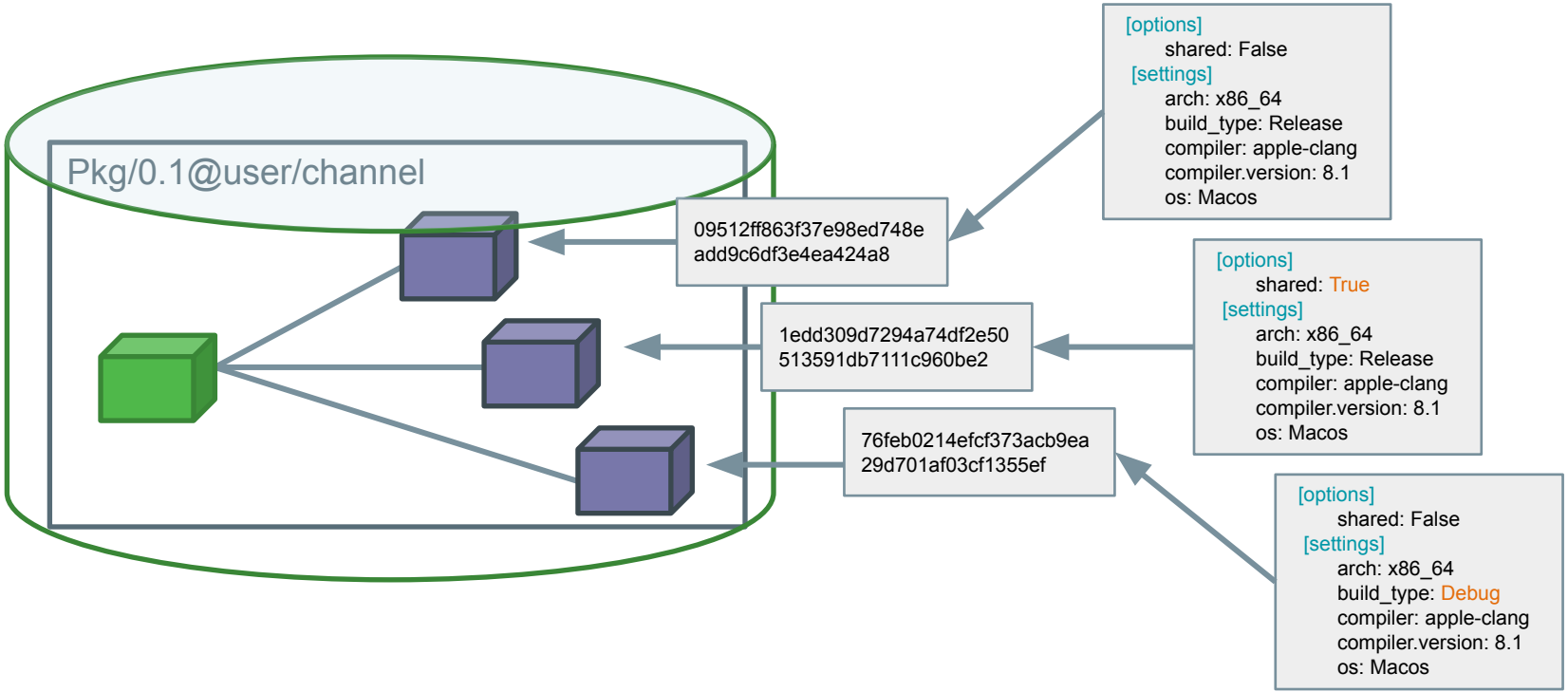
Binary Management

Servers



Client

Binary Management

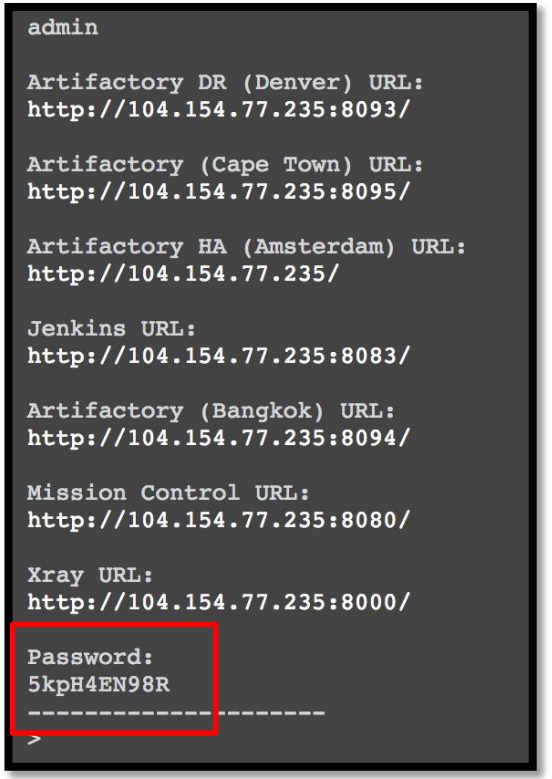


Outline

- Introduction
- **Consume Conan Packages**
- Create Conan Packages
- Uploading Packages to Artifactory
- Build Configuration & cross-build
- Special requirements
- Extensions and configuration
- Versioning approaches
- Jenkins Artifactory Conan CI

Exercise 1 - Setup

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

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



Exercise 2 - Consume

Servers
(artifact storage)

https://bintray.com/conan/conan-center

Developer
machine / CI

Client

Exercise 2 – Consume with CMake

```
$ cd consumer
```

timer.cpp

```
#include "Poco/Timer.h"
#include "Poco/Thread.h"
#include "Poco/Stopwatch.h"

#include <boost/regex.hpp>
#include <string>
#include <iostream>
```

conanfile.txt

```
[requires]
boost/1.67.0@conan/stable
Poco/1.9.0@pocoproject/stable

[generators]
cmake

[options]
Boost:shared=False
Poco:shared=False
```

CMakeLists.txt

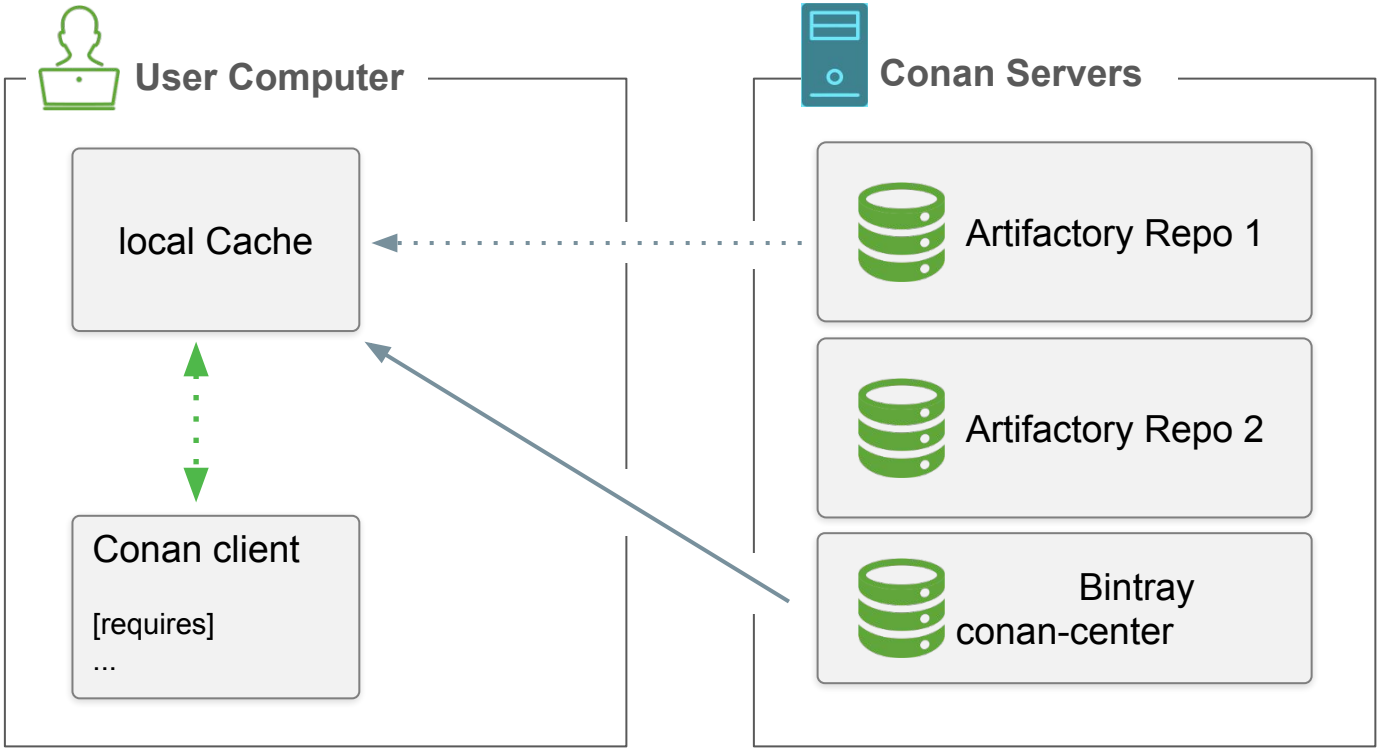
```
cmake_minimum_required(VERSION 2.8)
project(BoostPoco)
add_compile_options(-std=c++11)

include(${CMAKE_BINARY_DIR}/
                conanbuildinfo.cmake)
conan_basic_setup()


add_executable(timer timer.cpp)
target_link_libraries(timer ${CONAN_LIBS})
```

Exercise 2 - Consume with CMake

```
$ mkdir build && cd build
$ conan install .. # check the generated conanbuildinfo.cmake
$ cmake .. –DCMAKE_BUILD_TYPE=Release
$ cmake --build .    # or make
$ bin/timer
>...
```

$ ../catchup.sh # option 2

Exercise 2 – How Conan Installs Packages



Installed Packages (search)

```
$ conan search
$ conan search zlib/1.2.11@conan/stable
```

Exercise 3 - Consume (debug mode)

```
$ conan install .. – s build_type=Debug
# note that new packages are installed
$ cmake .. –DCMAKE_BUILD_TYPE=Debug
$ cmake --build .
$ bin/timer
>…
$ conan search zlib/1.2.11@conan/stable
```

$ ../catchup.sh # option 3

Conan Info & Search

```
$ conan search
$ conan search zlib/1.2.11@conan/stable   # add --table=file.html
$ conan info ..   # --graph=file.html
```

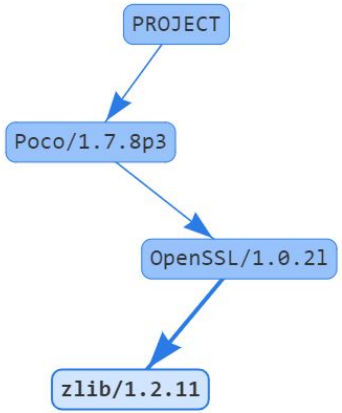
zlib/1.2.11@conan/stable

	x86 Debug shared=False	x86 Debug shared=True	x86 Release shared=False	x86 Release shared=True	x86_64 Debug shared=False	x86_64 Debug shared=True	x86_64 Release shared=False	x86_64 Release shared=True
Linux clang 3.9								
Linux clang 4.0								
Linux gcc 4.9								
Linux gcc 5.4								
Linux gcc 6.3								
Macos apple-clang 7.3								
Macos apple-clang 8.0								
Macos apple-clang 8.1								
Windows Visual Studio 10 (MD)								
Windows Visual Studio 10 (MDd)								
Windows Visual Studio 10 (MT)								
Windows Visual Studio 10 (MTd)								
Windows Visual Studio 12 (MD)								
Windows Visual Studio 12 (MDd)								
Windows Visual Studio 12 (MT)								
Windows Visual Studio 12 (MTd)								
Windows Visual Studio 14 (MD)								
Windows Visual Studio 14 (MDd)								
Windows Visual Studio 14 (MT)								
Windows Visual Studio 14 (MTd)								
Windows Visual Studio 15 (MD)								
Windows Visual Studio 15 (MDd)								
Windows Visual Studio 15 (MT)								
Windows Visual Studio 15 (MTd)								
Windows gcc 4.9 (dwarf2) (posix)								
Windows gcc 4.9 (seh) (posix)								
Windows gcc 4.9 (sjlj) (posix)								

Selected:
52365c918e417dff048f3ad367c434eb2c362d08

Legend

Outdated from recipe
Updated
Non existing



Exercise 4 - Consume (gcc generator)

```
$ cd ../consumer_gcc
$ ls # Look Ma, no build system!
$ conan install . – g gcc
# check conanbuildinfo.gcc
$ g++ timer.cpp @conanbuildinfo.gcc -o timer -std=c++11
$ ./timer
>…
```

$ ../catchup.sh # option 4

Generators

- Visual Studio
  - Legacy
  - Multi
- Cmake
  - Multi
- XCode
- pkg-config
- boost
- qmake, qbs, premake
- virtualrunenv, virtualbuildenv
- YOUR OWN!

Outline

- Introduction
- Consume Conan Packages
- **Create Conan Packages**
- Uploading Packages to Artifactory
- Build Configuration & cross-build
- Special requirements
- Extensions and configuration
- Versioning approaches
- Jenkins Artifactory Conan CI

Exercise 5 – Create Package (from github src)

- “Hello” library in https://github.com/memsharded/hello.git
- All we need is a conanfile.py “recipe”:
  - source()
  - build()
  - package()
  - package_info()

```python
class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"
    generators = "cmake"

    def source(self):
        self.run("git clone https://github.com/memsharded/hello.git")
        self.run("cd hello && git checkout static_shared")

    def build(self):
        cmake = CMake(self)
        cmake.configure(source_folder="hello")
        cmake.build()

    def package(self):
        self.copy("*.h", dst="include", src="hello")
        self.copy("*.lib", dst="lib", keep_path=False)
        self.copy("*.a", dst="lib", keep_path=False)

    def package_info(self):
        self.cpp_info.libs = ["hello"]
```

Exercise 5 – Create Package (from github src)

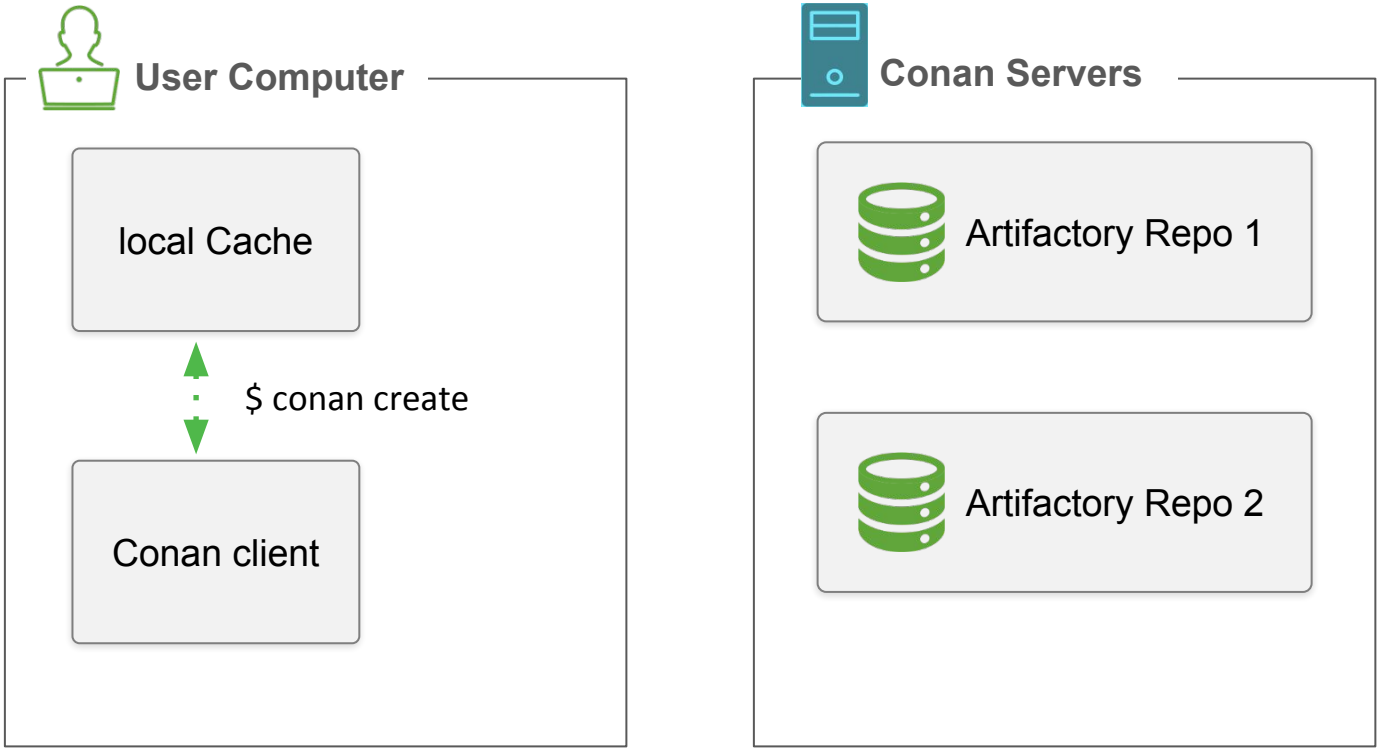
```
$ cd ../create
$ conan new Hello/0.1  # just a template
# check the conanfile.py
$ conan create . user/testing
> …
$ conan search
$ conan search Hello/0.1@user/testing
```

Fetching the sources from: https://github.com/memsharded/hello

Exercise 5 – Create Package (from github src)

```
$ conan create . user/testing –s build_type=Debug
> …
$ conan search Hello/0.1@user/testing
```

Conan Create is Local



Exercise 5 – Create & Test Package (from github src)

```
$ conan new Hello/0.1 -t # The –t generates test_package
```

conanfile.py

```
class HelloConan(ConanFile):
    name = “Hello"
    version = "1.0"
    def source(self):
    def build(self):
    def package(self):
    def package_info(self):
```

test_package/conanfile.py
(reuse conanfile, similar to conanfile.txt)

```
from conans import ConanFile, CMake


class MylibTestConan(ConanFile):
    def build(self):
    def imports(self):
    def test(self):
```

test_package/example.cpp
(reuse cpp example)

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

int main() {
    hello();
}
```

Exercise 5 – Create & Test Package (from github src)

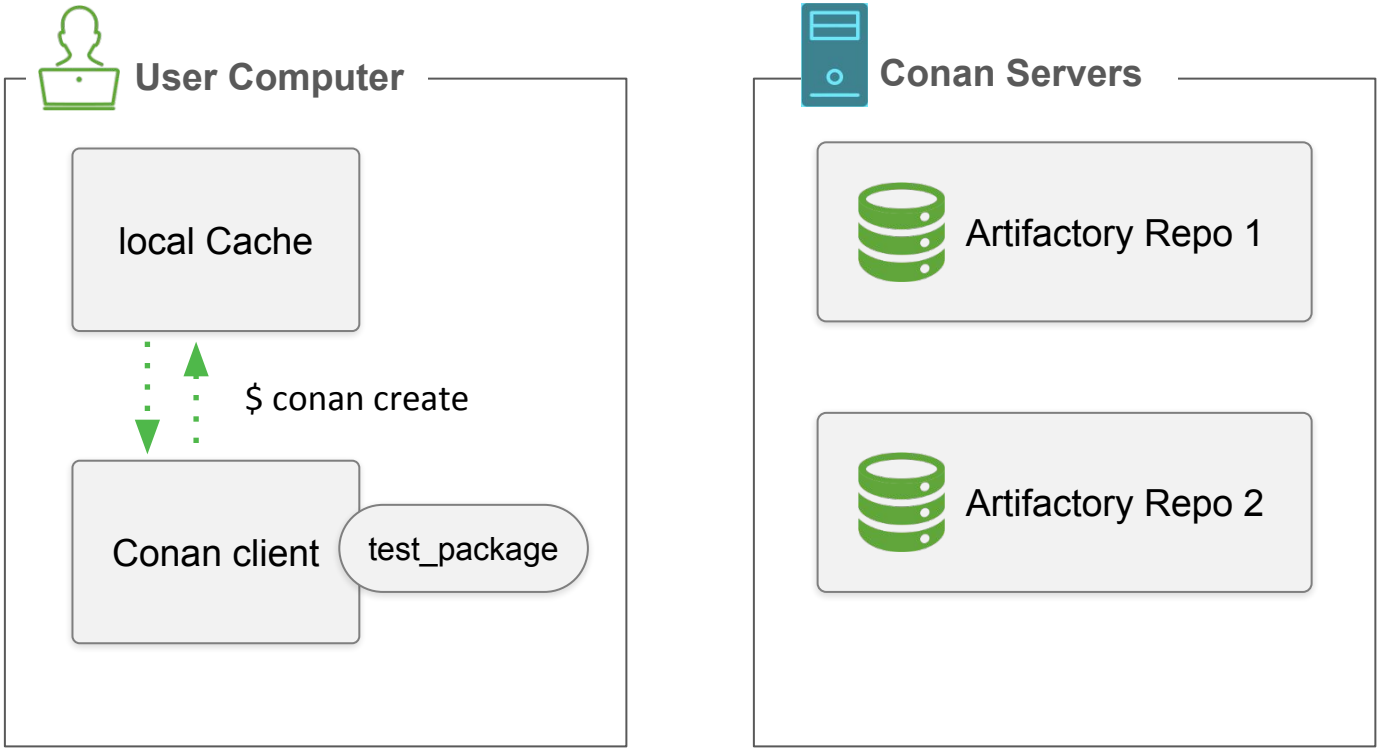
```
$ conan new Hello/0.1 -t # -t generates test_package
$ conan create . user/testing
> …# check output
> Hello World!
```

Exercise 5 – Create & Test Package (from github src)

```
$ conan create . user/testing  -s build_type=Debug
> …# check output
> Hello World!
```

```
$ ../catchup.sh # option 5
```

Conan Create (with test_package) is Local



Exercise 6 – Create (from src repo)

```
$ cd ../create_sources
$ conan new Hello/0.1 -t -s # The –s generates example src
```

conanfile.py

```
class HelloConan(ConanFile):
    name = “Hello"
    version = “0.1"
    def source(self):
    def build(self):
    def package(self):
    def package_info(self):
```

src/CMakeLists.txt

```
project(MyHello CXX)
cmake_minimum_required(VERSION 2.8)

include(${CMAKE_BINARY_DIR}/
            conanbuildinfo.cmake)
conan_basic_setup()

add_library(hello hello.cpp)
```

src/hello.h & src/hello.cpp

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

void hello(){
 #ifdef NDEBUG
    std::cout << "Hello World Release!"
<<std::endl;
#else
    std::cout << "Hello World Debug!"
<<std::endl;
 #endif
}
```

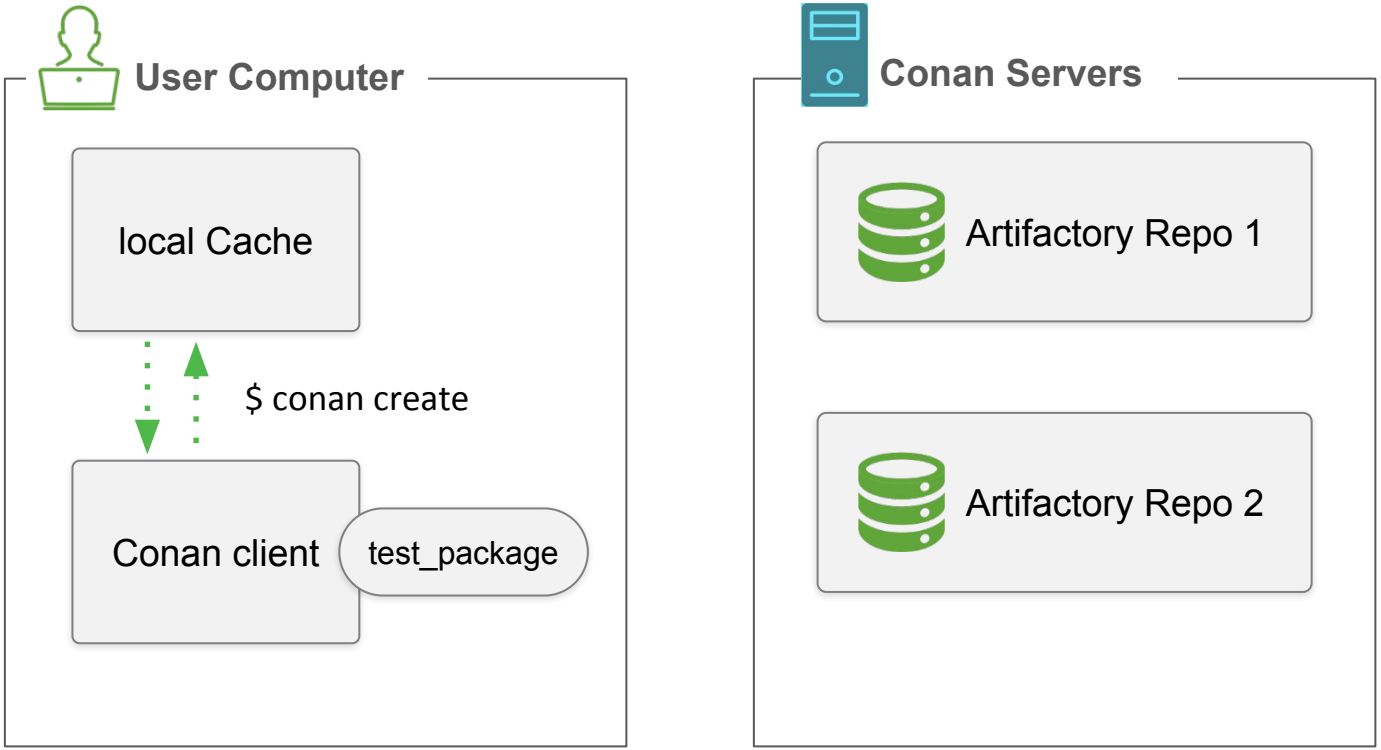
```
class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"
    generators = "cmake"
    exports_sources = "src/*"

    def build(self):
        cmake = CMake(self)
        cmake.configure(source_folder="hello")
        cmake.build()

    def package(self):
        self.copy("*.h", dst="include", src="hello")
        self.copy("*.lib", dst="lib", keep_path=False)
        self.copy("*.a", dst="lib", keep_path=False)

    def package_info(self):
        self.cpp_info.libs = ["hello"]
```

Exercise 6 – Create (from src repo)

```
$ conan create . user/testing
> …# check output
> Hello World Release!
$ conan create . user/testing –s build_type=Debug
> Hello World Debug!
```

$ ../catchup.sh # option 6

Conan Create (with test_package) is Local

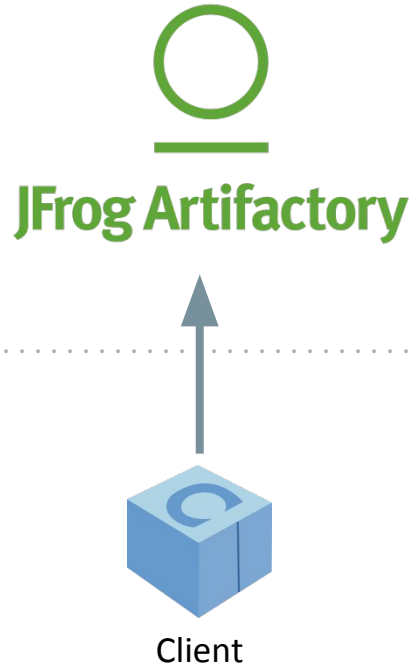


Outline

- Introduction
- Consume Conan Packages
- Create Conan Packages
- **Uploading Packages to Artifactory**
- Build Configuration & cross-build
- Special requirements
- Extensions and configuration
- Versioning approaches
- Jenkins Artifactory Conan CI

Exercise 7 – Upload to Artifactory

Servers
(artifact storage)



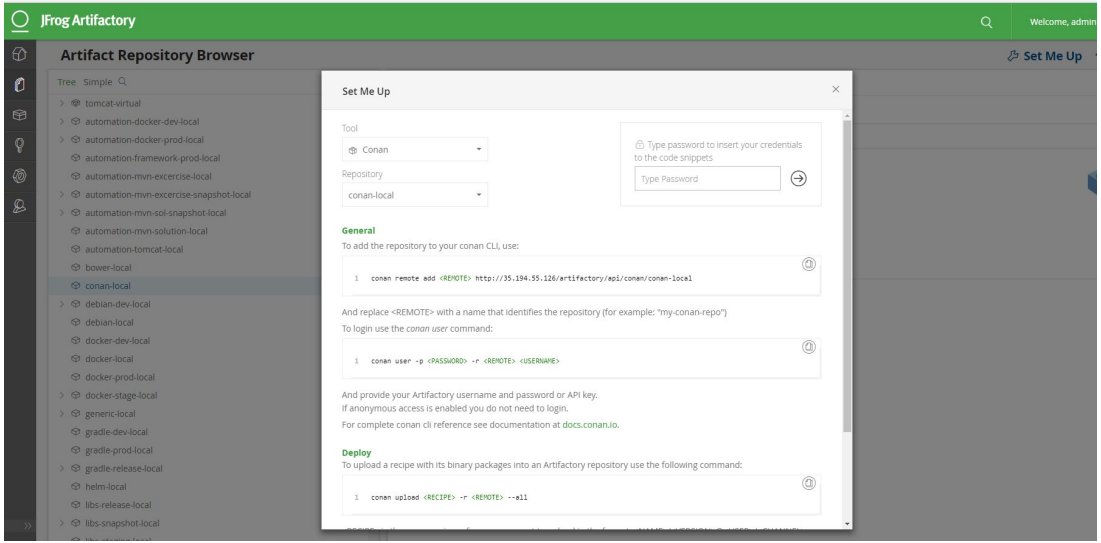
Developer
machine / CI

Conan Remotes

```
$ conan remote list
```

Artifactory

- Navigate to IP
  - Admin->Repositories->Local->New
- Create new conan repo **“myconanrepo”**
- Navigate to “Artifact browser”
  - Set Me Up



Exercise 7 – Upload Packages to Artifactory

```
$ conan remote add artifactory <URL from SetMeUp>
$ conan upload "Hello*" -r artifactory --all
$ conan search "*" -r=artifactory
$ conan search Hello/0.1@user/testing -r=artifactory
# Navigate to Artifactory WebUI and check!
```

Exercise 7 – Test Uploaded Packages

```
$ conan remove “Hello*”
$ conan test test_package Hello/0.1@user/testing
> …
$ conan test test_package Hello/0.1@user/testing -s build_type=Debug
> …
```

Exercise 7 – Upload ALL Packages to Artifactory

```
$ conan upload "*" -r artifactory --all --confirm
$ conan search "*" -r=artifactory
# Navigate to Artifactory WebUI and check!

$ conan remote remove conan-center
$ conan remove "*" –f
# go back to consumer, do install
```

$ ../catchup.sh # option 7

Outline

- Introduction
- Consume Conan Packages
- Create Conan Packages
- Uploading Packages to Artifactory
- **Build Configuration & cross-build**
- Special requirements
- Extensions and configuration
- Versioning approaches
- Jenkins Artifactory Conan CI

Conan Profiles

Conan allows to build/reuse packages with different configurations:

- Different build_type
- Different compiler versions
- Different compilers
- Cross building to a different architecture…
- Different options, (shared, static, active FPU, etc)

conan install . -s r=gcc -s compiler=4.8 -s arch=armv7 -s pe=Release -o zlib:shared=Tru

Conan Profiles

- Plain text files with settings + options + environment variables
  - **~/.conan/profiles**
- Can be applied to both conan install and conan create
- Can be shared between the team (standard confs for a company)
  - **$ conan config install**
- Env vars are very useful to enable cross building toolchains (CC, CXX)

```
[settings]
os=Linux
compiler=gcc
compiler.version=4.9
compiler.libcxx=libstdc++
build_type=Debug
arch=armv7

[env]
CC=arm-linux-gnueabihf-gcc
CXX=arm-linux-gnueabihf-g++
```

Conan Profiles

```
$ conan profile list
$ conan profile show default
```

Conan Settings (~/.conan/settings.yml)

```
os:
    Windows:
    Linux:
    Macos:
arch: [x86, x86_64, ppc64le, ppc64, …, armv7s, armv7k]
compiler:
    gcc:
        version: ["4.1", …"7.3"]
        libcxx: [libstdc++, libstdc++11]
    Visual Studio:
        runtime: [MD, MT, MTd, MDd]
        version: ["8", "9", "10", "11", "12", "14", "15"]
        toolset: [None, v90, ..v141_clang_c2]
    clang:
        version: ["3.3", … "6.0"]
```

Exercise 8 – Cross Build Hello Package to R-PI

```
$ cd exercises/create_sources
$ less ../profile_arm/arm_gcc_debug.profile
# press “q” to exit less
```

```
[settings]
os=Linux
compiler=gcc
compiler.version=6
compiler.libcxx=libstdc++11
build_type=Debug
arch=armv7
os_build=Linux
arch_build=x86_64

[env]
CC=arm-linux-gnueabihf-gcc
CXX=arm-linux-gnueabihf-g++
```

Exercise 8 – Cross Build Hello Package to R-PI

```
$ conan create . user/testing -pr=../profile_arm/arm_gcc_debug.profile
> ...
$ conan search
$ conan search Hello/0.1@user/testing
```

```
$ ../catchup.sh # option 8
```

Exercise 9 – Consume & Cross Build Zlib

$ cd ../profile_arm

conanfile.py

```
class HelloConan(ConanFile):
    def build(self):
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)
project(ZlibARM)

include(${CMAKE_BINARY_DIR}/
                conanbuildinfo.cmake)
conan_basic_setup()

add_executable(example example.c)
target_link_libraries(example ${CONAN_LIBS})
```

arm_gcc_debug.profile

```
[settings]
os=Linux
compiler=gcc
compiler.version=6
compiler.libcxx=libstdc++11
build_type=Debug
arch=armv7
os_build=Linux
arch_build=x86_64

[env]
CC=arm-linux-gnueabihf-gcc
CXX=arm-linux-gnueabihf-g++
```

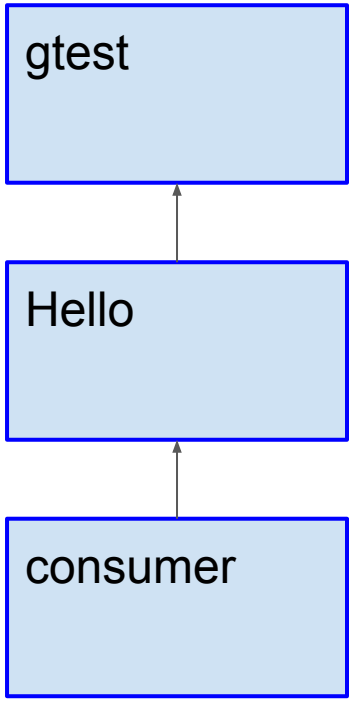
Exercise 9 – Consume & Cross Build Zlib

```
$ mkdir build && cd build
$ conan install .. -pr=../arm_gcc_debug.profile

ERROR: Missing prebuilt package for 'zlib/1.2.11@conan/stable'
Try to build it from sources with "--build zlib“
$ conan install .. -pr=../arm_gcc_debug.profile --build=missing
$ conan search zlib/1.2.11@conan/stable
$ conan build .. # local flow
$ bin/example # error! It is armv7
                                        $ ../catchup.sh # option 9
```

Outline

- Introduction
- Consume Conan Packages
- Create Conan Packages
- Uploading Packages to Artifactory
- Build Configuration & cross-build
- **Special requirements**
- Extensions and configuration
- Versioning approaches
- Jenkins Artifactory Conan CI

Exercise 10 – Unit Tests with gtest



Exercise 10 – Unit Tests with gtest

```
$ cd gtest/package
```

```
// In the test file
#include <gtest/gtest.h>
#include "hello.h"

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

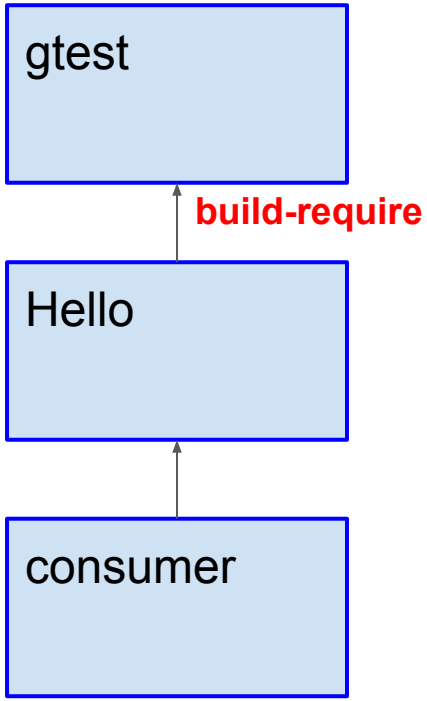
Exercise 11 – Unit Tests with gtest

```
class HelloConan(ConanFile):
 ...
    requires = "gtest/1.8.0@bincrafters/stable"
    default_options = "gtest:shared=False"

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

Exercise 10 – Unit Tests with gtest

```
# add remote conan-center: https://conan.bintray.com
# search in conan-center for gtest package
$ conan create . user/testing
# Check dependencies
$ cd ../consumer
$ conan install .
# check dependencies (gtest installed!)
```

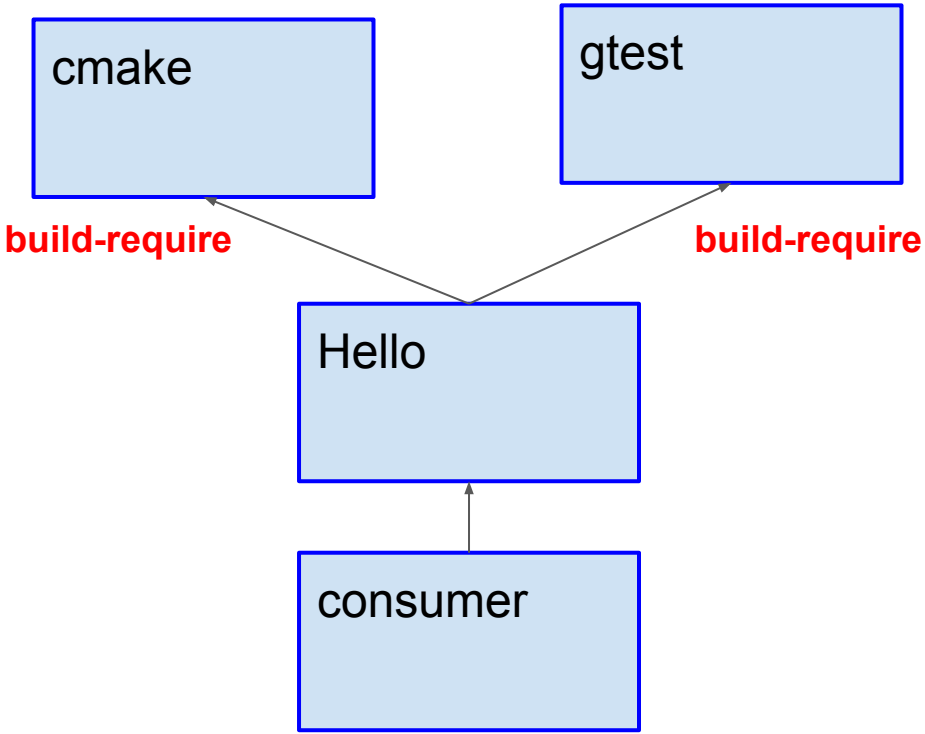
Exercise 11 – Unit Tests with gtest (build-require)



Exercise 11 – Unit Tests w. gtest (build-require)

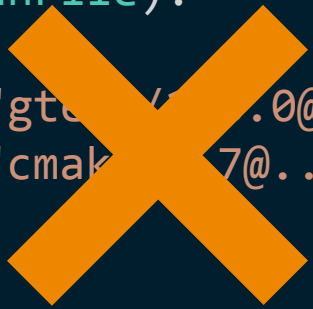
```
$ cd ../package
# change “requires” → “build_requires”
$ conan create . user/testing
$ cd ../consumer
$ conan install .
# check dependencies
```

Exercise 12 – CMake as build-require



Exercise 12 – CMake as build-require

```
class HelloConan(ConanFile):
 ...
    build_requires = "gtest/1.8.0@bincrafters/stable",
                     "cmake/3.7@...."
```



Exercise 12 – CMake from build_require

```
$ cmake --version
# add line in CMakeLists:
   message(STATUS "CMAKE VERSION ${CMAKE_VERSION}")
$ conan create . user/testing
# search for a “cmake” package in conan-center
$ vim myprofile
```

Exercise 12 – CMake from “build_require”

```
include(default)

[build_requires]
cmake_installer/3.3.2@conan/stable
```

```
$ conan create . user/testing –pr=myprofile
# Check cmake version!
$ cmake --version
```

A few notes about build_requires

- They shouldn’t change the binary
  - They are not taken into account in the package ID
- Use them 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)

Exercise 13 – CI Package Creation for PicoJson

- Try to package the open source library Pico JSON: https://github.com/kazuho/picojson.git
- Go to header_only folder,
  use the **example.cpp** for your test_package
- Hint: Use “conan new --help”

Exercise 14 - Python requires (mytools)

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

```
from conans import ConanFile

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

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

Exercise 14 - Python requires (reuse)

$ cd ../consumer

```
from conans import ConanFile, python_requires

mytools = python_requires("mytools/0.1@user/testing")

class ConsumerConan(ConanFile):
    def build(self):
        mytools.mymsg(self)
```

Exercise - Python requires (reuse)

```
$ conan create . consumer/0.1@user/testing
> … MyTool working cool message!!!
```

NOTES

- python-requires DO NOT have binary packages, only python code
- They do not affect the package-ID
- 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)

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, python_requires
mytools = python_requires("mytools/0.1@user/testing")

class Pkg(mytools.BaseConanFile):
    # inherits the source(), build()...
```

Outline

- Introduction
- Consume Conan Packages
- Create Conan Packages
- Uploading Packages to Artifactory
- Build Configuration & cross-build
- Special requirements
- **Extensions and configuration**
- Versioning approaches
- Jenkins Artifactory Conan CI

Exercise - Hooks

- Hooks are users extensions, written in python,at some 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
-

Exercise - Hooks

$ cd ../hooks && less 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)
    if "-" in ref:
        raise Exception("Use _ instead of -")
```

Exercise - Hooks

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

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

$ conan new Hello/0.1
$ conan create . user/testing
```

Exercise - conan config install

- Command that can install/update in cache:
  - Add/update: hooks, profiles, settings.yml
  - 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 - conan config install

```
$ rm  ~/.conan/hooks/check_name

# Deactivate in conan.conf
$ vim ~/.conan/conan.conf
#[hooks]

$ conan config install .
```

Outline

- Introduction
- Consume Conan Packages
- Create Conan Packages
- Uploading Packages to Artifactory
- Build Configuration & cross-build
- Special requirements
- Extensions and configuration
- **Versioning approaches**
- Jenkins Artifactory Conan CI

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, and need to do a change to the recipe?
    - 1.64.1? Mismatch to the original Boost version
  - Versions might use version ranges requirements
- Revisions:
  - pkg/version@user/channel#revision
  - revision is internal, automatic (hash)

Version ranges

```
$ cd ..
$ mkdir version_ranges && cd version_ranges
$ conan remove Hello* -f
$ conan new hello/0.1 -s
$ conan create . user/testing

# install a version range
$ conan install "hello/[>0.0 <1.0]@user/testing"
```

Version ranges

```
$ conan new hello/0.2 -s
$ conan create . user/testing
$ conan search

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

Version ranges

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

Revisions (creating packages with revisions)

```
$ conan config set general.revisions_enabled=True
# check the conan.conf

$ mkdir revisions && cd revisions
$ conan remove hello* -f
$ conan new hello/0.1 -s
$ conan create . user/testing
$ conan create . user/testing -s build_type=Debug
$ conan upload hello* --all -r=artifactory --confirm
# check in Artifactory
```

Revisions (creating packages with revisions)

```
$ echo "#comment" >> conanfile.py
$ conan create . user/testing
$ conan create . user/testing -s build_type=Debug
$ conan upload hello* --all -r=artifactory --confirm
# check in Artifactory
```

Revisions (consuming)

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

Outline

- Introduction
- Consume Conan Packages
- Create Conan Packages
- Uploading Packages to Artifactory
- Build Configuration & cross-build
- Special requirements
- Extensions and configuration
- Versioning approaches
- **Jenkins Artifactory Conan CI**

Exercise – Jenkins CI

```
$ docker exec -it jenkins /bin/bash
$ cd /var/lib/jenkins # We are going to create a new repo
$ mkdir hello && cd hello
$ conan new hello/0.1 –s –t   # lowercase!
$ git init .
$ git checkout -b release/0.1
$ git add .
$ git commit -m “initial release”
```

Exercise – Jenkins CI

```
# Go to Jenkins (IP:8083)
```

Exercise 10 – Jenkins CI

Configure Jenkins Job:

- New Item -> Multibranch Pipeline -> Give Name (conan-hello) -> OK
- Branch sources -> Add source -> Enter path to repo “/var/lib/jenkins/hello”
- Scan Multibranch Pipeline Triggers => Check “periodically” => 1 min
- Save button

Then:

- Check build, check logs

Jenkinsfile

```
def artifactory_name = "artifactory-ha"
def artifactory_repo = “conan-local"

node {
   def server = Artifactory.server artifactory_name
   def client = Artifactory.newConanClient()
   def serverName = client.remote.add server: server, repo: artifactory_repo
   stage("Get recipe"){
       checkout scm
   }

   stage("Build package"){
       client.run(command: "create . team/stable")
   }

   stage("Upload packages"){
       String command = "upload * --all -r ${serverName} --confirm"
       def b = client.run(command: command)
       server.publishBuildInfo b
   }
}
```

Exercise – Jenkins CI

```
$ wget
https://raw.githubusercontent.com/conan-io/training/master/jenkins/Jenkinsfile
$ git add .
$ git commit -m “Jenkinsfile”
```

Exercise – Jenkins CI

Generate a new package version

- Create new branch “release/0.2”
- Bump the version number in “conanfile.py” (and the .cpp code if you want)
- Commit the changes
- Check CI logs and Artifactory

Exercise – Jenkins CI

Generate revisions of every release version

- Enable revisions in the Jenkinsfile
- Do changes to the source code
- Commit
- Wait for Jenkins to create the revisions
- Check in Artifactory

Exercise - SCM

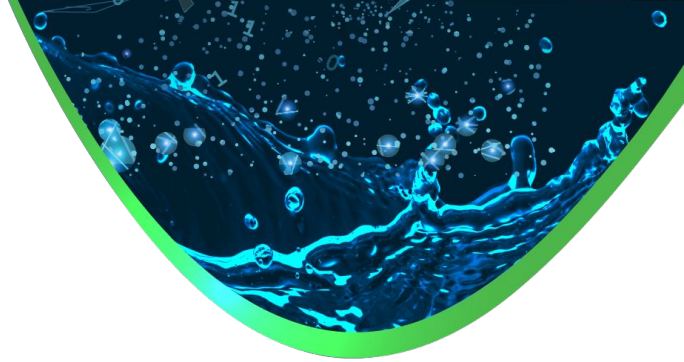
```
$ cd
$ cd training/scm
$ vim conanfile.py
```

- NO source() method necessary
- NO exports_sources necessary
- It captures the url & revision
- It does NOT capture the sources
- It can reproduce the build

```
scm = {"type": "git",
       "url": "auto",
       "revision": "auto"}
```

Exercise - SCM

```
$ conan create . user/testing
$ conan get hello/0.1@user/testing
```



THANK YOU!